



An Analysis of the Causes of Code Growth in Genetic Programming

TERENCE SOULE
ROBERT B. HECKENDORN

tsoule@cs.uidaho.edu
heckendo@cs.uidaho.edu

Computer Science, University of Idaho, Moscow, ID 83844-1010

Submitted September 21, 2001; Revised January 23, 2002

Abstract. This research examines the cause of code growth (bloat) in genetic programming (GP). Currently there are three hypothesized causes of code growth in GP: protection, drift, and removal bias. We show that single node mutations increase code growth in evolving programs. This is strong evidence that the protective hypothesis is correct. We also show a negative correlation between the size of the branch removed during crossover and the resulting change in fitness, but a much weaker correlation for added branches. These results support the removal bias hypothesis, but seem to refute the drift hypothesis. Our results also suggest that there are serious disadvantages to the tree structured programs commonly evolved with GP, because the nodes near the root are effectively fixed in the very early generations.

Keywords: genetic programming, code growth, code bloat, crossover

1. Introduction

The tendency of programs generated with genetic programming (GP) to grow without corresponding increases in fitness (known as code growth or code bloat) is well documented in the GP literature [1, 7, 12, 13, 16, 20, 23] and has been demonstrated in non-tree based evolutionary paradigms 15–17. Further, the current research on code growth in GP strongly suggests that it will occur in any evolutionary technique which uses variable size representations [13, 20] and Langdon has shown that growth can occur in non-population based search techniques [8].

Importantly, most code growth consists of code which does not significantly contribute to a program's performance. Thus, code growth is a significant problem, as the additional code consumes considerable resources without directly improving the solution. Additionally, code growth may interfere with continued exploration of the search space, since most of the code manipulation by evolutionary operators will occur in regions of relatively low importance.

However, code growth also demonstrates the strengths of evolutionary techniques. In part, code growth is believed to be a protective response to the destructive effects of crossover and mutation. This protective effect was an unanticipated side effect of the evolutionary process. Thus, code growth is a demonstration that the evolutionary process is capable of recognizing and evolving solutions to problems that the GP practitioner may not have anticipated. Clearly this is a highly desirable feature for a machine learning system.

To date three likely explanations for code growth of been proposed: growth for protection [1, 13, 16], a form of genetic drift towards larger solutions [11], and a removal bias in crossover that leads to growth [22]. These three causes are not mutually exclusive and there is some evidence in support of each cause. However, significant questions have been raised concerning each of these causes [12].

In this paper we present very strong experimental evidence for the existence of two of these causes, protection and removal bias, while calling into question the existence of a drift based cause of growth. Protective growth is shown to be a very sensitive feature of evolutionary algorithms, with very small amounts of selective pressure creating very large increases in growth. Additionally, removal bias is shown to be a very significant factor in code growth with serious negative ramifications for population diversity and successful evolution in tree based evolutionary techniques.

2. Background

2.1. Crossover and program size

In a basic GP, with crossover and without subtree mutation, crossover produces the variation in tree sizes. Removing a small branch and adding a large branch during crossover creates a larger tree, whereas removing a large branch and adding a small branch creates a smaller tree. However, if both offspring created by crossover are included in the population the average program size does not change. One offspring is smaller, the other is larger and the average is unchanged. Even if only one offspring is included, the average size of trees within the population should not change, as the included offspring is equally likely to be larger or smaller than its parent. Thus, the only way the average program size can increase during a GP run is if larger offspring are preferentially chosen during selection.

This is less true of subtree mutation as, depending on how the mutation is implemented, it may preferentially generate larger (or smaller) trees. However, a ‘fair’ subtree mutation will generate new programs whose average size is the same as the population’s average size. Again steady growth will only occur if larger offspring are preferentially chosen during selection, which will only happen if they tend to have a greater fitness than smaller offspring.

Thus, in examining the causes of code growth we must ask the question: what is it about crossover that on average leads to the larger offspring having a better fitness than the smaller offspring?

2.2. Types of code

In order to understand the theories regarding code growth it is necessary to examine the types of code being produced by GP. From its inception it was realized that GP has a tendency to create programs with large sections of code that do not significantly effect the program’s behavior or performance [7]. A few examples of

such code include:

if (FALSE){ X } else { Y } (1)

+ ($0 * X$) (2)

+ ($X - X$) (3)

+ ($1/X$) where $X \gg 1$ (4)

In each of these cases the code marked with an X does not significantly contribute to the program's behavior or performance (for case 3 it is the difference of the two sections that does not contribute). Historically the generic term 'intron' has been applied to code that doesn't have an effect and the term 'exon' to code that does have an effect. (Some authors have termed example 4 an intron, although it does have a small effect, whereas other authors consider it to be an exon.) Although there are problems with these terms (see for example [22]) they will be used here for simplicity.

Although none of these sections of code have a significant effect on performance, their other properties do vary. In the first two examples, X *cannot* change the program's behavior, regardless of how X is changed. In the third example, changing either of the sections of code labeled X could change the performance of the program. In the last example, the code labeled X does influence the output, but probably to an insignificant degree. The value of X would have to be significantly changed to change the output significantly.

Several researchers have attempted to categorize these types of code. Soule and Foster categorized code as inoperative, code that does not currently influence the output but could if changed (example 3 above), versus inviable code, code that can not influence the output even if changed (examples 1 and 2) [22]. (This assumes that none of the instructions have side effects.) Nordin and Banzhaf differentiated between code that has no effect for any input case, code that doesn't have an effect on any of the training cases and code that has an effect for some training cases [16]. Nordin et al. further refined this to five types of code depending on whether regions of code could be replaced by no-operation instructions without changing fitness [17]. Smith and Harries systematically studied the types of code defined by Nordin, Francone and Banzhaf and found that all five types, including code that has an actual, if insignificant effect on output (case 4 above), can produce code growth [19].

Luke has pointed out that many types of introns depend on the presence of an *invalidator*, a section of code that nullifies a second section of code thereby producing an intron [12]. Examples 1 and 2 above include invalidators (if(FALSE) and *0 respectively).

Theoretical and experimental work has typically focused on the more restrictive types of code, those that can not influence performance. In part this occurred because it is simpler to rigorously predict the effect (or noneffect) of code that never influences performance. Also, for some time it was assumed that the more restrictive types of code had the biggest impact on code growth. However, the studies by Smith and Harries and more recently by Luke have shown that code with a

very small effect on performance (example 4 above) can be as important for code growth as code with absolutely no effect.

In this paper we will ignore the different types of introns and only distinguish between introns and exons. Further, we will use the term introns only for code that could be removed without changing the program's output. However, we do this with the implicit understanding that some types of introns may be more important to the phenomenon of code growth than others and that some code we classify as exons may serve the same role as classical introns.

2.3. *The protective hypothesis*

In roughly equivalent theories Nordin and Banzhaf, McPhee and Miller, and Blickle and Thiele have argued that code growth occurs to protect programs against the destructive effects of crossover [1, 13, 16]. Several studies have shown that crossover is much more likely to decrease fitness than to increase fitness (destructive crossovers) [11, 18]. In addition, these studies show that a large proportion of crossover operations result in no change in fitness (neutral crossovers).

As noted above, evolved programs often contain large sections of introns that cannot have a significant effect on fitness even when changed by either crossover or mutation. The protective hypothesis proposes that there is an evolutionary benefit to increasing the proportion of introns. Crossover (or mutation) in these regions is likely to be neutral, which is evolutionarily preferable to destructive crossover. Thus, there is evolutionary pressure to decrease the ratio of exons to introns, which effectively 'hides' the exons from crossover.

Studies with non-destructive or hill-climbing crossover have supported this hypothesis [4, 20, 21]. In these versions of crossover, the fitness of the offspring is compared to the fitness of the parent. If the offspring is less fit (or, in some cases, as fit) as the parent, the offspring is discarded and the parent is kept instead. Thus, destructive crossovers are not allowed to occur. These forms of crossover severely curtailed code growth, suggesting that destructive crossovers do produce the evolutionary pressure that causes code growth. However, several researchers have questioned this research, pointing out that nondestructive crossover typically decreases the effective crossover rate, which could also decrease code growth [12, 19]. In this paper we use a form of fixed number mutation (described in Section 4.1) to show that some code growth does occur as a (presumably) protective response to destructive operators.

2.4. *The drift hypothesis*

A second theory of code growth is based on the structure of program search spaces. It has been experimentally observed that for many problems the number of programs of a given fitness that are larger than a given size is much greater than the number of programs with the given fitness that are smaller than a given size [9, 11].

For example, given a program of fitness X and size Y there are many more programs with fitness X that are larger than Y than that are smaller than Y .

In part, this property of the search space is caused by introns. Given a program of fitness X and size Y it is possible to create infinite larger variations of that program by adding introns. Because larger programs are more common it has been proposed that an unbiased search is more likely to find larger programs of a given fitness than it is to find smaller programs of that fitness, simply because there are more larger programs within the search space. The general idea is that "... any stochastic search technique, such as GP, will tend to find the most common programs in the search space of the current best fitness" [9] and for any given fitness, larger programs are more common.

2.5. *The removal bias hypothesis*

The third theory of code growth is the removal bias hypothesis proposed by Soule and Foster [22]. It assumes that introns are more densely concentrated near the leaves of program trees. This has been proven for strictly inviable code and appears to be true of introns in general, but has not been proven. Work by Igel and Chellapilla has shown that for some problems subtrees further from the root node have less influence on the program's output, which is additional empirical support for this assumption [5].

Given that inviable code is concentrated near the leaves, removing a small subtree is more likely to only effect inviable code, whereas removing a large subtree is more likely to effect viable code. In contrast, if a branch is added to the middle of a section of inviable code it will, by definition, have no effect, regardless of the size of the added branch.

Thus, the removal bias hypothesis states that there is a bias in favor of offspring created by removing a small branch and against offspring created by removing a large branch. No such bias applies to added branches. The net effect of such a bias would clearly be a general pattern of growth.

Luke has proposed a generalization of this hypothesis [12]. He argues, based on his own results and those of Igel and Chellapilla [5], that there is a general bias towards deeper crossover points (crossover points furthest from the root). This would favor larger trees, which have a larger distance from root to leaf, and it would favor crossover involving small subtrees. However, just favoring crossover involving small branches (or more precisely favoring the offspring created from small branch exchange) would not seem to produce growth. The bias towards small *removed* branches in particular seems to be necessary.

3. Experiments

In this section we describe the series of experiments that we used to test the three hypothesized causes of code growth and the test problems for these experiments.

The first set of experiments (Section 4.1) applies a fixed number of single node mutations to evolving programs: resulting in increased growth rates. Of the three theories of growth outlined above only the protective hypothesis can explain this increase.

The second set of experiments (Section 4.2) compares the size of the branches exchanged during crossover to the average change in fitness during crossover. The results show a strong correlation between removed branch size and the resulting fitness change (decrease), but a much weaker correlation for the added branch size. On average removing a larger branch results in a lower fitness, but adding a larger branch does not increase neutral crossovers even though it does explore regions of larger programs. Thus, these results support the removal bias hypothesis, but seem to refute the drift hypothesis.

The final set of experiments (Section 4.3) compares the size of the branches exchanged during crossover to the *number* of destructive, neutral, and constructive crossovers. Again the results show a strong correlation between the removed branch size and the number of destructive crossovers, but no such correlation for the added branch size. These results also support the removal bias hypothesis, but not the drift hypothesis.

3.1. Test problems

The experiments are performed on four different problems: symbolic regression, intertwined spirals [6], even-parity [3], and the Santa Fe trail [10].

3.1.1. Symbolic regression. The symbolic regression problem is to evolve a function $g(x)$ that matches a sample points taken from a target function $f(x)$. Our target function is a sine wave over the range $(-\pi, \pi)$. Evolving solutions are tested at 128 evenly distributed test points. The function set is $\{+, -, *, /\}$. Division is protected; if $|\text{divisor}| < 0.000001$ then the value 10,000 is returned. The terminal set is $\{x, \text{constants}\}$. New constants are generated randomly in the range $(-1, 1)$. For constants, mutation (including FNSN mutations described below) adds a random value in the range $(-0.5, 0.5)$ to the constant's current value.

Fitness was the square root of the sum of the squared errors. (Lower fitnesses are better.) A maximum fitness of 10,000 is imposed. Other details of the problem are presented in Table 1.

The values used for protected division, constants and maximum fitness were chosen after fairly limited testing. They were chosen so that division by zero will usually result in a program with very poor fitness and to produce reasonable, but not particularly outstanding results. It appears that changing the values for protected division and maximum fitness would not have a significant effect on the GP's performance.

Research by Daida et al. suggests that the choice of constants can significantly effect performance for this problem [2]. However, because the constants are chosen from the same range for every trial they should not favor any particular trial.

Table 1. Parameters for the GP

	Symbolic regression	Intertwined spirals	Ant	Even parity
Functions	+, -, /, *	+, -, *, /, iflte, sin, cos	prog2, prog3, iffoodahead	NOT, AND, OR, XOR
Terminals	constant, x	constant, x , y	Left, Right, Forward	N inputs
Fitness	root squared errors	% misclassified	% food missed	% miscalculated

3.1.2. Intertwined spirals. For the intertwined spirals problem two spirals coil around the origin of the x - y plane. Each spiral is defined by 97 points along its length. The goal is to find a function that classifies the points as belonging to spiral 1 or spiral 2 based on the x , y value of the point (see for example [6]). An evolved program returns a real value. Values less than zero are mapped to spiral 1, values larger than or equal to zero are mapped to spiral 2.

The function set is $\{+, -, *, /, \text{iflte}, \text{sine}, \text{cosine}\}$. Division is protected as in the symbolic regression problem. The function *iflte* (“if less than else”) takes four arguments, if the first is less than the second then the third argument is returned, otherwise the fourth argument is returned.

The terminal set is $\{x, y, \text{constant}\}$; x and y are the x , y values of the point to be classified. Constants are real values, randomly generated in the range $(-5.0, 5.0)$. For constants, mutation (including FNSN mutations, described in Section 4.1) adds a random value in the range $(-0.5, 0.5)$ to the constant’s current value.

Fitness is the percent of points misclassified. 0 is best, no points misclassified; 1.0 is worst, all points misclassified. On average a random classification will receive a fitness of 0.5. Other details of the problem are presented in Table 1.

3.1.3. Santa Fe Trail. The Santa Fe Trail, or artificial ant, problem requires an artificial ant to follow a trail of 89 ‘food’ objects laid out on a toroidal 32×32 grid (see, for example, [10]). There are spaces between some of the food objects, so the complete trail is 144 squares long. The trail is shown in Figure 1.

The function set for this problem is $\{\text{prog2}, \text{prog3}, \text{iffoodahead}\}$. The functions *prog2* and *prog3* have 2 and 3 arguments respectively, which are executed in sequence. The function *iffoodahead* has 2 arguments. It examines the square directly in front of the ant. If food is present the function executes its first argument, otherwise it executes its second argument.

The terminal set consists of the instructions $\{\text{left}, \text{right}, \text{forward}\}$. Each of these instructions takes one time unit to execute. The artificial ant can face in any of the four cardinal directions and can move one square forward.

Food is automatically ‘eaten’ and removed from the grid if the ant moves over it. The raw fitness is an integer between 0 and 89. The adjusted fitness is the percentage of food missed; 0 is best, no food missed; 1.0 is worst, all food missed.

The ant is allowed a total of 600 time units. An evolved program is executed repeatedly until all of the available time is used up. Thus, for example, a program consisting of the single instruction *forward* would move forward 600 times (looping through the same sections of the grid many times).

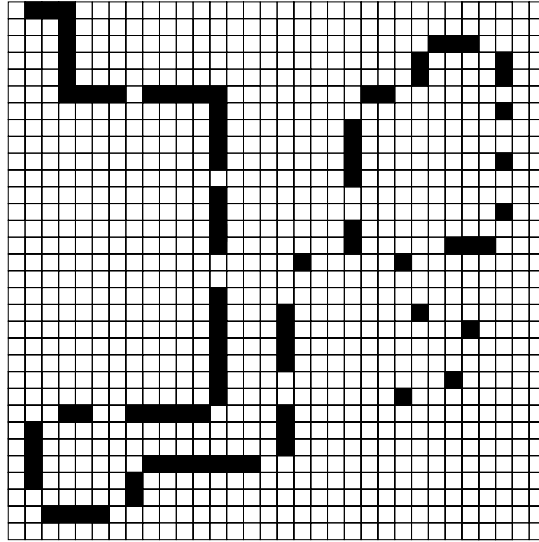


Figure 1. The Santa Fe Trail. The artificial ant begins in the upper left hand corner, facing right.

3.1.4. Even-parity. The even-parity problem requires evolving a Boolean function to determine the parity of N ($N = 5$ in these experiments) Boolean inputs. Thus, there are a total of 32 (2^5) different input cases.

The function set is {NOT, AND, OR, XOR}. This is a slightly unusual function set. It was chosen so that one of the test problems would be fairly easy to solve. The terminal set consists of the N input values. Fitness is the percentage of the input cases misclassified; 0 is best, no inputs misclassified, 1 is worst, all input cases misclassified.

3.2. The genetic program

The same basic GP was used for all of the test problems. The GP is generational and is run for 75 generations. The population size is 800. All of the results are the average of 50 trials. The ramped half-and-half technique is used to generate the initial populations. The initially generated programs are evenly distributed between depths of 3, 4, 5 and 6. For each of these maximum depths half of the programs are generated as full trees and the other half are random trees.

The 90/10 rule is used in selecting crossover points; in picking a crossover point there is a 90% chance that it will be an internal node and only a 10% chance that it will be a leaf node. Mutation randomly changes a single node to another node with the same arity (i.e., it is not subtree mutation). The mutations rate is 0.001 for all of the problems except symbolic regression, which uses a mutation rate of 0.01. The higher value is used to help evolution tune the constants in that problem. Constants subject to mutation are effected as explained under the individual problems.

Tournaments of 3 individuals are used for selection. Elitism consists of putting two copies of the best individual into the next generation's population. Other parameters

for the GP are listed in Table 1. No size or depth limit is placed on the programs and no parsimony pressure is used.

3.3. *The computations*

In order to generate sufficiently large statistical samples for the experiments in this paper, a PC based parallel computer in the Computer Science Department was used. This cluster computer uses the principles of commodity computing to build substantial computing power for considerably less money than traditional supercomputers.¹ Computers using this approach are often referred to as Beowulf computers [24, 25]. The University of Idaho's Beowulf cluster was programmed using C/C++ and MPI (<http://www.cs.uidaho.edu/thecollective>). Each trial was run on a single node in the cluster. Thus, all 50 trials could be run in parallel, which allowed experiments that would normally take almost a month to run to be completed in half a day.

4. Results

We begin by presenting the basic results with the four test problems. Figure 2 shows the best and average fitnesses, averaged across all 50 trials, for the four test

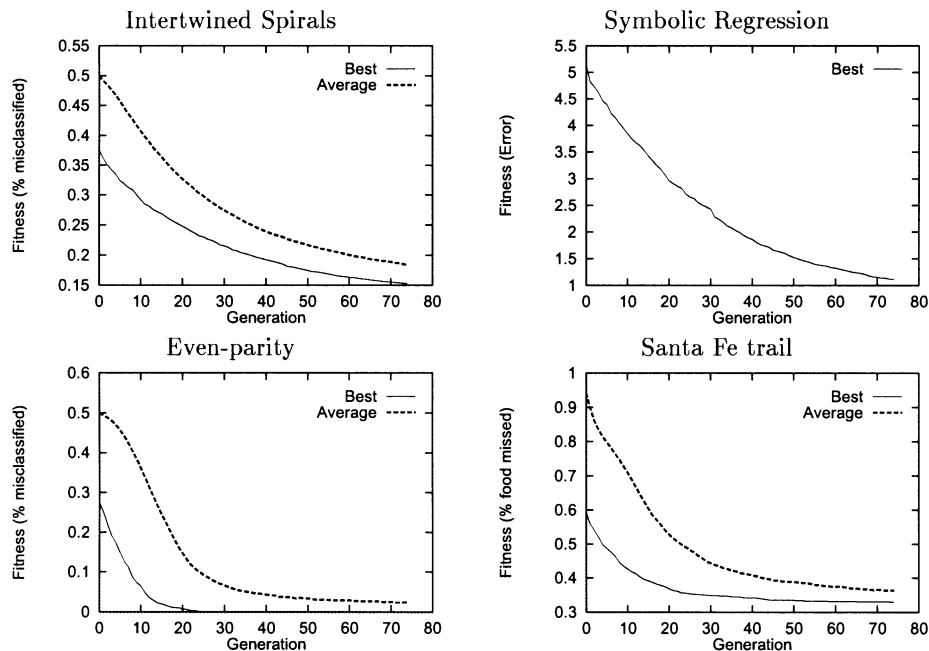


Figure 2. Average and best fitness for the four test problems. Only the best fitness is shown for the symbolic regression problem because the average fitnesses are quite large, due to a few outlying individuals.

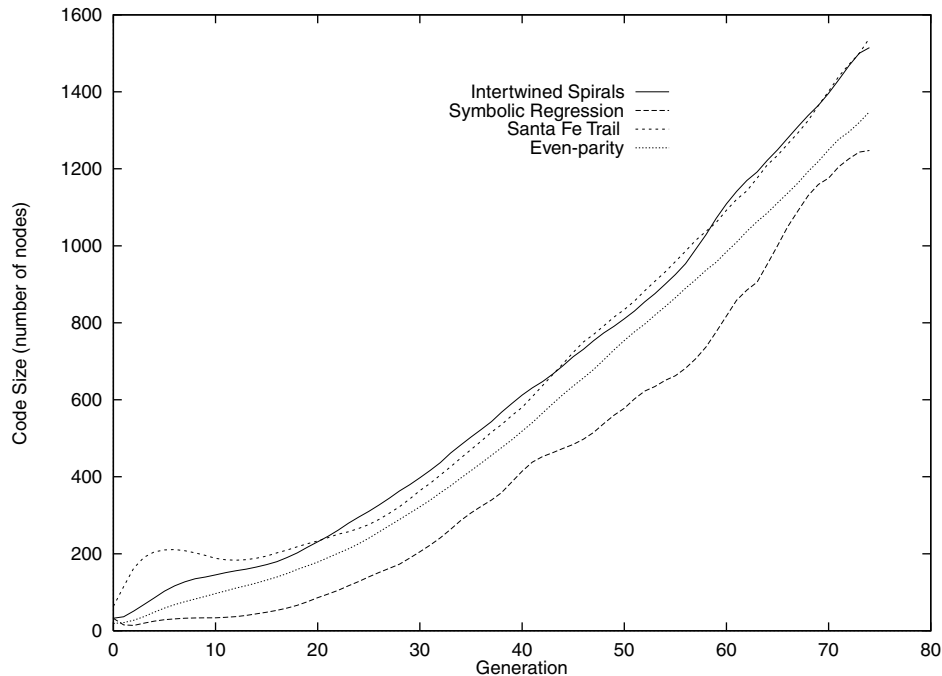


Figure 3. Average code size for intertwined spirals, symbolic regression, Santa Fe trail, and even-parity problems. All four test problems show considerable code growth.

functions. The average fitness is not shown for the symbolic regression problem. For that problem very large errors are possible for functions that are very different from the target function. A few individuals with these large errors skews the average fitness and makes it effectively meaningless.

The GP's performance is respectable for all of the test problems. However, because the GP is unoptimized, performance is not particularly outstanding.

Figure 3 shows the average code size for each of the four test problems. Clearly code growth is occurring. In particular the growth rates are rapid, very similar for different problems, and are not related to increasing fitness. These are the hallmarks of code growth. The similarity in growth rates is a clear indication that code growth is a phenomenon of GP (or of evolutionary techniques in general) and not caused by a particular problem or problem representation.

Although the growth rates are generally similar, there are some minor differences. Initially the programs solving the Santa Fe Trail grow unusually quickly. This is probably because the larger programs randomly cover more territory and so find more of the food and are preferentially selected. Once the programs begin to more systematically search for food the 600 step limit may help favor smaller programs, resulting in the temporary decrease in growth rate.

The programs solving the symbolic regression problem grow slightly slower than for the other problems. The larger initial programs are basically larger collections of random functions. They may be more likely to return larger or more varied

values, which would produce larger errors and lower fitnesses. Thus, there may be a slight initial bias towards smaller programs. Regardless of the cause of the minor variations in growth rates, all four of these problems produce fairly typical GP code growth.

4.1. *The protective hypothesis and fixed number mutations*

The protective hypothesis assumes that code growth is an evolutionary response to destructive changes to the evolving individuals. Decreasing the ratio of exons to introns helps protect individuals from destructive events. Thus, an increase in the number of destructive changes should result in greater evolutionary pressure to decrease this ratio and therefore in more rapid growth.

To test this we use Fixed Number, Single Node mutations (FNSN mutations). In an FNSN mutation a fixed number of nodes in the program are randomly changed into different nodes with the same arity (or if a constant is chosen for mutation the constant's value is randomly modified as described previously). It is essential that a fixed number of nodes be mutated, rather than applying a probability of mutation to each node, as is normally done.

If each node has the same, fixed probability of mutation (the standard case) then decreasing the ratio of exons to introns *will not* protect the exons, as their probability of mutation remains unchanged. In contrast, if exactly m nodes are mutated, regardless of the individual's total size, then decreasing the ratio of exons to introns *will* decrease the probability that an exon will be mutated. Thus, the destructive hypothesis predicts that FNSN mutations will increase the evolutionary pressure for growth.

Figure 4 shows the growth rates for the four test problems with varying numbers of FNSN mutations. We chose different numbers of mutations with different problems for two reasons. First, we wanted to make it clear that the exact number of FNSN mutations didn't matter, any increase in the number of mutations has an effect. Second, the size of the effect does vary slightly with the problem. Thus, to show a significant change in our limited number of generations more mutations were necessary with some of the test problems. This is discussed further below.

In all cases increasing the number of FNSN mutations increases the growth rate. For each of the test problems the average program sizes become significantly different between any two trials for the later generations (Student's 2-tailed t -test $P < 0.001$).

The generation at which significance first occurs depends on the test problem and which two trials are being compared. For example, the difference in size for the intertwined spirals problem with 4 FNSN mutations versus 8 FNSN mutations becomes statistically significant at generation 14, and remains significant for all later generations. Whereas, the difference in size for the symbolic regression problem with 3 FNSN mutations versus 6 FNSN mutations becomes significant at generation 39 and remains significant for all later generations. All pairs are significant by at most generation 39 and remain significant for all generations thereafter. Thus, over sufficient generations, FNSN mutations do produce increased growth.

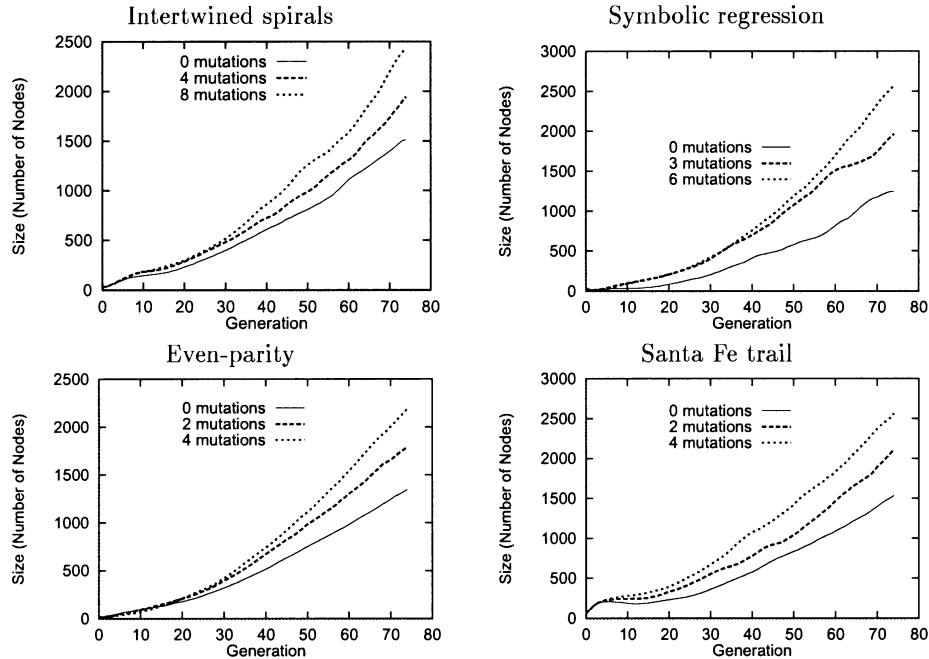


Figure 4. Change in growth rate as a function of Fixed Number, Single Node (FNSN) mutations. In all cases the increase in the number of single node mutations produces increased growth. Because only a fixed number of single nodes are mutated only the protective hypothesis explains the change in growth rates.

The occurrence of significance reflects the progressive effect of FNSN mutations over time. Significance takes longer to occur in some test problems because those problems create more natural variance in size, are less sensitive to the effects of FNSN mutations, or may even gain more benefit from mutations. For example, the symbolic regression problem tends to create much more variance in fitness, thus the negative effects of the FNSN mutations may be partially hidden. Additionally, FNSN mutations may provide some benefit in the symbolic regression problem by acting as a tuning mechanism for the constants. These influences could explain why statistically significant size differences take longer to appear for the symbolic regression problem and why somewhat more mutations were necessary to show significance in our limited number of trials.

Overall the FNSN mutations are clearly producing increased growth. (Note that the FNSN mutations themselves never change the programs' sizes, only a single node is changed to a different node. Size variations are still created by crossover.) Because FNSN mutations do not introduce additional crossovers there is neither removal nor addition of branches. So, removal bias cannot be the cause of the increased growth. Similarly, because FNSN mutations do not change the program size, the drift hypothesis does not apply. Thus, these results demonstrate increased growth that cannot be attributed to either drift or removal bias. However, by acting as an additional destructive operator FNSN mutations could increase the evolutionary pressure favoring protective code. Thus, this growth can be explained by the

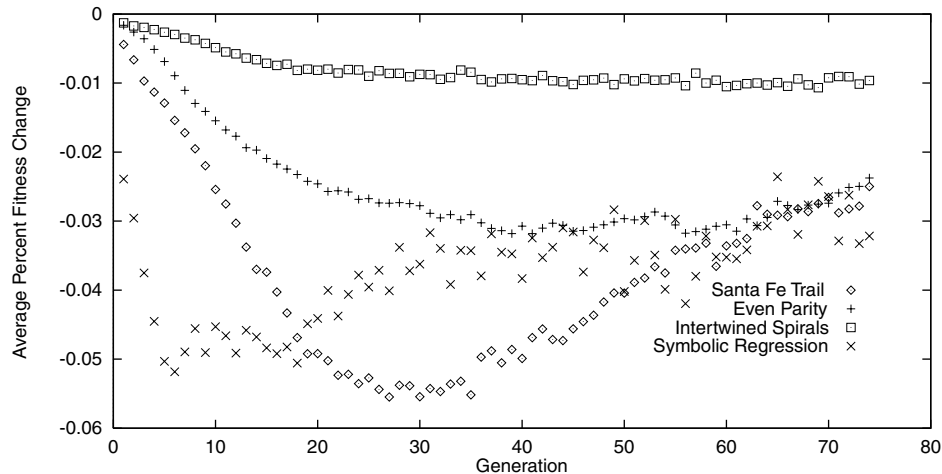


Figure 5. Average effect of a single FNSN mutation on fitness for each of the four test problems, by generation. The average fitness change for the symbolic regression problem is scaled down by 500 to fit on the graph. (In this graph a fitness change of 1% for the symbolic regression problem actually represents a 500% change.)

protective hypothesis, assuming that FNSN mutations are more likely to be destructive than constructive.

Although it is reasonable to assume that FNSN mutations are more destructive than constructive, we chose to empirically confirm the assumption. (If they were generally more constructive no search would be required, one could simply keep applying FNSN mutations until the program was sufficiently improved. However, there could be special cases in which FNSN mutations were beneficial, such as in tuning constants.) Figure 5 shows the average percent change in fitness caused by one FNSN mutation, by generation, for each of the four test problems. The average change is negative; i.e., an FNSN mutation usually lowers fitness, as one would expect. Thus, the protective hypothesis is a reasonable explanation for the increased growth under FNSN mutations.

The pattern of destructiveness shown in Figure 5 also indirectly supports the protective hypothesis. Initially, FNSN mutations are not very destructive because the programs of the first few generations are random or nearly random. Thus, a random change is almost as likely to improve a program as to worsen it. Further, negative changes tend to be relatively small because the fitness of the programs in the early generations are already quite poor. As the programs evolve and improve FNSN mutations become more likely to worsen fitness and to cause larger decreases in fitness. Thus, the mutations become more destructive. However, as introns build up the probability of a mutation having an effect decreases and the destructiveness of the FNSN mutations begins to decline. This pattern is seen most clearly for the Santa Fe ant problem.

Finally, Figure 6 shows the evolution of fitness under FNSN mutations for reference purposes. As before the average fitnesses are shown for all of the problems except symbolic regression. In most cases the FNSN mutations lower the average

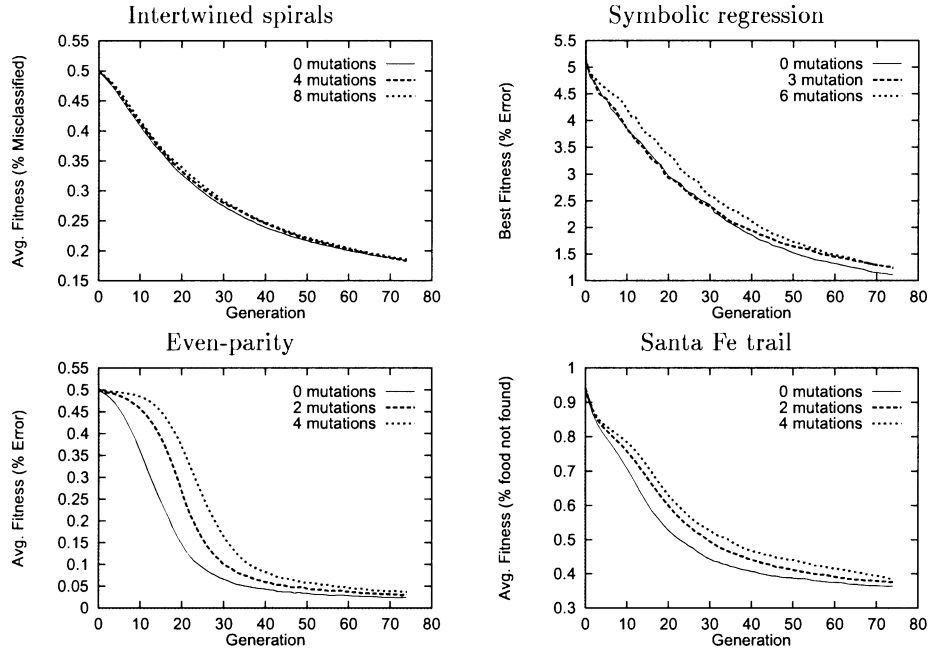


Figure 6. Fitness with varying numbers of FNSN mutations. Average fitness is shown for all problems except symbolic regression, for which average best fitness is shown.

fitness. The exception being the intertwined spirals problem, which seems to be unaffected by the mutations.

4.2. Branch size versus fitness change

Our next concern is to determine whether either drift or removal bias takes place. As described previously removal bias predicts that there is a negative correlation between the size of a branch removed during crossover and the change in fitness produced by crossover. Removal bias does not predict any correlation between the size of the added branch and the change in fitness.

The drift hypothesis predicts that crossovers leading to a smaller program are more likely to be destructive and that crossovers leading to a larger program are more likely to be neutral. This differs from the removal bias hypothesis in that it does not depend on whether the size change comes from removing a small branch or adding a large one. Thus, the drift hypothesis predicts a negative correlation between the removed branch size and the fitness change (removing a larger branch tends to produce smaller offspring). This is the same as the prediction of removal bias. However, the drift hypothesis also predicts a positive correlation between the size of the added branch and the change in fitness (adding a larger branch produces a larger offspring).

Thus, these theories differ in the following way: removal bias predicts a very small or nonexistent correlation between added branch size and fitness change (or

number of neutral crossovers); drift predicts a positive correlation between added branch size and fitness change.

4.2.1. Intertwined spirals. For this experiment we begin with an in-depth look at the results for the intertwined spirals. The data for the other test problems are shown later (Section 4.2.4) and are fundamentally similar. Figure 7 shows the ‘average percent change’ in fitness for removed and added branches during crossover in the intertwined spirals problem. By average percent change in fitness we mean the change in fitness between parent and offspring as a percentage of the parent’s fitness. Size of the added and removed branches is also given as a percentage of

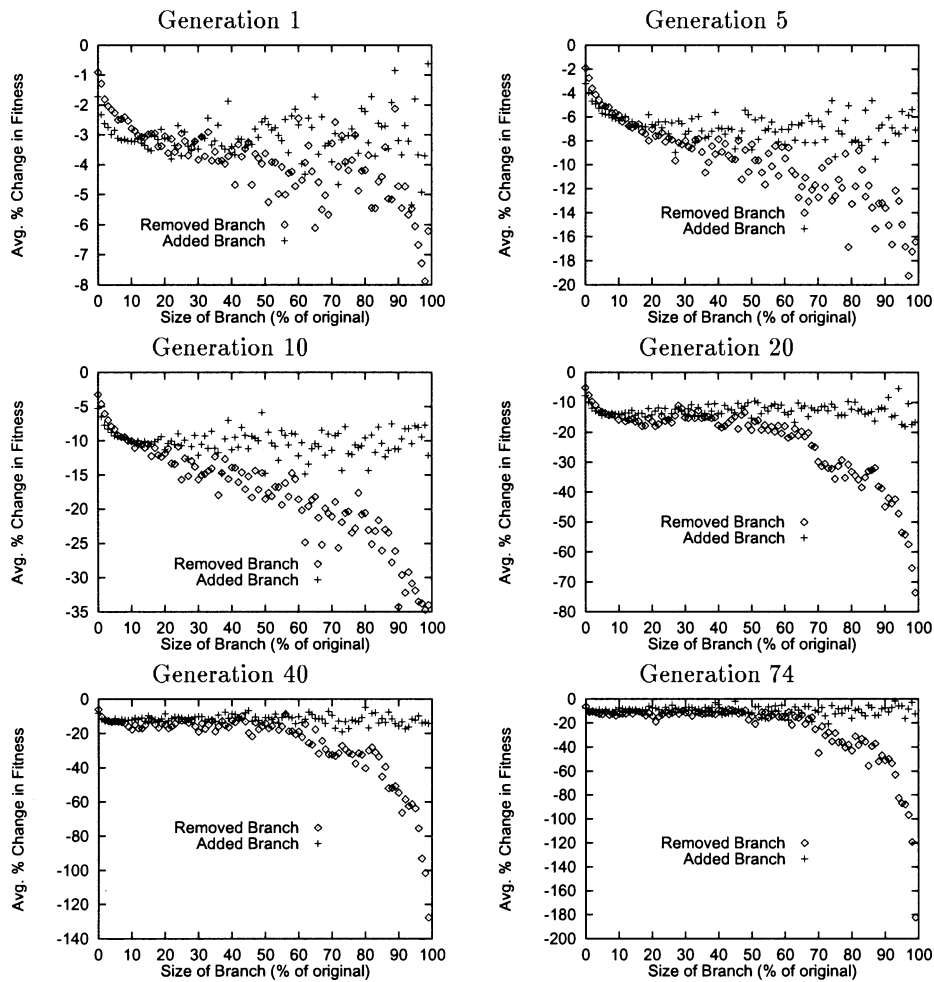


Figure 7. Intertwined Spirals. Average percent change in fitness for removed and added branches of different sizes during crossover. Size is given as a percent of the original program. Except in generation 1 there is a strong relationship between the size of the removed branch and the change (decrease) in fitness, as predicted by the removal bias hypothesis. The relationship for added branches is much weaker.

the original parent's size. For example, a point in the graph at 10% on the x-axis represents the average fitness change from parent to offspring, as a percentage of the parent's fitness, when branches that are 10–11% of the parent program are added (or removed). For example, in generation 1 adding a branch which is 10–11% of the size of the program it is being added to causes an average decrease in fitness of 3%.

To collect these data 5000 extra crossovers were performed at each of six generations (1, 5, 10, 20, 40, and 74). The offspring were measured and then discarded, so these extra crossovers did not change the evolutionary process or the later generations. The experiment was repeated fifty times for a total of 250,000 crossovers in each of the test generations.

(The change in fitness may be over 100% because fitness for the intertwined spirals problems is measured as percent miscategorized. Going from 10% miscategorized to 25% miscategorized shows up as a change of –150%.)

The average effect of crossover, regardless of the sizes of the branches involved, is always negative. This is to be expected. If there were a particular size of branch that on average lead to improved fitness it would be possible to simply add (or remove) branches of the given size until the fitness improved to the desired value.

4.2.2. Removed branches. Figure 7 shows that overall there is a strong inverse relationship between the size of the removed branch during crossover and the resulting change in fitness caused by that crossover. Removing a larger branch does, on average, produce larger decreases in fitness. This agrees with the prediction of both the removal bias hypothesis and the drift hypothesis.

In general the relationship between removed branch size and average change in fitness can be separated into three distinct regions. For the smallest removed branches (roughly 0 to 20 percent of the parent, depending on the generation) there is an inverse relationship between size and fitness change: increasing the size of the removed branch decreases the offspring's fitness.

For mid-sized branches (roughly 20 to 40 percent of the parent, depending on the generation) the behavior is very generation dependent. In many generations (1, 5, 10) there is a clear negative correlation. In other generations (40, 74) there appears to be no correlation. While in generation 20 there appears to be a slight *positive* relationship; removing a larger branch causes a smaller change in fitness.

Finally, for the largest branches (40+ percent of the parent) removing a larger branch again causes a larger decrease in fitness. This relationship becomes much stronger in later generations. Note that in these figures the y-scales have been increased in later generations to graph the decrease in fitness. Figure 8 shows the removed branch data for all 6 of the tested generations in one plot. Clearly, the negative effect of removing a large branch becomes much more important in later generations. One likely reason for the increasing effect in later generations is simply that the programs have higher average fitnesses. Thus, a destructive crossover can produce a larger decline in fitness.

4.2.3. Added branches. Figure 7 also shows the relationship between added branch size and change in fitness. The behavior for added branches is similar to the behavior for removed branches when the branches are small or medium sized. For the

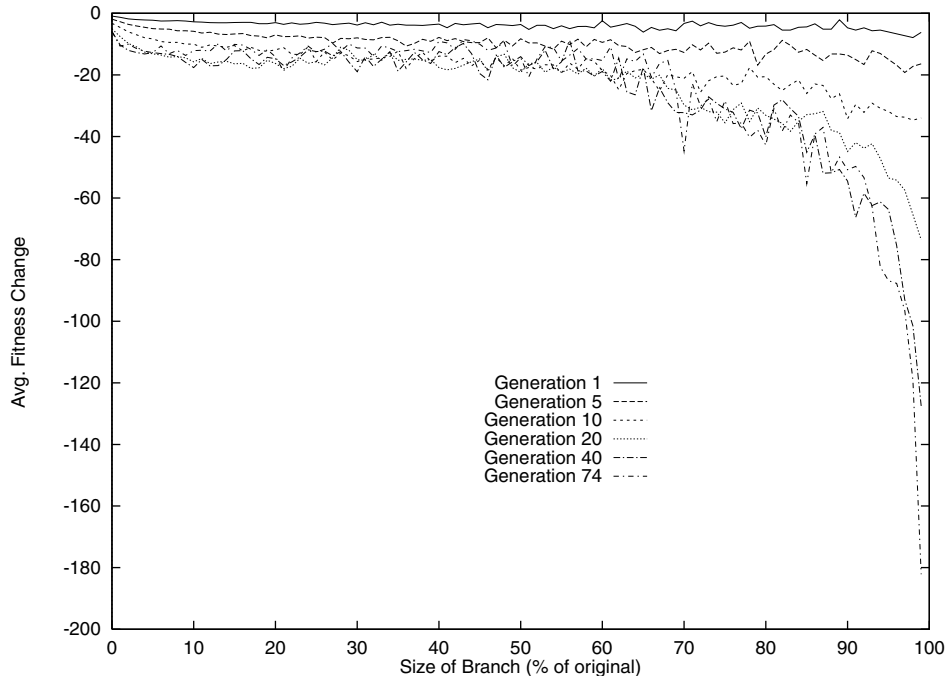


Figure 8. Intertwined Spirals. Average percent change in fitness for removed branches of different sizes during crossover. Size is given as a percent of the original program.

smallest branches (roughly 0 to 20 percent of the parent, depending on the generation) adding a larger branch results in a larger average decrease in fitness. The drift hypothesis predicts the opposite relationship; adding a larger branch creates a larger program, which should have a greater probability of the having the same fitness as its parent.

For medium sized added branches (roughly 20 to 40 percent of the parent, depending on the generation) size does not seem to be an important factor. In some generations a slight positive correlation exists and in other generations the correlation is negative.

For the largest added branches (40+ percent of the parent) the size of the added branch is also not strongly related to the fitness change. These results appear to contradict the predictions of the drift hypothesis. If a larger offspring is more likely to have the same fitness as its parent, then adding a larger branch should result in fewer fitness decreases, producing a positive correlation between the added branch's size and the resulting fitness change. These results show that adding a larger branch, creating a larger program, is not more likely to result in a neutral crossover.

A final issue must be considered in examining these data. Despite the use of the 90/10 rule in these experiments (only 10% of the time was a single leaf node chosen for crossover) the number of chosen branches (both removed and added) of a given size decreases exponentially with increasing size. Figure 9 shows the distribution of sizes of the branches chosen for crossover in the intertwined spirals problem for

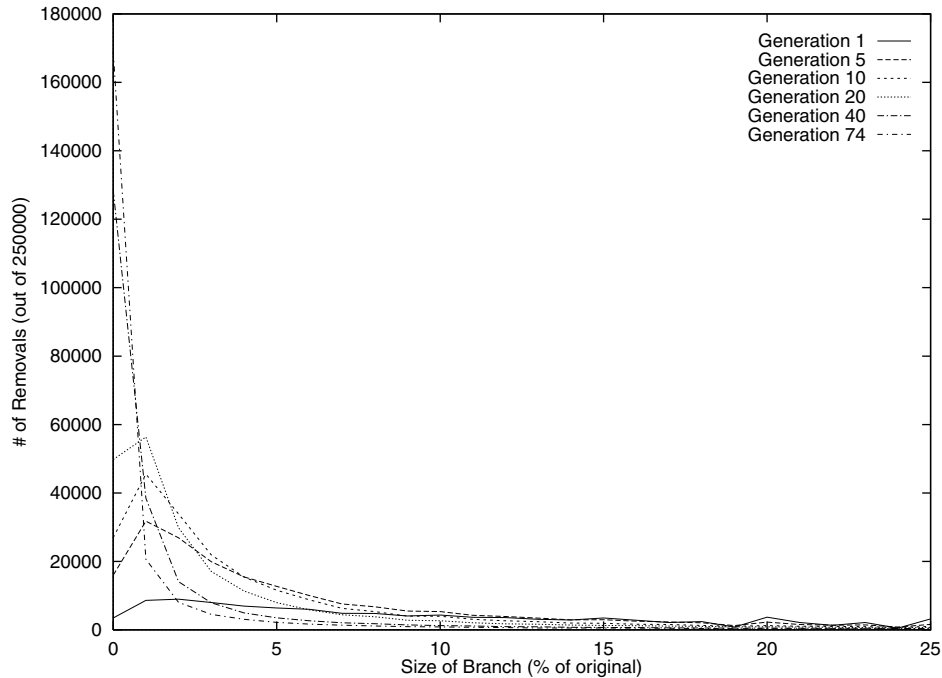


Figure 9. Intertwined Spirals. Number of branches chosen for crossover as a function of size. Smaller branches are chosen exponentially more often. The use of the 90/10 rule does decrease the number of very small branches (0–1% of the parent) chosen in the earlier generations, but only in the early generations when the parents are still relatively small.

each of the test generations. Clearly removing (or adding) a relatively small branch is exponentially more likely than removing (or adding) a relatively large branch. This is simply a function of the tree structures. Most nodes in a tree are the root of a relatively small subtree and thus, most crossovers involve relatively small branches.

This relationship is more pronounced in later generations, as the overall program sizes increase. The number of branches in the 0–1% range is artificially lowered in the earlier generations because of the 90/10 rule. But by the later generations even a branch consisting of several nodes is less than 1% of the total tree. The net effect is that the results for small branches are much more significant to the evolutionary process, because exponentially more crossover operations involve small branches.

4.2.4. Results for the other problems. The data for branch size versus fitness change for the even-parity, symbolic regression and Santa Fe Trial problems are shown in Figures 10, 11 and 12, respectively. For all three problems the basic pattern of the results is the same as for the intertwined spirals problem. The size of the removed branch is inversely correlated to the fitness change, with the strongest correlations occurring for very small or very large branches and during the later generations. The size of the added branch is *negatively* correlated to the fitness change for small sizes only.

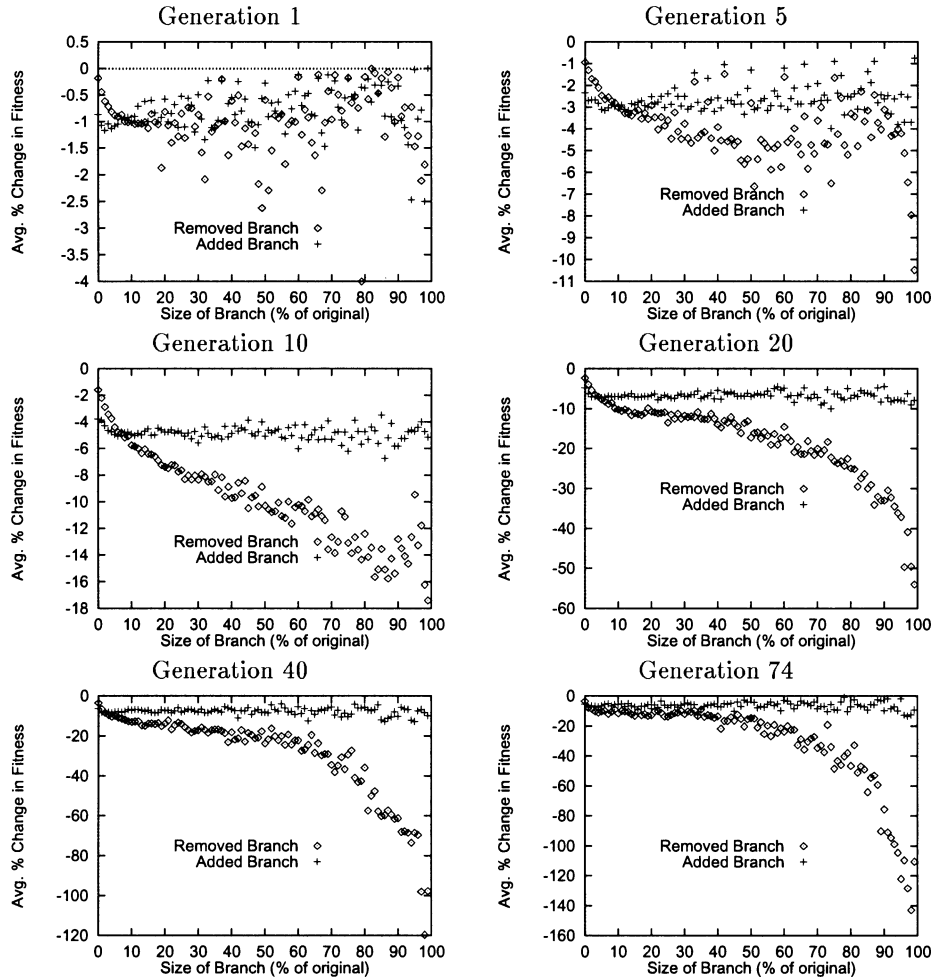


Figure 10. Even parity. Average percent change in fitness for removed and added branches of different sizes during crossover. Size is given as a percent of the original program. The results are very similar to those seen with the intertwined spirals problem.

The results are less clear for the symbolic regression problem, because the variations in fitness for that problem are so large. Consider an evolved solution that is fairly close to the target function. It will have a low error and a good (low) fitness. Now consider that crossover may add a branch that adds 100 to the result, increasing the error (and decreasing the fitness) by several orders of magnitude. Hence, the fitness change due to crossover is very large, with a large variation. Similar fitness changes do not occur with the other test problems.

These data clearly support the predictions of removal bias while refuting the drift hypothesis. In general, the probability that an offspring that is larger than its parent will have the same fitness as its parent only increases if the size change occurred because a relatively small branch was removed from the parent during crossover.

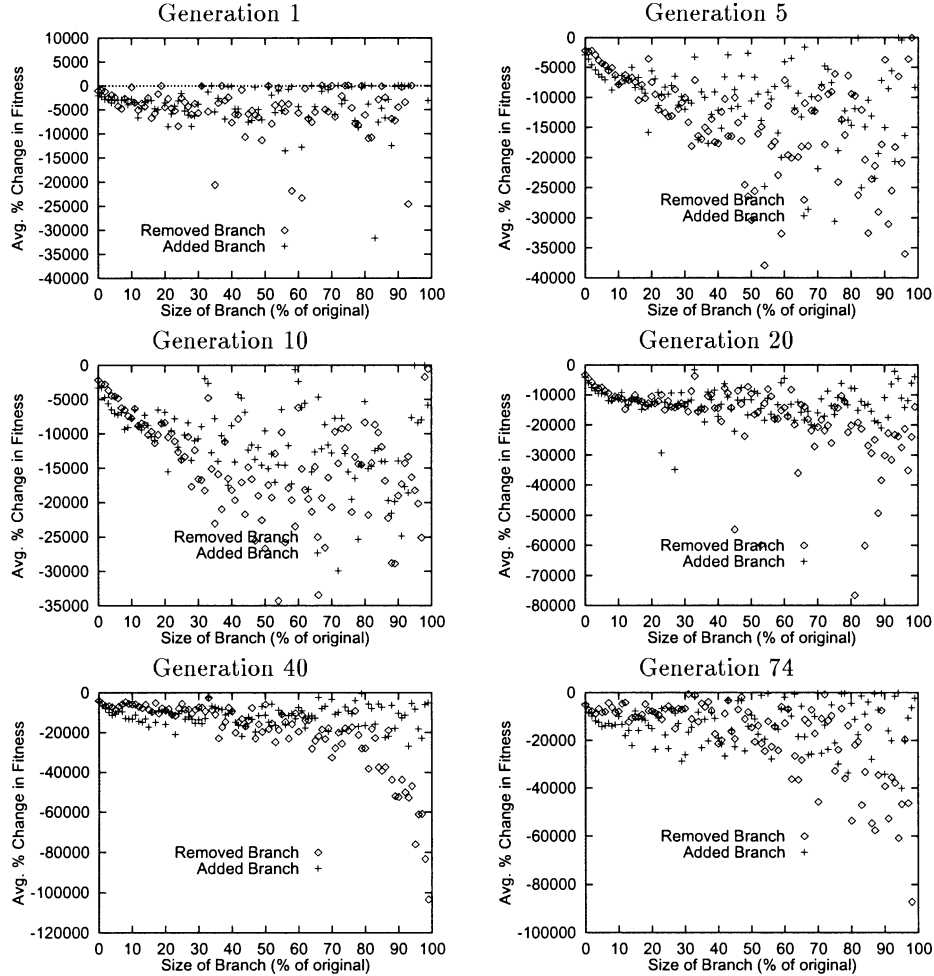


Figure 11. Symbolic regression. Average percent change in fitness for removed and added branches of different sizes during crossover. Size is given as a percent of the original program. For this problem a single crossover can create extremely large changes in the evolving function creating correspondingly large changes in error and hence in the fitness. This accounts for the very large observed changes in fitness.

4.3. Branch size and number of neutral crossovers

The previous experiments examined the average *fitness change* as a function of branch size. We also want to consider the average *number* of destructive, neutral and constructive crossovers as a function of branch size.

Although the average change in fitness during crossover as a function of branch size supports the removal bias hypothesis, it is possible that the number of neutral crossovers as a function of branch size supports the drift hypothesis. We expect that the number of destructive, neutral, and constructive crossovers is closely correlated to the fitness changes caused by crossover. However, it is possible that the number of

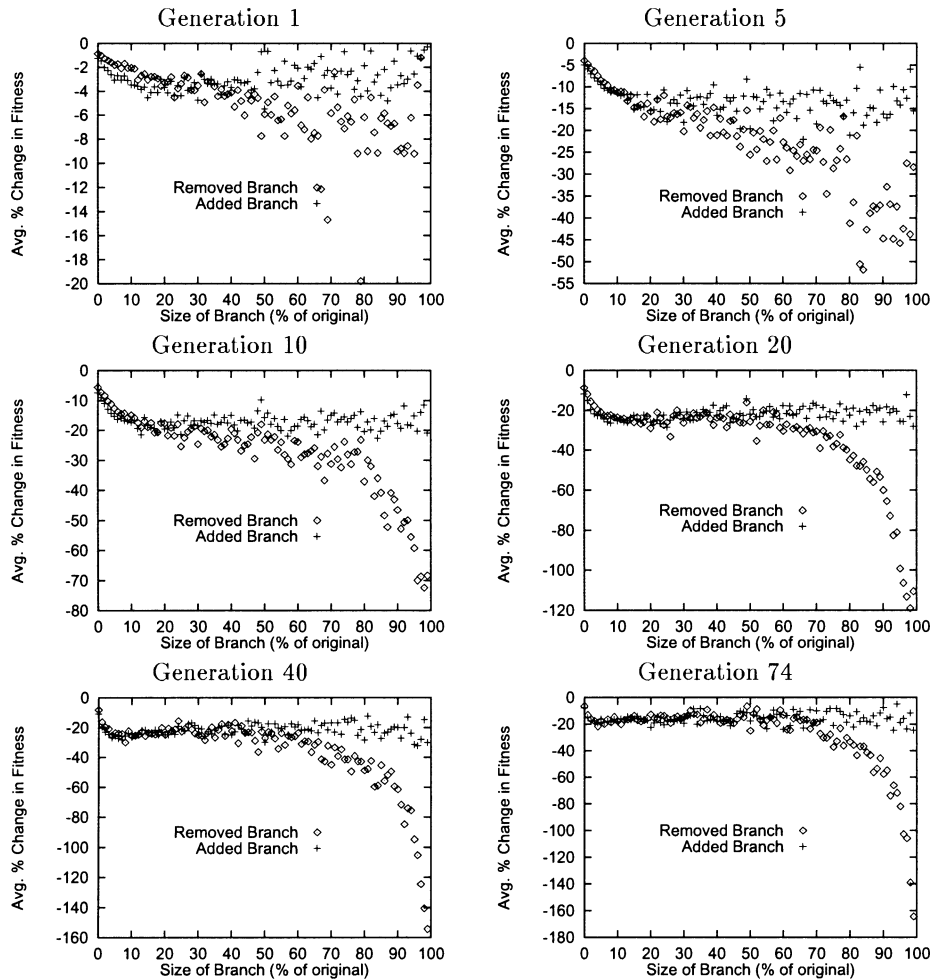


Figure 12. Santa Fe Trail. Average percent change in fitness for removed and added branches of different sizes during crossover. Size is given as a percent of the original program.

destructive crossovers could decrease when larger branches are added even though the average destructiveness remains unchanged. Thus, it is worth briefly examining the number of such crossovers.

Figure 13 shows the proportion of constructive, neutral, and destructive crossovers as a function of the removed branch sizes for the intertwined spirals problem for generation 10. Figure 14 shows the same data, but for the added branches. Data are only shown for generation 10 to save space.

As in the previous experiments with crossover the data was collected by performing 5000 extra crossovers in the tested generation in each of the 50 trials. The offspring of these crossovers were measured and discarded to avoid contaminating later generations. Thus, these data are the average of 250,000 crossovers.

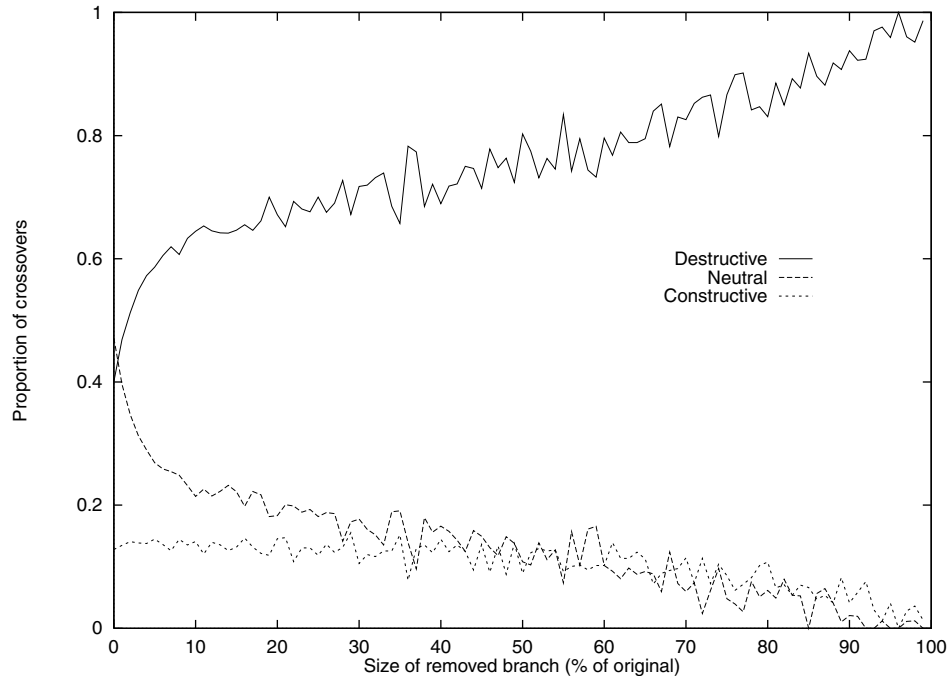


Figure 13. Intertwined spirals. Number of destructive, neutral, and constructive crossovers in generation 10 as a function of removed branch size. The number of destructive and neutral crossovers depend on the size of the removed branch.

Figure 13 shows that removing a larger branch increases the probability that a crossover will be destructive rather than neutral. These results are quite similar to the results for the average fitness change as a function of branch size.

Figure 14 shows that in general the number of destructive, neutral, and constructive crossovers are all independent of the size of the added branch. For the smallest branches (0–10% of the parent) adding a larger branch and creating a larger offspring is more likely to be destructive than adding a smaller branch. This matches the results for the change in fitness due to branch size and is the opposite of the drift hypothesis prediction.

The results (not shown) for the other tested generations (1, 5, 20, 40, and 74) produced similar results. The number of destructive crossovers increases as the size of the removed branch increases and is independent of the size of the added branch. The number of neutral crossovers decreases as the size of the removed branch increases and is independent of the size of the added branch. The number of constructive crossovers is generally very small and shows little relationship with the size of either the added or removed branch.

Figure 15 shows the data for the other three test problems. The general pattern is the same, the size of the removed branch is related to the number of destructive and neutral crossovers, but the size of the added branch is not related, except for the smallest branches. An interesting exception occurs for the symbolic regression problem. In this case for the very largest branches (85–100+% of the parent) adding a

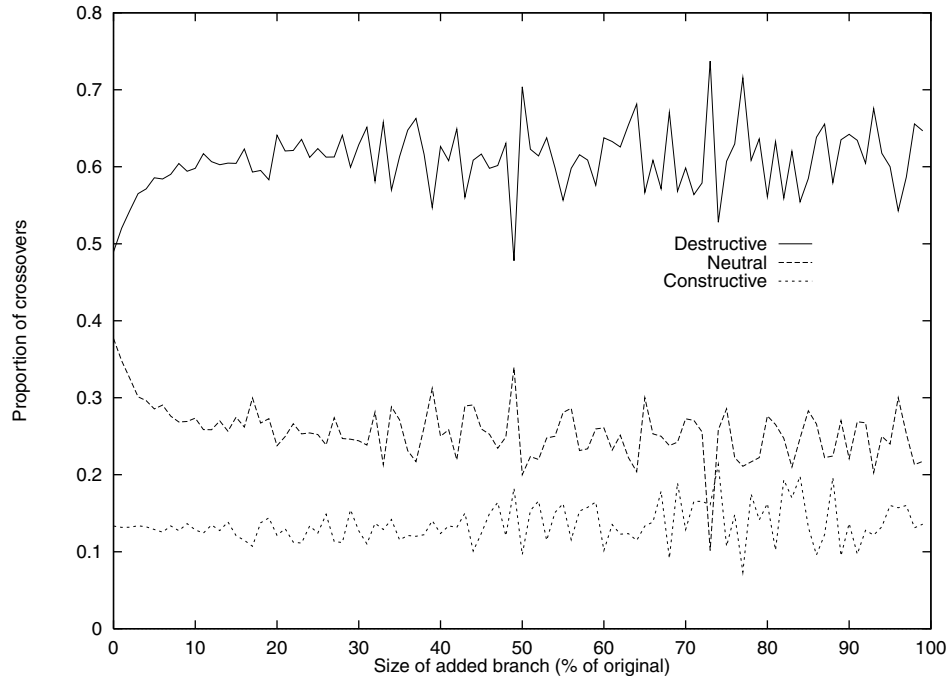


Figure 14. Intertwined spirals. Number of destructive, neutral, and constructive crossovers in generation 10 as a function of added branch size. The number of destructive and neutral crossovers do not depend on the size of the added branch.

larger branch does decrease the number of destructive crossovers and increases the number of neutral crossovers. It is unclear why this occurs, although it does match the drift hypothesis predictions.

5. Conclusions

These data clearly show that growth increases in response to destructive operations, in this case FNSN mutations. Thus, they show that some growth is a protective response to destructive operations. Further, this growth is very sensitive to changes in the frequency of destructive operations. Randomly changing a few additional nodes per program per generation results in significant increases in growth.

Next we found a strong correlation between removed branch size during crossover and both the probability that the resulting offspring would be worse than its parent and the amount of the fitness decrease. Removing larger branches during crossover creates more destructive crossovers. In contrast, adding a larger branch has almost no effect on the destructiveness.

These results confirm the removal bias hypothesis that some growth occurs simply because removing large branches during crossover is relatively destructive whereas adding large branches is not. These results also refute the drift hypothesis

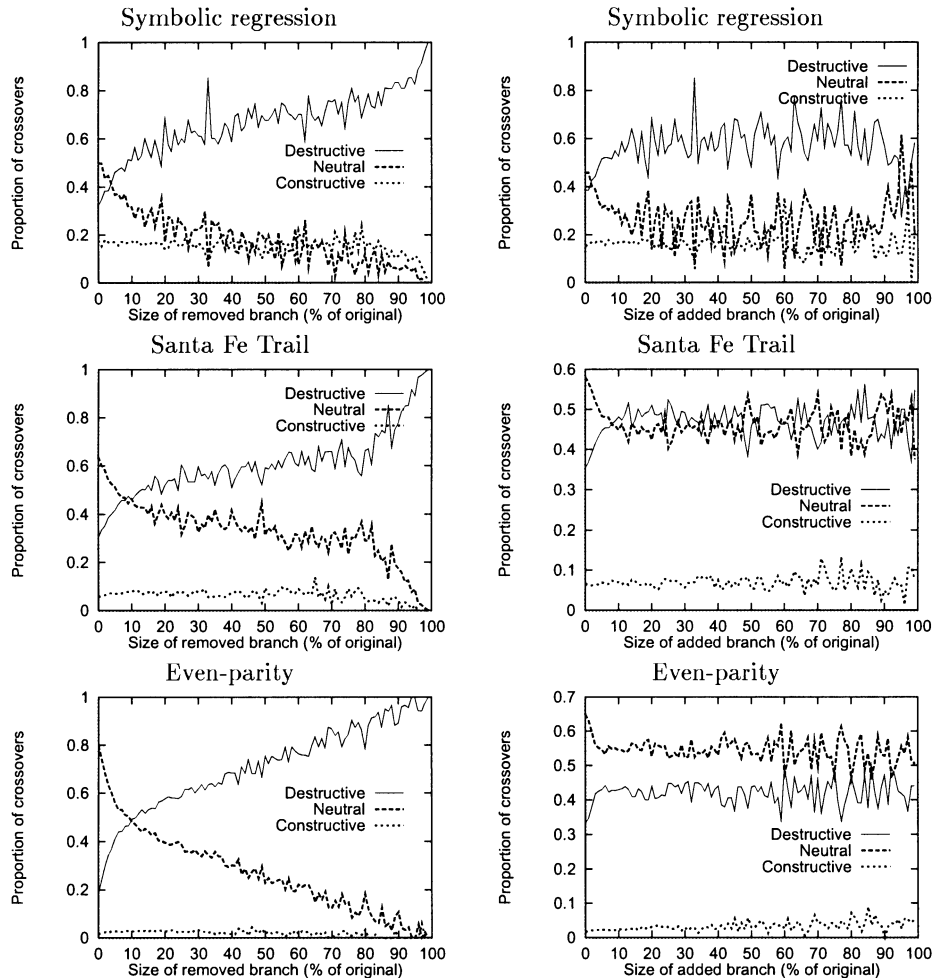


Figure 15. Number of destructive, neutral, and constructive crossovers in generation 10 as a function of removed (left column) and added (right column) branch sizes, for the remaining three test problems. As with the intertwined spirals problem, the number of destructive and neutral crossovers does depend on the size of the removed branch, but not on the size of the added branch except for the very smallest sizes.

as adding large branches, which creates larger offspring, is not correlated with neutral crossovers.

Importantly, the correlation with removed branch size is very strong and grows increasingly significant as evolution precedes. The strength of this correlation suggests that removal bias may be a much more important factor in causing code growth than was previously believed, particularly in later generations.

There are two likely failures of the drift hypothesis. One, it assumes that the search space is sampled without bias. However, crossover produces new individuals based on parent individuals that have been produced by many generations of selection. Thus, the offspring are clearly not random samples of the total program space.

Two, even if the search space is sampled without bias it may not be sufficient to say that there are more equally fit individuals of larger size. Rather for drift to occur it may be necessary that the percentage of equally fit individuals be larger for larger programs so that it is more that likely a random search will find them.

Experiments have been performed with ‘fairer’ versions of crossover. Notably Langdon used a version called size fair crossover, in which the removed and added branches must be of approximately the same size [9]. Growth was significantly reduced using this variant of crossover. This result supports both the drift hypothesis and the removal bias hypothesis. Slower growth could arise because the restriction in crossover points made it more difficult to reach larger, equally-fit programs. This is a result predicted by drift. Alternatively, slower growth could occur because the difference in size between removed and added branches is reduced, limiting the removal bias.

Finally, these results suggest that there are serious problems with the standard tree-based structures used in GP. First, we note that very few crossovers effect the nodes close to the root of a tree. This is a purely statistical effect; in choosing a random node for crossover it is relatively unlikely that the few nodes near a tree’s root will be selected, even when using the 90/10 rule.

Second, our data show that the few crossovers that do effect those nodes are almost invariably destructive and cause relatively large decreases in fitness. Thus, the resulting offspring are almost guaranteed to be discarded during selection.

Increasing the number of generations does not significantly improve the chances of changing nodes near the root because the probability of changing the root node (or nodes near the root) decreases as the program size increases. Additionally, in later generations, the larger branch removals (which are necessary to change nodes near the root) become progressively more destructive on average. The net effect is that the nodes near the root are effectively fixed in the initial, random population. Evolution is forced to make do with whatever nodes were placed near the root randomly.

This conclusion supports the experimental results of McPhee and Hooper [14]. They traced the genealogies of evolving populations and found that there was almost never a change in the root nodes of a population.

These results may also help to explain why diversity is so difficult to maintain in GP. If the nodes near the root are effectively fixed then only the offspring of programs that had ‘good’ nodes there to begin with are likely to survive. It seems likely that this is a small proportion of the initial population.

There are probably many ways of modifying crossover and GP that would reduce the problem of fixed root nodes. For example, the 90/10 rule could be replaced with a rule that selects more uniformly from all depths. This would increase the number of crossover operations that modify nodes near the root. However, these crossovers would still generally be destructive, so the total number of destructive crossovers would most likely increase and the nodes near the root would not change.

In general, it seems that the current tree structure simply places too much importance on nodes near the root. A possible solution to this problem is to allow offspring several ‘free’ generations during which they were subjected to reduced

selection pressure. This might allow offspring programs that had branches extending up to the root node changed by crossover a chance to 'recover' (through additional crossovers with successful individuals) from the negative affects of such large changes. Of course, these are only preliminary suggestions. In general, an increased understanding of the exact relationship between crossover, growth and offspring fitness should lead to better approaches, particularly program structures and crossover operations that are better suited to the evolutionary process.

Acknowledgment

This research is supported by NSF EPS 80935.

Note

1. The total cost of the machine is about \$44,000. All of the memory for the machine was generously donated by Mircon Technology. This cut the cost of the machine by 30%.

References

1. T. Blickle and L. Thiele, "Genetic programming and redundancy," in *Genetic Algorithms Within the Framework of Evolutionary Computation*, J. Hopf (ed.), Max-Planck-Institut für Informatik: Saarbrücken, Germany, 1994, pp. 33–38.
2. J. M. Daida, R. B. Bertram, S. A. Stanhpe, J. C. Khoo, S. A. Chaudhary, and O. A. Chaudhri, "What makes a problem gp-hard? Analysis of tunably difficult problems in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 2, no. 2, pp. 165–191, 2001.
3. C. Gathercole and P. Ross, "Tackling the boolean even n parity problem with genetic programming and limited-error fitness," in *Genetic Programming 1997: Proc. Second Annual Conf.*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. R. Riolo (eds.), Morgan Kaufmann: San Francisco, CA, 1997, pp. 119–127.
4. D. C. Hooper, N. S. Flann, and S. R. Fuller, "Recombinative hill-climbing: A stronger search method for genetic programming," in *Genetic Programming 1997: Proc. Second Annual Conf.*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. R. Riolo (eds.), Morgan Kaufmann: San Francisco, CA, 1997, pp. 174–179.
5. C. Igel and K. Chellapilla, "Investigating the influence of depth and degree of genotypic change on fitness in genetic programming," in *Proc. Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), Morgan Kaufmann, 1999, pp. 1061–1068.
6. J. Koza, "A genetic approach to the truck backer upper problem and the intertwined spiral problem," in *Proc. IJCNN Int. Joint Conf. Neural Networks*, IEEE Press, 1992, pp. 310–318.
7. J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, The MIT Press: Cambridge, MA, 1992.
8. W. B. Langdon, "Fitness causes bloat: Simulated annealing, hill climbing and populations," Technical Report CSRP-97-22, The University of Birmingham, Birmingham, UK, 1997.
9. W. B. Langdon, "Size fair and homologous tree genetic programming crossovers," in *Proc. Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), Morgan Kaufmann: 1999, pp. 1092–1097.
10. W. B. Langdon and R. Poli, "Why ants are hard," in *Genetic Programming 1998: Proc. Third Annual Conf.*, 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 (eds.), 1998, pp. 193–201.

11. W. B. Langdon, T. Soule, R. Poli, and J. A. Foster, "The evolution of size and shape," in *Advances in Genetic Programming III*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline (eds.), The MIT Press: Cambridge, MA, 1999, pp. 163–190.
12. S. Luke, "Code growth is not caused by introns," in *Late Breaking Papers, Proc. Genetic and Evolutionary Computation Conf.*, 2000, pp. 228–235.
13. N. F. McPhee and J. D. Miller, "Accurate replication in genetic programming," in *Proc. Sixth Int. Conf. Genetic Algorithms*, L. J. Eshelman (ed.), Morgan Kaufmann: San Francisco, CA, 1995, pp. 303–309.
14. N. F. McPhee and N. J. Hopper, "Analysis of genetic diversity through population history," in *Proc. Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), Morgan Kaufmann, 1999, pp. 1112–1120.
15. P. Nordin, *Evolutionary Program Induction of Binary Machine Code and Its Application*, Krehl Verlag: Muenster, 1997.
16. P. Nordin and W. Banzhaf, "Complexity compression and evolution," in *Proc. Sixth Int. Conf. Genetic Algorithms*, L. J. Eshelman (ed.), Morgan Kaufmann: San Francisco, CA, 1995, pp. 310–317.
17. P. Nordin, W. Banzhaf, and F. D. Francone, "Introns in nature and in simulated structure evolution," in *Proceedings Bio-Computing and Emergent Computation*, D. Lundh, B. Olsson, and A. Narayanan (eds.), Springer, 1997, pp. 22–35.
18. P. Nordin, F. Francone, and W. Banzhaf, "Explicitly defined introns and destructive crossover in genetic programming," in *Advances in Genetic Programming II*, P. Angeline and Jr. K. E. Kinneer (eds.), The MIT Press: Cambridge, MA, 1996, pp. 111–134.
19. P. Smith and K. Harries, "Code growth, explicitly defined introns, and alternative selection schemes," *Evolutionary Computation*, vol. 6, no. 4, pp. 339–360, 1998.
20. T. Soule, *Code Growth in Genetic Programming*, Ph.D. thesis, University of Idaho, University of Idaho, 1998.
21. T. Soule and J. A. Foster, "Code size and depth flows in genetic programming," in *Genetic Programming 1997: Proc. Second Annual Conf.*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. R. Riolo (eds.), Morgan Kaufmann: San Francisco, CA, 1997, pp. 313–320.
22. T. Soule and J. A. Foster, "Removal bias: a new cause of code growth in tree based evolutionary programming," in *ICEC 98: IEEE International Conf. on Evolutionary Computation*, IEEE Press, 1998, pp. 781–786.
23. T. Soule, J. A. Foster, and J. Dickinson, "Code growth in genetic programming," in *Genetic Programming 1996: Proc. First Annual Conf.*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. R. Riolo (eds.), MIT Press: Cambridge, MA, 1996, pp. 215–223.
24. D. H. M. Spector, *Building Linux Clusters: Scaling Linux for Scientific and Enterprise Applications*, O'Reilly: Sebastopol, CA, 2000.
25. T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese, *How to Build a Beowulf: A Guide to the Implementation and Application of PC Cluster*, MIT Press: Cambridge, MA, 1999.