Genetic Programming and Evolvable Machines, 1, 95–119 (2000)

Size Fair and Homologous Tree Crossovers for Tree Genetic Programming

W. B. LANGDON w.b.langdon@cwi.nl
Centrum voor Wiskunde en Informatica, Kruislaan 413, NL-1098 SJ Amsterdam

Received April 21, 1999; Revised August 26, 1999

Abstract. Size fair and homologous crossover genetic operators for tree based genetic programming are described and tested. Both produce considerably reduced increases in program size (i.e. less bloat) and no detrimental effect on GP performance.

GP search spaces are partitioned by the ridge in the number of program v. their size and depth. While search efficiency is little effected by initial conditions, these do strongly influence which half of the search space is searched. However a ramped uniform random initialisation is described which straddles the ridge.

With subtree crossover trees increase about one level per generation leading to sub-quadratic bloat in program length.

Keywords: genetic algorithms, genetic programming, bloat reduction, evolution of shape, sub-quadratic length growth, linear depth growth, uniform initialisation, binary tree search spaces.

1. Introduction

It has been known for some time that programs within GP populations tend to rapidly increase in size as the population evolves [13, 4, 30, 27, 25, 2, 41, 29]. If unchecked this consumes excessive machine resources and so is usually addressed either by enforcing a size or depth limit on the programs or by an explicit size component in the GP fitness measure which penalises larger programs, although other techniques may be used [13, 12, 42, 3, 36, 32, 39, 10]. Both main approaches have problems [13, 30, 38], [9, 20]. Recently there has been increased interest in the underlying causes of bloat [28, 38, 26].

It has been shown that the protective effect of inviable code (which does not effect the fitness of the program) [27, 4] is not sufficient to explain all cases of bloat and shown there are at least two mechanisms involved [26]. However we also suggest these are manifestation of an underlying cause, which is: any stochastic search technique, such as GP, will tend to find the most common programs in the search space of the current best fitness. Since in general there are more of these which are long than there are which are short (but GP starts with the shorter ones) the population tends to be filled with longer and longer programs [21, 24, 18]. This is a general explanation, which does not rely on GP mechanisms, indeed we have shown bloat occurs in several other stochastic search techniques using variable length representations [22, 16]. The exponential growth in the number of programs with size is a very strong driving factor. It may be the cause of bloat even if the fitness function changes rapidly or we penalise programs with the same fitness as their parents [23].

Using this argument we devised an unbiased tree mutation operator which carefully controls variation in size and produces much less bloat. In Section 3 we introduce the corresponding crossover operator and in Section 4 we describe means of extending it to increase the chance of crossover between like parts of parent trees yielding a more homologous operator. We compare the evolution of tree size and depth for the three crossover operators starting from three types of initial random populations: standard "ramped half-and-half" [13, pages 92–93], "ramped half-and-half" with bigger initial trees and ramped uniform random (described in Section 5). In Section 6 we compare both new operators with standard subtree crossover on two continuous domain problems (symbolic regression of the quintic and sextic polynomials) and two discrete problems (Boolean 6 multiplexor and 11 multiplexor). This is followed by a discussion in Section 7 and we conclude in Section 8. However first we review what is known about the distribution of programs and reiterate our claims about the distribution of their fitnesses.

2. Distribution of Programs and their Fitnesses

In genetic programming it is possible to have function sets that contain functions of different arities. I.e. for program trees to have internal nodes of more than one branching factor. However many GP experiments use only binary functions, e.g. the symbolic regression experiments described in Section 6. Therefore in this section we will concentrate upon the case where all the functions are binary (i.e. have two arguments) and so the programs are expressed as binary trees. This avoids dealing with mixtures of arities which complicates the analysis and for which there is little existing work (for example [37] deals with two cases: binary and unrestricted arity trees). We don't expect such additional complexity to increase our understanding at this stage.

There are

$$|T|^{(l+1)/2}|F|^{(l-1)/2} \times \frac{(l-1)!}{((l+1)/2)!((l-1)/2)!}$$

different programs of size l, where $|T|$ is the number of terminals and $|F|$ is the number of functions [13, 1, page 213]. Note this formula is relatively simple as each function (internal node) has two arguments. The number of programs rises rapidly with increasing program length l. Of course if no bounds are placed on the size or depth of programs then the number of them is unbounded, i.e. the search space is infinite. Figure 1 plots the number of different binary tree shapes against their size and the number of different functions for our four benchmark problems. (Size = number internal nodes + number leafs = l). Note while the multiplexor experiments use functions with one, two and three inputs the shape of their curves are similar to the binary cases. Figure 1 clearly shows the number of different programs (for all but the shortest) grows essentially exponentially with their size [1].

We now consider how the number of programs varies with their size and their maximum depth. These are of course related. A tree of a given size cannot exceed

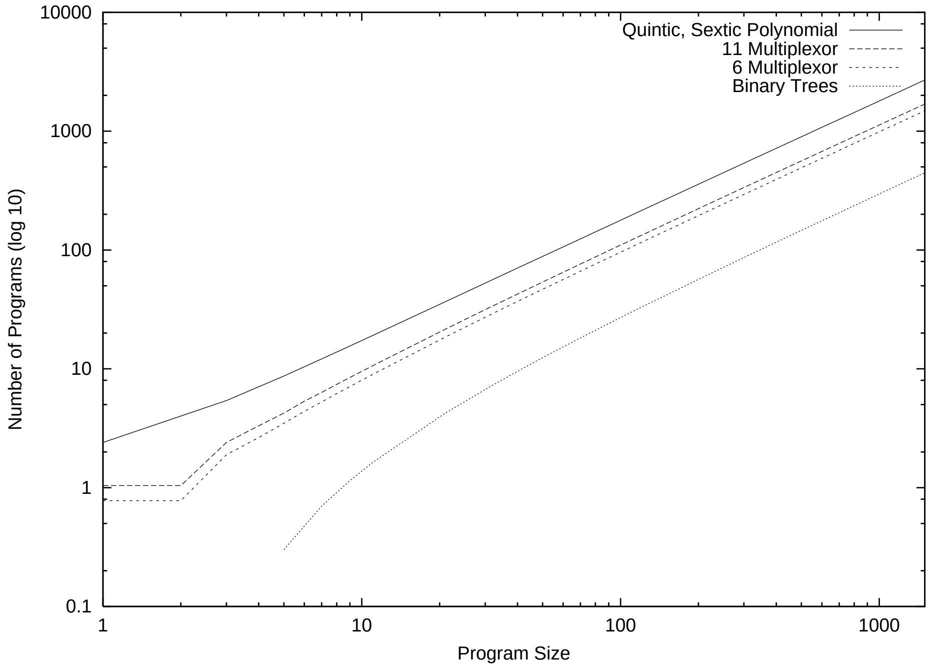

Figure 1. Size of Program Search Spaces (note log log scale)

a certain maximum depth (that of a tree of the chosen size but composed of only one long chain of functions, all side branches terminating immediately in leafs). Similarly its depth cannot be less than a minimum (given by a (nearly) full tree where every branch is continued and leafs only occur at the maximum depth (or one depth closer to the root)). In the case of binary programs (i.e. those composed only of two input functions) the maximum and minimum depths are given by $(l + 1)/2$ and $\lceil \log_2(l + 1) \rceil$. In fact most programs lie between these two extremes, see Figures 2–4.

In binary trees the number of programs of a given size and depth is

$$|T|^{(l+1)/2}|F|^{(l-1)/2}$$

times the number possible tree shapes of that size and depth. Note this depends on tree size but not tree depth. Therefore Figure 2 can be readily converted from number of tree shapes to number of programs by increasing the gradient parallel to the size axis and so retaining its basic shape. (This also holds for non-binary trees if their internal nodes all have the same branching factor).

Figure 3 shows a plan view of Figure 2 in which the spread in the distribution of number of trees can be seen. The distribution is slightly asymmetric and so the peak lies to one side of the mean but close to it. Also note the theoretical large tree quadratic limit [8] to which the mean approaches slowly. Similarly Figure 4 shows a

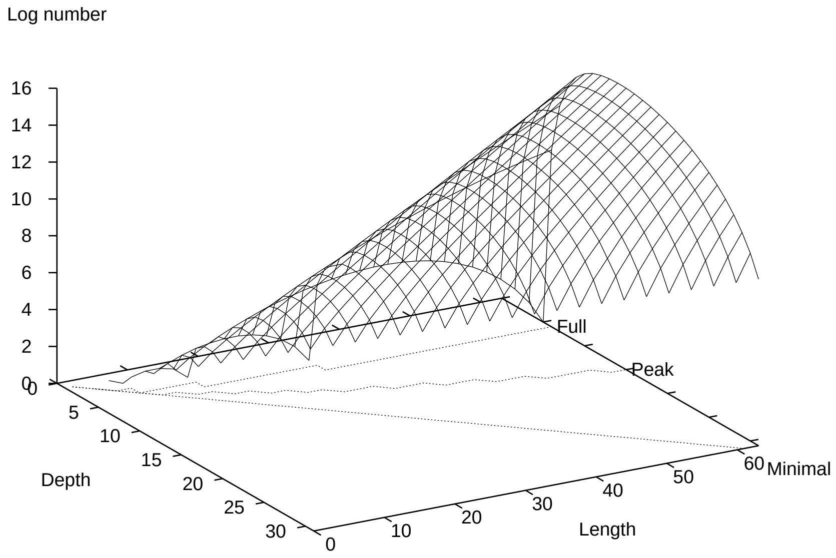

Figure 2. Distribution of binary trees by size (length) and height (depth). Note log scale. The dotted lines show the projection of the bushiest trees (Full), the (Peak) and the sparsest (Minimal), points onto the x-y plane.

plan view of part of Figure 2, where the arrows indicate the direction of maximum increase in numbers of trees (using a simple nearest neighbour three point fit).

In earlier work [24, 18] we suggest in general the distribution of fitness values does not change much with their length, provided they are bigger than some problem and fitness level dependent threshold. (A few special case counter examples have been found). In all examples so far, bloat continues above the threshold and so the threshold can be ignored for the purposes of explaining bloat. We suggest that in general for the bulk of the search space in simple GP problems the proportion of programs with a given level of performance is independent of their size (and further we assume independent of their shape). Thus the number of programs with a given level of performance will be distributed like the total number of programs, i.e. it will have the same shape as the curves shown in Figures 2–4.

To restate our explanation for bloat it is: after a period GP (or any other stochastic search technique) will find it difficult to improve on the best trial solution it has found so far and instead most of the trial solutions it finds will be of the same or worse performance. Selection will discard those that are worse, leaving only those that are as good as the best-so-far active. In the absence of bias, the more plentiful programs with the current level of performance are more likely to be found. But

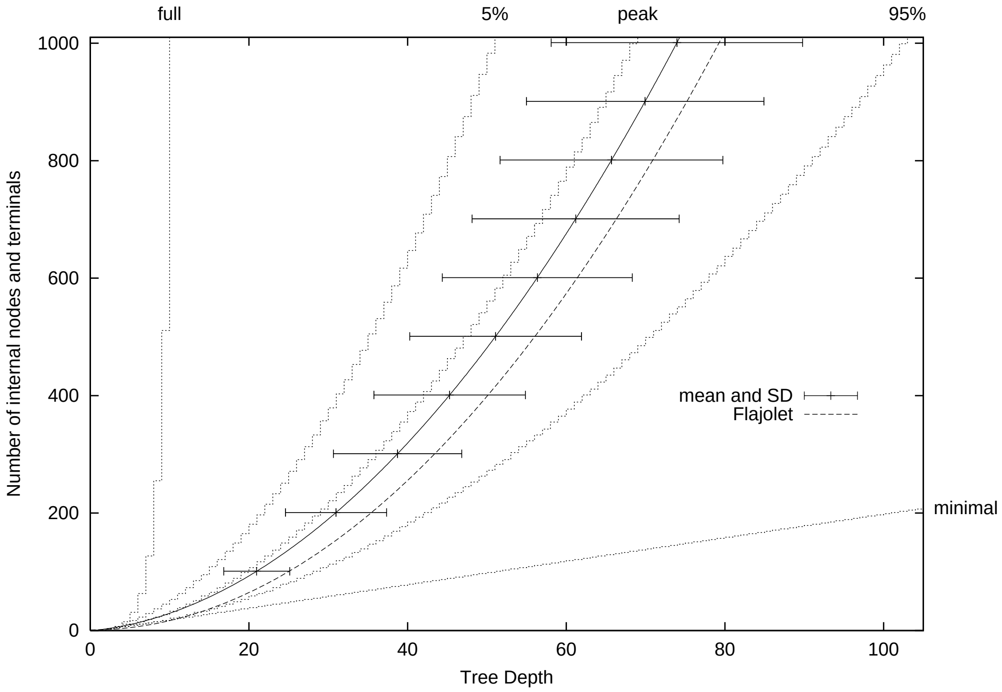

Figure 3. Distribution of binary trees by size and maximum depth, cf. Figure 2. Solid line and error bars indicate the mean and standard deviation of the depth for trees of a give size. The dash line is the large tree limit for the mean, i.e. $2\sqrt{\pi(\text{internal nodes})}$ (ignoring terms $O(N^{1/4})$). The full tree and minimal tree limits are shown with dotted lines, as are the most likely shape (peak) and the 5% and 95% limits (which enclose 90% of all programs of a given size).

as the previous paragraph has argued, the distribution of these is similar to that shown in Figures 2–4, therefore we expect the search to evolve in the direction of the arrows given in Figure 4. [26] confirms this in various diverse problems when using GP with standard crossover.

In the remainder of this paper we discuss two new crossover operations which are carefully constructed so that the fitness landscape they provide to the GP population is unbiased. Instead of the population seeing the huge exponential growth in programs the landscape is tailored to be more even, with an equal chance of selecting a link in the landscape to a shorter program as to a longer one. In this way the population (once the performance plateau has been reached) can be expected to execute a random walk in the space of program lengths rather than in the space of all possible programs. On average very little change in size will be produced by such a random walk whereas a random walk on the landscape shown in Figures 2–4 results on average in rapid motion in the direction of the arrows. Like subtree crossover, both new crossover operators produce offspring that are on average the same size as their parents.

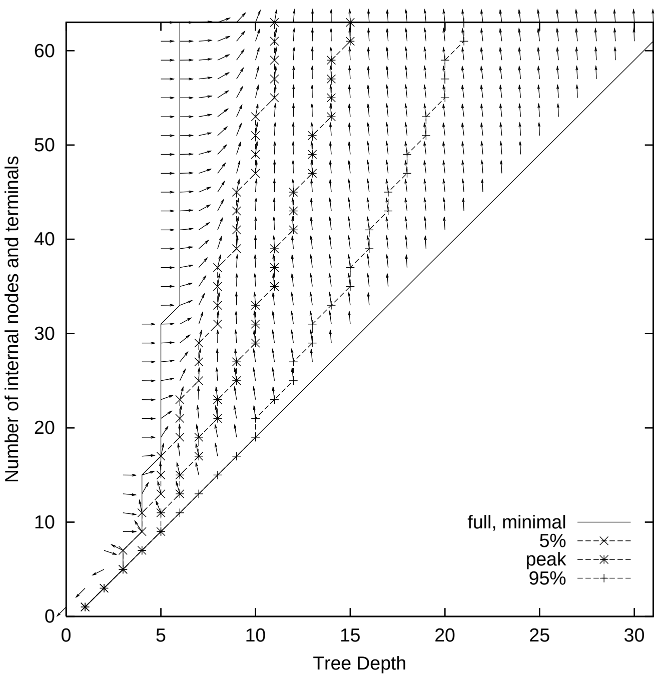

Figure 4. Arrows show gradient in distribution of binary trees by size and height, cf. Figure 2. Note near vertical arrows actually point towards ridge (peak).

3. Size Fair Crossover

In size fair crossover we select two parents and one crossover point in the normal way. (I.e. conduct two independent tournaments, each between seven randomly chosen individuals in the population). The crossover point in the first parent, i.e. the one from which the child inherits its root node, is selected at random from all the nodes in the first parent. We follow standard GP and ensure on average 90% of crossover points are internal nodes, while the remaining 10% (pUnRestrictWt 10) are chosen at random from both terminals (leafs) and functions. As with standard crossover, a crossover point in the other tree is chosen, and the subtree rooted at it, is copied and inserted into (a copy of) the first parent at its crossover point, deleting the subtree that was there originally. The difference between size fair and normal crossover is the choice of the second crossover point.

The size of the subtree to be deleted is calculated and this is used to guide the random choice of the second crossover point. In a single pass across the second parent, the size of every subtree in it is calculated. As with size fair mutation [16, 26] we place a bound on the amount of genetic change in one operation. Subtrees bigger than $1{+}2 \times |\text{subtree to be deleted}|$ are prevented from being inserted into the first parent. (Note each offspring will be no more than $|\text{subtree to be deleted}| + 1$

nodes longer than its first parent. Although apparently generous, removing the upper bound appears to encourage bloat. On average, at the end of 50 runs of the quintic problem, programs were 2.5 times larger when this restriction was removed.) Excluding these big subtrees, we count the number that are shorter (n_-), the same (n_0) and longer (n_+) than the subtree to be deleted. We also calculate the mean size difference for both smaller ($mean_-$) and bigger ($mean_+$) subtrees. If there are no smaller or no bigger trees then the only way to ensure a balance between increasing and decreasing the size of the tree is to not change it, therefore we set the size of the inserted subtree to be equal to that of the subtree to be deleted. Note this means a terminal is always replaced by another terminal. If there are no subtrees in the second parent the same size as the subtree to be deleted, we go back and randomly select a crossover point in the first parent and start again.

If there are both smaller and bigger suitable subtrees then we choose between them all at random using a biased roulette wheel to select the length of the subtree. If there is more than one subtree of the desired length, we choose between them uniformly at random. Thus the chance of a subtree being selected falls in proportion to the number of other subtrees in the second parent of the same size.

The roulette wheel is biased so if there are subtrees of the same size as the subtree to be deleted the chance of choosing one of them is $p_0 = 1/|\text{subtree to be deleted}|$. This somewhat arbitrary choice was made by analogy with conventional subtree crossover where the chance the child is the same size as the parent falls rapidly as the size of the subtree crossed over increases. All the shorter lengths have the same probability of being selected, as do all the longer lengths. However we use the mean size difference to balance these two probabilities so that on average there is no change of length. I.e. the chance of choosing any of the longer sizes is

$$p_+ = \frac{1 - p_0}{n_+(1 + mean_+/mean_-)}$$

4. Homologous Crossover

Standard GP crossover moves code fragments from one program to another. It is assumed that since the code fragment has survived the selection process, it must have some worth and so using it to create a new program is more likely to produce a better program. However it can be anticipated that the worth of a code fragment will depend upon the context within which it is executed. Moving into a different program at a random location may destroy this context [33]. Secondly the presence of bloat may indicate that the code fragment is not good, only that it has survived the selection process by being not harmful. With this in mind several context preserving crossovers have been suggested [7, 34] (and [31] for linear GP). These aim to increase the chance of moving the code fragment to a (syntactically) similar part of the recipient program and thus preserve its context and so worth. Some of these have only had mixed success and so we propose a new homologous crossover operator based on fair crossover described in the previous section.

The homologous crossover operator works identically to the size fair crossover operator up to the last step. Instead of randomly choosing between all the avail-

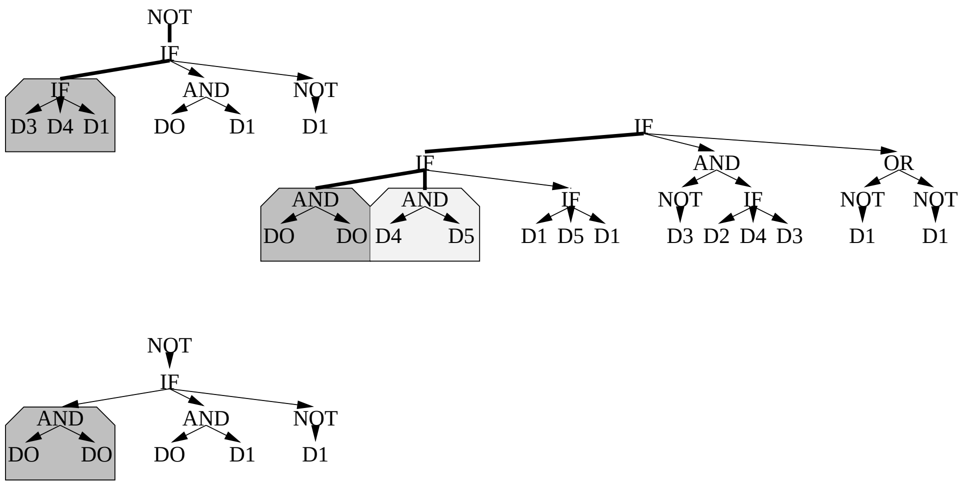

Figure 5. An example of homologous crossover. The shaded subtree (size 4) is chosen in the first parent (top left) to be removed. In the second parent (middle, size 24) all subtrees except the root node and its left argument are eligible to be crossed over. A crossover fragment size 3 is chosen. There are two possibilities (shaded). The left hand one is chosen because the path (heavy lines) connecting it to the root is more similar to the path connecting the subtree to be removed from the first parent than the that for the right hand. The child produced is shown at the bottom.

able subtrees in the second parent of the desired size, in homologous crossover we deterministically choose the one closest to the subtree in the first parent.

Here we define the distance between the two crossover points using only their locations and the shapes of the two trees, i.e. ignoring the functions at each node within the trees. We do this by tracing back up the tree to the root node. The closeness of two points within the trees is given by the depth at which their routes back to the root diverge. See Figure 5. Note homologous crossover on two identical trees will produce an identical offspring if the offspring is of the same size.

5. Ramped Uniform Initialisation

In [38] binary tree populations are shown evolving away from both full or sparse trees. In fact towards the most common tree shape [26]. In this section we describe a new means of creating the initial population in which the population starts with common trees of a range of lengths. We anticipate that such a population will evolve to bigger trees but remain near the most common tree shape (for a given length).

There are enormously more long programs than short ones, so uniform sampling as described by [11] not only ensures almost all the initial population has one of the common shapes but also ensures they are near the maximum possible length. We adopt a more gradualist approach similar to "ramped half-and-half" [13, pages 92–93] and [6], and instead generate a uniform range of program sizes. ([5] provides another initialisation algorithm based upon exact uniform sampling trees of a bounded depth and therefore it predominately generates programs of nearly the maximum depth).

The first stage of our algorithm is to chose uniformly at random a program length between the minimum and maximum allowed and then generate a random program of this length. Thus choosing the program size is a simple procedure and this avoids some of the numerical problems reported by [11] where a more complex procedure is required. The algorithm to generate a random tree of a given length is, like Iba's, based upon Alonso's bijective algorithm [1]. If the functions set contains more than one arity, e.g. the multiplexor function set includes functions which take one, two and three arguments, then, in general, there are multiple combinations of function arities which yield a program of the chosen length. Before Alonso's algorithm can be used one of these must be chosen. Since each combination of arities corresponds to a different number of programs, the random choice is biased in proportion to this number. Tables for each length are precalculated before the GP run starts. Once a random tree has been created it is converted to a random program by labeling its internal nodes with functions of the same arity chosen at random from the function set. Similarly the tree leafs are labeled with terminals chosen at random from the terminal set. (In the case of the two symbolic regression problems on average the input variable x is chosen half the time and one of the constants is chosen the other half). Figure 6 shows ramped uniform produces more programs with shapes near the peak in the search space, while "ramped half-and-half" produces many more large full trees.

Our algorithm is similar to Iba's but is fast and since our implementation is based upon logarithms it is stable even for large trees. (It can readily generate random trees in excess of 1000 nodes even if they contain functions of several arities). It is substantially the same as that given in [15, Appendix A] [2].

6. Experiments

The four benchmark problems are symbolic regression of the quintic polynomial [14] symbolic regression of the sextic polynomial [14] learning the Boolean 6-multiplexer [13, page 187] and the Boolean 11-multiplexer functions [13]. Apart from the use of different crossover operators and different means of creating the initial populations, the absence of size or depth restrictions and the use of tournament selection, our GP runs are essentially the same as [14] and [13]. Parameters are summarised in Tables 1 and 2. We speed up GP on the two Boolean problems by extending the bit packing technique described in [35] to IF. This enabled us to evaluate 32 fitness cases simultaneously.

To test the importance of the initial population we carried out experiments with both the standard "ramped half-and-half" method and also using it to create bigger trees with maximum depths between 5 and 8, corresponding to binary trees up to a length of 255. (In the multiplexor runs maximum depth 8 gives trees up to 3280 in principle although the maximum observed was 611). Duplicate prevention was not used. The range of random program sizes created using the ramped uniform method

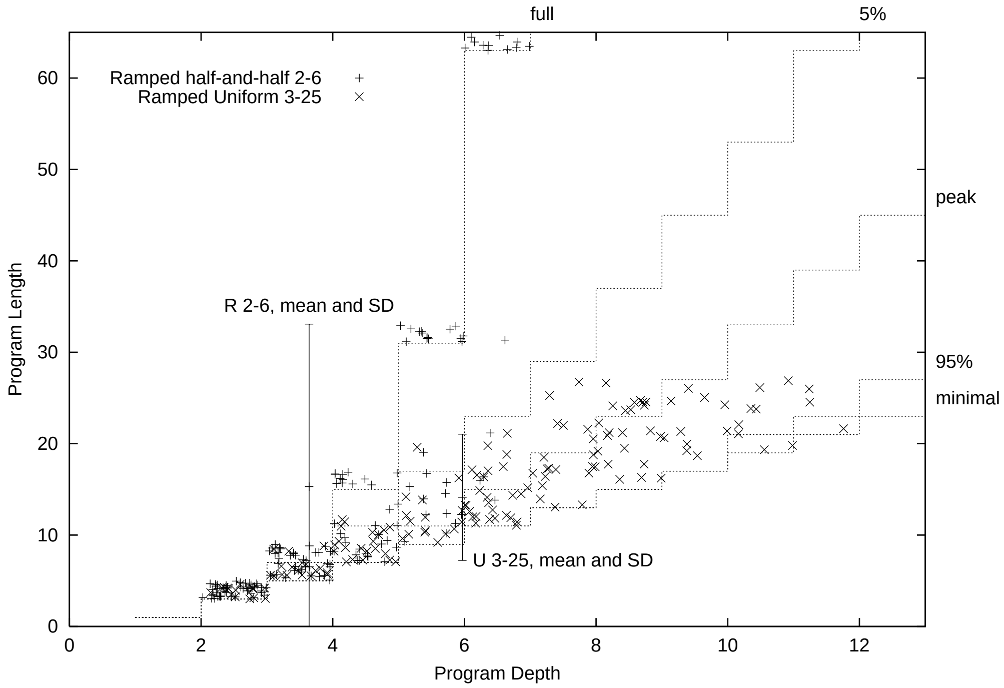

Figure 6. 100 random quintic polynomial program shapes produced by ramped half-and-half (2–6) and ramped uniform. Error bars indicate the means and standard deviations. The full tree and minimal tree limits are shown with dotted lines, as are the most likely shape (peak) and the 5% and 95% limits (which enclose 90% of all programs of a given size). Noise ($\delta x = 0 \ldots 1$, $\delta y = 0 \ldots 2$) has been added to spread data points.

was chosen to have the same minimum size and similar mean size to standard "ramped half-and-half". (Note "ramped half-and-half" produces a small fraction of very big trees; much bigger than the biggest we created using ramped uniform). See Figure 6.

For each of the four problems we performed fifty independent runs for each combination of crossover type and means of creating the initial population. The results of these $4 \times 50 \times 3 \times 3$ runs (about 50 billion fitness evaluations) are summarised in Table 3.

6.1. Evolution of Size

In all 36 cases we see the GP population bloats. (The initial populations start with mean sizes near 14, or 75 for R 5–8). However there is a clear separation between standard crossover and the two new crossovers. In all cases standard crossover produces far bigger trees. (The mean length of programs at the end of the runs is given in column 9 of Table 3. While the last column gives the average size of the biggest program at the end of the run). This is also reflected in the fact that it also

Table 1. GP Parameters for the Symbolic Regression Problems

Objective:	Find a program that produces the given value of the quintic polynomial $x^5 - 2x^3 + x$ (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 $x^5 - 2x^3 + x$ ($x^6 - 2x^4 + x^2$).
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
Pop Size:	4000
Initial pop:	Created using "ramped half-and-half" with depths between 2 and 6, between 5 and 8 or using ramped uniform between 3 and 25 (63). (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 50
Experiments:	50 independent runs performed with each parameter setting

Table 2. GP Parameters for Multiplexor Problems (as Table 1 unless stated)

Objective:	Find a Boolean function whose output is the same as the Boolean 6 (11) multiplexor function
Terminal set:	D0 D1 D2 D3 (D4 D5 D6 D7) A0 A1 (A2)
Functions set:	AND OR IF NOT
Fitness cases:	All the 2^6 or 2^{11} combinations of the 6 (11) Boolean arguments
Fitness:	number of correct answers
Pop size:	500 (4000)
Initial pop:	as Table 1 except ramped uniform between 2 and 25

produces bigger solutions. There isn't such a clear cut difference between fair and homologous crossover.

Figure 7 shows the evolution of program lengths in the population for the quintic symbolic regression problem starting from R 2–6 initial populations. It shows the typical behaviour, where both program size and the spread of sizes in the population in runs using standard crossover grow rapidly and non-linearly. In contrast both fair crossover and homologous crossover show the hoped for reduction in bloat. In both these cases growth in program size is much slower and more linear.

6.2. Evolution of Depth

Figure 8 shows the evolution of program depths in the population for the quintic symbolic regression problem starting from a normal population. It shows the typical behaviour, where both program depth and the spread of depths in the population

Table 3. Results of 50 runs on each crossover and method of creating the initial population. Columns 4–8 refer just to the runs which evolved a solution (num sol). While other data are the means of 50 runs.

Problem	Initiali-sation	Crossover	Num sol	Effort ×1000	Solution size mean	Solution size min–max	End of run size mean	End of run size max	time secs
Quintic	R2–6	stand	39	660	218	15–1205	752	3276	324
	R2–6	fair	38	630	63	15– 153	116	251	92
	R2–6	homo	37	670	61	17– 157	85	162	81
Quintic	R5–8	stand	29	1000	352	27–1871	815	3169	495
	R5–8	fair	32	880	106	27– 337	147	277	146
	R5–8	homo	29	970	77	25– 177	113	213	129
Quintic	U3–25	stand	42	520	337	15–1485	1188	5124	514
	U3–25	fair	39	610	60	15– 145	157	381	96
	U3–25	homo	28	950	50	17– 119	147	354	100
Sextic	R2–6	stand	13	3100	451	53–1209	735	2852	297
	R2–6	fair	7	4400	75	15– 139	119	251	76
	R2–6	homo	9	3900	61	15– 177	105	209	77
Sextic	R5–8	stand	32	920	408	31–1019	919	3415	527
	R5–8	fair	26	1300	116	29– 321	164	307	150
	R5–8	homo	22	1300	88	27– 181	122	219	136
Sextic	U3–63	stand	26	1300	633	61–2037	1332	5446	664
	U3–63	fair	25	1300	123	35– 235	190	408	134
	U3–63	homo	19	1900	107	15– 205	171	360	135
6 Multiplexor	R2–6	stand	39	38	96	15– 275	731	2573	13
	R2–6	fair	47	24	47	10– 160	138	260	5
	R2–6	homo	46	32	42	10– 114	121	236	6
6 Multiplexor	R5–8	stand	45	42	205	34– 845	852	2734	14
	R5–8	fair	47	30	118	36– 324	206	349	6
	R5–8	homo	45	44	110	28– 266	177	308	7
6 Multiplexor	U2–25	stand	33	36	59	12– 435	655	2781	20
	U2–25	fair	26	64	36	14– 104	133	283	8
	U2–25	homo	24	75	35	10– 189	128	277	10
11 Multiplexor	R2–6	stand	37	750	292	57–1344	684	2832	383
	R2–6	fair	49	270	93	35– 228	176	368	138
	R2–6	homo	47	290	79	25– 207	156	338	133
11 Multiplexor	R5–8	stand	10	4100	439	223– 894	679	2349	532
	R5–8	fair	43	540	212	83– 452	248	481	220
	R5–8	homo	32	960	221	77– 504	244	463	221
11 Multiplexor	U2–25	stand	36	680	251	90– 896	686	3172	392
	U2–25	fair	18	1400	86	50– 116	179	399	140
	U2–25	homo	24	930	86	53– 142	173	390	141

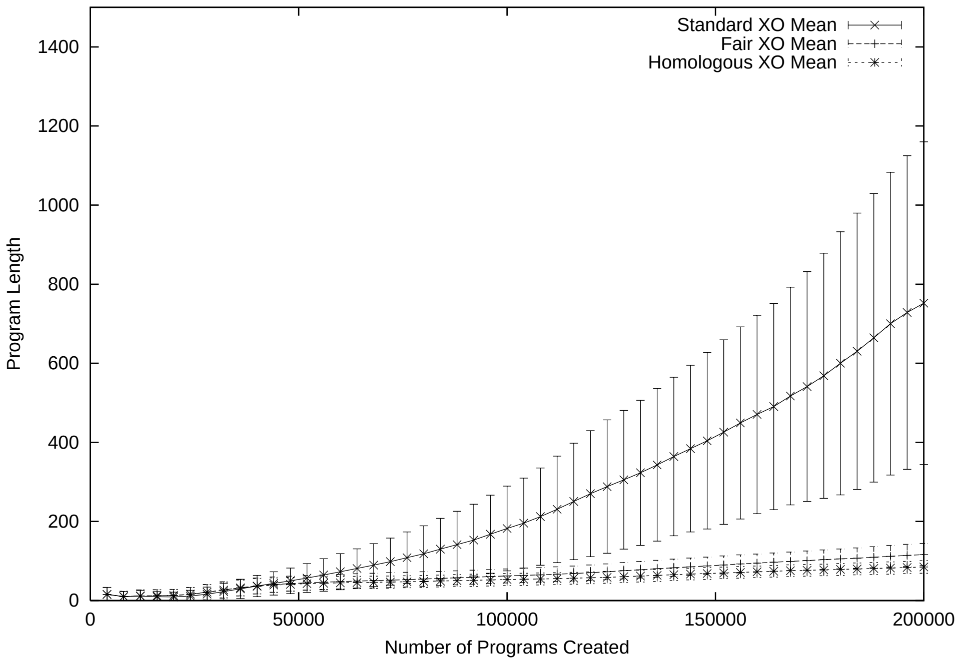

Figure 7. Evolution of population program length from R 2–6 populations. Error bars indicate standard deviation in population. Means of 50 runs of quintic polynomial problem.

in runs using standard crossover grow rapidly but apparently linearly. Over the last 3/4 of the run the mean growth is 1.2 layers per generation. Which greatly exceeds 0.2 measured in both fair and homologous crossover runs over the same period.

Figure 9 shows the evolution of program depths for each our four problems and each of the three methods of creating the initial population. It is evident that the linear growth in program mean depth is not a fluke but may be an important property of standard subtree crossover (in the absence of depth or size limits). Table 4 gives the mean and max program depths and their average rate of increase over the last 38 generations of the runs. While not problem independent, Table 4 shows the rate of increase in depth is consistently close to unity.

6.3. Evolution of Shape

Figure 10 shows the evolution of program depth compared to size in ten of the 50 standard crossover quintic populations, shown in Figure 7 and 8. Figure 10 shows for the quintic problem, GP population behave much as they do for other problems [26], with programs tending both to grow bigger and deeper but also tending to be near the combination of size and depth for which there are most programs.

Figure 11 shows the evolution program depth compared to size for the three crossover operations. For clarity only the average behaviour of each group of 50

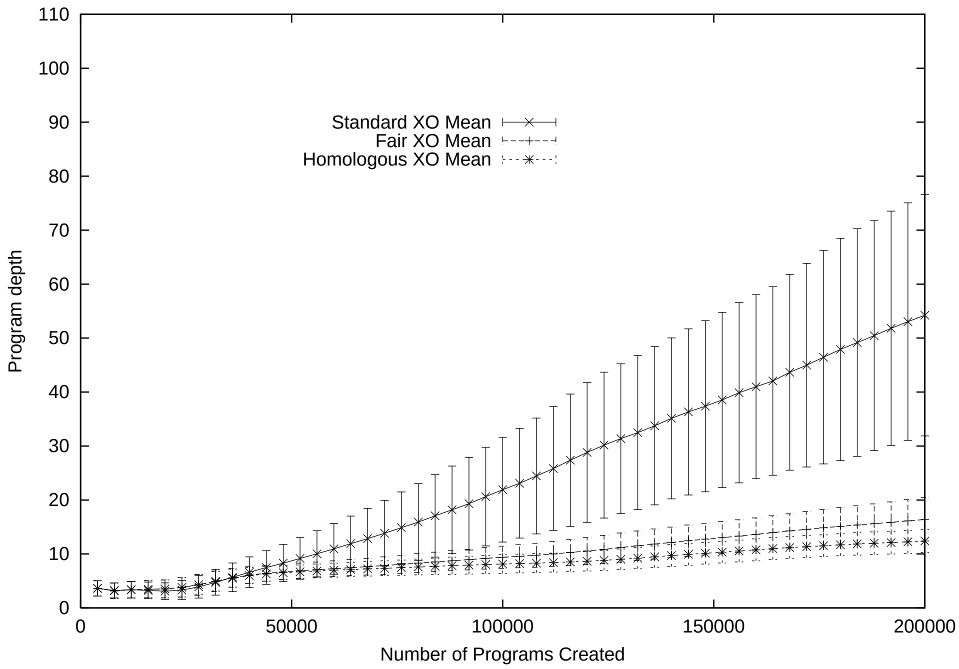

Figure 8. Evolution of population program depth. Error bars indicate standard deviation in population. Means of 50 quintic polynomial runs.

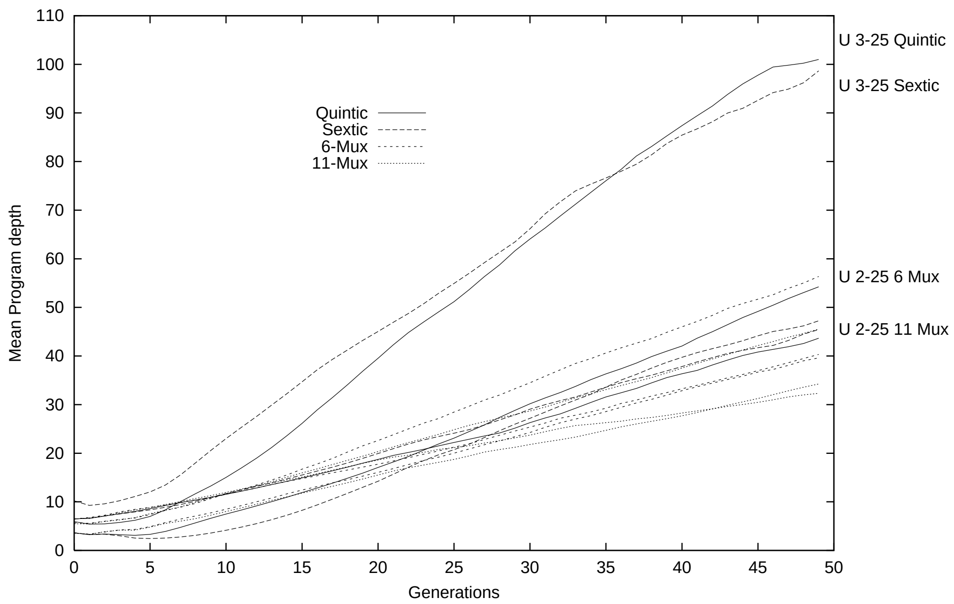

Figure 9. Evolution of population program depth. Means of 50 runs with standard crossover for each problem and initial populations.

Table 4. Program Depth, standard crossover 50 runs

Problem	Initialisation	Final pop		Growth per generation (average and standard deviation of last 38)	
		mean	max	mean	max
Quintic	R2–6	54	181	1.2 (.5)	4.0 (2.4)
	R5–8	43	128	0.8 (.4)	2.4 (1.2)
	U 3–25	101	332	2.2 (1.4)	7.0 (4.4)
Sextic	R2–6	47	150	1.1 (.6)	3.5 (2.2)
	R5–8	45	131	0.9 (.4)	2.5 (1.0)
	U 3–63	98	312	1.9 (.7)	5.8 (2.9)
6 Multiplexor	R2–6	39	101	0.8 (.3)	2.1 (.8)
	R5–8	40	97	0.7 (.2)	1.9 (.9)
	U 2–25	56	172	1.2 (.4)	3.6 (1.5)
11 Multiplexor	R2–6	34	107	0.7 (.2)	2.1 (.7)
	R5–8	32	90	0.5 (.3)	1.4 (1.0)
	U 2–25	45	157	0.9 (.2)	2.9 (1.0)

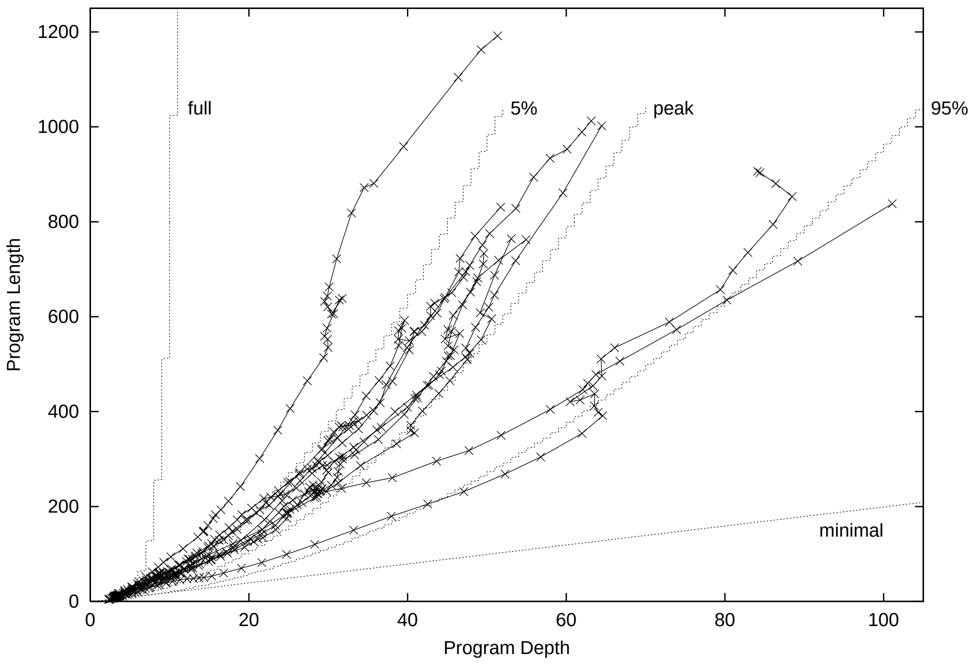

Figure 10. Evolution of mean population program shape, tick marks every generation. The full tree and minimal tree limits are shown with dotted lines, as are the most likely shape (peak) and the 5% and 95% limits (which enclose 90% of all programs of a given size). The first 10 standard crossover runs of quintic polynomial problem.

runs is plotted. We see both fair ($\times$) and homologous ($\square$) crossover producing trees of similar shapes as standard crossover (+) (again near the peak number of programs) but moving much more slowly along the same trajectory. (Because the average size of programs is a non-linear function of their depth, large programs have a disproportionate effect on the arithmetic mean leading to the population mean depth v. size appearing to be initially outside the range of feasible trees).

Figure 12 shows the evolution of all 450 initial populations used in the quintic polynomial problem. For clarity only the mean of each group of fifty runs is plotted. As shown in [26, 38] for very different problems standard crossover evolves the population towards the peak in the distribution of programs versus their shape. However like [38] the population retains a long term memory of how it was initialised and the mean evolutionary curves do not coalesce. This is consistent with the view that on average populations follow close to the steepest gradient in the density of programs. Figure 12 also plots (using black squares) the average of each of 50 runs using standard crossover but without the customary bias to chose functions as crossover points rather than terminals (pUnRestrictWt 100). Figure 12 also shows the evolution of populations initialised with sparse trees of depths between 2 and 6. It is clear that all populations using subtree crossover evolve towards the ridge. However the normal crossover bias to chose functions rather than leafs appears

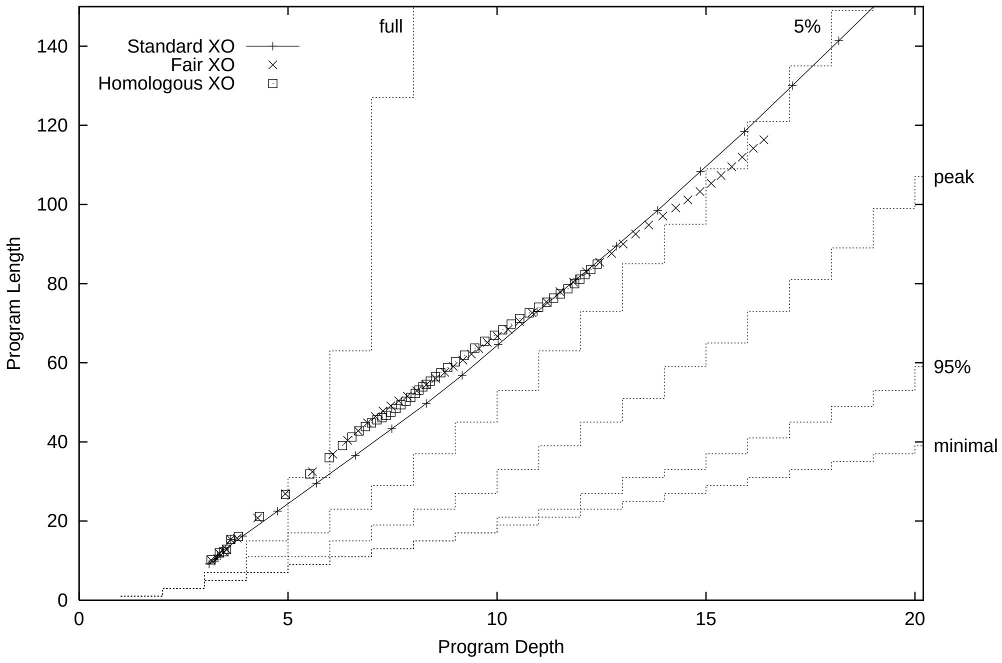

Figure 11. Evolution of mean population program shape from R 2–6 initial populations. Tick marks every generation. Means of 50 runs of quintic polynomial problem.

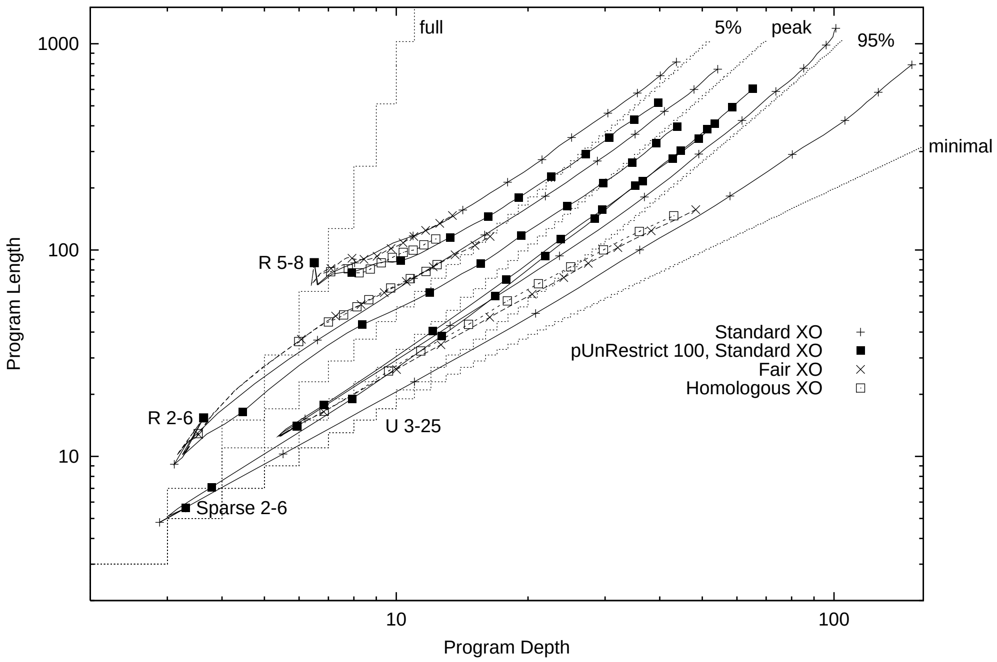

Figure 12. Evolution of mean population program shape showing effect of four types of initial populations. Ramped half-and-half 2–6, ramped half-and-half 5–8, ramped uniform and sparse. Tick marks every 5 generations. Means of 50 of quintic polynomial runs. Note log scales.

to bias evolution slightly, so fuller trees remain on average on the fuller side of the ridge and sparse trees remain to the sparse side. Without this bias all initial populations appear to grow to become close to the ridge. However the ridge in the distribution of program versus their shape is quite wide and the population mean in individual runs wonders considerably either side of it as shown for example in Figure 10. (It is not statistically significant that the R 2-6 subtree crossover plot without bias (pUnRestrictWt 100) crosses to left of the ridge).

Again we see fair and homologous runs show much reduced bloat (the tick marks every five generations are much closer together) and lie close to each other. However both runs with bigger initial populations and those produced by ramped uniform deviate from the mean shape followed by standard crossover runs and create deeper trees. This may be because, while size change is carefully controlled, no specific restrictions are placed on depth exploration, allowing the population to move more freely in this direction. Future genetic operators might consider this aspect of bloat too.

6.4. *Sub-quadratic Bloat*

As discussed in [26] and Section 2, if the programs within the population remain close to the ridge in the number of programs versus their shape and they increase their depth at a constant rate this leads to a prediction of sub-quadratic growth in their lengths'. (A best fit of the ridge for sizes between 50 and 500 gives a power law exponent of 1.3 (see dotted curve in Figure 13). Thus we expect for modest size programs size $O(\text{gens}^{1.3})$, rising to a limit of quadratic growth for $|\text{program}| \gg 1000$, cf. [8, Table II]. Over the last 38 generations the mean measured values are near $O(\text{gens}^{1.3})$ for the quintic and sextic problems (which are solved with binary trees) see Table 5. (The multiplexor problems have more complex trees and so the distribution of number of programs v. their shape differs in detail).

It is clear that our simple model works reasonably well on average. There are several reasons why the fit can not be exact.

1. The distribution of programs can only be approximately described by a power law. The exponent obtained by fitting a power law curve to the ridge varies slowly according to which part of the curve we try and fit. As bigger, deeper parts of the curve are fitted the exponent rises. Thus even if our crude model was exactly correct the measured exponent would vary according to how big the programs in the population were.

2. Individual runs differ from the average behaviour. For example Figure 13 shows the evolution of the population mean statistics in one quintic run plus the best fit obtained by linear regression of log size v. generation. While the mean depth (dashed line) increases approximately linearly over the last part of the run, the population remains somewhat bushier than the ridge in the program shape distribution. If depth increased exactly linearly and the average shape (dotted line with +) coincided with the power law prediction (dotted line) exactly then bloat (solid line with +) would fit a power law of $\text{time}^{1.33}$. The best power law fit over the last 38 generations (solid line) suggests size $\propto \text{time}^{1.2}$ however the mean figure for 50 runs is 1.31 (cf. Table 5). I.e. our simple model gives an indication of bloat in individual runs and works better as a predictor of average behaviour across many runs.

Initial experiments extending evolution to many hundreds of generations and fitting power laws to the bloated program sizes show the exponent does indeed increase towards 2.0. Standard crossover ceases to be effective in producing changes to large programs ($> 100,000$ functions) behaviour and selection pressure falls. This may explain why the predicted limit of quadratic increase in program size with number of generations is not reached.

6.5. *Search Efficiency*

As shown in Table 3 in all four problems most of the nine experiments have similar search efficiency in terms of number of solutions found or "effort" [13, page 194].

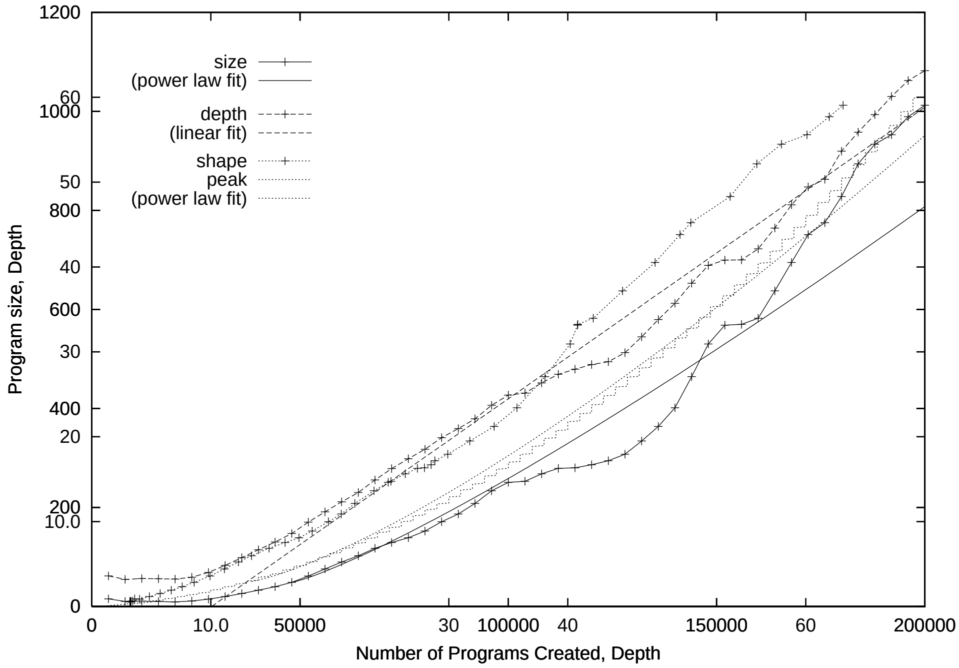

Figure 13. Evolution of population mean statistics for one run quintic run with standard crossover from a standard population. Solid line (size) and dash line (depth) are plotted against time (horizontal) while the dotted lines show size v. depth. Lines without crosses (shown every generation) are Size $= a + b \times \text{gens}^{1.2}$, depth $= c + 1.37 \times \text{gens}$, the ridge in the distribution of binary trees (steps) and power law fit to it, size O(depth$^{1.33}$).

Even with 50 runs, there are two cases where the difference can be thought statistically reliable, even though in others the differences may be large. 1) all three crossover operators perform slightly better with the two new means of creating the initial random programs in the sextic polynomial and 2) in the 11-multiplexor problem standard crossover performs slightly worse on large initial programs. I.e. the new operators perform at least as well as the original.

[18] shows the density of programs of a given fitness level will be much the same in most of the search space. The similarity of the search efficiency of the operators suggests that although they are searching different parts of the search space, these different regions are similar. I.e. not only are their fitness levels similar but so too are the adjacency links in the fitness landscapes provided by the various crossover operators.

6.6. Homologous Measurements

It is disappointing that homologous crossover shows little performance gain over fair crossover. In this section we investigate why this might be. We expect the

Table 5. Power law fit of mean program size in population over last 38 generations v. generation. Means (and standard deviations) of 50 standard crossover runs.

Problem	Initialisation	Exponent
Quintic	R2–6	1.34 (.28)
Quintic	R5–8	1.32 (.35)
Quintic	U 3–25	1.31 (.21)
Sextic	R2–6	1.49 (.37)
Sextic	R5–8	1.27 (.30)
Sextic	U 3–63	1.20 (.18)
6 Multiplexor	R2–6	1.28 (.21)
6 Multiplexor	R5–8	1.34 (.34)
6 Multiplexor	U 2–25	1.32 (.20)
11 Multiplexor	R2–6	1.24 (.16)
11 Multiplexor	R5–8	1.17 (.26)
11 Multiplexor	U 2–25	1.26 (.15)

use of homologous crossover to increase the convergence of the GP populations. In particular, in the multiplexor runs, we would hope to see common trees evolving with combinations of address bits as the first arguments of `IF` functions and data bits as the second and third arguments. Using population variety and number of duplicate children produced we do see a little evidence for some convergence but these are crude measures and the degree of commonality in the population may be higher than they indicate. (They say two trees are different even if the difference is small or they differ only in inviable code). However, if this higher level of convergence does exists, it doesn't appear to impact the spread of fitness values. E.g. the spread of performance in the final populations, as measure by the standard deviation in fitness, is not markedly different between homologous and fair crossovers.

A possible explanation for the similarity in the results produced by size fair and homologous crossovers might have been that the "homologous" aspect was not operating, i.e. homologous crossover was not making a directed choice of second crossover point (where size fair was making a random choice) because it had no choice. So we measured how often this happened. In the 11 multiplexor runs the homologous aspect influenced the outcome of crossover in 54% (uniform), 63% (half-and-half 2–6), 78% (R 5–8) of the time. (The variation is probably accounted for by the variation in program bloat between these three cases). However looking in more detail at the first 11 multiplexor run, while in most crossovers the homologous aspect was active, in only half of these did it produce a different offspring to that which size fair would have produced and only a fraction of these had a different fitness. In fact only about 10-15% of the population behaved differently because of the directed choice of crossover points.

These results may indicate (in these problems and representations):

1. subtrees have no particular value
2. the value is destroyed outside their particular context
3. the simple syntax rule used doesn't describe the context well
4. even good subtrees are little help in improving programs to solve additional fitness cases.

7. Discussion

The impressive suppression of bloat produced by size fair and homologous crossovers was expected as it concurs with our theory of bloat [26] and similar results for fair mutation. While they are designed with a view to reducing bloat by carefully controlling how the search space is sampled (i.e. by sampling programs of neighbouring lengths) an alternative view of their success, is by closely correlating the size of the inserted subtree with that of the removed they suppress the "removal bias" [40] bloat mechanism and remaining bloat is due to some other mechanism probably inviable code [16].

Some of the reduced rate of growth derives from the upper bound on the size of the replacement subtree in both cases. When this bound is removed larger programs are evolved. To ensure size increases and decreases produced by size fair crossover are balanced, bigger increasing crossovers will also mean more size reducing crossovers. Thus the increased bloat from removing the limit indicates that big increases in size have a disproportionate effect.

In cases where the initial population is much smaller than the necessary size of the solution it may be growth restricting crossover or mutation operators will not perform as well as subtree crossover. If the minimum size of the solutions is known or can be estimated in advance, then it may be an advantage to start with an initial population containing programs of approximately this size. On the other hand if this is not known in advance, and a small initial population is created, then rapid growth in size may help. If such growth is driven by fitness based selection it is not what we normally call bloat (which by definition is not driven by fitness). While the maximum rate of program size increase is more restricted in size fair crossover and mutation and homologous subtree crossover than in normal subtree crossover, they do allow growth in program size at much higher rates than are observed in bloating populations. Hence, while they may be at a disadvantage in these circumstances, the difference need not be large. (As an illustration: suppose all solutions are eight times bigger than the initial population. When two programs of the same size are crossed over using subtree crossover the biggest possible child is twice as big as its parents. But if size fair is used it is only 50% bigger. Thus in a big population it will take at least 3 generations for subtree crossover to create a program of the required size, while subtree sized will need at least 5.1 generations. Whether a 2.1 generations head start is important depends upon what happens after a program of the required size has been created. We suspect that in general in problems requiring such large solutions, a difference of 2.1 generations will be unimportant. Admittedly this is somewhat artificial example.)

The linear growth in mean depth of near one level per generation gives a simple problem independent prediction of when a population will be severely affected by a depth limit. In particular we predict that standard GP starting from an initial population created with the usual "ramped half-and-half" parameters will run into the common depth limit of 17 levels in about 12 generations [17].

Using the curve indicating the peak in the distribution of programs against their size and shape, a predicted depth can be converted into a predicted program size. The curve is known for programs with only two inputs [37] and can be precalculated for more complex function sets. Thus we predict that generally standard GP will run into common size limits (which can be as low as 50 or 200 nodes), within a few generations and certainly before the 50 generations commonly use, see [17].

8. Conclusions

We have presented and demonstrated on four benchmark problems a new bloat reduced crossover operator, a new homologous crossover operator and a new mechanism for creating random populations for tree based genetic programming. The results in terms of reduction in growth of both mean and maximum program and solution sizes are impressive and are achieved without reduction in search efficiency.

While we have demonstrated the homologous crossover operator is effective at finding solutions and reducing bloat, we have not yet shown it to be greatly more efficient. Growth in program sizes was found not to depend overly on the initial population. However it does have a dominant role in the evolution of program shapes. The ridge in the distribution of number of programs for each size and shape acts to divide the search space. "Ramped half-and-half" does not search a large part of the search space corresponding to long thin trees (and vice-versa an initial population of long thin trees does not search the part of the search space corresponding to short bushy trees).

Average growth in program depth when using standard subtree crossover is near linear in these problems. When combined with the known distribution of number of programs of any given size and depth, this yields a prediction of subquadratic growth in program size. This indicates GP populations using standard crossover (and no parsimony techniques) will quickly reach bounds on size or depth commonly used.

Acknowledgments

I would like to thank Paul Vitanyi and Andrei Kotlov for helpful suggestions and references concerning the distribution of random trees.

Notes

1. C++ code to calculate the number of different programs can be found in `ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code/ntrees.cc`.
2. C++ code to generate programs uniformly at random can be found in `ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code/rand_tree.cc`.

References

1. Laurent Alonso and Rene Schott. *Random Generation of Trees.* Kluwer Academic Publishers, Boston, MA, USA, 1995.
2. 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.
3. Tobias Blickle. Evolving compact solutions in genetic programming: A case study. 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 564–573, Berlin, Germany, 22-26 September 1996. Springer-Verlag.
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. Walter Bohm and Andreas Geyer-Schulz. Exact uniform initialization for genetic programming. In Richard K. Belew and Michael Vose, editors, *Foundations of Genetic Algorithms IV*, pages 379–407, University of San Diego, CA, USA, 3–5 August 1996. Morgan Kaufmann.
6. Kumar Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, September 1997.
7. Patrik D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.
8. Philippe Flajolet and Andrew Oldyzko. The average height of binary trees and other simple trees. *Journal of Computer and System Sciences*, 25:171–213, 1982.
9. Chris Gathercole and Peter Ross. An adverse interaction between crossover and restricted tree depth 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 291–296, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
10. Dale C. Hooper, Nicholas S. Flann, and Stephanie R. Fuller. Recombinative hill-climbing: A stronger search method for genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 174–179, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
11. Hitoshi Iba. Random tree generation for genetic programming. 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 144–153, Berlin, Germany, 22-26 September 1996. Springer Verlag.
12. Hitoshi Iba, Hugo de Garis, and Taisuke Sato. Genetic programming using a minimum description length principle. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press, 1994.
13. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.
14. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

15. William 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.
16. William B. Langdon. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.
17. William B. Langdon. Linear increase in tree height leads to sub-quadratic bloat. In Thomas Haynes, William B. Langdon, Una-May O'Reilly, Riccardo Poli, and Justinian Rosca, editors, *Foundations of Genetic Programming*, Orlando, Florida, USA, 13 July 1999.
18. William B. Langdon. Scaling of program tree fitness spaces. *Evolutionary Computation*, 7(4), 1999.
19. William B. Langdon. Size fair and homologous tree genetic programming crossovers. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1092–1097, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
20. William B. Langdon and Riccardo Poli. An analysis of the MAX problem in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 222–230, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
21. William B. Langdon and Riccardo Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997.
22. William B. Langdon and Riccardo Poli. Fitness causes bloat: Mutation. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 37–48, Paris, 14-15 April 1998. Springer-Verlag.
23. William B. Langdon and Riccardo Poli. Genetic programming bloat with dynamic fitness. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 96–112, Paris, 14-15 April 1998. Springer-Verlag.
24. William B. Langdon and Riccardo Poli. Boolean functions fitness spaces. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 1–14, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.
25. William B. Langdon. *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!* Kluwer, Boston, 1998.
26. William B. Langdon, Terry Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.
27. 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.
28. Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Introns in nature and in simulated structure evolution. In Dan Lundh, Bjorn Olsson, and Ajit Narayanan, editors, *Bio-Computation and Emergent Computation*, Skovde, Sweden, 1-2 September 1997. World Scientific Publishing.
29. Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.
30. 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.
31. Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector,

William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

32. 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.
33. Una-May O'Reilly and Franz Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 73–88, Estes Park, Colorado, USA, 31 July–2 August 1994 1995. Morgan Kaufmann.
34. Riccardo Poli and William B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.
35. Riccardo Poli and William B. Langdon. Sub-machine-code genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 13, pages 301–323. MIT Press, Cambridge, MA, USA, June 1999.
36. Justinian P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
37. Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, 1996.
38. Terence Soule. *Code Growth in Genetic Programming.* PhD thesis, University of Idaho, Moscow, Idaho, USA, 15 May 1998.
39. Terence Soule and James A. Foster. Code size and depth flows in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 313–320, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
40. Terence Soule and James A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.
41. 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.
42. Byoung-Tak Zhang and Heinz Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, 1995.