Fitness Causes Bloat: Simulated Annealing, Hill Climbing and Populations

W. B. Langdon
School of Computer Science, The University of Birmingham, Birmingham B15 2TT, UK
W.B.Langdon@cs.bham.ac.uk http://www.cs.bham.ac.uk/~wbl
Tel: +44 (0) 121 414 4791, Fax: +44 (0) 121 414 4281

Technical Report: CSRP-97-22

2 September 1997

Abstract

In many cases programs length's increase (known as "bloat", "fluff" and increasing "structural complexity") during artificial evolution. We show bloat is not specific to genetic programming and suggest it is inherent in search techniques with discrete variable length representations using simple static evaluation functions. We investigate the bloating characteristics of three non-population and one population based search techniques using a novel mutation operator.

An artificial ant following the Santa Fe trail problem is solved by simulated annealing, hill climbing, strict hill climbing and population based search using two variants of the the new subtree based mutation operator. As predicted bloat is observed when using unbiased mutation and is absent in simulated annealing and both hill climbers when using the length neutral mutation however bloat occurs with both mutations when using a population.

We conclude that there are two causes of bloat.

1 Introduction

In earlier work [Langdon and Poli, 1997a] we claimed the widely reported [Tackett, 1993; Tackett, 1994; Angeline, 1994; Tackett, 1995; Langdon, 1995; Nordin and Banzhaf, 1995; Soule *et al.*, 1996] phenomenum of programs' length increasing as artificial populations are evolved (known variously as "bloat", "fluff" and increasing "structural complexity") is not a specific to genetic programming (GP). Instead we argued in general bloat is inherent in any discrete variable length representation using simple static evaluation functions provided there is no length bias. We have demonstrated [Langdon and Poli, 1997b] that bloat is not specific property of GP's crossover operator but can occur with other operators, such as random subtree change.

In this paper we seek to further justify our claim by investigating the bloating characteristics of three non-population based search techniques using a novel mutation operator on the same problem as [Langdon and Poli, 1997a] and [Langdon and Poli, 1997b] (i.e. evolving an artificial ant to follow the Santa Fe trail). For comparison we also demonstrate bloat using our new mutation operator in a population.

We have solved the ant problem using 10 different techniques (8 in this paper), in 6 cases bloat was observed. In two of the four cases where bloat was not found, strict hill climbing was used. In these cases a change in length can only happen with an increase in fitness and so fitness independent bloat is not possible. In the remaining two cases, the search strategy was designed to sample uniformly the search space by nearby program sizes. NB it is biased to concentrate upon the smaller trees. Also note that bloat occurred when this search strategy was used in a population based search.

We conclude that there are two causes for bloat. Firstly that due to the tendency of length bias free genetic operators to sample bigger trees and secondly competition within populations favours those that can reproduce most accurately which usually favours longer programs.

In Sections 2 and 3 we describe a typical GP demonstration problem and and summarise the four search techniques. The detailed parameters used are given in Section 4, while the two variants of our new mutation operator are described in Section 5. Section 6 contains the results of the eight sets of experiments, which are discussed in Section 7, while our conclusions are given in Section 8.

Appendix A describes in detail the algorithm for creating random trees, while Appendix B describes the expected change in size due specifically to mutation alone.

2 The Artificial Ant Problem

The artificial ant problem is described in [Koza, 1992, pages 147–155]. It is a well studied problem and was chosen as it has a simple fitness function. Briefly the problem is to devise a program which can successfully navigate an artificial ant along a twisting trail on a square 32×32 toroidal grid. The program can use three operations, Move, Right and Left, to move the ant forward one square, turn to the right or turn to the left. Each of these operations take one time unit. The sensing function IfFoodAhead looks into the square the ant is currently facing and then executes one of its two arguments depending upon whether that square contains food or is empty. Two other functions, Prog2 and Prog3, are provided. These take two and three arguments respectively which are executed in sequence.

The artificial ant must follow the "Santa Fe trail", which consists of 144 squares with 21 turns. There are 89 food units distributed non-uniformly along it. Each time the ant enters a square containing food it eats it. The amount of food eaten is used as the fitness measure of the controlling program.

3 Search Strategies

3.1 Simulated Annealing

An initial random individual is created using the ramped "half and half" method. A 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 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 10 to 0.1 after 25,000 trial programs had been created. Whichever program does not become the current one is discarded. However a run is deemed successful if at any point it finds a program which causes the ant to eat all the food.

3.2 Hill Climbing

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).

3.3 Strict Hill Climbing

As hill climbing except the new program must be better than the current one in order to replace it.

Table 1: Ant Problem

Objective:	Find an ant that follows the "Santa Fe trail"
Terminal set:	Left, Right, Move
Functions set:	IfFoodAhead, Prog2, Prog3
Fitness cases:	The Santa Fe trail
Fitness:	Food eaten
Selection:	SA, HC, SHC or Tournament group size of 7, non-elitist, generational
Wrapper:	Program repeatedly executed for 600 time steps.
Population Size:	1 or 500
Max program size:	32,767
Initial trial:	Created using ramped "half and half" with a maximum depth of 6
Parameters:	Initial temp 10, final 0.1, 10^{-7} or 10^{-79}, exponential cooling; max inserted mutation subtree 30; mutation points chosen uniformly
Termination:	Maximum number of trials 25,000, 100,000 or 1,000,000

3.4 Population Based

As a mutation only genetic algorithm with 91% of each generation being created by performing one mutation on of a parent in the previous generation and 9% being direct copies of parents in the previous generation. Tournaments of 7 used to select parents in both cases.

4 Search Parameters

Apart from replacing the population used in [Langdon and Poli, 1997a] and [Langdon and Poli, 1997b] by a single active trial program the parameters are substantially unchanged. However some effects believed to be due to limiting the maximum size of the programs to 500 had been noted and accordingly the maximum size restriction was effectively removed by raising it from 500 to 32,767. The details are given in Table 1, parameters not shown are as [Koza, 1994, page 655]. On each version of the problem 50 independent runs were conducted, except in the case of the longest runs where runtime constrains meant we only ran the first 20 runs.

New trials solutions are created from the currently active one by mutation. This randomly selects a subtree within the program tree and replaces it with a new subtree. The replacement subtree is created randomly.

Preliminary runs were conducted using a mutation operator substantially the same as the mutation operator used in [Langdon and Poli, 1997b]. While results with this operator are substantially the same as we present later, it does have an appreciable parsimony bias. To make interpretation of the results clearer we developed a new subtree mutation operator which, of itself, produces no change in program size.

5 New Tree Mutation Operators

In earlier work [Langdon, 1997; Langdon and Poli, 1997b] we demonstrated bloat despite a slight parsimony bias in the tree based mutation operator. For these experiments it is important to be clear about the causes behind the absence 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. However it proved difficult to ensure a tree based mutation operator based upon the ramped "half and half" method [Koza, 1992, Page 92–93], such as we used in [Langdon, 1997; Langdon and Poli, 1997b], did not have any length bias. 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 given in Appendix A). We investigate two different size distributions with quite different bloat characteristics, even though they both produce on average no size change.

In the first method the size of the replacement subtree is chosen uniformly in the range $l \mp l/2$ (where l is the size of the subtree selected to be deleted). We refer to this as 50%–150% 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. Checks are also made that the mutation does not violate size or depth limits but they are irrelevant to the current experiments. This loop continues until a successful mutation has been performed. Note this operator samples programs near the current one uniformly according to their length. Thus nearby programs (i.e. those that can be created by one mutation of the current one) which have the same length as many other nearby programs are less likely to be sampled than nearby programs where there are few with the same length. There are many more long programs than short ones so 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. (For example in the ant problem with modest program sizes there are about 5.5 times as many programs which are one node longer than a specific length than there are of that length. This ratio increases very slowly with increasing tree size. See also Figure 10 on page 21).

In the case of the ant problem trees of any size except two nodes are possible. As this rules out some of the mutations for subtrees of size three and four, special code deals with these cases to avoid bias. Subtrees of size three are mutated to other subtrees of size three, while those of size four can be mutated to trees of containing three, four or five nodes.

In the second method the size of the replacement subtree is the size of a second subtree chosen at random within the same individual. 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. Should this be exceeded a new mutation point is selected and another attempt to create a new random tree is made. Checks are also made that the mutation does not violate size or depth limits but they are irrelevant to the current experiments. This loop continues until a successful mutation has been performed.

In both types of mutation programs can degenerate to very small trees from which mutation can never escape. Mutation at the root node of such programs results in replacing the whole program by a new one created using the same ramped "half and half" method as used to create the initial population. This occurs with 50%–150% mutation (in the ant problem) with programs of three nodes and with subtree sized mutation when the program contains a single node.

The second means of choosing the size of the inserted subtree is in essence the same as that used in standard GP crossover. The subtree size mutation operator is like performing self GP crossover with one parent acting as both (of course with random code inserted rather than pre-adapted code).

Table 2: Initial tree sizes

	Mean	(SD)
SA and Hill climbers	21	29
Population based	23	33

Table 3: Santa Fe Problem: Means of 50 runs after 25,000 trials

	50%–150%			Subtree-sized		
	Success	Score	Size	Success	Score	Size
Simulated Annealing	4	62	95	2	55	1186
Hill Climbing	3	62	41	2	60	1074
Strict Hill Climbing	8	67	32	3	59	78
Population (data for best)	12	70	40	6	69	127

6 Results

Initially we performed 50 runs on each of the four search techniques using both variants of the mutation operator (i.e. eight sets of 50 runs). The six non-population techniques all started from the same 50 individuals while the two population experiments both used the same 50 initial populations.

All eight experiments where initially run with a limit of 25,000 trials (results are summarised in Table 3). In two bloat was obvious, the remaining six where extended to 100,000 (cf. Table 4). Finally three experiments where little bloat had been observed were extended to 1,000,000 trials (cf. Table 5). Due to run time considerations only the first 20 runs in each category were run to 1,000,000. In simulated annealing runs the final temperature was reduced in the longer runs so as to smoothly extend the original exponential cooling schedule.

Figures 1 and 2 plot the evolution of the fitness and length of programs in each experiment. In the two population based runs the "best" of the population is plotted.

6.1 Remaining Bias in Mutation Operator

In this section we investigate whether the observed change in program sizes are due to the mutation operator alone. We conclude that, apart from restarting when the search collapses to trivial programs, the changes produced directly by mutation are consistent with the expected random fluctuations.

Table 4: Santa Fe Problem: Means of 50 runs after 100,000 trials

	50%–150%			Subtree sized		
	Success	Score	Size	Success	Score	Size
Simulated Annealing	11	71	86	na	na	na
Hill Climbing	18	74	53	na	na	na
Strict Hill Climbing	17	76	33	3	63	96
Population (data for best)	19	76	68	6	72	329

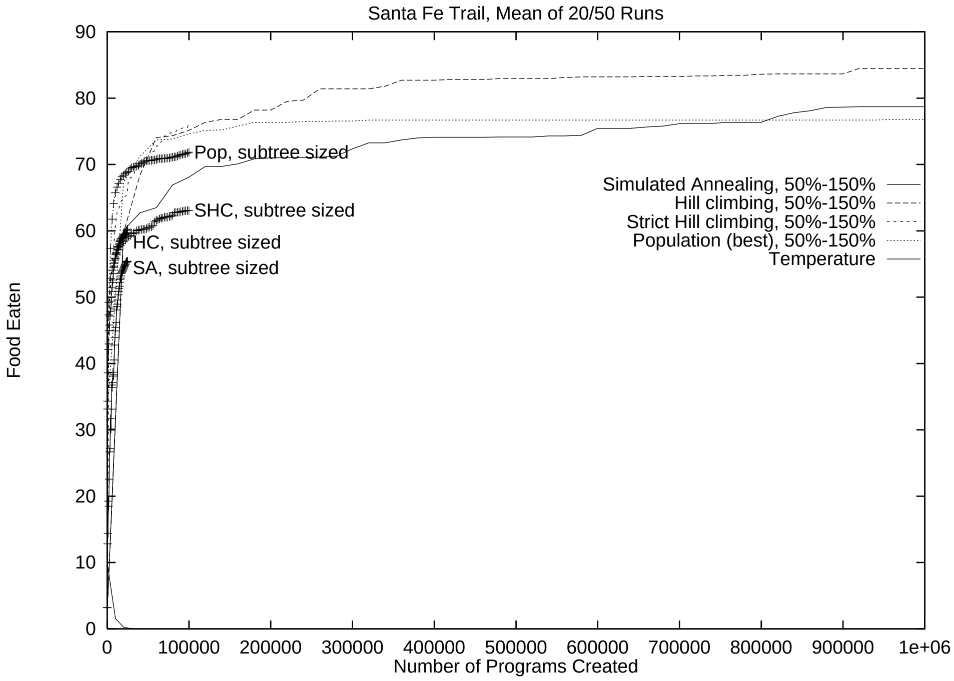

Figure 1: Fitness of Programs, Means of 20/50 runs

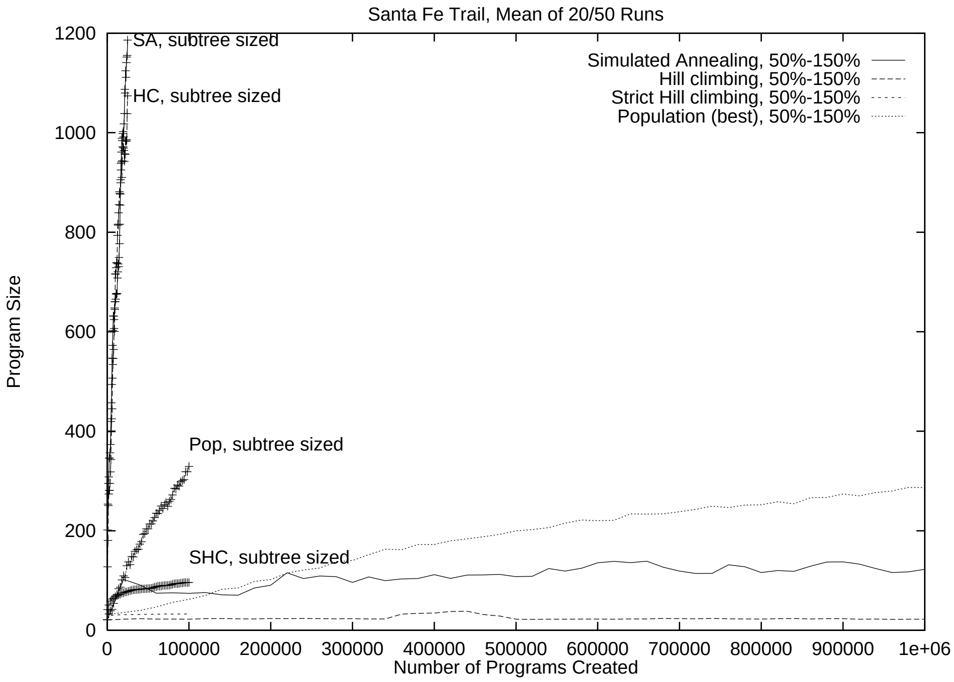

Figure 2: Size of Programs, Means of 20/50 runs

Table 5: Santa Fe Problem: Means of 20 runs after 1,000,000 trials

	50%–150%		
	Success	Score	Size
Simulated Annealing	10	79	122
Hill Climbing	14	85	22
Strict Hill Climbing	na	na	na
Population (data for best)	8	77	287

Table 6: Sum of Mutation size changes: Means of 50 runs after 25,000 trials

	50%–150%				Subtree sized			
	Total	(SD)	Re init	Adj	Total	(SD)	Re init	Adj
Simulated Annealing	112	238	3.84	28	116	1223	6.12	-30
Hill Climbing	-3	302	2.86	-59	59	1164	0.44	49
Strict Hill Climbing	28	309	2.82	-34	-12	989	0.18	-17
Population based	504	268	26.32	-72	-178	1055	0.10	-180
Predicted SD				30				109

The columns headed "Total" in Tables 6, 7 and 8 contain the total change in length produced by all mutations (whether the resulting offspring was accepted or not) during the whole of each run. In the case of the population based runs data is given for the whole population.

In Table 6 the columns headed "Re init" indicate the mean number of times an individual was completely replaced using the ramped "half and half" method. This only happened at the beginning of the runs (i.e. before 2,000 trials had been completed). The columns headed "Adj" contain an estimate of the total change in length produced by all mutations except these reinitialisations. The estimate is based on subtracting the estimated mean size of programs produced by ramped "half and half" (24.874) minus the previous size of the program (3 or 1, depending on mutation type) times the mean number of times this happened.

In Appendix B we develop a crude model of the expected variation of the change in size produced by the two mutation operator. The predicted standard deviations are summarised in Table 13 on page 24 and (taking into account averaging over a number of runs) in the row labelled "predicted SD" in Tables 6, 7 and 8. In all sets of runs the measured residual change in size produced by the mutation operator is within 2.5 predicted standard deviations. Given the crudeness of the model it is reasonable to ascribe remaining changes in size to random fluctuations.

In Table 7 the columns headed "-25k" contain the total change in length produced by all mutations (whether the resulting offspring was accepted or not) between the 25,000th trial and the end of each run. Data is the mean of 50 runs in each case.

In Table 8 the columns headed "-100k" contain the total change in length produced by all mutations (whether the resulting offspring was accepted or not) between the 100,000th trial and the end of each run. Data is the mean of 20 runs in each case.

Figure 3 plot the total change in size produced by the mutation operator (regardless of whether the offspring is accepted or not) in each experiment. In the two population based runs the total for the whole population is plotted. Data is averaged over 50 runs.

Table 7: Sum of Mutation size changes: Means of 50 runs after 100,000 trials

	50%–150%			Subtree sized		
	Total	(SD)	-25k	Total	(SD)	-25k
Simulated Annealing	124	452	12	na	na	na
Hill Climbing	51	475	55	na	na	na
Strict Hill Climbing	1	499	-27	116	1972	129
Population based	451	484	-53	-342	2255	-164
Predicted SD			52			188

Table 8: Sum of Mutation size changes: Means of 20 runs after 1,000,000 trials

	50%–150%		
	Total	(SD)	-100k
Simulated Annealing	105	1574	12
Hill Climbing	320	1295	404
Strict Hill Climbing	na	na	na
Population based	295	1328	-206
Predicted SD			283

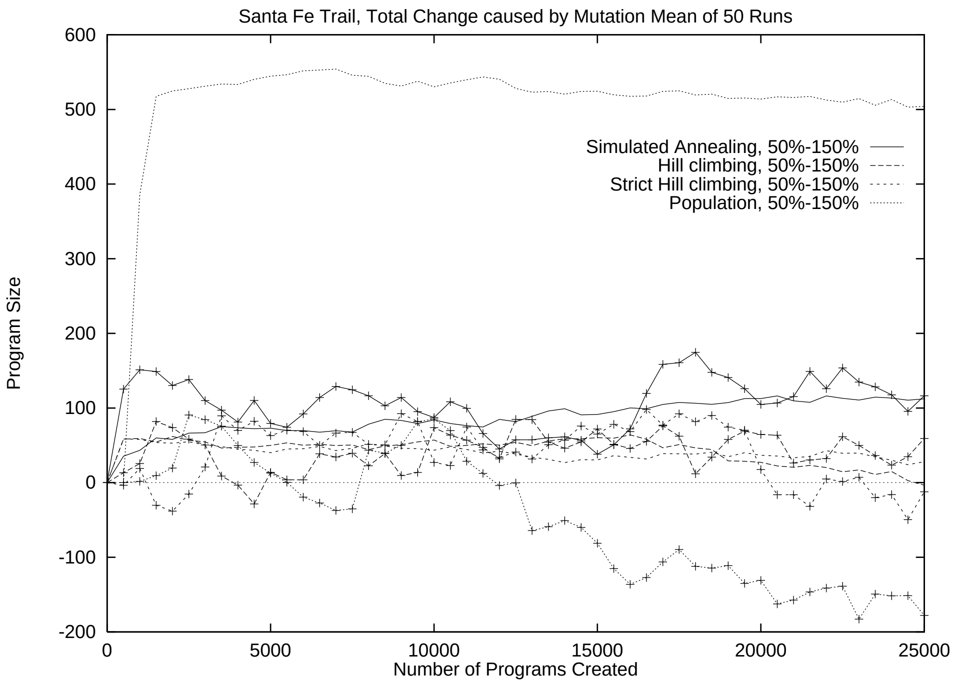

Figure 3: Change in size directly due to Mutation operator, Subtree sized mutation runs shown with crosses, Means of 50 runs

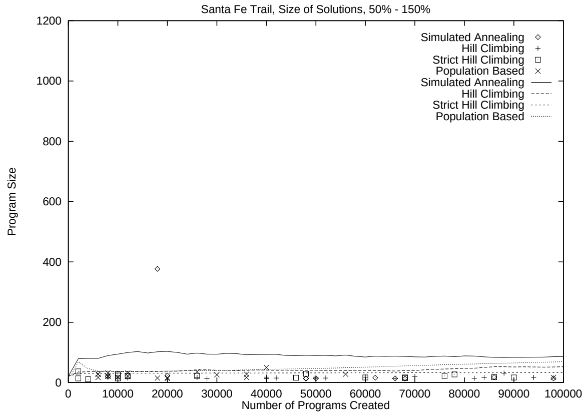

Figure 4: Size of First Solutions and Mean Size, 50%–150%, 50 runs

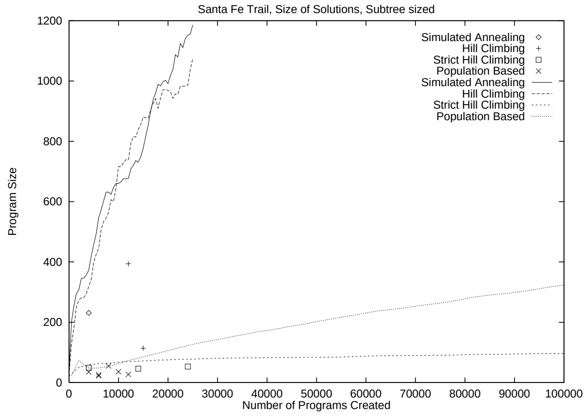

Figure 5: Size of First Solutions and Mean Size, Subtree Sized, 50 runs

6.2 Strict Hill Climbing

In strict hill climbing the current trial individual is only replaced by a new one if the new individual is better than it. This binds the search tightly to the current position and a change in length is only possible with an increase in fitness. With the 50%–150% mutation operator between the $20{,}000^{th}$ and $100{,}000^{th}$ trial fitness (averaged over 50 runs) increased from 65.0 to 76.1 while the mean length changed from 31.3 to 32.8. Of the 32 runs where fitness increased during the interval, in 25 cases the length increased, in two it was the same overall and in 5 cases the final solution was shorter than the one at 20,000.

In contrast with the subtree sized mutation operator the mean fitness increased from 58.4 at 20,000 to 63.0 after 100,000 trails. This was accompanied by an increase in mean length from 75.0 to 96.0 during the same interval. In one run an increase in fitness occurred without increasing the trial solution's length but in the remaining 32 runs where a better solution was found the length also increased. (In 17 runs no improvement was found between 20,000 and 100,000).

6.3 Length of Solutions

Figures 4 and 5 show the size of the first solution in each run. While there are four points above 100, 80% lie in the range 11 . . . 31. This suggests bloat encountered with the subtree sized mutation operator is responsible for its comparatively poor performance, since bloat rapidly moves the search away from short trees where it appears to be easier to find initial solutions.

6.4 Evolution of Program Sizes

In this section we investigate the evolution of the change in program length when using our 50%–150% mutation operator. As we said in Section 6.1 we expect both mutation operators to produce some growth in the very beginning of runs and discovery of improved solutions may result in some change in size. However once program fitness has saturated we expect the 50%–150% to produce random fluctuations in size, i.e. we expect size to perform a bounded random walk. When we look at the average of 50 or 20 runs (e.g. in Figure 2) the initial growth and random fluctuations can be seen, however the average behaviour conceals a trapping phenomena where program size changes very little. In this section we investigate this and show population search behaves differently from simulated annealing and hill climbing.

6.4.1 Simulated Annealing and Hill Climbing

In these experiments simulated annealing and hill climbing perform similarly. This is reasonable since the temperature is cooled below the smallest gap between fitness values by the 12,500th trial. This means after the 12,500th trial the chance of any fitness decreasing step being accepted to is small and so simulated annealing behaves like hill climbing and we will treat them together.

Once the temperature has cooled and the local fitness landscape has been well explored there is little chance of accepting any new trial unless it has the same fitness. Under these circumstances we expect the 50%–150% mutation operator to effectively execute a random walk in the space of computer program lengths. However the walk is bounded as a certain amount of program code is required in order to follow the current food trail. Therefore some movement away from this end point is expected but once away from the limit, the expected program size will not change. However the expected variation in program size will increase (Figure 6 shows a typical run).

We can contrast this with the subtree sized mutation operator which, under the same circumstances, samples the space of programs which have the same score. When moving at random in this space the chance of moving back to the start point is small, instead a random move is much more likely to be to a bigger program.

6.4.2 Random Walk and Trapping

Initially program length seems to be adequately described as a bounded random walk, however as Figure 6 shows the fluctuations become much smaller. Close investigation of the first simulated annealing run shows after it finds a program with a score of 67 and length 81 no further improvement in fitness is made and only very small changes in length occur. Looking in detail at one of these long static periods we see for 5783 trials 50%–150% mutation is unable to change the structure of the program at all and then it finds a minor change which produces a functionally identical program which is one node longer.

Examining the program, we see of the 81 nodes 9 are the roots of trees which exceed 20 nodes and so cannot be mutated. Of the remaining 72, mutation only succeeds in changing (and having the change accepted) 10 nodes (plus the one more which leads to the change of length). Of these ten nodes, six are terminals and four are subtrees of three nodes, NB 50%–150% mutation is unable to change the length or structure of them. Of these ten subtrees we can show 7 are can never executed. Mutation is able to the change the remaining three in only a few cases and it seems likely it replaces them with different but logically identical code, e.g. replacing (Prog2 Left Right) with (Prog2 Right Left) which has the same effect.

This suggests that without a population 50%–150% mutation is likely to become trapped by convoluted programs which contain small amounts of useless code, especially when it is broken into separate subtrees with only 1 or 3 nodes. (NB it may be difficult for any operator to increase the fitness of such code).

6.4.3 Trapping in Populations

Given the trapping behaviour of 50%–150% mutation in simulated annealing we might expect similar behaviour to occur in populations. Such trapping is observed, however eventually populations are able to escape from small programs and bloat occurs. This section describes the

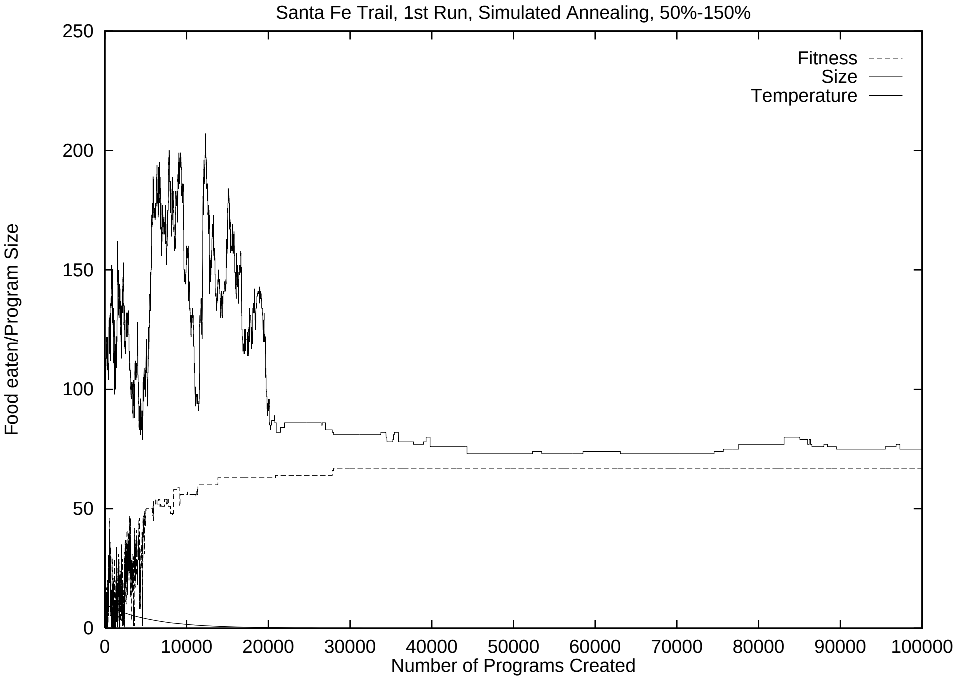

Figure 6: Random walk in program size followed by trapping, First Simulated Annealing run with 50%–150%.

trapping phenomenum in populations and explains how non-executable code allows populations to escape small program traps.

Between generations 30 and 40, the size of the "best" individual did not change in 22 of the 50 population runs using 50%–150% mutation. In the remaining 28 runs the change was small and only in 3 did it exceed six. I.e. trapping does occur, however populations are able to escape being trapped by small programs (bloat occurred before generation 600 in all 20 runs).

Figure 7 shows part of the first 50%–150% run in a population where the whole population is effectively trapped at a single short 14 node individual which scores 53. (Figure 9 on page 17 shows more of the run). Between generation 20 and 56 the population has converged towards the best individual in the population with approximately 60% of the population being duplicates while 88% of children are produced from 14 node programs which score 53. Most of the mutations which score 53 produced during this interval are identical to their parent and only 380 high fitness changes are found, in all but 11 of them mutation has simply replace one terminal with another. The population eventually escapes when a large mutation replaces most of this program and creates a child with higher fitness.

It is not necessary for mutation to find an improved offspring for a population to escape a small program. For example the second run finds a solution of length 14 in generation 38 (19,000 trials) and the population remains stuck near it for many generations (cf. Figure 8). However a series of small changes increase the length of programs in the population so by generation 166 the "best" individual has grown to 33 nodes. Of these 33, 17 either cannot be executed or have no effect and they are concentrated primarily in one large group of 15 nodes. The executable part of the program is very similar to the 14 node solution found in generation 38. From generation 240 onwards the population bloats. The rot seems to have started in generation 50 when a mutation of the 14 node solution created a functionally identical program containing two useless nodes by replacing a tree of five nodes with one of seven. While the non-executed code in this tree is too small for 50%–150% mutation to increase directly, it

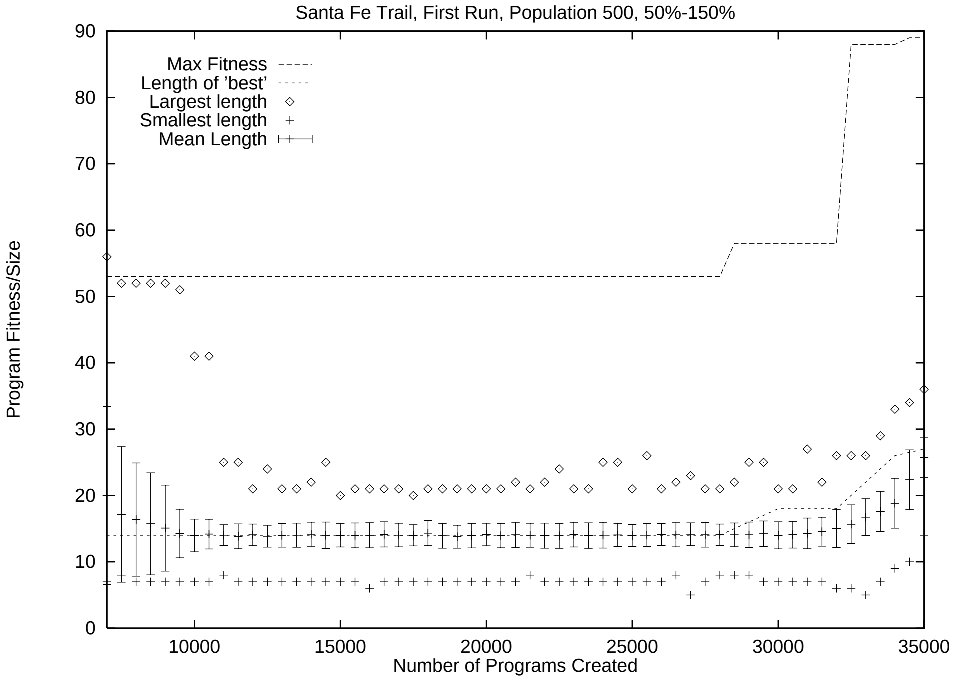

Figure 7: Evolution of Fitness and Size in First Population run with 50%–150%, Error bars indicate the standard deviation of lengths within the population.

acts as a stepping stone for the discovery of trees of sizes 18, 21, 23 and 25 (shown with arrows in Figure 8). Each of these has progressively more non-executed nodes and so finds it easier to survive in the population. The 25 node tree contains a subtree of 7 non-executed nodes which is large enough for 50%–150% mutation to change in size and so bloat can proceed directly simply by mutation acting upon this cluster of junk code.

Table 9 gives the number of times programs scoring 89 where mutated before trial 160,000 by size up to size 33. The third column shows how often the child also scored 89 and the next column gives the ratio between these. It is clear that as the size of the programs increase, 50%–150% mutation produces a higher proportion of equally fit children. Almost all children which also score 89 are the same size as their parent. Together this gives longer programs which score 89 a competitive advantage in the population over shorter ones with the same score. This because, as they have the same fitness, they are selected for reproduction equally often but a higher proportion of longer programs have a fitness of 89. This gradually leads to them taking over the population and then in turn being replaced by still longer programs. Figure 8 shows the rise in the program size in the population.

Columns 5–8 of Table 9 investigate the children with the same fitness as their parent. We split them into those that are identical to their parents and those that are different. We can model those that are identical by considering the probability of 50%–150% mutation generating a random tree which is identical to the one it replaces. The chance of a large random trees being identical are very small, so we can reasonably estimate the total chance of a program being unchanged by mutation by considering only mutation of its terminals and functions all of whose arguments are themselves terminals. Based on this proportion and column 2 we can predict the number of clones produced by 50%–150% mutation, which is given in column 6. We see good agreement between columns 5 and 6. The predicted proportion of clones remains fairly constant at 0.20 until solutions longer than 20 nodes are found. These are slightly more likely to produces clones as 50%–150% mutation excludes the largest subtrees they contain, so

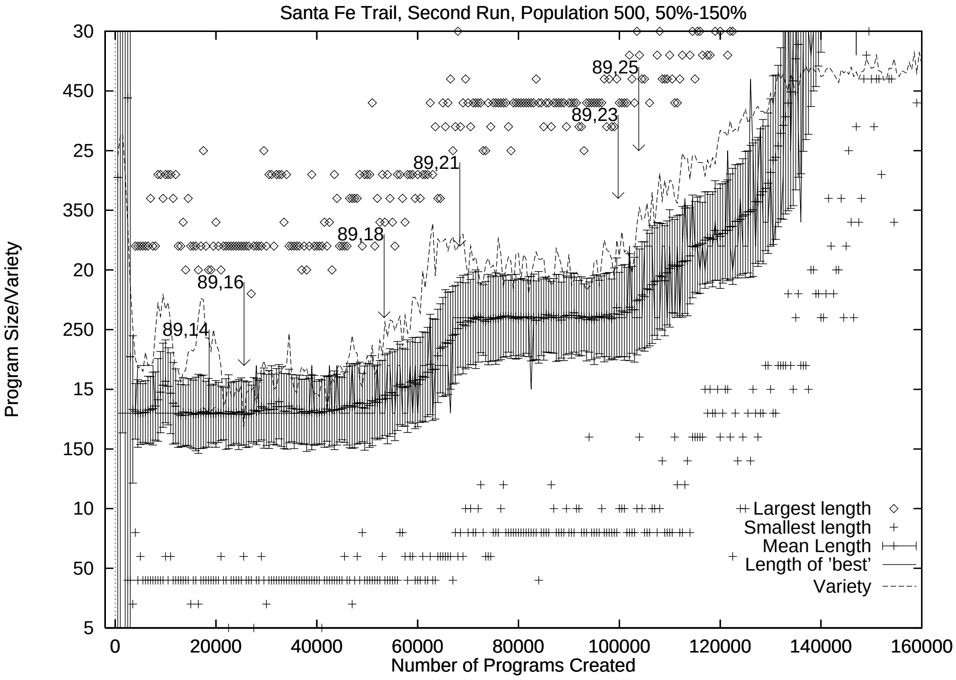

Figure 8: Evolution of Size and Variety in Second Population run with 50%–150%. First solutions of selected lengths shown with arrows. Error bars indicate the standard deviation of lengths within the population.

slightly increasing the chance of choosing a small one.

The bulk of the change in column 4 is due to the increasing chance of creating a child of the same fitness which is not identical to its parent. By investigating in detail a number of programs and determining which nodes cannot be executed we can model the production of high fitness non-clone mutations. Our model assumes all such children are produced by mutating non-executing code and ignores other mutation which can change executed code in such away as to leave the program's score unchanged. As such it is an underestimate. The last column contains the predicted number of non-identical children which score 89, for a number of program sizes. We see there is reasonable agreement between columns 7 and 8. However the predictions are based upon detailed examination of only one program for each size, which may explain why agreement is not quite so good for sizes 21 and 33, as in these sizes there are known to be programs with different structures, particularly w.r.t. non-executable code.

The predicted proportion of mutations which yield different programs with the same score rises with program size from zero to 0.38 for the program of length 33. This accounts for the variation with length seen in column 4 and so in turn for the increase in programs' length's in the population. *I.e. introns cause bloat* in fitness selected populations even when using a length neutral operator.

7 Discussion

In Table 10 we summarise Tables 3, 4, 5, and Figure 2 to yield the bloating characteristics of our two mutation operators and four search techniques. The columns headed "bias?" give the adjusted figures for the last part of the longest runs made (see Tables 6, 7 and 8). In all cases the bias is small, "0" indicates the bias is too small to have been responsible for the observed bloat and only in the case of strict hill climbing is it of the correct sign to have assisted

Table 9: Non disruption of solutions by 50%–150% mutation in second population run (initial pop to 160,000 trials)

Size	Mutated	Child=89	Ratio	Identical	pred	Not identical	pred
14	26370	5310	0.20	5308	5232	2	0
15	726	172	0.24	155		17	
16	7843	1814	0.23	1456	1525	358	327
17	13	0	0	0		0	
18	36189	11734	0.32	7090	7037	4644	4579
19	380	108	0.28	73		35	
20	773	250	0.32	163		87	
21	12819	4827	0.37	2561	2635	2266	1887
22	2172	915	0.42	467		448	
23	2599	1041	0.40	549	525	492	427
24	1987	891	0.45	394		497	
25	1466	671	0.46	275	309	396	428
26	809	395	0.49	161		234	
27	1297	670	0.52	286		384	
28	667	375	0.56	151		224	
29	610	356	0.58	123		233	
30	1316	810	0.62	308		502	
31	494	285	0.58	85		200	
32	720	487	0.68	159		328	
33	814	554	0.68	183	181	371	307

Table 10: Santa Fe Problem: Bloating

	50%–150%		Subtree sized	
	Bloat	bias?	Bloat	bias?
Simulated Annealing	no	12	Big	0
Hill Climbing	no	404	Big	0
Strict Hill Climbing	no	-27	limited	129
Population based	yes	$-ve$	Yes	$-ve$

bloat/non-bloating.

Table 10 indicates the two mutation operators have very different bloat characteristics despite using the same mechanism to create new code and neither having an explicit length bias. The subtree sized mutation operator bloats. This is entirely in keeping with our earlier predictions [Langdon and Poli, 1997a] that bloat would occur if there is no length bias. 50%–150% does not bloat except when used in a population.

7.1 Strict Hill Climbing

As with both simulated annealing and hill climbing, when using strict hill climbing programs in runs with the subtree sized mutation grow in size. However, as described in Section 6.2, strict hill climbing considerably restricts program growth as it only allows changes in size during the rare event of finding an improved program.

7.2 Population Approach

Table 11 gives the rate of change in program size per generation. In 50%–150% populations mean size rises by 0.13 per generation, whereas in subtree sized runs it increased by 1.3. Both are less than in other population based solutions to this problem, e.g. 7.6 in crossover runs and 5.0 in mutation only runs. (The difference between subtree sized runs and earlier mutation only runs may be due to the size limit of 30 on each mutation).

At first sight Figure 2 suggests that bloat in 50%–150% population runs is less dramatic than bloat in simulated annealing or hill climbing using subtree sized mutation. However in simulated annealing runs using subtree sized mutation program size increased at only 0.04 per generated trial solution and little when using 50%–150% mutation. What is it about parallel search in populations which overcomes the implicit length bias in 50%–150% mutation and leads to bloat?

7.3 Populations as Parallel Search

Each individual in the population is created by random change to an individual in the previous generation using 50%–150% mutation, i.e. its length is uniformly randomly changed from that of its parent. So on average each individual has the same length as their parents in the previous generation and each program's size should execute a random walk. The mean of several independent unbiased random walks is itself a random walk. Therefore if the population approach was the same as running n searches in parallel we would expect it to bloat or not in the same manner as the simulated annealing and hill climbing approaches. To explain why it doesn't, we have to consider the different way they handle rejection of trial solutions.

Table 11: Santa Fe Problem: Rate of Increase in Mean Program Size, Means of 20/50 runs

	50%–150%		Subtree sized		SA Subtree sized		GP		Mutation	
	Gen	Size	Gen	Size	Trial	Size	Gen	Size	Gen	Size
	39	35	3	73	2,000	308				
	1,999	284	199	324	25,000	1186	25		50	
Per generation		0.13		1.28		0.04		7.6		5.0

GP[Langdon and Poli, 1997a, Figure 3]
Mutation[Langdon and Poli, 1997b, Figure 11]

With simulated annealing and hill climbing rejecting a mutation implies retention of the current point, NB there is no change in length. In contrast in a population, rejection implies the death of that germ line and its replacement by another more successful one. Therefore the length may change. Bloat arises where there is a systematic variation between the length of the dead individual and its replacement. Such variation arises in the ant problem (and we suggest this is generally true) because shorter programs in the previous generation are more likely to produce children that are worse (i.e. eat less food) than those of average or longer lengths.

Thus when these low fitness programs are deleted, they are, on average, replaced by longer ones. Figure 9 plots the mean size of the parents of offspring which have a lower score than themselves. We see after 80 generations (after the maximum fitness has been reached) they are consistently shorter than the average and the average increases, i.e. the population bloats. So even though offspring are on average the same size as those from which they where produced, fitness based selection uses the variation in program size across the population to increase size from one generation to the next. *I.e. fitness causes bloat.*

The *evolution of evolvabilty* view states that the population evolves to be more evolvable, i.e. more able to produce offspring that are fitter than their parents [Altenberg, 1994]. However in cases of bloat the population does not change over time to increase its chances of finding improved solutions but instead it changes over time to reduce the chance of finding worse solutions. Bloated populations tend to have little chance of improvement.

7.4 Introns

The principal explanation advanced for bloat has been the growth of "introns" or "redundancy", i.e. code which has no effect on the operation of the program which contains it. ([Wu and Lindsay, 1996] contains a survey of recent research in biology on "introns"). Such introns are said to protect the program containing them from crossover [Blickle and Thiele, 1994; Blickle, 1996; Nordin *et al.*, 1995; Nordin *et al.*, 1996]. [McPhee and Miller, 1995] presents an analysis of some simple GP problems designed to investigate bloat. This shows that, with some function sets, longer programs can "replicate" more "accurately" when using crossover. That is offspring produced by crossover between longer programs are more likely to behave as their parents than children of shorter programs. [Rosca and Ballard, 1996] provides a detailed analysis of bloat using tree schemata specifically for GP.

Whilst not disagreeing with this explanation, we have sought a more general one in terms of the general characteristics of search spaces. This predicts in general bloat with any unbiased search operator. We have shown this is true for a particular problem with five different types of search. However our experiments with a length neutral (i.e. implicit parsimony bias) mutation

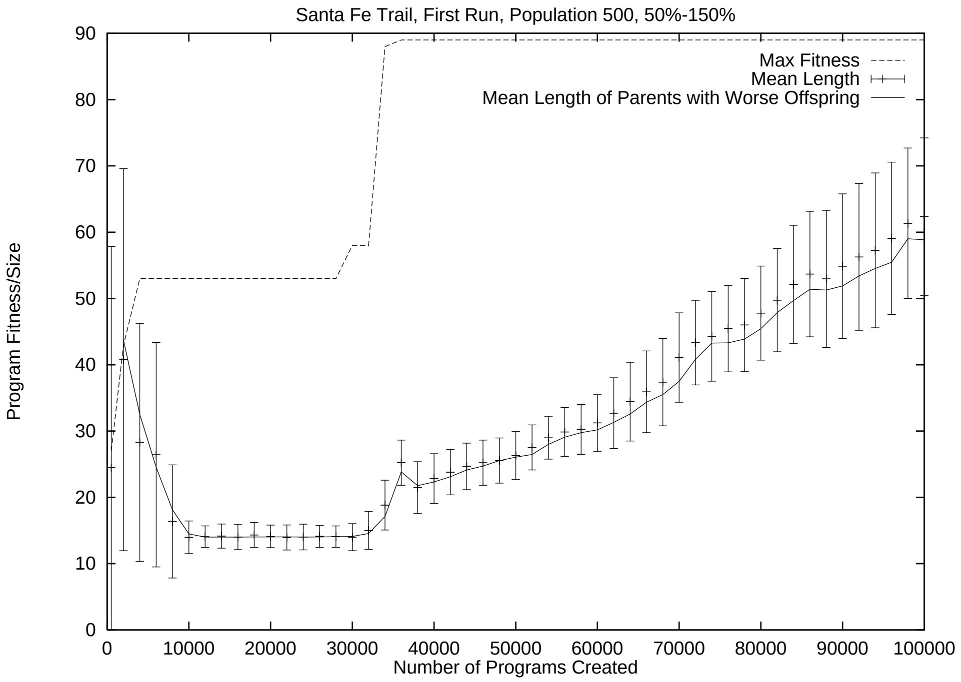

Figure 9: Evolution of Fitness and Size in First Population run with 50%–150%, Error bars indicate the standard deviation of lengths within the population.

operator reveal bloat (albeit at much reduced levels) in parallel search mechanism such as GP. In Section 6.4.3 we showed bloat with our 50%–150% mutation operator can be explained by non-executable code (i.e. by introns).

8 Conclusions

In previous work [Langdon and Poli, 1997a] we advanced a general explanation for bloat which should apply generally to any discrete variable length representation and generally to any progressive search technique. That is bloat is not specific to genetic programming applied to trees and tree based crossover but should also be found with other genetic operators and non-population based stochastic search techniques such as simulated annealing and stochastic iterated hill climbing. In [Langdon and Poli, 1997b] we demonstrated bloat can occur when crossover is replaced by mutation and in this paper we have demonstrated it can indeed occur with simulated annealing and hill climbing although strict hill climbing stifles evolution after it reaches a local optima and in the process cuts off further bloat. However the occurrence of low levels of bloat when using the length neutral (i.e. with an implicit parsimony bias) version of our new mutation operator leads us to suggest the conventional intron explanation for bloat can be a second cause for bloat which applies specifically to parallel search techniques such as GP.

We have described two versions of a new mutation operator. Both on average produce children with the same length as their parent but have different size distributions. This leads to very different bloating characteristics, one form produces no bloat, except in population based search, where bloat is at a reduced rate.

Acknowledgements

This research was funded by the Defence Research Agency in Malvern. I would like to thank Hitoshi Iba for a copy of ETL-95-35.

References

[Altenberg, 1994] Lee Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 3, pages 47–74. MIT Press, 1994.

[Angeline, 1994] Peter John Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

[Blickle and Thiele, 1994] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241).

[Blickle, 1996] Tobias Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich, November 1996.

[Iba, 1995] Hitoshi Iba. Random tree generation for genetic programming. Technical Report ETL-TR-95-35, ElectroTechnical Laboratory (ETL), 1-1-4 Umezono, Tsukuba-city, Ibaraki, 305, Japan, 14 November 1995.

[Iba, 1996] Hitoshi Iba. Random tree generation for genetic programming. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 75–82, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

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

[Koza, 1994] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[Langdon and Poli, 1997a] W. B. Langdon and R. Poli. Fitness causes bloat. In *Second Online World Conference on Soft Computing in Engineering Design and Manufacturing*, 23-27 June 1997.

[Langdon and Poli, 1997b] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In John Koza, editor, *Late Breaking Papers at the GP-97 Conference*, pages 132–140, Stanford, CA, USA, 13-16 July 1997. Stanford Bookstore.

[Langdon, 1995] W. B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Langdon, 1997] W. B. Langdon. Fitness causes bloat in variable size representations. Technical Report CSRP-97-14, University of Birmingham, School of Computer Science, 14 May 1997. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97.

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

[Nordin and Banzhaf, 1995] Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Nordin *et al.*, 1995] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, California, USA, 9 July 1995.

[Nordin *et al.*, 1996] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 6, pages 111–134. MIT Press, Cambridge, MA, USA, 1996.

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

[Soule *et al.*, 1996] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Tackett, 1993] Walter Alden Tackett. Genetic programming for feature discovery and image discrimination. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 303–309, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[Tackett, 1994] Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.

[Tackett, 1995] Walter Alden Tackett. Greedy recombination and genetic search on the space of computer programs. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 271–297, Estes Park, Colorado, USA, 31 July–2 August 1994 1995. Morgan Kaufmann.

[Wu and Lindsay, 1996] Annie S. Wu and Robert K. Lindsay. A survey of intron research in genetics. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 101–110, Berlin, Germany, 22-26 September 1996. Springer-Verlag.

A Random Tree Algorithm

The random tree creation algorithm used by our mutation operator is based upon the random tree algorithm proposed in [Iba, 1995] (and summarised in [Iba, 1996]) for creating initial populations in genetic programming. The algorithm calculates the number of distinct trees shapes which have the required size. It then selects one of these at random in proportion to its count, labels its nodes at random and returns the result. If there are no trees of the requested size, or the size exceeds a preset table size limit, then an error indication is given.

A.1 Tree counting

Iba indicates numeric overflow problems with his algorithm which appear to limit its usefulness [Iba, 1995, page 15], therefore we describe in some detail our implementation of it, indicating how we have avoided this problem whilst restating the algorithm and giving examples of its use in the ant problem.

If all the functions available have the same number of arguments then it is easy to decide if trees of the requested size exists and if so to calculate how many functions and terminals they contains. Further all trees of this size contain the same number functions and the same number of terminals. However where the function set includes functions with different arities there may be different combinations of function arities which give trees of the required size. As the size increases the number of different combinations of arities increases rapidly. (This rise is even more rapid if the number of different function arities is also increased). As a further complication, in general, the number of different trees with each combination of arities will be different. When generating a random tree we follow [Iba, 1995] and choose between such legal combinations at random but in proportion to the number of different trees that can be created using them. Like [Iba, 1995] for simplicity, we count only the tree shapes and do not consider the impact of different numbers of functions (or terminals) of each arity. (This is equivalent to assuming there is only one function or terminal of each particular arity).

Following [Iba, 1995] let n be the size of the tree, A be the available function arities, k be the largest arity and l_i be the number of nodes whose arity is i. Then

$$\sum_{i \in A} i \times l_i = n - 1 \tag{1}$$

If $i \notin A$ we define $l_i = 0$.

In general there are multiple solutions to (1), for each solution the number of terminals r is given by

$$r = n - \sum_{i=1}^{k} l_i \tag{2}$$

For example in the case of the artificial ant problem (1) and (2) become

$$\begin{aligned} 2l_2 + 3l_3 &= n - 1 \\ r &= n - l_2 - l_3 \end{aligned}$$

The number of distinct tree shapes is (see [Iba, 1995, pages 8–9])

$$\overline{Cn} = \frac{n!}{l_1! \times l_2! \times \cdots \times l_k! \times r!} \times \frac{1}{n} \tag{3}$$

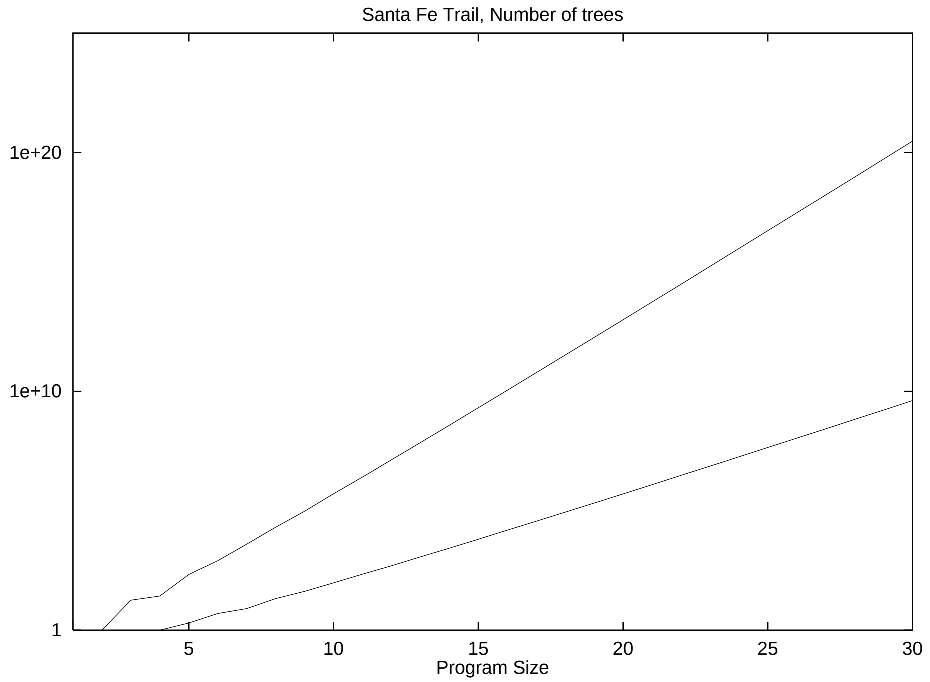

Figure 10: Number of different ant programs of a given size ($\sum \overline{Cn} \times 2^{l_2} \times 3^r$) lower line number of different tree shapes ($\sum \overline{Cn}$)

where $l_1, l_2, \ldots, l_k$ are solutions to (1) and r is given by (2).

Multiplying $\overline{Cn}$ by the number of different ways to label each shape given the available terminals and functions of each arity and summing over all the solutions to (1) gives the total number of different programs possible

$$\text{Number of Programs} = \sum_{j \in \text{ solution to (1)}} \overline{Cn(s_j)} \times |F_1|^{l_1} \times |F_2|^{l_2} \cdots \times |F_k|^{l_k} \times |T|^r \qquad (4)$$

Where $|F_i|$ is the number of distinct functions in the function set with arity i and $|T|$ is the size of the terminal set. The upper line of Figure 10 plots values of (4) for the ant problem (NB log scale). Values for $\sum_{j \in \text{ solution to (1)}} \overline{Cn(s_j)}$ in the ant problem are also plotted in Figure 10 (lower line).

A.2 Creating Lookup Tables

For each tree size a table of $\overline{Cn}$ and $(l_1, l_2, \ldots, l_k)$ values is required. As this is potentially expensive and is required each time a random tree is created all the tables up to a specified size limit are calculated and stored before the main search algorithm commences.

When creating a random tree of a particular size the corresponding table is used to select one of the possible solutions to (1) (i.e. a set of $(l_1, l_2, \ldots, l_k)$ values) with a probability proportionate to its $\overline{Cn}$ value. I.e. the probability of choosing a particular solution s to (1) is

$$\frac{\overline{Cn(s)}}{\sum_{j \in \text{ solution to (1)}} \overline{Cn(s_j)}} \qquad (5)$$

When implementing (3) and (5) to avoid numeric overflow the multiplications of factorials are replaced by additions of logarithms of factorials. (For speed a table of logarithms of factorials is precalculated). Again to avoid overflow each $\log \overline{Cn}$ is normalised by subtracting $\max(\log \overline{Cn(s_j)})$ before performing the summation in (5) which requires converting back from logs by taking the exponent of $\log \overline{Cn}$.

With modest tree size (e.g. 500) and where the range of arities is small but more than one (e.g. 4) the range of $\overline{Cn}$ is enormous. That is, there are some solutions to (1) which are very very unlikely to ever be chosen. Correspondingly the number of solutions to (1) will also be enormous. As each requires space in a table, the space required can be reduced by ignoring small (in comparative terms) $\overline{Cn}$ values. This is implemented by a second pass through all the $\overline{Cn}$ during which we normalise by dividing by $\sum_{j \in (1)} \overline{Cn(s_j)}$ and rejecting all values smaller than floating point precision (FLT_EPSILON). A larger value could be used, if it where desired to save more space. (Removing values smaller than FLT_EPSILON is unnecessary for the ant problem with the mutation size limits used since the tables are small anyway). To allow rapid access via a binary chop algorithm, a running total is stored in the table rather than the normalised probabilities.

A.3 Runtime Creation of Random Subprograms

When a new random subtree is required, a solution to (1) is probalistic selected using the pre-calculated table for the specified tree size, n. To do this we start by generating a random number in the range 0 to the normalised value of $\sum_{j \in (1)} \overline{Cn(s_j)}$ (calculated as described in the previous section). As the table is stored in ascending order the first table entry with a value less than the random number can be rapidly found using a binary chop. The corresponding $(l_1, l_2, \ldots, l_k)$ vector specifies the number of nodes of each arity to use.

We create a random list containing the specified number of nodes of each function arity and the required number of terminals. This list is permuted one element at a time until it corresponds to a 1-dominant word of n x letters and $n-1$ y letters, i.e. until it can be readily converted to a tree. ([Iba, 1995, pages 7–6] says it will always be possible to find such a permutation). We check if a permutation does correspond to a 1-dominant word by starting from the beginning of the list and counting the number of links currently dangling. I.e. we initialise a counter to zero, if the current list element corresponds to a terminal we decrement the counter, if a function we add $1 -$ the function's arity to it. If the counter ever falls below one we reject this permutation and try the next.

The permuted list can now be converted serially into a tree with the right arities at each node. The algorithm to do this is given in [Iba, 1995, pages 5–6]. Briefly we start by initialising a stack of trees. Starting at the beginning of the list, for each element we pop the current list item number of trees (NB if the current element corresponds to a terminal, i.e. zero arity, we pop zero times). We then create a tree composed of a root node with this arity and use as its arguments the trees we have just popped. This tree is then pushed onto the stack. When we reach the end of the list the required tree will be at the top of the stack.

The tree is now recursively converted into a program. Starting at the root, each node is replaced by a function (or terminal) selected at random from those available with the required arity.

Table 12: Large Full Binary Trees, 50%–150% Change in Size

$\lvert\Delta\rvert$	Probability	SD
7	$1/15\ 2/15$	$7\sqrt{n(1/15\ 2/15)}$
6	$1/15\ 2/15$	$6\sqrt{n(1/15\ 2/15)}$
5	$1/15\ 2/15$	$5\sqrt{n(1/15\ 2/15)}$
4	$1/15\ 2/15$	$4\sqrt{n(1/15\ 2/15)}$
3	$1/15\ 2/15 + 2/15\ 2/7$	$3\sqrt{n(1/15\ 2/15 + 2/15\ 2/7)}$
2	$1/15\ 2/15 + 2/15\ 2/7$	$2\sqrt{n(1/15\ 2/15 + 2/15\ 2/7)}$
1	$1/15\ 2/15 + 2/15\ 2/7$	$1\sqrt{n(1/15\ 2/15 + 2/15\ 2/7)}$

B Expected Variation in Mutated Tree Size

In this appendix we describe a crude model of the change in program size due to the mutation operator. We start by considering the 50%–150% mutation operator.

B.1 Expected Size Variation Due to 50%–150%

For simplicity we will assume the population (or current trial program) consists entirely of copies of a large full binary tree. With this gross simplification there are only four types of nodes we need consider. They are the roots of subtrees with 1, 3, 7 and 15 nodes (larger subtrees will exceed the $1.5 \times \text{size} \leq 30$ limit and so can be ignored). There are twice as many subtrees of size 1 than those of size 3, which in turn are twice as numerous as those with 7 nodes, which again are twice as numerous as those with 15 nodes. Given this we can calculate the probability of each particular change in size. These are given in the second column of Table 12. The calculations are simplified by the fact that mutation of subtrees of size 1 and 3 never changes the length.

After a large number of independent trials the mutation operator will on average not change the size of the program, instead it will have performed a random walk. Where there are a large number of mutations it is reasonable to base our estimates upon the assumption that each change in length occurs exactly in accordance with the probabilities given in Table 12 and these can be treated as being independent of each other. Thus the total length will be described a random walk of 7 independent variables and its variance will be the sum of the individual variances and so the expected standard deviation of the change in length will be

$$\begin{aligned}
\text{SD} &= \sqrt{n}\sqrt{\begin{array}{r}49\ 2/225 + 36\ 2/225 + 25\ 2/225 + 16\ 2/225 + 9\ 2/225\ + 4\ 2/225\ \ +2/225 \\ +9\ 4/7\ 15 + 4\ 4\ /7\ 15 + 4/7\ 15\end{array}} \\
&\ \sqrt{n}\sqrt{280/225 + 14\ 4/7\ 15} \\
&\ \sqrt{n}\sqrt{280/225 + 8\ 15/225} && (6) \\
&\ \sqrt{n}\sqrt{400/225} \\
&\ 1.3333\sqrt{n} && (7)
\end{aligned}$$

From (7) we see the expected standard deviation of the total change in size produced by the 50%–150% mutation operator grows slowly as we increase the number of mutation. However (6) emphasises the importance of large mutation on the overall rate of change, even though

Table 13: Large Full Binary Trees, 50%–150% and Subtree Sized Expected Size Change

n	$1.3333\sqrt{n}$	$1.3333\sqrt{n/50}$	$1.3333\sqrt{n/20}$	$4.8589\sqrt{n}$	$4.8589\sqrt{n/50}$
25,000	211	30		768	109
75,000	365	52		1,331	188
900,000	1,265		283	4,610	

Table 14: Large Full Binary Trees, Subtree Sized Mutation Change in Size

$\|\Delta\|$	Probability	SD
14	2 1/15 8/15	$14/15\sqrt{16n}$
12	2 1/15 4/15	$12/15\ \sqrt{8n}$
8	2 1/15 2/15	$8/15\ \sqrt{4n}$
6	2 2/15 8/15	$6/15\sqrt{16n}$
4	2 2/15 4/15	$4/15\ \sqrt{8n}$
2	2 4/15 8/15	$2/15\sqrt{16n}$

large changes are rare events. We should therefore expect increasing the limit of 30 on mutation tree size to have a significant impact on rates of change in program size. Values for particular numbers of mutation are tabulated in columns 2 and 3 of Table 13.

B.2 Expected Size Variation Due to Subtree Size Mutation

We will use the same crude model as we used for 50%–150% mutation. Again there are only four types of nodes we need consider. They are the roots of subtrees with 1, 3, 7 and 15 nodes (larger subtrees will exceed the 30 node limit and so can be ignored). Again we calculate the probability of each particular change in size. These are given in the second column of Table 14.

As with the 50%–150% mutation operator after a large number of independent trials the mutation operator will on average not change the size of the program, instead it will have performed a random walk. We again base our estimates on the assumption that each change in length occurs exactly in accordance with the probabilities given in Table 14 and these can be treated as being independent of each other. So the total length will be described a random walk of 6 independent variables and so the expected standard deviation of the change in length will be

$$\begin{aligned} \mathrm{SD} &= \sqrt{n}/15\ \sqrt{196\ 16 + 144\ 8 + 64\ 4 + 36\ 16 + 16\ 8 + 4\ 16} && (8) \\ & \sqrt{n}/15\ \sqrt{5312} \\ & 4.86\sqrt{n} && (9) \end{aligned}$$

From (9), like with 50%–150%, we see the expected standard deviation of the total change in size produced by the subtree sized mutation operator grows slowly as we increase the number of mutation. Again (8) emphasises the importance of large mutation on the overall rate of change, even though large changes happen less frequently. We should therefore expect increasing the limit of 30 on mutation tree size to have a significant impact on rates of change in program size. Values for particular numbers of mutation are tabulated in columns 5 and 6 of Table 13.