PPSN VI, Sixth International Conference on Parallel Problem Solving from Nature, Marc Schoenauer, editor, Paris, 16-20 September 2000, LNCS 1917, Springer-Verlag

Genetic Programming Bloat without Semantics

W.B. Langdon[1] and W. Banzhaf[2]

[1] Computer Science, University College, London, Gower Street, London, WC1E 6BT
[2] Department of Computer Science, University of Dortmund, D-44221 Dortmund

Abstract. To investigate the fundamental causes of bloat, six artificial random binary tree search spaces are presented. Fitness is given by program syntax (the genetic programming genotype). GP populations are evolved on both random problems and problems with "building blocks". These are compared to problems with explicit ineffective code (introns, junk code, inviable code). Our results suggest the entropy random walk explanation of bloat remains viable. The hard building block problem might be used in further studies, e.g. of standard subtree crossover.

1 Introduction

It has been suggested that ineffective code (introns, junk code) is essential for bloat found in genetic programming (GP). An alternative interpretation is based on random expansion into the accessible parts of the search space. To investigate the differences between these a series of artificial random binary tree search spaces are presented. These avoid the complications associated with the semantics of the programming language by defining fitness directly from the syntax of each program (the GP genotype). In order to avoid introducing hidden assumptions, the remaining linkage within the search spaces is random.

A total of six parameterisable problems are investigated: random and two variations where fitness is composed of "building blocks" (hard and easy variants). For each of these a second version is obtained by explicit introduction of syntax elements which do not change the fitness of a program containing them.

In the next section we give the background and indicate why studies of bloat remain important and then Sect. 3 summarises theories about bloat. Section 4 describes the six artificial problems. GP is evaluated on them in Sect. 5. Finally Sect. 6 draws some conclusions for automatic programming.

2 Background

Many researchers have investigated the importance of crossover in genetic programming (GP), particularly its effectiveness compared to mutation. Mostly these have been empirical studies [27, 10, 2, 25, 7],[14, Chapter 56]. While this may be viewed as a GP issue, there is a crossover-related controversy in bit string genetic algorithms (GAs). Proponents of GAs argue that their effectiveness in real problems comes from their use of genetic crossover in a population of trial solutions. They argue crossover and selection would allow improved individuals to be created from components of the better samples of the search space that have already been tested. Note the assumption that in soluble real problems better (or ideally the best) solutions can be constructed from parts, and that the parts themselves are good (or at least better than average). Further it is assumed that solutions to the problem have been represented by the GA user in such a way that crossover is capable of doing the assembly at random. This is the well known building-block hypothesis. Even in simple fixed representation GAs this remains disputed, although there is a growing body of theory about building blocks in GAs.

From the empirical studies of GP it has been known for some time that programs within GP populations tend to rapidly increase in size as the population evolves [13, 1, 33, 4, 26, 16, 32, 24]. 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 [13, 12, 34, 29] although other techniques have been proposed [30, 6, 32, 31, 18]. Depth or size limits [9, 20] and simple parsimony pressure [32] may have unexpected and untoward effects, while [11] shows that addition of duplicated code segments (i.e. addition of ineffective code, code that has no impact on the behaviour of the program) can sometimes be helpful. Therefore it is still interesting to explore why such bloat happens, and the factors influencing its speed and its limits.

3 Bloating

Tackett [33, page 112] and Altenberg [1] both suggest the common "intron" explanation for bloat is due to Singleton (however James Rice and Peter Angeline may also have contributed). Briefly this says program size tends to increase over time because programs which are bigger contain more ineffective code ("junk" code, code that has no effect on the program). Since changes to ineffective code have no impact on the execution of the program, a higher proportion of ineffective code means a higher chance programs produced by crossover will act like their parents. Therefore they will also be of high fitness and so themselves have a higher chance of being selected to reproduce. Various experiments have shown this to be essentially correct. However Soule [24] shows that there are at least two mechanisms involved. It needs to be noted that this implicitly assumes that the problem is static, i.e. behaving as your parents (who must have been good to have been selected to have children) will also be good for you. Yet bloat has also been observed in dynamic problems [23]. An alternative suggestion [3] that ineffective code could act as safe storage areas for code which is not needed at present but may be needed in future has received only little experimental support [11].

While the ineffective code mechanism is essentially correct one of us (W.B.L.) has proposed a simpler alternative explanation which is independent of mechanisms and indeed has been applied to non-GP search: After a period GP (or any other stochastic search technique) will find it difficult to improve on the

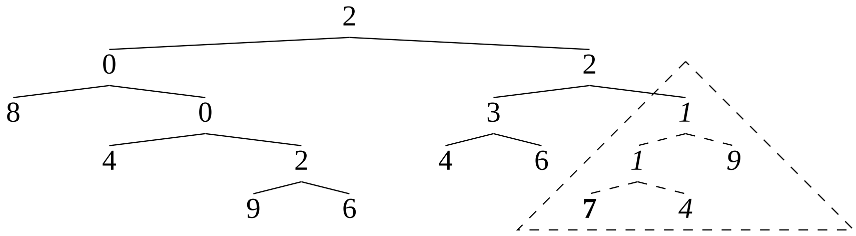

Fig. 1. To calculate fitness of this tree, it is traversed depth first, i.e. 208042962, 34611749. The first nine opcodes are packed into one integer and the rest into a second. These are combined using XOR, 208042962 $\overline{\vee}$ 3461749 = $C667BD2_{16}$ $\overline{\vee}$ 2102225_{16} = $E7659F7_{16}$ = 242637303. 242637303 is the hash value of the tree. 242637303 is randomised by Park-Miller to yield $65D2E653_{16}$ which has 17 bits set and so the fitness of the tree is 17. If ineffective code is enabled all subtrees labelled with "1" are replaced by their first terminal. So 208042962, 34611749 becomes 208042962, 3467. XOR becomes $C667BD2_{16}$ $\overline{\vee}$ $D8B_{16}$ = $C667659_{16}$ and randomising yields $4AA4B33_{16}$, which has 13 bits set.

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 active only those that are as good as the best-so-far. In the absence of bias, the more plentiful programs with the current level of performance are more likely to be found [21]. But as [17] proves the distribution of these is similar to the distribution of trees, therefore we expect the search to evolve in the direction of the most popular tree shape. I.e. trees whose depth lies near $2\sqrt{\pi(\text{internal nodes})}$ (ignoring terms $O(N^{1/4})$) [8]. See Flajolet parabola curve in (lower right graph in) Figs. 3 and 4. [24] and [17] confirm this in various problems when using GP with standard crossover. The simplicity of this explanation makes it easy to apply to non-GP search [15, 22], to devise new genetic operators [18], to explain the evolution of shape [24] and to produce quantitative predictions of the evolution of both size and shape [19].

In [5] we consider bloat in abstract representation-less random search spaces and highlight the importance of inviable code. In the next section we extend this approach to more concrete but still artificial representations.

4 Artificial Binary Tree Search Spaces

To repeatably assign a fitness value to a program based only on its syntax, the syntax is first reduced to a single hash value. This is deterministically converted into a random fitness value. Hashing techniques are used to ensure similar programs have different hash values and so different fitnesses.

Ineffective code is introduced by a special function within the function set (opcode 1). When calculating fitness, subtrees starting with opcode 1 are treated as if they had been replaced by their first leaf, i.e. the rest of the subtree is ignored (see dashed subtree in Fig. 1). This means ineffective code always lies away from the root and towards the leafs, as is usually the case in GP [32].

4.1 Random Binary Tree Search Spaces

The programs are composed of four functions, opcodes $0 \ldots 3$, and six terminals (leafs), opcodes $4 \ldots 9$ (cf. Fig. 1). To hash the program tree it is traversed in conventional depth-first order. At each node the opcode is packed into an integer. When the 31 bit integer (the sign bit is not used) is full a new integer is used. (Since $10^9 \leq 2^{31}$ nine syntax elements can be packed into each integer.) Successive 31 bit values are combined using XOR. Thus the whole tree is reduced in a single pass to an integer value. This is converted to a random fitness by adding a large constant and feeding the sum into a pseudo-random number generator [28]. Alternative problems can be generated by changing the constant (As a confidence check, many runs were repeated replacing Park-Miller with Mersenne Twister. No important differences were noted). The fitness is the number of bits set $(1 \ldots 31)$ in the random number produced.

4.2 Random Binary Sub-Tree Building Block Search Spaces

In the building block (BB) problems we define fitness as the combined fitness of the building blocks within the program. Every subtree within the program is treated as a building block and given a random fitness using the mechanisms described above in Sects. 4 and 4.1. (With a suitable stack, fitness can still be calculated in a single pass through the tree.) To simulate idealised building blocks, the randomised value of each subtree is converted to a bit location $(0 \ldots 31)$ which is set. The behaviour of the whole tree is the union of these bits and its fitness is the number of bits set.

In the easy problem each bit is equally likely to be set, so finding a tree which sets all 32 is relatively easy. Each subtree's bit is given by the least significant five bits of its randomised value.

The more difficult case is where some bits are much more likely to be set than others. Each subtree's bit is now calculated by counting the number of set bits in the random value of the subtree. This gives a binomial distribution $(1 \ldots 31)$ centered in the middle of the word. Setting the bits far from the middle is very rare and achieving maximum fitness (31) is very difficult.

5 Experiments

On each search space we ran 10 independent GP runs for each of two or four different ways of creating the initial population. Apart from the search space, the absence of size or depth restrictions and the use of tournament selection the GP runs are essentially the same as [13]. Table 1 gives parameters.

The average evolution of each GP population from initial trees of two different ranges of sizes (r2:6 and r7:8) on the six problems (random, easy and hard building blocks, with and without explicit ineffective code) is plotted in Figs. 2–4 and summarised in Table 2 (numbers in round brackets indicate the standard deviation across ten runs).

Table 1. GP Parameters for Random Search Problems

Functions set:	b0 b1 (intron) b2 b3
Terminal set:	t4 t5 t6 t7 t8 t9
Fitness:	Determined by tree syntax
Selection:	Tournament group size of 7, non-elitist, generational
Wrapper:	none
Pop Size:	500
Max program:	no limit
Initial pop:	Created using "ramped half-and-half" with depths between 2 and 6 (r2:6), 7 and 8 (r7:8), ramped uniform (u7:55) or ramped sparse depths 4 and 28 (i.e. size s7:55) (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:	50 generations

As expected, all but the easy building block landscape prove difficult for GP, and programs of the maximum possible fitness are only found in the easy landscapes, Fig. 3. The presence of explicit ineffective code in the search space makes little difference in the best of run fitness but, particularly in the random landscape (Fig. 2, top left), it does reduce disruption by crossover so increasing average fitness (plotted with lines) in the population. Also the size of the initial programs makes little difference to the overall behaviour but the shape of the initial programs is important. In all but the random landscape, bloat occurs.

5.1 Anti-Bloat in Random Binary Tree Landscape

Only in the case of the totally random landscape do we not see bloat. Here runs started with both normal and large random trees converge to tiny trees whose parents are identical. (Similar convergence is reported in [19],[16, page 184]). This is explained by the difficulty of the search, so in all most all generations no improved programs are found. Non-elitist selection means improved solutions are removed from the population at the end of the generation. However they have children but they are in competition with each other as well as the rest of the population. Thus only genetic lines which reproduce themselves fastest continue. Most children are produced by subtree crossover. Therefore programs which crossover is more likely to reproduce exactly (clone) have an advantage. The chance subtree crossover between two identical parents producing a clone falls as the parent trees get bigger. This means smaller programs have an advantage (when crossover cannot find either better programs or programs of the same fitness which are not identical to their parents) [16, pages 197–201]. An equilibrium is reached between the local optima and its unfit offspring. These equilibria are stable for at least 1000 generations.

5.2 Evolution of Depth

With bushy initial trees (r2:6 and r7:8) and except for both the easy landscapes, average program height increases roughly linearly by about one level per gen-

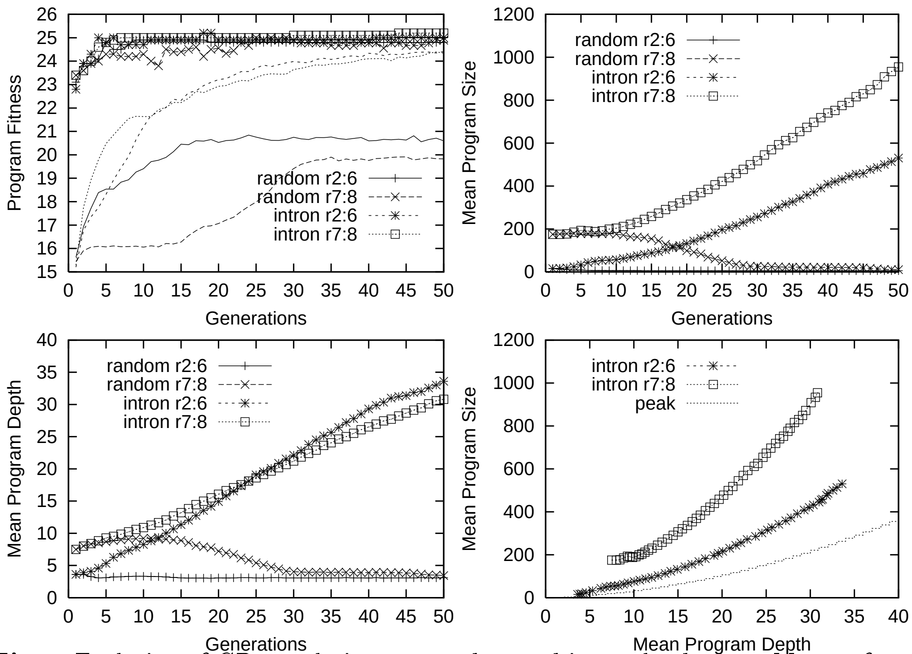

Fig. 2. Evolution of GP populations on random and intron landscapes. Means of ten runs with standard (r2:6) and large (r7:8) initial populations

Table 2. Mean of 10 runs on each landscape and method of creating the initial population. Power law fit of mean program size in population over last 38 gens. v. gen.

Problem	Initiali-sation	Fitness		Final population mean size		max size		Depth mean	per gen		Power law Exponent	
Random	R2–6	25	(1)	6	(0.9)	11	(3)	3				
	R7–8	25	(1)	10	(12)	57	(130)	3				
intron	R2–6	25	(1)	530	(200)	2100	(1300)	33	0.6	(0.2)	1.23	(.09)
	R7–8	25	(.7)	950	(210)	2700	(480)	30	0.5	(0.1)	1.19	(.08)
BBlock (easy)	R2–6	32	(0)	870	(100)	2800	(480)	33	0.4	(0.1)	1.02	(.12)
	R7–8	32	(0)	730	(59)	2100	(490)	23	0.3	(0.0)	1.10	(.08)
	U7–55	32	(0)	1500	(170)	5900	(2000)	93	1.2	(0.3)	.96	(.09)
	S7–55	32	(0)	2900	(680)	14000	(4200)	527	8.6	(2.4)	.98	(.11)
intron	R2–6	32	(0)	1100	(130)	3900	(880)	38	0.5	(0.1)	.98	(.09)
	R7–8	32	(0)	1000	(160)	2800	(470)	26	0.3	(0.1)	1.03	(.09)
	U7–55	32	(0)	1700	(220)	5600	(920)	97	1.3	(0.2)	.94	(.08)
	S7–55	32	(0)	2600	(340)	13000	(2200)	413	7.1	(1.6)	1.00	(.06)
BBlock (hard)	R2–6	26	(.8)	9000	(4600)	20000	(11000)	132	2.8	(1.7)	1.34	(.19)
	R7–8	25	(.8)	5900	(2800)	13000	(5100)	73	1.3	(0.4)	1.50	(.64)
	U7–55	26	(.4)	14000	(7100)	33000	(15000)	306	6.5	(2.6)	1.32	(.20)
	S7–55	28	(.9)	76000	(47000)	200000	(110000)	3145	77.4	(42.3)	1.60	(.17)
intron	R2–6	25	(.5)	6300	(2200)	14000	(3700)	93	1.9	(0.8)	1.33	(.14)
	R7–8	24	(.9)	4500	(2400)	11000	(6600)	69	1.3	(0.8)	1.18	(.25)
	U7–55	25	(.7)	17000	(9000)	44000	(26000)	359	8.2	(5.4)	1.44	(.24)
	S7–55	26	(.8)	24000	(17000)	71000	(55000)	1078	25.6	(19.8)	1.49	(.25)

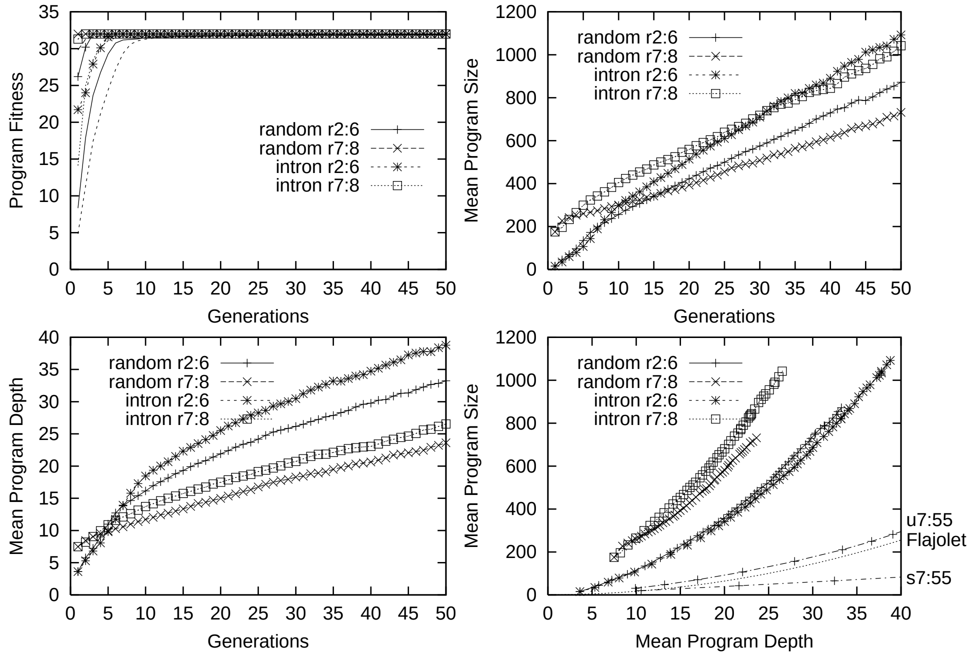

Fig. 3. Evolution of GP populations on easy correlated landscapes. Means of ten runs with standard (r2:6) and large (r7:8) initial populations. Lower right (evolution of shape) also includes average evolution of runs started with uniform (u7:55) and sparse populations (s7:55). Tick marks at each generation

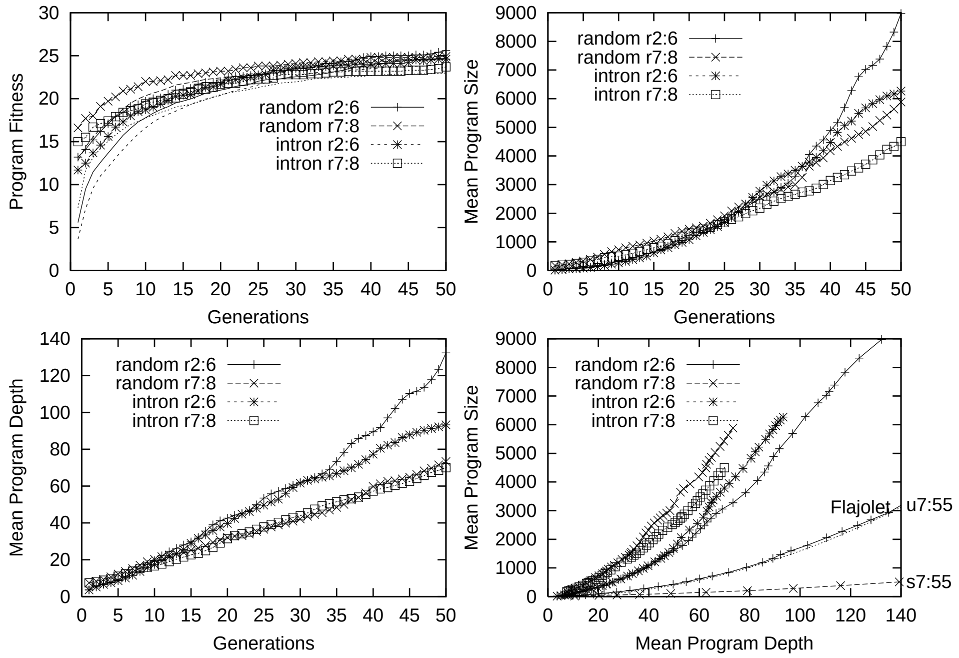

Fig. 4. Evolution of GP populations on hard correlated landscapes

eration (cf. Table 2 col 10). This has been observed in several GP bench mark problems [24, 18].

The non-linear, slower depth increase in the easy search spaces is produced by reduced selection pressure. GP populations converge in the sense that most of the programs within them have identical fitness values. On average in the last ten generations 79–86% of tournaments are between potential parents of identical fitness and so parents are chosen at random.

Unexpectedly populations which initially contain random trees (u7:55) increase their depth faster than those starting with bushy trees (r2:6 and r7:8). However depth increase in sparse trees (s7:55) is even bigger and is non-linear but faster bloat of thin trees can be expected.

5.3 Evolution of Shape

The average shape of trees within the populations evolves away from the bushy nearly full trees created by "ramped half-and-half" in the direction of the most popular part of the search space (denoted by "peak" or "Flajolet" in Figs. 2–4). However the population mean lies to the left of it. There are several possible reasons for this: 1) The ridge itself is quite wide and, except near the boundaries, change in program depth makes comparatively little difference to the number of programs, i.e. the local gradient lies nearly parallel to the size axis. 2) The initial population is to the left. 3) As is usual, our subtree crossover is biased to select functions as crossover points rather than terminals. 4) Interaction with the search space. To test the fourth option, ten runs were conducted on both building block landscapes (with and without explicit introns) starting with a) random sparse trees and b) trees selected uniformly at random between two sizes [18]. As observed in GP benchmark problems [24] populations initialised with sparse trees evolve towards the most populated part of the search space but remain on the sparse (right) side of it. These trees are taller than used in [24] and they remain comparatively sparse, i.e. they move more slowly towards the ridge. While those initialised with common tree shapes remain near this ridge in the search space, see Figs. 3–4 (lower right). Therefore it does not appear that these search spaces promote the evolution of busier trees.

5.4 Sub-Quadratic Bloat

As discussed in [24, 18, 19] 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'. For modest size programs we expect size $O(\text{gens}^{1.3})$ rising to a limit of quadratic growth for $|\text{program}| \gg 1000$ cf. [8, Table II]. For traditional bushy initial tree and excluding the very hard and the easy landscapes, Table 2, column 10, reports variation between runs but values near 1.3 on average.

6 Conclusions

We have investigated subtree crossover, building blocks and bloating by using random search spaces where fitness is based only on syntax. Thus avoiding consideration of the semantics of the programming language. We find GP behaves like it does on real program spaces (i.e. with semantics).

Only in the case of a totally random landscape does bloat not occur. In this case crossover is unable to find children which are different from their parents (i.e. to explore) and who have a fitness at least as good as their parents. I.e. the extreme nature of the problem prevents bloat. In the other problems there is correlation and bloat occurs, whether the correlation is due to building blocks or the explicit introduction of ineffective code. Thus the entropy explanation of bloat [21, 24] remains viable.

The failure of GP to solve the harder problem with building blocks might be due to premature convergence. This may be a common failing in similar genetic search and these artificial search spaces may be useful in future research to investigating this and other aspects of GP and related techniques.

C++ code may be found in `ftp://cs.ucl.ac.uk/wblangdon/gp-code`

Acknowledgements

We acknowledge funding of the "Deutsche Forschungsgemeinschaft", under project B2 of "Sonderforschungsbereich SFB 531".

References

1. Lee Altenberg. Emergent phenomena in genetic programming. In A. V. Sebald and L. J. Fogel, editors, *Evolutionary Programming — Proceedings of the Third Annual Conference*, pages 233–241. World Scientific Publishing, 1994.
2. Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In John R. Koza *et. al.*, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, 13-16 July 1997. Morgan Kaufmann.
3. Peter J. Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4. MIT, 1994.
4. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, January 1998.
5. W. Banzhaf, and W. B. Langdon. *Some considerations on the reason for bloat.* In preparation.
6. Tobias Blickle. Evolving compact solutions in genetic programming: A case study. In Hans-Michael Voigt *et. al.*, editors, *Parallel Problem Solving From Nature IV*, volume 1141 of *LNCS*, pages 564–573, 22-26 September 1996. Springer-Verlag.
7. Kumar Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, September 1997.
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. *An Investigation of Supervised Learning in Genetic Programming.* PhD thesis, University of Edinburgh, 1998.
10. David E. Goldberg and Una-May O'Reilly. Where does the good stuff go, and why? In Wolfgang Banzhaf *et. al.*, editors, *Proceedings of the First European Workshop on Genetic Programming*, *LNCS 1391*, pp. 16–36, Paris, 14-15 April 1998. Springer.

11. Thomas Haynes. Collective adaptation: The exchange of coding segments. *Evolutionary Computation*, 6(4):311–338, 1998.
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, 1992.
14. J. R. Koza, D. Andre, F. H Bennett III, and M. Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, 1999.
15. W. B. Langdon. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638.
16. W. B. Langdon. *Data Structures and Genetic Programming*. Kluwer, 1998.
17. W. B. Langdon. Scaling of program tree fitness spaces. *Evolutionary Computation*, 7(4):399–428, 1999.
18. W. B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1):95–119, 2000.
19. W. B. Langdon. Quadratic bloat in genetic programming. In Whitley *et. al.*, editors, *GECCO'2000*, Las Vegas, USA, 8-12 July 2000. Morgan Kaufmann.
20. W. B. Langdon and R. Poli. An analysis of the MAX problem in genetic programming. In John R. Koza *et. al.*, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 222–230, 1997. Morgan Kaufmann.
21. W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry *et. al.*, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997.
22. W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In Wolfgang Banzhaf *et. al.*, editors, *Proceedings of the First European Workshop on GP*, 1998. Springer.
23. W. B. Langdon and R. Poli. Genetic programming bloat with dynamic fitness. In W. Banzhaf *et. al.*, editors, *Proc. First European Workshop on GP*, 1998. Springer.
24. W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In Lee Spector *et. al.*, *Advances in Genetic Programming 3*, ch. 8. 1999.
25. S. Luke and L. Spector. A revised comparison of crossover and mutation in GP. In J. R. Koza *et. al.*, *Genetic Programming 1998*, pp. 208–213. Morgan Kaufmann.
26. Nicholas F. McPhee and Justin D. Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *(ICGA95)*, pages 303–309. Morgan Kaufmann.
27. Una-May O'Reilly. *An Analysis of Genetic Programming*. PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, 1995.
28. Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 32(10):1192–1201, Oct 1988.
29. J. P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza *et. al.*, editors, *Genetic Programming 1997*, pp. 286–294. Morgan Kaufmann.
30. Conor Ryan. Pygmies and civil servants. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 11, pages 243–263. MIT Press, 1994.
31. Peter W. H. Smith and Kim Harries. Code growth, explicitly defined introns, and alternative selection schemes. *Evolutionary Computation*, 6(4):339–360, 1998.
32. Terence Soule. *Code Growth in Genetic Programming*. PhD thesis, University of Idaho, Moscow, Idaho, USA, 15 May 1998.
33. Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, Southern California, Dept. EE Systems, 1994.
34. Byoung-Tak Zhang and Heinz Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, 1995.