The Evolution of Size in Variable Length Representations

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

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) search operators with no length bias tend to sample bigger trees and 2) competition within populations favours longer programs as they can usually reproduce more accurately.

I. Introduction

In earlier work [9] we claimed the widely reported [3], [7], [13], [16], [17] phenomenon of programs' length increasing as artificial populations are evolved is not specific to genetic programming (GP). The increase in programs' size from one generation to the next while the performance of programs within the population is essentially the same as in previous generations is known as bloat. In our earlier work 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 [10] that bloat is not specific to 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 [9] and [10] (i.e. evolving an artificial ant to follow the Santa Fe trail [6, pages 147–155], cf. Table II). 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 (4 in this paper, cf. bold font in Table I). 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 (rows 14 and 15 in Table I) the search strategy was designed to sample uniformly the search space of nearby program sizes. I.e. 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 (last row in Table I).

TABLE I
Bloat in Ant Problem, 4 search techniques × 2 operators

Successful runs, Mean Score and Program Length

	50%–150%			Subtree-sized		
	Ok	Score	Size	Ok	Score	Size
Means of 50 Runs after 25,000 trials						
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 (best of)	12	70	40	6	69	127
Means of 50 runs after 100,000 trials						
Simulated Annealing	11	71	86			
Hill Climbing	18	74	53			
Strict Hill Climbing	17	76	33	3	63	96
Population (best of)	19	76	68	6	72	**329**
Means of 20 runs after 1,000,000 trials						
Simulated Annealing	10	79	122			
Hill Climbing	14	85	22			
Strict Hill Climbing						
Population (best of)	8	77	**287**			

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.

The two variants of our new mutation operator are described in Section II while Section III contains the results of the eight sets of experiments, these are discussed in Section IV, while our conclusions are given in Section V.

II. New Tree Mutation Operators

Our new mutation operators are motivated by the requirements that they operate on variable length tree structures. Koza's [6] subtree replacement mutation operator does this but it has a length bias. I.e. it can, depending upon the size and shape of the parent, produce children which are on av-

TABLE II
SANTA FE TRAIL ANT PROBLEM

Objective:	Find an ant that scores 89
Terminal set:	Left, Right, Move
Functions set:	IfFoodAhead, Prog2, Prog3
Fitness cases:	The Santa Fe trail
Fitness:	Food eaten
Selection:	Tournament group size of 7 in non-elitist generational populations
Wrapper:	Program repeatedly executed for 600 time steps.
Population:	1 or 500
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; In populations 9% reproduction 91% mutation
Termination:	Maximum number of trials 25,000, 100,000 or 1,000,000

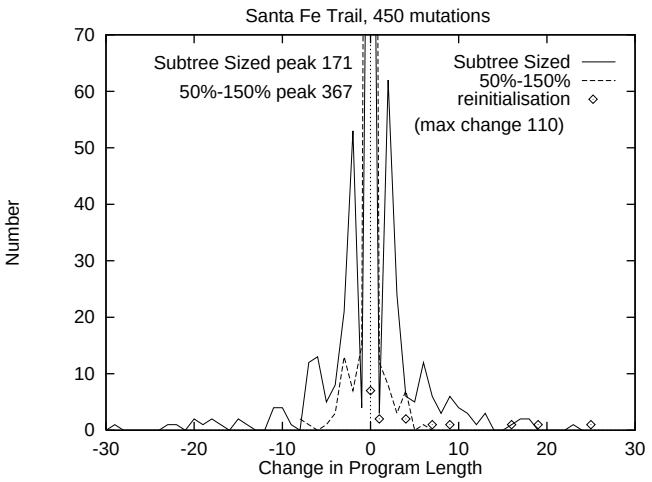

Fig. 1. Change in size produced by mutating individuals created by "ramped half-and-half". 17 50%–150% mutations completely reinitialise very small trees.

erage either larger or smaller. Our mutation operator has no such bias.

The new mutation operator generates random subtrees with one of two different size distributions. The two distributions have quite different bloat characteristics, even though they both produce on average no size change, cf. Figure 1. (The algorithm used to create random trees uses the bijective random tree creation algorithm described in [1, Chapter 4], cf. [5] and [8, Appendix A]).

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. Note this operator samples programs near the current one uniformly according to their length. Thus nearby programs that 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. (In the ant problem there are about 5.5 times as many programs of length $n + 1$ than there are of length n).

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.

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 (shown with diamonds in Figure 1).

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 (with random code inserted rather than pre-adapted code).

III. 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

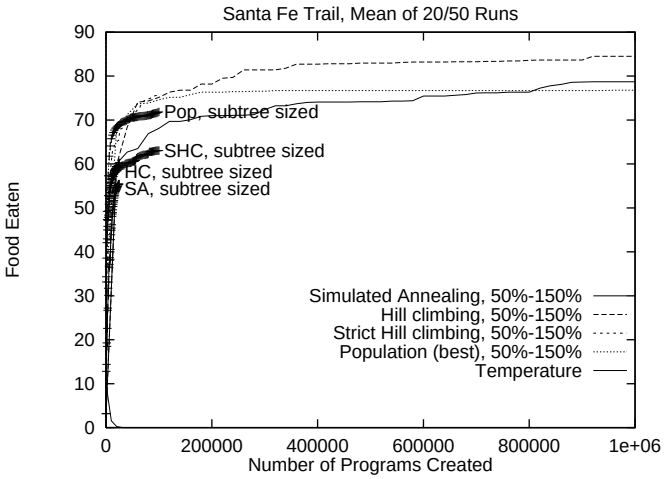

Fig. 2. Fitness of Programs, Means of 20/50 runs

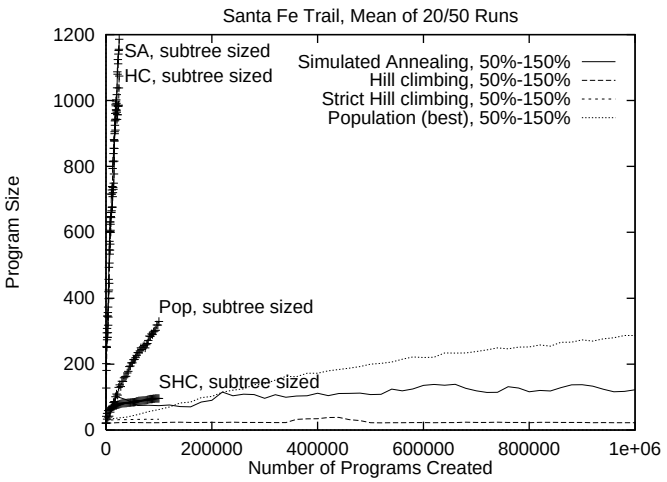

Fig. 3. Size of Programs, Means of 20/50 runs

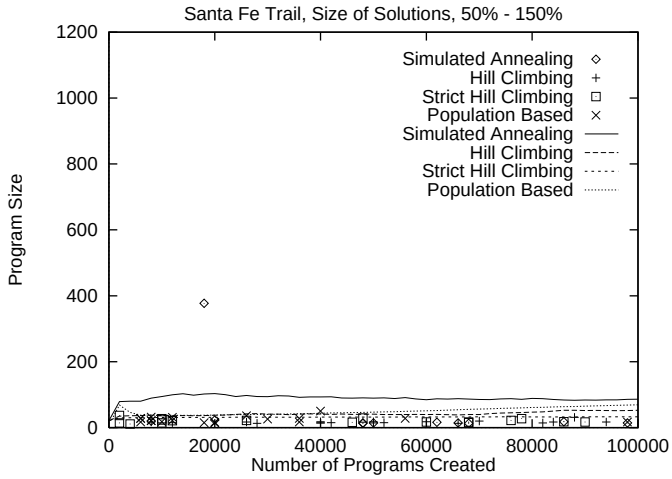

Fig. 4. Size of First Solutions and Mean Size, 50%–150%, 50 runs

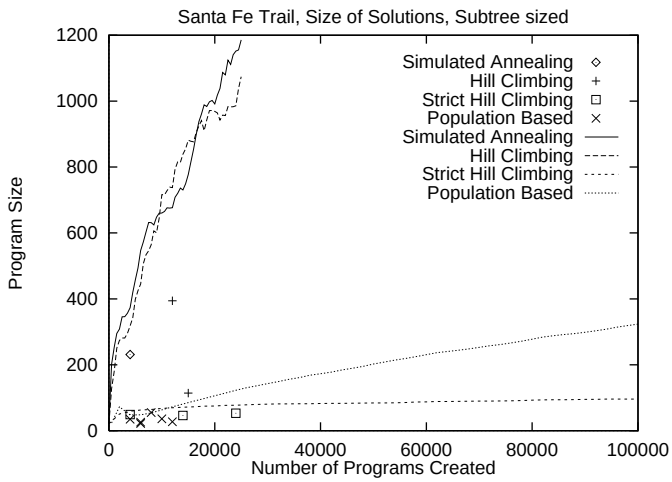

Fig. 5. Size of First Solutions and Mean Size, Subtree Sized, 50 runs

25,000 trials (results are summarised in Table I). Runs that found solutions bloat much like those that get stuck at sub-optimal solutions. In two experiments bloat was obvious, the remaining six where extended to 100,000. Finally three experiments where little bloat had been observed were extended to 1,000,000 trials. 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 2 and 3 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.

A. 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 $11 \ldots 31$. 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 (cf. [11]).

B. Evolution of Program Sizes

B.1 Simulated Annealing and Hill Climbing

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 that is one node longer. (This and other runs are discussed more fully in [8]). Without a population 50%–150% mutation is likely to become trapped by convoluted

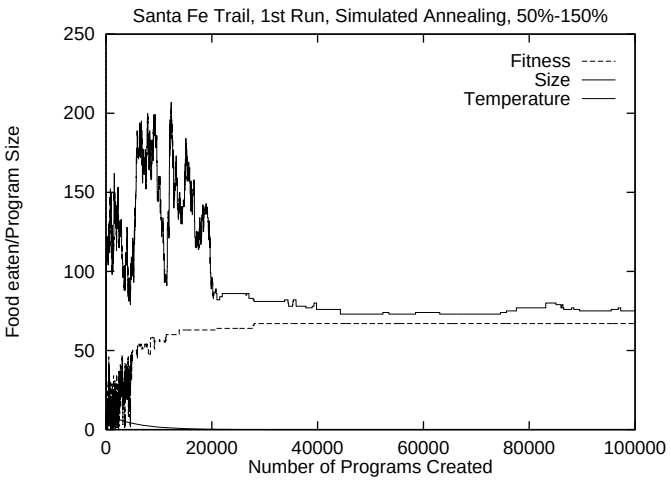

Fig. 6. Random walk in program size followed by trapping, First Simulated Annealing run with 50%–150%.

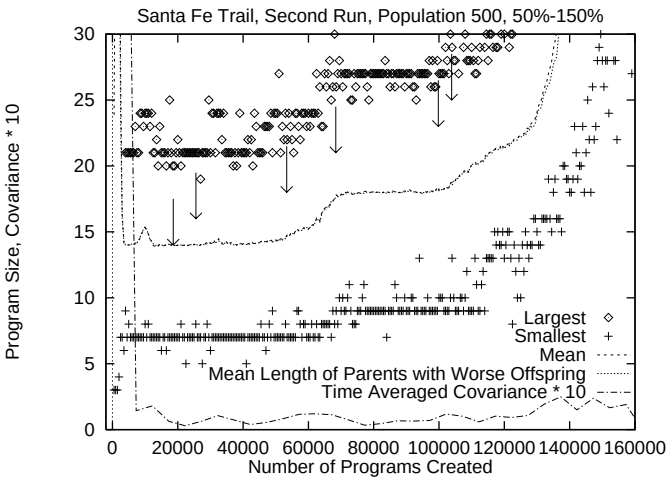

Fig. 7. Evolution of Second Population run with 50%–150%. Covariance of normalised fitness and length averaged over ten generations. First solutions of selected lengths shown with arrows.

programs that contain small amounts of useless code, especially when it is broken into separate subtrees with only 1 or 3 nodes.

B.2 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 trapping phenomenon 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). In the remainder of this section we consider a typical population run using 50%–150% mutation. We build a simple numeric model of how introns or junk code cause longer programs to be able to produce more fitter children. Simple selection within the population exploits this correlation between length and fitness to produce bloat. This is a numerical model of the conventional argument that introns cause bloat by protecting against crossover in GP [13] applied to our mutation operator.

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 7). 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 after which program size increase rapidly.

Table III 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 it as a percentage. 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 that also score 89 are the same size as their parent. Together this gives longer programs that score 89 a competitive advantage in the population over shorter ones with the same score.

Columns 5–8 of Table III investigate the children with the same fitness as their parent. We can model those that are identical by considering the probability of 50%–150% mutation generating a random tree that is identical to the one it replaces. (Measurement and prediction, i.e. columns 5 and 6, are in reasonable agreement). The predicted proportion of clones remains fairly constant at 20%.

The bulk of the change in column 4 is due to the increasing chance of creating a child of the same fitness that 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. The last column contains the predicted number of non-identical children which score 89, for a number of program sizes. In most cases there is reasonable agreement between columns 7 and 8.

The predicted proportion of mutations which yield different programs with the same score rises with program size from zero to 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.

IV. Discussion

In Table IV we summarise Table I and Figure 3 to yield the bloating characteristics of our two mutation operators

TABLE III
NON DISRUPTION OF SOLUTIONS BY 50%–150% MUTATION IN SECOND POPULATION RUN (INITIAL POP TO 160,000 TRIALS)

Size	Childs	89	%	Same	pred	Diff	pred
14	26370	5310	20	5308	5232	2	0
15	726	172	24	155		17	
16	7843	1814	23	1456	1525	358	327
17	13	0	0	0		0	
18	36189	11734	32	7090	7037	4644	4579
19	380	108	28	73		35	
20	773	250	32	163		87	
21	12819	4827	37	2561	2635	2266	1887
22	2172	915	42	467		448	
23	2599	1041	40	549	525	492	427
24	1987	891	45	394		497	
25	1466	671	46	275	309	396	428
26	809	395	49	161		234	
27	1297	670	52	286		384	
28	667	375	56	151		224	
29	610	356	58	123		233	
30	1316	810	62	308		502	
31	494	285	58	85		200	
32	720	487	68	159		328	
33	814	554	68	183	181	371	307

TABLE IV
SANTA FE PROBLEM: BLOATING

	50%–150%	Subtree sized
Simulated Annealing	no	**Big**
Hill Climbing	no	**Big**
Strict Hill Climbing	no	limited
Population based	**yes**	**Yes**

and four search techniques. Table IV 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 [9] that bloat would occur if there is no length bias. 50%–150% does not bloat except when used in a population. (In all cases the mean change in length produced by all mutations is small and consistent with random fluctuations [8, Appendix B]).

A. 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 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. I.e. strict hill climbing considerably restricts program growth.

TABLE V
SANTA FE PROBLEM: RATE OF INCREASE IN MEAN PROGRAM SIZE

	50%–150%		Subtree sz		SA Subtree sz	
	Gen	Size	Gen	Size	Trial	Size
Start	39	35	3	73	2,000	308
End	1,999	284	199	324	25,000	1186
Per gen		0.13		1.28		19
GP[9, Figure 3]				7.6		
Mutation[10, Figure 11]				5.0		

B. Population Approach

Table V 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).

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.

With simulated annealing and hill climbing rejecting a mutation implies retention of the current point, note 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 7 plots the covariance of length with normalised fitness and shows it to be on average positive. Thus, using Price's Theorem, we expect this correlation between size and fitness to drive mean size of programs in the population upwards. Figure 7 also plots the mean size of the parents of offspring which have a lower score than themselves. We see after 250 generations (125,000 trials) they are 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 evolvability* view states that the population evolves to be more evolvable, i.e. more able to produce offspring that are fitter than their parents [2]. 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.

C. 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. Such introns are said to protect the program containing them from crossover [4], [12], [14], [15]. Whilst not disagreeing with this explanation (indeed in Section III-B.2 we showed bloat with our length neutral (i.e. implicit parsimony bias) mutation operator can be explained by non-executable code, i.e. by introns) 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.

V. Conclusions

In previous work [9] 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 [10] 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 population 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

[1] Laurent Alonso and Rene Schott. *Random Generation of Trees*. Kulwer Academic Publishers, 1995.

[2] 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.

[3] 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.

[4] 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).

[5] 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.

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

[7] 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.

[8] W. B. Langdon. Fitness causes bloat: Simulated annealing, hill climbing and populations. Technical Report CSRP-97-22, University of Birmingham, School of Computer Science, 2 September 1997.

[9] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pan, editors, *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London, 23-27 June 1997.

[10] 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.

[11] W. B. Langdon and R. Poli. Why ants are hard. Technical Report CSRP-98-4, University of Birmingham, School of Computer Science, January 1998.

[12] 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, 15-19 July 1995. Morgan Kaufmann.

[13] 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, 15-19 July 1995. Morgan Kaufmann.

[14] 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, 1996.

[15] Justinian P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza *et al.*, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, 13-16 July 1997. Morgan Kaufmann.

[16] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza *et al.* editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, 28–31 July 1996. MIT Press.

[17] 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, 17-21 July 1993. Morgan Kaufmann.