Advances in Genetic Programming 3, L. Spector, W. B. Langdon, U-M O'Rielly, P. J. Angeline (eds.)

8 The Evolution of Size and Shape

W. B. Langdon, T. Soule, R. Poli and J. A. Foster

The phenomenon of growth in program size in genetic programming populations has been widely reported. In a variety of experiments and static analysis we test the standard protective code explanation and find it to be incomplete. We suggest bloat is primarily due to distribution of fitness in the space of possible programs and because of this, in the absence of bias, it is in general inherent in any search technique using a variable length representation.

We investigate the fitness landscape produced by program tree-based genetic operators when acting upon points in the search space. We show bloat in common operators is primarily due to the exponential shape of the underlying search space. Nevertheless we demonstrate new operators with considerably reduced bloating characteristics. We also describe mechanisms whereby bloat arises and relate these back to the shape of the search space. Finally we show our simple random walk entropy increasing model is able to predict the shape of evolved programs.

8.1 Introduction

The rapid growth of programs produced by genetic programming (GP) is a well documented phenomenon [Koza, 1992; Blickle and Thiele, 1994; Nordin and Banzhaf, 1995; McPhee and Miller, 1995; Soule et al., 1996; Greeff and Aldrich, 1997; Soule, 1998]. This growth, often referred to as "code bloat", need not be correlated with increases in the fitness of the evolving programs and consists primarily of code which does not change the semantics of the evolving program. The rate of growth appears to vary depending upon the particular genetic programming paradigm being used, but exponential rates of growth have been documented [Nordin and Banzhaf, 1995].

Code bloat occurs in both tree based and linear genomes [Nordin, 1997; Nordin and Banzhaf, 1995; Nordin et al., 1997] and with automatically defined functions [Langdon, 1995]. Recent research suggests that code bloat will occur in most fitness based search techniques which allow variable length solutions [Langdon, 1998b; Langdon and Poli, 1997b].

Clearly, an exponential rate of growth precludes the extended use of GP or any other search technique which suffers from code bloat. Even linear growth seriously hampers an extended search. This alone is reason to be concerned about code growth. However, the rapid increase in solution size can also decrease the likelihood of finding improved solutions. Since no clear benefits offset these detrimental effects, practical solutions to the code bloat phenomenon are necessary to make GP and related search techniques feasible for real-world applications.

Many techniques exist for limiting code bloat [Koza, 1992; Iba et al., 1994; Zhang and Mühlenbein, 1995; Blickle, 1996; Rosca, 1997; Nordin et al., 1996; Soule and Foster, 1997; Hooper et al., 1997]. However, without definitive knowledge regarding the causes of code bloat, any solution is likely to have serious shortcomings or undesirable side effects. A robust solution to code bloat should follow from, not precede, knowledge of what actually causes the phenomenon in the first place.

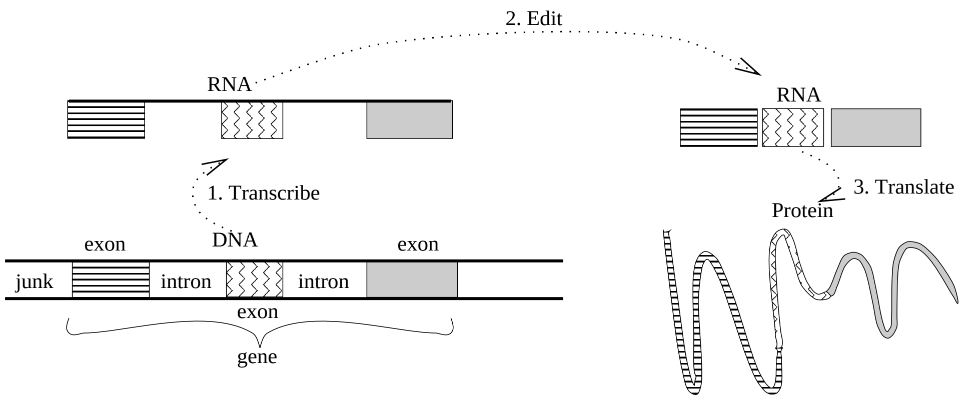

Figure 8.1
How cells express DNA: 1. Transcribe DNA to RNA; 2. Remove introns; 3. Translate to proteins (not to scale).

We present the latest research into the causes of code bloat. This research clearly demonstrates that there are several distinct causes of code bloat. Each of these causes appears to operate in both GP and other, related, search techniques. Thus, any general strategy for countering code bloat should address all of these causes.

This research promises to do more than merely lead to more feasible controls for code bloat. It also sheds some much needed light on the process of evolution, or, at least, artificial evolution. Code bloat research helps identify more of the many, often conflicting, forces which influence an evolving population.

In the next section we describe the historical background to bloat including previous work on it. In Section 8.3 we suggest program spaces may be to a large extent independent of program length, in that over a wide range of program lengths the distribution of fitness values does not change dramatically. Section 8.4 reconciles this with bloat and indeed suggests bloat, in the absence of bias, is general. Sections 8.5 to 8.7 experimentally test these theories. The experiments are followed by a discussion of their significance and of bloat more generally in Section 8.8 and conclusions in Section 8.9. Finally we give some suggestions for future work.

8.2 Background

In living organisms, molecules transcribe the DNA in each gene into RNA, edit out portions of the RNA, and then translate the remaining RNA into a protein. *Exons* are gene segments which produce protein building blocks, and *introns* are the non-expressed segments. See Figure 8.1.

Many natural genomes contain both *genic* DNA, which encodes the genes, and *non-genic* (sometimes called “junk”) DNA. Many genomes are predominantly non-genic. For example, human DNA is approximately 95% non-genic. There is no correlation between genome size and the ratio of genic to non-genic DNA, the complexity of the organism, or the ancestry of the organism [Cavalier-Smith, 1985; Li and Graur, 1991].

There are many distinctions between introns and non-genic DNA. Introns provide vital functions for the organism and perhaps even for evolution itself [Mattick, 1994; Li and Graur, 1991]. Non-genic DNA apparently contributes little to an organism’s fitness, though it may serve some structural role or provide an environment for genetic parasites [Li and Graur, 1991]. The origins of both non-genic DNA and introns are unclear. However, organisms with selective pressure toward streamlined genomes, such as bacteria and viruses, have little non-genic DNA and few, if any, introns. In some cases, such as ΦX-174 (a virus which lives in *E. coli*) [Kornberg, 1982] or some genes coding for human mitochondrial RNA [Anderson et al., 1981], a single sequence of DNA codes simultaneously for more than one protein—a kind of natural data compression.

In GP fitness neutral sections of code are commonly referred to as introns, whereas sections of code which effect fitness are called exons. There are several problems with the intron/exon distinction as it is used in GP. First, these terms are often used without precise definitions. The formal definitions which have been published are not always compatible. Perhaps more importantly the terms intron and exon have quite different meanings in the biological community. The lack of a transcription process in typical GP makes it impossible to reasonably associate biological introns and exons with types of GP code. Thus, the terms intron and exon can make communication with biologists difficult. Finally, in many cases dividing GP code into more than two categories is necessary to understand the evolution of bloat. For these reasons we chose to introduce two entirely new terms: operative and viable.

Definition 1 *A node* n *in a program’s syntax tree is* operative *if the removal of the subtree rooted at that node (and replacing it with a null operation) will change the program’s output. 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 subtree such that replacing the subtree 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.*

Although we chose to define these terms for tree based genomes, modifying these definitions to apply to other genomes is not difficult. For example if a linear genome was used the definition would need to refer to linear subsections of code rather than subtrees.

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

As an example consider two code fragments:

$$\texttt{X+}\underline{\texttt{(1-(4-3))}}\texttt{ and}$$
$$\texttt{Y+}\underline{\texttt{(0*Z)}}$$

where X,Y, and Z are additional sections of code. In each fragment the underlined section contributes nothing and could be removed without affecting the output or fitness of the individual containing this code. Thus, both underlined sections are inoperative. In addition, assuming there are no side-effects, fragment `Z` is inviable since no replacement for fragment `Z` will affect performance. While the fragment `(1-(4-3))` is viable, because changes to this fragment could change performance. Quite often inviable code is code which does not get executed, such as code following an `if(false)` statement.

In roughly equivalent theories, [Nordin and Banzhaf, 1995], [McPhee and Miller, 1995] and [Blickle and Thiele, 1994] have argued that code growth occurs to protect programs against the destructive effects of crossover and similar operators. Clearly any operator which only affects inviable code cannot be destructive (or beneficial) and any operator affecting only inoperative code is less likely to be destructive because the code which is being changed doesn't contribute to the fitness. Thus individuals which contain large amounts of inviable or inoperative code and relatively small amounts of operative code are less likely to have damaged offspring, and therefore enjoy an evolutionary advantage. Inviable and inoperative code have a protective role against the effects of crossover and similar operators.

[McPhee and Miller, 1995] argue more generally that evolution favors programs which replicate with semantic accuracy, i.e. that there is a Replication Accuracy Force acting on the population. This is a general force which should respond to replication inaccuracies caused by crossover, mutation or any other primarily destructive operator. This force also favors maximizing total code while minimizing viable code.

Although code bloat apparently serves a protective function, this does not mean that it is necessarily beneficial in producing improved solutions. These hypotheses suggest that code bloat performs a purely conservative role. Code bloat preserves existing solutions, but makes it difficult to modify, and thereby improve upon, those solutions. Thus code bloat is a serious problem for sustained learning.

It has also been argued that code bloat could act as a storehouse for subroutines which may be used later [Angeline, 1994]. However, there is no clear experimental evidence that this generally occurs.

Several techniques for limiting code bloat have been proposed. One of the first and most widely used is to set a fixed limit on the size or depth of the programs [Koza, 1992]. Programs exceeding the limit are discarded and a parent is kept instead. This technique is effective at limiting code bloat but has certain drawbacks. Some prior domain knowledge is necessary to choose a reasonable limit and code bloat occurs fairly rapidly until the average program approaches the limit. Recent research also suggests that a limit can interfere with searches once the average program size approaches the size limit [Gathercole and Ross, 1996; Langdon and Poli, 1997a].

Parsimony pressure attempts to use the evolutionary process to evolve both suitable and small solutions. A term is added to the fitness function which penalizes larger programs, thereby encouraging the evolution of smaller solutions. Commonly the penalty is a simple linear function of the solution size, but other more, and less, subtle approaches have been used. Of these, variable penalty functions, which respond to the fitness and size of individuals within the population appear to be the most robust [Iba et al., 1994; Zhang and Mühlenbein, 1995; Blickle, 1996]. Other studies have shown a degradation in performance when parsimony pressure is used [Koza, 1992; Nordin and Banzhaf, 1995]. Recent research suggests that the effect of parsimony pressure depends on the magnitude of the parsimony function relative to the size-fitness distribution of the population [Soule, 1998]. Populations with a stronger correlation between high fitness and large size are less likely to be negatively affected by parsimony pressure. When the correlation is low, smaller programs are heavily favored and the population tends towards minimal individuals, which seriously hampers further exploration.

Another approach to reducing code bloat has been to modify the basic operators. Notable operator modification approaches include varying the rate of crossover (and mutation) to counter the evolutionary pressure towards protective code [Rosca, 1997], varying the selection probability of crossover points by using explicitly defined introns [Nordin et al., 1996], and negating destructive crossover events (a form of hill climbing) [Soule and Foster, 1997; O'Reilly and Oppacher, 1995; Hooper et al., 1997]. Each of these approaches has the goal of reducing the evolutionary importance of inviable and inoperative code. Although each has shown some promise none of them appear to be universally successful.

8.3 Program Search Spaces

The problem of automatically producing programs can be thought of as the problem of searching for and finding a suitable program in the space of all possible programs. The first requirement is that we choose a search space which does contain suitable programs. In GP this means that the function and terminal sets are suffiently powerful to be able to express a solution. We must also ensure that limits on the size of programs don't exclude solutions. Given finite terminal and function sets and a bound on the size or depth of programs we have a finite number of possible programs, i.e. a finite search space. However even in simple GP problems, the size of the search spaces are huge, typically growing approximately exponentially with the size of the largest program allowed.

Like the number of different programs of a given size, the number of different tree shapes of a given size also grows approximately exponentially with size. For binary trees of length l (i.e. comprised of $l/2$ internal nodes and $(l+1)/2$ external nodes or leafs) the shortest or most compact tree has a depth of $\lceil \log_2 l + 1 \rceil$ and the tallest $(l+1)/2$. The most popular height lies between these extremes (for reasonable programs sizes it is near $l^{0.63}$, while the average height converges slowly to $2\sqrt{\pi(l-1)/2} + O(l^{1/4})$ as l increases [Flajolet and Oldyzko, 1982, page 200]) and almost all programs have a maximum height near this peak (see Figure 8.9).

It is often assumed that we know almost nothing about the distribution of solutions within these vast search spaces, that they are neither continuous nor differentiable and so classical search techniques will be incapable of solving our problems and so we have to use stochastic search techniques, such as genetic programming. However random sampling of a range of simple GP benchmark problems suggests a common features of program search spaces is that over a wide range of program lengths the distribution of fitness does not vary greatly with program length [Langdon and Poli, 1998d; Langdon and Poli, 1998a; Langdon and Poli, 1998e]. These results suggest in general longer programs are on average the same fitness as shorter ones. I.e. there is no intrinsic advantage in searching programs longer than some problem dependent threshold. Of course, in general, we will not know in advance where the threshold is. Also it may be that some search techniques perform better with longer programs, perhaps because together they encourage the formation of smoother more correlated or easier to search fitness landscapes [Poli and Langdon, 1998a]. However in practice searching at longer lengths is liable to be more expensive both in terms of memory and also time (since commonly the CPU time to perform each fitness evaluation rises in proportion to program size). Given this why should progressive search techniques which decide where to explore next based on knowledge gained so far, such as genetic programming, encounter bloat?

8.4 Bloat Inherent in Variable Length Representations

In general with variable length discrete representations there are multiple ways of representing a given behaviour. If the evaluation function is static and concerned only with the quality of each trial solution and not with its representation then all these representations have equal worth. If the search strategy were unbiased, each of these would be equally likely to be found. In general there are many more long ways to represent a specific behaviour than short representations of the same behaviour. For example in the sextic polynomial problem, Section 8.5, there are about 3,500 times as many high scoring programs of length $n + 2$ as there are with the same score and a length of n. Thus, assuming an unbiased search strategy, we would expect a predominance of long representations.

Practical search techniques are biased. There are two common forms of bias when using variable length representations. Firstly search techniques often commence with simple (i.e. short) representations, i.e. they have an in built bias in favour of short representations. Secondly they have a bias in favour of continuing the search from previously discovered high fitness representations and retaining them as points for future search. I.e. there is a bias in favour of representations that do at least as well as their initiating point(s).

On problems of interest, finding improved solutions is relatively easy initially but becomes increasingly more difficult. In these circumstances, especially with a discrete fitness function, there is little chance of finding a representation that does better than the representation(s) from which it was created. (Cf. "death of crossover" [Langdon, 1998a,

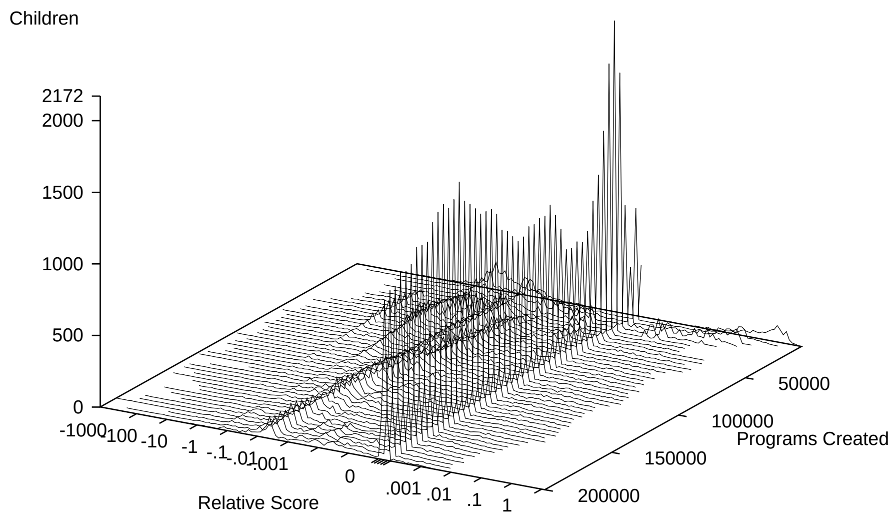

Figure 8.2
Fitness relative to first parent. First GP run of sextic polynomial problem. Peak at no change in fitness has zero width in the initial generation. In later generations the peak widens and by the end of the run 15% of children have a fitness different from their first parent but the difference is less than 10^{-5}.

page 206]). So the selection bias favours representations which have the same fitness as those from which they were created.

For example in our experiments with the artificial ant problem by the end of 50 runs crossover made no improvements at all in any of them [Langdon and Poli, 1997b, Figure 9]. Similarly in continuous problems most crossovers do not improve programs. Figure 8.2 shows while about 50% of crossovers in the sextic polynomial problem (see Section 8.5) did not change the measured fitness of the programs or changed it by less than 10^{-5}, in the last generation of the run only 3% of crossovers produced children fitter than their first parent.

In general the easiest way to create one representation from another and retain the same fitness is for the new representation to represent identical behaviour. Thus, in the absence of improved solutions, the search may become a random search for new representations of the best solution found so far. As we said above, there are many more long representations than short ones for the same solution, so such a random search (other things being equal) will find more long representations than short ones. In Section 8.7 we show another aspect of this random expansion towards the parts of the search space containing most programs; the search drifts towards the most popular program shapes.

Bloat can be likened to diffusion where there is a macroscopic change which appears to be directed but it is in fact the effect of many microscopic random fluctuations which cause the physical system to move from an initial highly unlikely (low entropy) state to a more likely one (high entropy). In the same way the initial GP population is usually constrained to be in one of a relatively small number of states (as the programs in it start relatively short). Over time the effect of the many random changes made by crossover, mutation and selection cause the population to evolve towards the part of the search space containing the most programs simply because this means there are many more states the population can be in. I.e. if we choose a state uniformly at random is likely to be one in which the population contains long programs as there are many more long programs than short ones. The law of exponential growth in number of programs gives a very strong bloat pressure which is difficult for random fluctuations produced by mutation and crossover to ignore. However in Section 8.5.3 we describe a mutation operator which does have much reduced bloating characteristics.

While most previous attempts to explain bloat during evolution have concentrated upon inoperative or inviable code in genetic programming the above explanation is more general in two important ways. Firstly it predicts code growth is general and is expected in all unbiased search techniques with variable length representations. In Section 8.5 we investigate bloat in continuous domain non-GP search. [Langdon, 1998b] showed bloat in a discrete problem under a range of non-GP search techniques. Secondly it is able to explain the evolution of program shapes as well as sizes. That is not to say the other approaches are wrong, only that we suggest they are less general.

Like physical entropy, this explanation only says the direction in which change will occur but nothing about the speed of the change. Price's Covariance and Selection Theorem [Price, 1970] from population genetics can be applied to GP populations [Langdon, 1998a; Langdon and Poli, 1997a]. In particular it can be applied to program size. Provided the genetic operators are random and unbiased, given the covariance between program's length and the number of children they have (which is given by their fitness and the selection technique), Price's theorem says what the expected mean change in length will be between this generation and the next. The increase is proportional to the covariance. So the greater the correlation between size and fitness the faster bloat will be. In practice most of the covariance, and hence most of the bloat, is due not to long children being better than their parents but due relatively short ones being worse than average. (See, for example, Section 8.5.4).

Essentially Price's theorem gives a quantitative measurement of the way genetic algorithms (GAs) search. If some aspect of the genetic material is positively correlated with fitness then, other things being equal, the next generation's population will on average contain more of it. If it is negative, then the GA will tend to reduce it in the next generation.

Table 8.1
GP Parameters for the Sextic Polynomial Problem

Objective:	Find a program that produces the given value of the sextic polynomial $x^6 - 2x^4 + x^2$ as its output when given the value of the one independent variable, x, as input
Terminal set:	x and 250 floating point constants chosen at random from 2001 numbers between -1.000 and +1.000
Functions set:	$+ - \times \%$ (protected division)
Fitness cases:	50 random values of x from the range -1 … 1
Fitness:	The mean, over the 50 fitness cases, of the absolute value of the difference between the value returned by the program and $x^6 - 2x^4 + x^2$.
Hits:	The number of fitness cases (between 0 and 50) for which the error is less than 0.01
Selection:	Tournament group size of 7, non-elitist, generational
Wrapper:	none
Population Size:	4000
Max program:	8000 program nodes (however no run was effected by this limit)
Initial population:	Created using "ramped half-and-half" with a maximum depth of 6 (no uniqueness requirement)
Parameters:	90% one child crossover, no mutation. 90% of crossover points selected at functions, remaining 10% selected uniformly between all nodes.
Termination:	Maximum number of generations G = 50

8.5 Sextic Polynomial

In studies of a number of benchmark GP problems which have discrete representations and simple static fitness functions we tested the predictions of Section 8.4 and show they essentially hold (see [Langdon and Poli, 1997b; Langdon and Poli, 1998b; Langdon, 1998b] and Section 8.6). In this and the following sections we extend this to a continuous problem which uses floating point operations and has a continuous fitness function, i.e. it has an effectively unlimited number of fitness value. We use the sextic polynomial $x^6 - 2x^4 + x^2$ regression problem [Koza, 1994, pages 110–122]. The fitness of each program is given by its error averaged over all 50 test cases (as given in Table 8.1). We used two ways to test the generality of the evolved solutions. Either using 51 test points chosen to lie in the interval -1 to +1 at random points between the 50 fitness test case points or we used 2001 points sampling every 0.001 between -1 and +1.

8.5.1 GP Runs

In 46 out of 50 runs bloat occurs (see Figure 8.3). In the remaining 4 runs, the GP population remains stuck with the best of generation individual yielding a constant value with mean error of 0.043511. In these four runs the lengths of programs within the population converge towards that of the best individual and no bloat occurs. In 45 of the remaining 46 runs there was a progressive improvement in the fitness of the best of generation individual. In one run the population bloated and there was no improvement in fitness. In some runs the generalised performance of the best of population individual (calculated from 2001 test points) improves steadily with the measured fitness. In other runs generalised perfor-

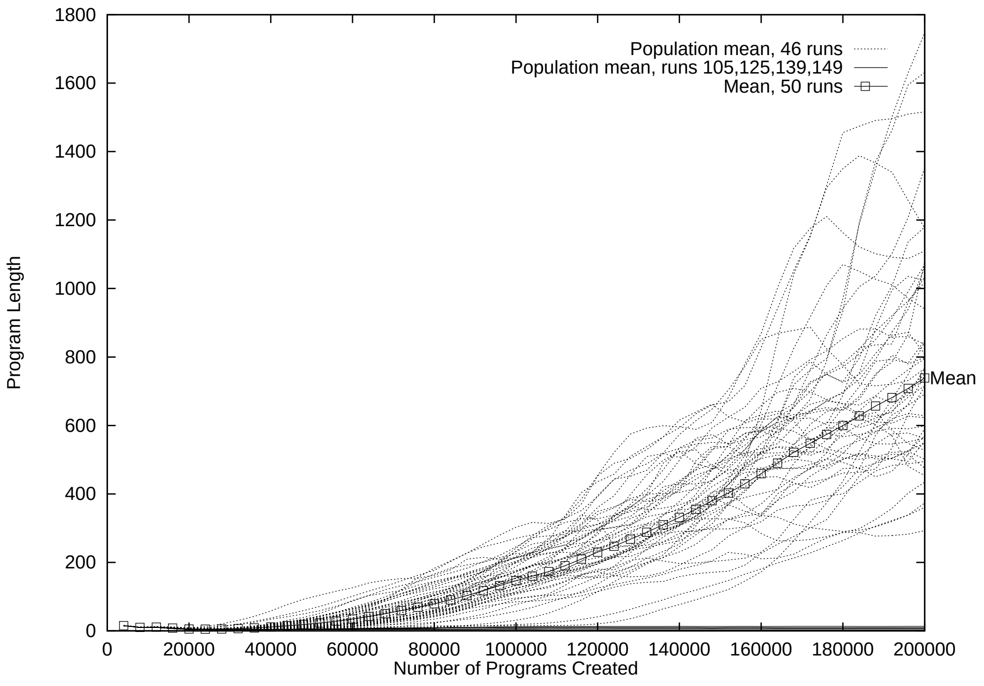

Figure 8.3
Evolution of population mean program length. 50 GP runs of sextic polynomial problem.

mance varies widely and may be exceedingly poor, even for apparently very fi t programs. Figure 8.4 plots the fi tness and generality for three representative runs.

Figure 8.5 shows the evolution of the behaviour of the best of generation individual in the fi rst run. This shows the typical behaviour that the best of the initial random population is a constant. After a few generations typically the GP fi nds more complex behaviours which better match the fi tness test cases. Later more complex behaviours often “misbehave” between points where the fi tness test cases test the programs behaviour. In fact the behaviour of the best of generation individual (including its misbehaviour) is remarkably stable. Note this is the behaviour of single individuals not an average over the whole population. We might expect more stability from an average. This stability stresses GP is an evolutionary process, making progressive improvements on what it has already been learnt.

8.5.2 Non GP Search Strategies

In Section 8.4 we predicted bloat in non-GP search. In this section we repeat experiments conducted on discrete benchmark problems but on a continuous domain problem using four non-GP search techniques.

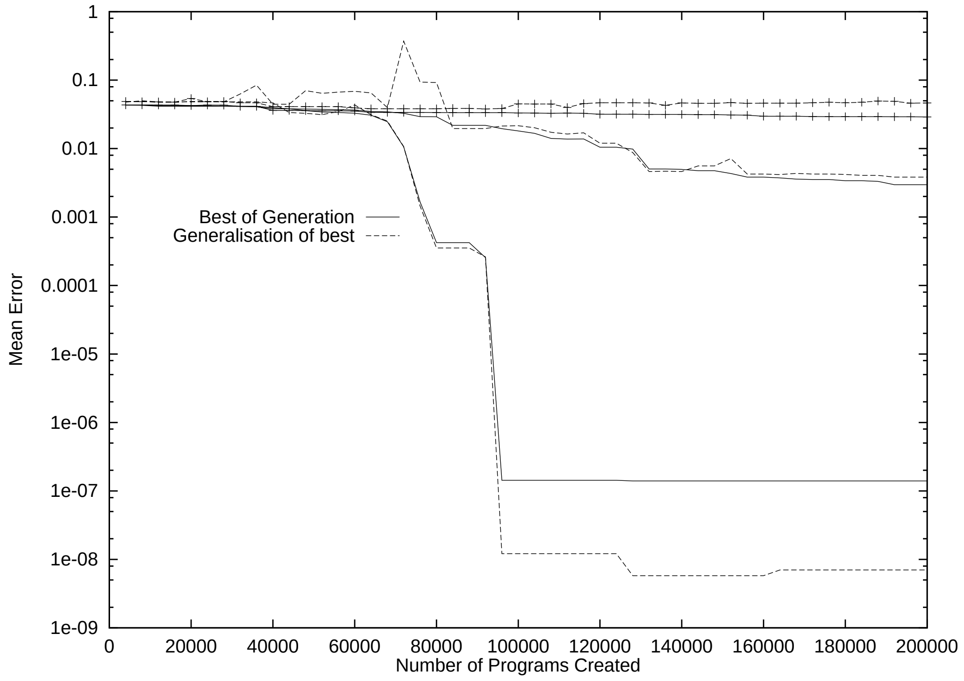

Figure 8.4
Evolution of fitness and generalisation of best individual in population. Best of 50 GP runs, first successful run and first unsuccessful run on sextic polynomial problem.

In Simulated Annealing (SA) an initial random individual is created using the ramped "half and half" method. Each new trial program is created using the mutation operator. It is then evaluated. If its score is better or the same as that of the current one, it replaces the current one. If it is worse then its chance of replacing the current one is $\exp(-\Delta\text{fitness}/T)$. Where T is the current *temperature*. In these experiments the temperature falls exponentially from 0.1 to 10^{-5} after 200,000 trial programs have been created. Whichever program does not become the current one is discarded. A run is deemed successful if at any point it finds a program which scores 50 hits.

Hill climbing (HC) can be thought of as simulated annealing with a zero temperature, i.e. a worse program is never accepted. The runs do not restart (except in the sense that mutation at the root replaces the whole program with a new randomly created one). Strict hill climbing (SHC) is like normal hill climbing except the new program must be better than the current one in order to replace it. Finally population search is a mutation only genetic algorithm with 91% of each generation being created by performing one mutation on a parent in the previous generation and 9% being direct copies of parents in the previous generation. Tournaments of 7 were used to select parents in both cases.

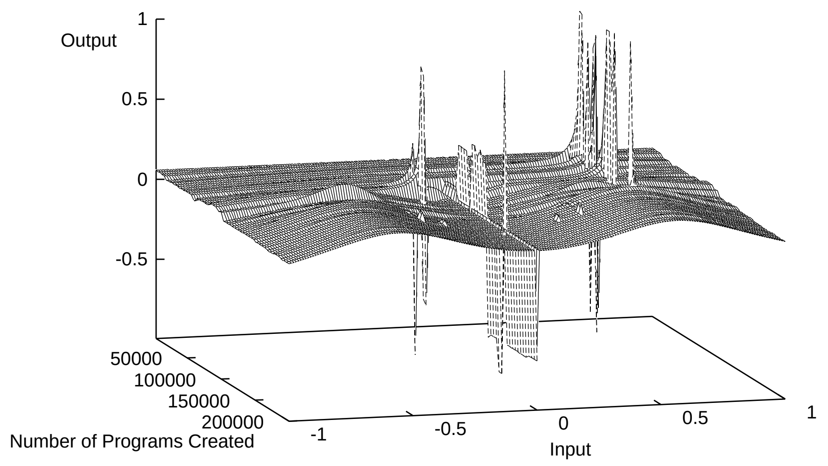

Figure 8.5
Evolution of phenotype. Value returned by the "best" program in the population. First of 50 GP runs of the sextic polynomial problem.

The parameters used are substantially the same as were used in [Langdon, 1998b] on the artificial ant problem. The mutation runs described in these sections use the same parameters as in the GP runs in Section 8.5.1, however a smaller population of 500 rather than 4,000 was used. Also the maximum program size was 2,000 rather than 8,000, see Table 8.2.

Table 8.2
Parameters used on the Sextic Polynomial mutation runs.

Objective etc:	as Table 8.1
Selection:	SA, HC, SHC or Tournament group size of 7, non-elitist, generational
Population Size:	1 or 500
Max program size:	2,000
Initial trial:	Created using ramped "half and half" with a maximum depth of 6
Parameters:	Initial temp 0.1, final 10^{-5} exponential cooling; max inserted mutation subtree 30; mutation points chosen uniformly
Termination:	Maximum number of trials 200,000

8.5.3 New Tree Mutation Operators

For these experiments it is important to be clear about the causes of bloat and so it is more important that the mutation operator should not introduce either a parsimony bias or a tendency to increase program size. Accordingly we introduced a new mutation operator which generates random trees with a specific distribution of sizes, choosing this distribution so on average the new subtree is the same size as the one it replaces. (The algorithm used to create random trees is substantially the same as that given in [Langdon, 1997, Appendix A]. C++ code can be found at `ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code/rand_tree.cc`).

In the first method the size of the replacement subtree is chosen uniformly in the range $l \pm l/2$ (where l is the size of the subtree selected to be deleted). We refer to this as 50%–150% fair mutation. Thus on average the new subtree is the same size as the subtree it is to replace. Should it be impossible to generate a tree of the chosen size or $l + l/2$ exceeds 30 a new mutation point is selected and another attempt to create a new random tree is made. This loop continues until a successful mutation has been performed. Note 50%–150% fair mutation samples programs near their parent uniformly according to their length. Thus neighbouring programs which have the same length as many other neighbouring programs are less likely to be sampled than neighbouring programs which have the same length as few others. As there are many more long programs than short ones each long one is relatively less likely to be sampled compared to a shorter one. That is the 50%–150% size distribution has an implicit parsimony bias.

In the second method the size of the replacement subtree is the size of a second subtree chosen at random within the same individual. We call this subtree fair mutation. Since this uses the same mechanism as that used to select the subtree to replace, the new subtree is on average the same size as the subtree it replaces. It should always be possible to generate a tree of the chosen size, however a limit of 30 was imposed to keep certain tables within reasonable bounds.

8.5.3.1 50–150% Fair Mutation Runs

In simulated annealing runs at initial high temperatures fitness and length vary rapidly but as the temperature falls the variability of program fitness also falls. In contrast the size of the current program continues to vary rapidly as it appears to execute a random walk. However on average program size shows little variation after an initial rise.

In runs using 50–150% mutation with both hill climbing and strict hill climbing there is very little variation in either length or fitness. Indeed 50–150% mutation hill climbing finds it difficult to progress past the best constant value. Few runs are successful but bloat does not happen.

When 50–150% mutation is used in a population it is easier to pass the false peak associated with returning constant value and more runs are successful (albeit only 6 out of 50, see Table 8.3). There is limited growth in program size in the first few generations (as-

Table 8.3
Sextic Polynomial and Artificial Ant: Means at the end of 50 runs. The number of Sextic Polynomial runs which found a program which scored 50 hits, the number where a best of generation individual scored 2001 hits on the generalisation test and the mean length of programs in the final population. The number of Ant runs where a program was able to complete the Santa Fe trail (89 hits) within 600 time steps and the the mean length of programs in the final population.

	Sextic Polynomial						Artificial Ant, 25,000 trials			
	50%–150%			Subtree-sized			50%–150%		Subtree-sized	
	50 hits	2001	Size	50 hits	2001	Size	89	Size	89	Size
Simulated Annealing	6	0	217	32	4	1347	4	95	2	1186
Hill Climbing	1	1	21	15	0	1838	3	41	2	1074
Strict Hill Climbing	2	1	22	16	0	1517	8	32	3	78
Population	6	2	32	28	0	553	12	40	6	127
Population after 10^6 and 10^5 trials							19	287	6	329

sociated with improvement in the population fitness) followed by a long period where the population size average size is almost constant. As in the artificial ant problem[Langdon, 1998b], very slight growth in the programs within the population can be observed.

8.5.3.2 Subtree Fair Mutation Runs

At the start of simulated annealing runs while the temperature is relatively high the fitness of the current program fluctuates rapidly. As does its size. If we look at the average behaviour less fluctuation is seen, with mean error falling to a low value but on average programs grow rapidly to about half the available space (2,000 nodes) see Table 8.3. On average this slows further growths. However there is still considerable fluctuation in program size in individual runs.

Similar behaviour is seen when using hill climbing or strict hill climbing. Subtree Sized strict hill climbing runs either bloat very rapidly or get trapped at very small (3 or 5 node) programs. In ten of 50 runs programs of fitness 0.043511 were rapidly found but no improvements were found and no bloat occurred. While initially the same happens in the hill climbing runs, eventually in all 50 runs the hill climber was able to move past 0.043511 and rapid bloat follows. With both search techniques average program length grows rapidly towards the maximum size allowed.

When subtree sized fair mutation is used in a population there is a steady, almost linear, increase in program length. This is in contrast to the initial fall and subsequent rapid non-linear growth when using crossover (albeit with a different population size, see Figure 8.3).

The right hand side of Table 8.3 summarises our results when using the same mutation operators and search strategies on the artificial ant problem[Langdon, 1998b]. Comparing program sizes for the sextic polynomial and the artificial ant we essentially see the same bloating characteristics (except in one case).

In the sextic polynomial problem using subtree sized fair mutation and strict hill climbing bloat occurs whereas it did not in the artificial ant problem. We suspect this is simply due to the continuous nature of the problem's fitness function. With gaps between fitness values, strict hill climbing imposes a barrier protecting the current point on the search. This barrier is lowered when the fitness function is continuous and any improvement in performance (no matter how small) can now displace the current program. Thus there is little difference between hill climbing and strict hill climbing.

Very slow bloat is observed in the sextic polynomial when 50–150% fair mutation is used in a population, as was bloat on the ant problem. The rate is even slower in the sextic polynomial at about 0.009 nodes per generation, compared to 0.13. Studying individual runs indicate the populations converge towards the size of the best individual in the population. While the fitness of this program may vary a little from generation to generation it does not show steady improvement and the search remains trapped, often only marginally better than the best fitness a constant can achieve. The slower bloat may indicate it is more difficult for small Sextic Polynomial programs to contain inviable code than it is for small artificial ant programs.

The difference in the performance of the two mutation operators may indicate 50-150% mutation is searching programs that are too short. Certainly the shortest solutions found by the 50 GP runs (at 67 nodes) were bigger than almost all the programs tested by 50-150% mutation. Unfortunately the low density of solutions means that we have not been able to explore the search space using random search to plot the density of solutions w.r.t. length. It would be nice to repeat these experiments using bigger initial programs, i.e. in the neighbourhood of 67 nodes.

8.5.4 Direct Measurement of Genetic Operators Effects on Performance

In this section we isolate the effect different genetic operators have from other influences (e.g. selection, noise, population size) by performing all possible variations of each genetic operation on representative programs and measuring the change in fitness and change in size. We used the 223 best of generation programs from our 50 GP runs which are between 101 and 201 nodes in length and with a fitness no better than 0.02. We know due to convergence of the GP populations these and their descendents are responsible for bloat.

8.5.4.1 Self Crossover

A new program was created by crossing over each program with itself at each possible pair of crossover points As expected most crossovers produce very small changes in both size and fitness. The effects of self crossover are asymmetric. In almost all cases on average children which do worse than their parents are smaller than them. While in about half the cases on average children which have the same fitness as their parent are nearly the same length, the remainder are on average at least one node longer. In most cases it is possible for self crossover to find improvements. In all but 16 of 233 cases these improved children are on average at least one node longer than their parents.

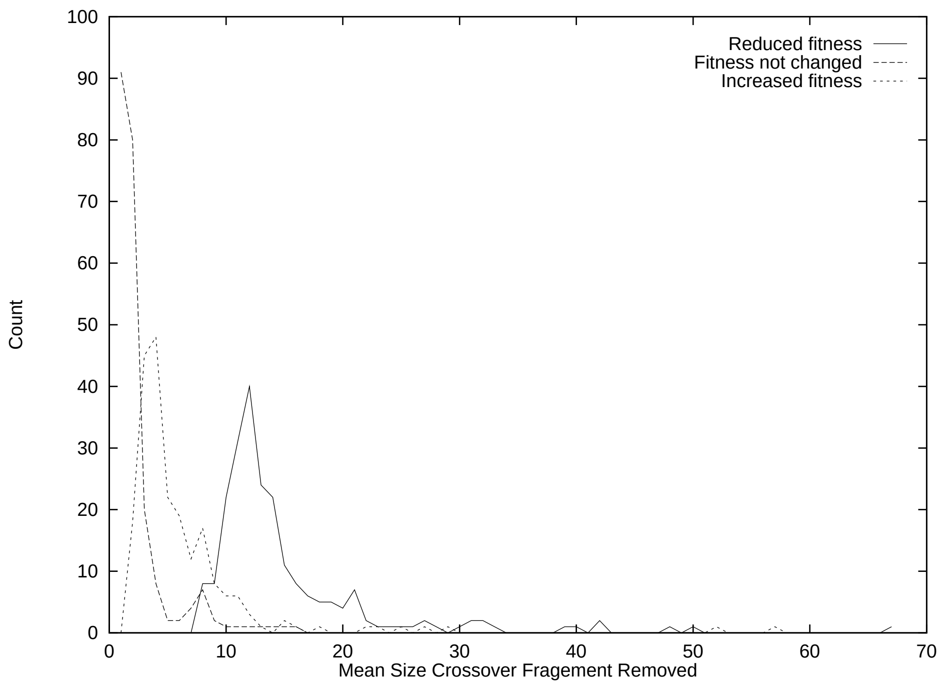

Figure 8.6
Effect of self crossover on best of generation individuals in the Sextic Polynomial problem. For all possible children of the 223 programs the plots show the mean size of the subtree removed by self crossover averaged over those with improved, same and worse measured fi tness. "Removal bias" is indicated as worse children have more code removed.

By considering the sizes of the subtree removed from the parent and the size of that inserted we can discover the cause of this asymmetry. Figure 8.6 shows the size of the code removed by self crossover in most cases is on average bigger when the children perform worse than when either they perform the same or perform better. Figure 8.6 offers clear evidence of "removal bias" (as discussed in Section 8.6.2). In contrast the size of new inserted code is not particularly asymmetric.

8.5.4.2 Mutation Operators

As expected 50–150% Fair Mutation is nearly symmetric. In all cases the mean change in length of worse children is within -0.5 and +0.5. The mean change in length for better children and children with the same fi tness are also almost symmetric.

The affects of subtree sized fair mutation are similar to those of self crossover, especially w.r.t. the asymmetry of change in fi tness and change in size.

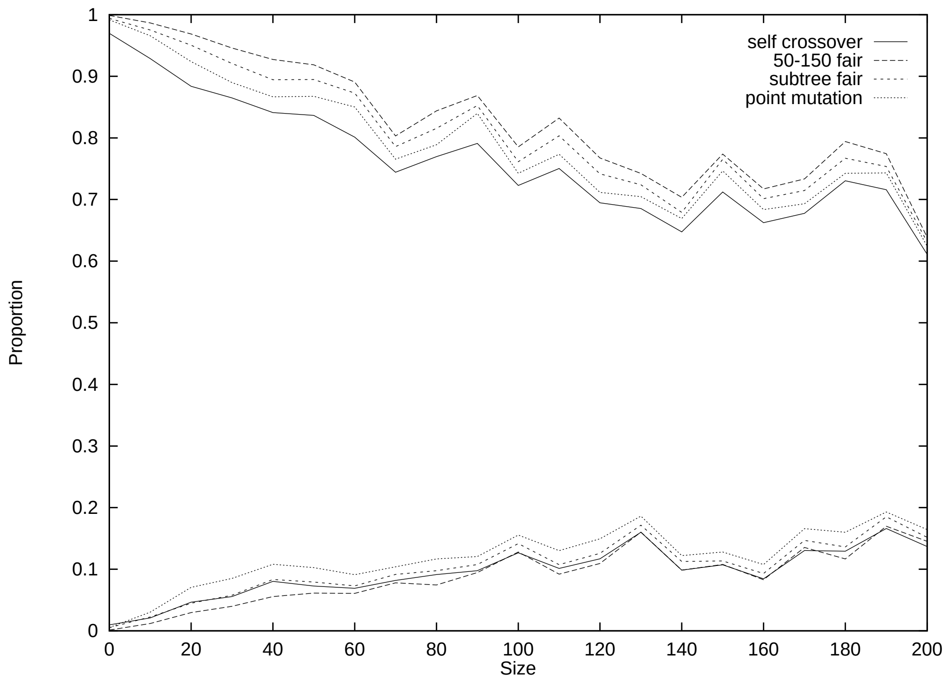

Figure 8.7
Effect of mutations on best of generation individuals in the Sextic Polynomial problem. (Means of data grouped into bins of size 10). The protective effect of inviable code is shown in the general trend for longer programs to be less likely to be disrupted.

Figure 8.7 shows the proportion of single point changes that increase fi tness and those that make it worse. Initially programs are short but increase as the population evolves, so a similar plot is obtained if we replace size as the horizontal axis by generations. As expected initially all high fi tness individuals are fragile and over 95% of point mutation reduce fi tness. As programs grow, the chance of a single point mutation reducing fi tness decreases (and the chance of it improving fi tness grows). This is entirely as expected and corresponds to larger programs containing more inviable code. The proportion of worse, same and better programs produced by self crossover, 50–150% fair mutation and subtree sized fair mutation are essentially the same as that of point mutation. This indicates the chance the offspring has a changed behaviour depends mainly on the point in the program which is changed (particularly whether it is viable or not) rather than how it is changed.

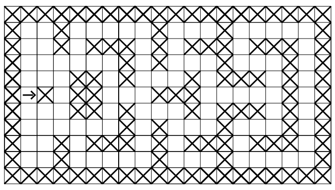

Figure 8.8
The maze used for the maze navigation problem. Walls are indicated by crosses and the start by the arrow.

Table 8.4
Summary of the maze navigation and even 7 Parity problems

Objective:	To navigate a simple maze	To find parity of 7 boolean inputs
Terminal set:	`forward, back, left, right, no_op, wall_ahead, no_wall_ahead`	The 7 input values
Function set:	`if_then_else, while, prog2`	`AND, NAND, OR, XOR`
Restrictions:	Programs were halted after 3000 instructions to avoid infinite loops	
Fitness:	Distance traveled from left wall (0 to 18)	Number of correct cases (of $2^N = 127$)
Selection:	stochastic remainder	
Population size:	500	
Initial population:	random trees	
Parameters:	66.6% crossover, no mutation, results averaged over fifty trials	
Termination:	fixed number of generations	
# of Trials:	50	

8.6 Bloat in Discrete Problems

8.6.1 Code Bloat as Protection

This series of experiments tests the hypothesis that code bloat is a mechanism to protect existing solutions from the destructive effects of crossover and similar code modifying operations. This hypothesis was described in detail in Section 8.2. We began by using a non-destructive (hill-climbing) version of crossover. In the modified operation the fitness of an offspring produced with crossover is compared to the fitness of the parent which supplied the offspring's root node. The offspring replaces the parent only if the offspring's fitness equals or exceeds the parent's fitness, otherwise the parent remains in the population and the offspring is discarded. Thus, survival does not depend on avoiding the destructive effects of crossover and the presumed evolutionary benefit of code bloat is removed.

These experiments were performed on two test problems: a simple maze navigation problem, and the even 7 parity problem. These two problems are summarized in Table 8.4. The maze used with the maze navigation problem is shown in Figure 8.8.

Table 8.5
Code size and fitness at generation 75 with normal or non-destructive crossover.

	Size		Fitness	
	Maze navigation	Even 7 parity	Maze navigation	Even 7 parity
Normal Crossover	590.58	542.81	15.23	91.39
Non-destructive Crossover	185.67	158.64	16.68	99.24

Table 8.5 compares the size and performance of programs evolved using normal and non-destructive crossover. The trials with normal crossover show obvious bloat, in contrast with non-destructive crossover the program sizes are much smaller. The use of non-destructive crossover significantly lowers the amount of code growth observed but there is no significant change in fitness by the end of the runs. These results agree with other results using non-destructive crossover [O'Reilly and Oppacher, 1995; Hooper et al., 1997; Soule and Foster, 1997] This is very strong evidence that code growth occurs, at least partially, as a protective mechanism against the destructive effects of crossover.

If code bloat is a protective, conservative mechanism it should occur to protect against other, primarily destructive, operations. We tested this possibility by looking at the rates of code growth when mutation was used in addition to crossover.

If mutation is applied at a constant rate per node then the probability of a mutation occurring at a given, viable, node is not diminished by the presence of inviable nodes and there is no evolutionary advantage to excess inviable code. Therefore mutation at a *constant rate per node* should not produce code growth.

To test if inviable code is advantageous we used a modified mutation rate. A program was selected for mutation with probability p_m. Once a program was selected for mutation the program's size was used to fix the mutation rate so that an average of N_m nodes would be mutated (i.e. $mutationrate = N_m/programsize$). We refer to this form of mutation as *constant number* mutation. Because the average number of mutated nodes is constant, a larger number of inviable nodes makes it less likely that viable code is affected, presumably producing an evolutionary advantage in favor of code bloat.

For these experiments p_m = 0.3 and N_m = 4. It is important to note that a single tree node is mutated. Thus, this mutation operation does not change the size of the program. Because code size is not directly affected by the inclusion of mutations any changes in code bloat must be an evolutionary effect. The rate of crossover was reduced from 0.667 to 0.333. This decreases the total amount of bloat making the effects of mutation on size, if any, easier to observe. Crossover was applied, with offspring replacing their parents, to produce a new population. Then mutation was applied to that population.

We also applied mutation at a constant rate per node. The probability of a node mutating was 0.02. In this case additional bloat cannot protect against the destructive effects of mutation and additional bloat is not expected.

Table 8.6
The effects of mutations on code bloat in the maze navigation and even 7 parity problems. The data are averaged over 50 trials and taken after generation 50.

	Size		Fitness	
	Maze navigation	Even 7 parity	Maze navigation	Even 7 parity
No mutations	245.6	240.6	14.3	90.4
Constant Rate	206.4	285.8	10.8	85.9
Constant Number	363.5	420.3	13.0	95.7

Table 8.7
Code size and fitness at generation 75 with normal, non-destructive, or rigorous non-destructive crossover.

	Size		Fitness	
	Maze	Even 7 parity	Maze	Even 7 parity
Normal Crossover	590.58	542.81	15.23	91.39
Non-destructive Crossover	185.67	158.64	16.68	99.24
Rigorous Non-destructive Crossover	69.68	66.93	16.03	92.49

Table 8.6 shows the effects of mutation for the maze navigation and even 7 parity problems at generation 50. As expected the baseline rate of code bloat is lower for these data because of the decreased crossover rate. It is clear from these results that constant number mutations cause a dramatic, significant increase in code bloat whereas constant rate mutations have a much smaller, more ambiguous effect.

These results make it clear that, at least in part, code bloat is a protective mechanism against the destructive effects of code modifying operations. When the possibility of damage from an operator is removed, in this case with non-destructive crossover, the amount of code bloat decreases. When the probability of damage increases, in this case with the addition of mutation, the amount of code bloat increases. Further, code bloat only increases if inviable code can play a protective role. When mutations were applied at a constant rate per node additional viable code could not shield viable code and no additional growth was observed.

8.6.2 Code bloat due to “Removal Bias”

In this section we suggest “Removal Bias” is a second cause of bloat and conduct dynamic experiments to show its effects in evolving GP populations. We use two different types of non-destructive crossover. The first version is identical to the non-destructive crossover described previously. While the second is a more rigorous form in which an offspring replaces its parent only if its fitness *exceeds* its parent’s fitness. Table 8.7 shows rigorous non-destructive crossover produces significantly less bloat.

The only difference between the two forms of crossover is that the non-rigorous form allows offspring of equal fitness to be preserved. Thus, the change in code bloat indicates that the offspring of equal fitness are, on average, larger than their parents. Larger, equivalent solutions are more plentiful and, thus, easier to find than smaller, equivalent solutions. This produces a general increases in size even though fitness may not be improving. However, when rigorous non-destructive crossover is used, the larger equivalent programs are no longer kept and most of the bloat vanishes.

Equivalently we can view this in terms of the program landscapes of the three crossover operators. Subtree crossover densely links the search space allowing ready access to the neighbouring programs. Most of these with similar fitness are bigger than the start point and bloat follows. With non-destructive crossover all the links to worse programs are replaced with links back to the current program. This reduces the rate of bloat because the chance of moving away from the current program is significantly reduced. Strict non-destructive crossover replaces all the links to programs of the same fitness by links back to self. This leaves only links to better programs connected but naturally these are few in number and although they on average lead to bigger programs (because there are more bigger programs) there is much less chance of any movement at all through the program landscape, so bloat is dramatically reduced. In continuous fitness problems there are many links to programs with better fitness (albeit the improvement may be tiny) so we would expect non-destructive crossover not to be so effective in such cases.

In addition, we can hypothesize a particular mechanism which produces these larger, equivalent programs. The instructions in a program syntax tree are distributed in such a way that inviable instructions cluster near the branch tips except in extremely pathological trees [Soule and Foster, 1998]. This means that removing a relatively small branch during crossover will decrease the probability of affecting viable code, whereas removing a larger branch increases the probability of affecting viable code. Thus, removing a small branch is less likely to be damaging, because any random change to viable code is more likely to be harmful than to be beneficial. In contrast the size of a replacement branch is not connected to changes in fitness. Thus, there is an overall bias in favor of offspring which only had a small branch removed during crossover.

This "removal bias" leads to steady growth. At each generation the offspring which grew during crossover, because a smaller than average branch was removed, will be favored.

The total average size change during crossover is zero, as every branch removed from one individual is added to another individual and vice versa. However measurements shows that offspring which are at least as fit as their parent are on average slightly bigger, exactly as predicted by the notion of removal bias. Initially the percentage increase is large but within 10–20 generations these transients vanish and steady growth from parent to offspring of the order of 5% is observed. Although the change in size is relatively small it is compounded at each generation and, over many generations, it leads to exponential growth. Thus, this apparently minor bias can have a significant effect on code size.

Removal bias is not limited to crossover. In most versions of subtree mutation a randomly selected subtree is removed and replaced by a randomly generated one. The previous argument regarding the size of the removed and replacement branches applies equally well to this type of subtree mutation. However, removal bias can only occur when the size of the removed branch and the replacement branch are independent. Thus, with the 50%–150% fair mutation used earlier removal bias is not expected to occur, whereas with subtree-sized fair mutation removal bias should take place. This agrees with the results presented previously.

These results make it clear that code bloat has at least two causes. Code bloat occurs as a protective mechanism which minimizes the destructive affects of code modifying operations. Code bloat also occurs because the search space is more heavily weighted towards larger solutions. These solutions are easily found because of the bias in the removal stage of crossover or subtree mutation.

8.7 Evolution of Program Shapes

In addition to changing size, programs with a tree structured genome can also change shape, becoming bushier or sparser as they evolve. In this section we consider the size and depth of the trees. While the density of trees affects the size of changes made by subtree crossover and many mutation operators [Rosca, 1997; Soule and Foster, 1997] our experiments show bloating populations evolve towards shapes that are of intermediate density. As Figure 8.9 shows this can be explained as simple random drift towards the most popular program shapes. In the case of three radically different problems the populations evolve in very similar ways. We suggest this is because all three contain only binary functions and so while the number of different programs in the three cases are very different, the location of the most common shape is the same. For problems with functions with more than two arguments or mixtures of numbers of arguments the exact distribution of depth v. size in the search space will be different but will have the same general characteristics.

Experiments where maze navigation populations were initialised as either all full trees or as all minimal trees of the same size (31 nodes) show in both cases the population evolves away from full trees or minimal trees towards the most common tree shape. However they don't appear to converge (within 75 generations) to the peak, i.e. most common, tree shape. This may be because, as the 5% and 95% lines in Figure 8.9 show, there is a wide spread of probable sizes around the peak.

While we have not yet completed a mathematical analysis of the rate of tree growth with crossover between random trees, such analysis may be tractable. Figure 8.10 gives a strong indication that the average depth of binary trees in a population grows linearly at about one level per generation. Using the relationships between size and depth for random binary trees given in Section 8.3, this corresponds to growth in size of $O(\text{generations}^{1.6})$ for reasonable size programs rising to a limit $O(\text{generations}^{2})$ for programs of programs of more than 32,000 nodes. Note this indicates quadratic or sub-quadratic growth rather than exponential growth. Also the actual program sizes will depend upon their depth when the linear growth begins and so will be problem dependent.

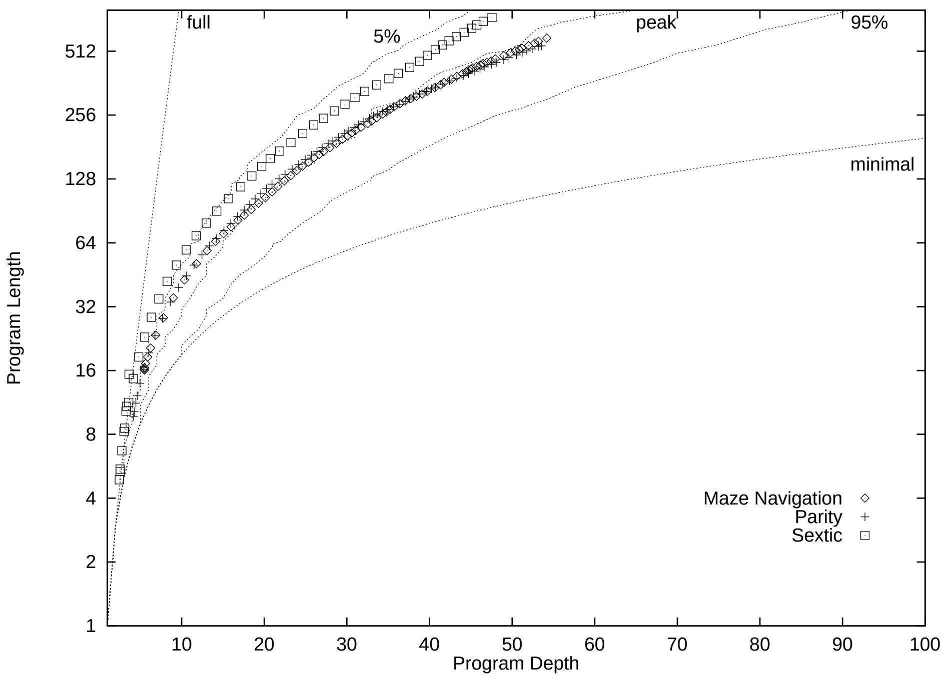

Figure 8.9
Evolution of program tree shapes for the maze navigation, even parity, and sextic polynomial problems. Each point represents the mean size and mean depth of trees in a single generation. Means of 50 GP runs on each problem. For comparison full and minimal trees are plotted with dotted lines, as are the most popular shapes and the boundary of the region containing 5% and 95% of trees of a given length. Note log scale.

This analysis and these data only looked at tree based genomes. It is clear that shape considerations will not apply to linear genomes. However, it is possible that the linear distribution of viable and inviable nodes are subject to some similar considerations. For example, a very even distribution of viable nodes in a linear genome may make it more likely that at least a few viable nodes will be affected by most operations. In which case an even distribution of viable nodes is unlikely to be favored evolutionarily. More complex genomes, such as graph structures, do have shapes and it seems likely that they are also subject to the evolutionary pressures discussed here.

8.8 Discussion

As programs become longer it becomes easier to find neighbouring programs with the same performance. Indeed in our continuous problem it became easier to find neighbours

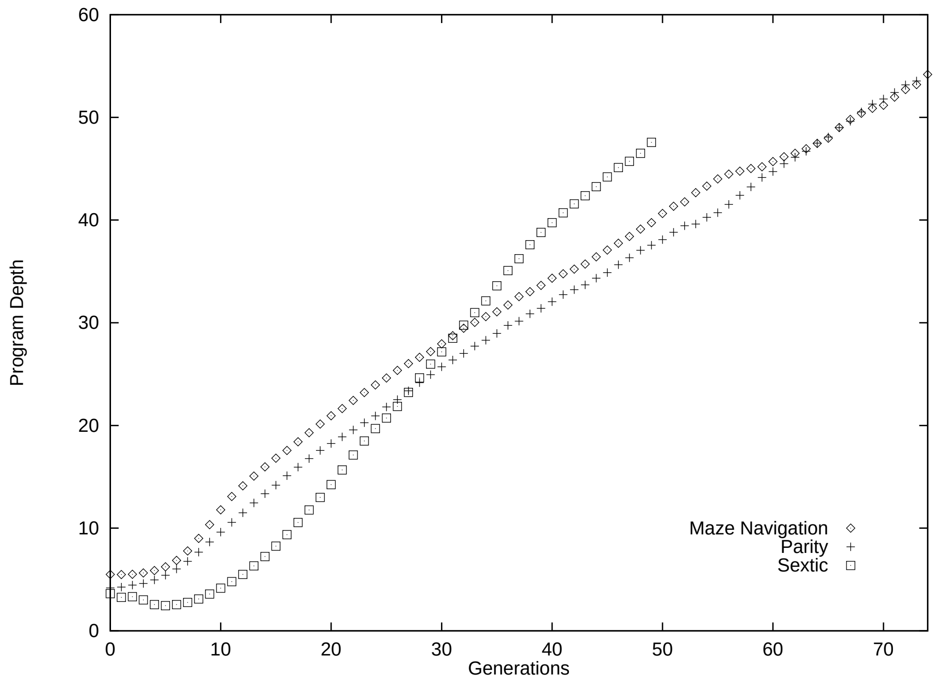

Figure 8.10
Evolution of program tree depth for the maze navigation, even parity, and sextic polynomial problems. Means of 50 GP runs on each problem. Note apparently linear growth in tree depth.

of slightly better fitness. I.e. the fitness landscape becomes smoother for long programs. Alternatively we may view this as it becomes more difficult to make large moves as programs get bigger. Figure 8.5 shows GP is an evolutionary process with stable populations evolving only gradually. From an optimisation point of view this is of course very vexing since it means the stable GP population is not learning. This problem that evolution isn't an optimisation process has been faced before in Genetic Algorithms [De Jong, 1992].

The exponential growth in the number of programs is a very strong driving factor. It may be the cause of bloat even if the fitness function changes rapidly[Langdon and Poli, 1998c]. If we conduct the reverse experiment to Section 8.6 and instead of rewarding programs with the same fitness we penalise them we may still get bloat[Langdon and Poli, 1998c]. Effectively instead of replacing links to worse programs with links back to the current point we remove links to programs of equal fitness. This increases the chance of moving to either better or worse programs but there are still remain overwhelmingly more links to longer programs. So the population still bloats even though inviable code would appear to be a liability.

In Section 8.5.4.2 we showed 50–150% fair mutation removes the large correlation between offspring size and change in fitness seen in other genetic operators. This indicates that by carefully controlling the size of the new code we can avoid "removal bias" as a cause of bloat.

In these experiments it appears that there is faster bloat with subtree crossover than common mutation operators firstly because it does not have an implicit size bias and secondly because it allows larger size changes in a single operation. We can also speculate that random code generated by mutation is less likely to contain inviable code than code swapped by crossover. This potential cause of bloat, if it exists, would be specific to crossover.

8.9 Conclusions

Code growth is a general phenomenon of variable length representations. It is not just a feature of GP. While it is not possible to show it occurs in every case, in Section 8.4 we have argued it is can be expected where there isn't a parsimony bias and we have shown here and elsewhere [Langdon, 1998b] that code growth happens in a variety of popular stochastic search techniques on a number of problems.

Code bloat is such an insidious process because in variable length representations there are simply more longer programs than short ones. It appears to be common for the proportion of programs of a given fitness to be more-or-less independent of program size. In the absence of length bias, the effect of fitness selection on the neighbourhoods of common genetic operators is to prefer programs which act like their parents but exponentially more of these are long than are short. Therefore code growth can be explained as the population evolving in a random diffusive way towards the part of the search space where most of the programs are to be found. Another aspect of this is the shape of trees within the population also evolves towards the more common shapes.

Our research shows that code bloat arises in at least two separate points in the evolutionary process.

1. The mechanics of crossover and subtree mutation typically involve the removal and replacement of a subtree. Often comparatively small changes are more likely not to reduce performance (meaning successful offspring differ little from their primary parent). If the size of the inserted code is independent of that removed this means the added code in successful children is on average bigger than that removed. Thus offspring are more likely to maintain their parent's fitness if the net effect of crossover or subtree mutation is to increase their size. We have called this *removal bias*.

2. It appears to be common that larger programs are on average more likely to produce offspring which retain their parents fitness and thus are more likely to survive. Thus being larger is an evolutionary benefit because a larger program is more likely to have equally fit offspring. This evolutionary benefit arises because the extra code in a larger program has

a protective effect. This makes it less likely that the program's offspring will be damaged during crossover or mutation, regardless of whether the offspring increased or decreased in size.

We have shown in one problem that the proportions of worse, better and unchanged programs are similar for a range of genetic operators. This is consistent with the view that the primary reason for offspring to behave as their parent is that their parent contained inviable code which makes no difference to the program when it is modifi ed. We also show the proportion of inviable code grows with parent size.

In the sextic polynomial problem these proportions are much the same as those of randomly chosen programs of similar fi tness suggesting similar behaviour may be expected in large parts of the search space. The implication of this is GP is mainly sampling "typical" programs. We of course want it to fi nd solutions, i.e. to sample extraordinary programs.

We have proposed a number of new genetic operators. Two of these show promise in controlling bloat. 50–150% fair mutation is carefully constructed to avoid bloat due to the exponential nature of tree search spaces. In discrete problems, non-destructive crossover may limit code growth due to the evolutionary advantage of inviable code.

8.10 Future Work

The success of 50-150% fair mutation at controlling bloat suggests it is worth investigating size fair crossover operators. Such new operators might not only control the size of the replacement subtree but also additional benefi ts might be found by controlling more tightly from where in the other parent the replacement subtree is taken. One point [Poli and Langdon, 1998b] and uniform crossover [Poli and Langdon, 1998a] and Nordin's homologous crossover (Chapter 12) suggest this later step might improve the crossover operator in other ways as well as controlling bloat.

Acknowledgments

This research was partially funded by the Defence Research Agency in Malvern. I would like to thank Lee Spector, Paul Vitanyi and Martijn Bot for helpful suggestions and criticism.

Bibliography

Anderson, S., Bankier, A. T., Barrell, B. G., de Bruijn, M. H. L., Coulson, A. R., Drouin, J., Eperon, I. C., Nierlich, D. P., Roe, B. A., Sanger, F., Schreier, P. H., Smith, A. J. H., Staden, R., and Young, I. G. (1981), "Sequence and organization of the human mitochondrial genome," *Nature*, 290:457–464.

Angeline, P. J. (1994), "Genetic programming and emergent intelligence," in *Advances in Genetic Programming*, K. E. Kinnear, Jr. (Ed.), Chapter 4, pp 75–98, MIT Press.

Blickle, T. (1996), "Evolving compact solutions in genetic programming: A case study," in *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation*, H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel (Eds.), volume 1141 of *LNCS*, pp 564–573, Berlin, Germany: Springer-Verlag.

Blickle, T. and Thiele, L. (1994), "Genetic programming and redundancy," in *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, J. Hopf (Ed.), pp 33–38, Im Stadtwald, Building 44, D-66123 Saarbr ̈ucken, Germany: Max-Planck-Institut f ̈ur Informatik (MPI-I-94-241).

Cavalier-Smith, T. (1985), *The Evolution of Genome Size*, John Wiley & Sons.

De Jong, K. A. (1992), "Are genetic algorithms function optimisers?," in *Parallel Problem Solving from Nature 2*, R. Manner and B. Manderick (Eds.), pp 3–13, Brussels, Belgium: Elsevier Science.

Flajolet, P. and Oldyzko, A. (1982), "The average height of binary trees and other simple trees," *Journal of Computer and System Sciences*, 25:171–213.

Gathercole, C. and Ross, P. (1996), "An adverse interaction between crossover and restricted tree depth in genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 291–296, Stanford University, CA, USA: MIT Press.

Greeff, D. J. and Aldrich, C. (1997), "Evolution of empirical models for metallurgical process systems," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), p 138, Stanford University, CA, USA: Morgan Kaufmann.

Hooper, D. C., Flann, N. S., and Fuller, S. R. (1997), "Recombinative hill-climbing: A stronger search method for genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 174–179, Stanford University, CA, USA: Morgan Kaufmann.

Iba, H., de Garis, H., and Sato, T. (1994), "Genetic programming using a minimum description length principle," in *Advances in Genetic Programming*, K. E. Kinnear, Jr. (Ed.), Chapter 12, pp 265–284, MIT Press.

Kornberg, A. (1982), *Supplement to DNA Replication*, Freeman.

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

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

Langdon, W. B. (1995), "Evolving data structures using genetic programming," in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, L. Eshelman (Ed.), pp 295–302, Pittsburgh, PA, USA: Morgan Kaufmann.

Langdon, W. B. (1997), "Fitness causes bloat: Simulated annealing, hill climbing and populations," Technical Report CSRP-97-22, University of Birmingham, School of Computer Science.

Langdon, W. B. (1998a), *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, Boston: Kluwer.

Langdon, W. B. (1998b), "The evolution of size in variable length representations," in *1998 IEEE International Conference on Evolutionary Computation*, pp 633–638, Anchorage, Alaska, USA: IEEE Press.

Langdon, W. B. and Poli, R. (1997a), "An analysis of the MAX problem in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 222–230, Stanford University, CA, USA: Morgan Kaufmann.

Langdon, W. B. and Poli, R. (1997b), "Fitness causes bloat," in *Soft Computing in Engineering Design and Manufacturing*, P. K. Chawdhry, R. Roy, and R. K. Pant (Eds.), pp 13–22, Springer-Verlag London.

Langdon, W. B. and Poli, R. (1998a), "Boolean functions fi tness spaces," in *Late Breaking Papers at the Genetic Programming 1998 Conference*, J. R. Koza (Ed.), University of Wisconsin, Madison, Wisconsin, USA: Stanford University Bookstore.

Langdon, W. B. and Poli, R. (1998b), "Fitness causes bloat: Mutation," in *Proceedings of the First European Workshop on Genetic Programming*, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty (Eds.), volume 1391 of *LNCS*, pp 37–48, Paris: Springer-Verlag.

Langdon, W. B. and Poli, R. (1998c), "Genetic programming bloat with dynamic fitness," in *Proceedings of the First European Workshop on Genetic Programming*, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty (Eds.), volume 1391 of *LNCS*, pp 96–112, Paris: Springer-Verlag.

Langdon, W. B. and Poli, R. (1998d), "Why ants are hard," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 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.), pp 193–201, University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.

Langdon, W. B. and Poli, R. (1998e), "Why "building blocks" don't work on parity problems," Technical Report CSRP-98-17, University of Birmingham, School of Computer Science.

Li, W.-H. and Graur, D. (1991), *Fundamentals of Molecular Evolution*, Sinauer.

Mattick, J. S. (1994), "Introns - evolution and function," *Current Opinions in Genetic Development*, pp 1–15.

McPhee, N. F. and Miller, J. D. (1995), "Accurate replication in genetic programming," in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, L. Eshelman (Ed.), pp 303–309, Pittsburgh, PA, USA: Morgan Kaufmann.

Nordin, P. (1997), *Evolutionary Program Induction of Binary Machine Code and its Applications*, PhD thesis, der Universitat Dortmund am Fachereich Informatik.

Nordin, P. and Banzhaf, W. (1995), "Complexity compression and evolution," in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, L. Eshelman (Ed.), pp 310–317, Pittsburgh, PA, USA: Morgan Kaufmann.

Nordin, P., Banzhaf, W., and Francone, F. D. (1997), "Introns in nature and in simulated structure evolution," in *Bio-Computation and Emergent Computation*, D. Lundh, B. Olsson, and A. Narayanan (Eds.), Skovde, Sweeden: World Scientific Publishing.

Nordin, P., Francone, F., and Banzhaf, W. (1996), "Explicitly defined introns and destructive crossover in genetic programming," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr. (Eds.), Chapter 6, pp 111–134, Cambridge, MA, USA: MIT Press.

O'Reilly, U.-M. and Oppacher, F. (1995), "Hybridized crossover-based search techniques for program discovery," in *Proceedings of the 1995 World Conference on Evolutionary Computation*, volume 2, p 573, Perth, Australia: IEEE Press.

Poli, R. and Langdon, W. B. (1998a), "On the search properties of different crossover operators in genetic programming," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 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.), pp 293–301, University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.

Poli, R. and Langdon, W. B. (1998b), "Schema theory for genetic programming with one-point crossover and point mutation," *Evolutionary Computation*, 6(3):231–252.

Price, G. R. (1970), "Selection and covariance," *Nature*, 227, August 1:520–521.

Rosca, J. P. (1997), "Analysis of complexity drift in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 286–294, Stanford University, CA, USA: Morgan Kaufmann.

Soule, T. (1998), *Code Growth in Genetic Programming*, PhD thesis, University of Idaho, Moscow, Idaho, USA.

Soule, T. and Foster, J. A. (1997), "Code size and depth flows in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 313–320, Stanford University, CA, USA: Morgan Kaufmann.

Soule, T. and Foster, J. A. (1998), "Removal bias: a new cause of code growth in tree based evolutionary programming," in *1998 IEEE International Conference on Evolutionary Computation*, pp 781–186, Anchorage, Alaska, USA: IEEE Press.

Soule, T., Foster, J. A., and Dickinson, J. (1996), "Code growth in genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 215–223, Stanford University, CA, USA: MIT Press.

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