Size Control via Size Fair Genetic Operators in the PushGP Genetic Programming System

Raphael Crawford-Marks
Box 820
Hampshire College
Amherst, MA 01002
rpc01@hampshire.edu

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

Abstract

The growth of program size during evolution (code "bloat") is a well-documented and well-studied problem in genetic programming. This paper examines the use of "size fair" genetic operators to combat code bloat in the PushGP genetic programming system. Size fair operators are compared to naive operators and to operators that use "node selection" as described by Koza. The effects of the operator choices are assessed in runs on symbolic regression, parity and multiplexor problems (2,700 runs in total). The results show that the size fair operators control bloat well while producing unusually parsimonious solutions. The computational effort required to find a solution using size fair operators is about equal to, or slightly better than, the effort required using the comparison operators.

1 INTRODUCTION

Code bloat in genetic programming has been documented since the field came into existence a decade ago. In the past few years bloat has been studied extensively, with researchers examining the causes of bloat as well as testing new operators designed to limit code bloat (D'haeseleer, 1994; Angeline, 1994; Langdon and Poli, 1997; Poli and Langdon, 1997; Banzhaf, et al., 1998; Soule and Foster, 1998; Langdon, et al., 1999; Francone, et al., 1999; Langdon, 1999; Luke, 2000). Recently, "size fair" operators have been shown to limit bloat significantly without decreasing the problem-solving ability of the genetic programming system (Langdon, et al., 1999; Langdon, 1999).

This paper extends Langdon's work by testing size fair operators in a genetic programming system that uses unusual representations for programs. The PushGP system is conventional in most respects but it manipulates and produces Push programs rather than the Lisp-like program trees used in more conventional genetic programming systems (for example (Koza, 1992)). Push programs, like Lisp programs, are variably sized strings of symbols and balanced (possibly nested) sets of parentheses. On the other hand, Push programs are interpreted quite differently from Lisp programs; Push program interpretation is more similar to the interpretation of stack-based languages like Forth or Postscript. The applicability of Langdon's work to PushGP is therefore an interesting test of the generality of his findings.

A detailed description of the Push programming language is beyond the scope of this paper; see (Spector and Robinson, 2002) for a full introduction and language reference, or (Spector, 2001; Robinson, 2001) for brief introductions. The essential feature of the Push language for the present study is just that the programs are syntactically similar to, yet semantically quite different from, the Lisp-like programs used in traditional genetic programming systems. Push's unique structure supports many enhancements to genetic programming systems (for example, efficient and fully automatic evolution of modular programs) but none of these are relevant to the present study; see (Spector and Robinson, 2002) for details.

In Langdon's prior work he tested a 50%-150% fair mutation operator in stochastic problem solving systems (e.g., hill climbing and simulated annealing systems) but not specifically in genetic programming systems. In this study we apply a variant of this operator in PushGP and demonstrate its utility for genetic programming. We also describe a new size fair crossover operator and describe the performance of the size fair operators in all possible combinations with

naive operators and operators that use node selection (a technique based on Koza's 90%-10% tree/leaf selection method (Koza, 1992)).

2 Bloat in PushGP

Nested parentheses in the Push syntax make it possible to model a Push program as a tree, and therfore to apply standard, tree-based genetic operators in PushGP. The original version of PushGP used what we will call "naive" operators which were meant to be as simple as possible while capturing the essential ideas of traditional genetic programming operators. The naive mutation operator selects a random "point" of the program to replace, with each point having an equal probability of being chosen. Each symbol and each parenthesized expression in the program counts as a point. The chosen point is then replaced with a new randomly generated expression, which will have a size uniformly selected from the range $[1, n]$, where n is a system parameter. The naive crossover operator selects random points in both parent programs (again with all points having an equal probability of being chosen) and returns a copy of one of the parents with the chosen point from its mate replacing its own chosen point.[1]

Bloat is quite strong in PushGP when the naive operators are used, in large part because the naive mutation operator generates random subtrees that are larger, on average, than the subtrees they replace. In order to keep program lengths manageable, PushGP implements a size ceiling. Any program exceeding the size ceiling is discarded, and a clone of one of its parents is used in its place; in the tables below we refer to this as a "size limit replication." Most runs with the naive crossover and mutation operators exhibit rapid code bloat, with program sizes climbing steadily toward the size ceiling. It has been shown that when put to work on simple symbolic regression problems, 20%-45% of the children were over the size limit at Generation 50, and thus discarded in favor of a clone of the parent (Robinson, 2001). (Robinson, 2001) also discovered that the naive crossover and mutation operators were more likely to select leaf nodes than internal nodes, resulting in little variation in the internal structure of programs across the population.

3 What Causes Bloat?

Code bloat is not a phenomenon particular to GP. It has been shown to occur in several non-GP stochastic search techniques (Langdon, 1998). There are many studies on the origins of bloat. Some findings suggest that because there are more large programs than small ones in a search space, fitness-based selection on average finds larger programs with better fitness than smaller or equal-sized programs (Langdon and Poli, 1997). Others suggest that bloat occurs as an evolutionary defense mechanism against destructive operators. Traditional crossover and mutation can often be fatal when applied to a small, fit program as they randomly rip out a chunk of the fit program and replace it with a different random chunk of program. Thus, programs evolve "introns" (segments of neutral code) as a means to preserve fitness when subjected to destructive evolutionary operators (Nordin and Banzhaf, 1995). Similar to defense theory, (Soule and Foster, 1998) suggest that individuals are penalized when a large chunk of code is removed, but not so when a large chunk is inserted, thus driving up the size of the program. This is called "removal bias". More recently, (Luke, 2000) suggested that introns are not the cause of code growth, but rather a symptom, and that a bias towards deeper crossover points drives code growth.

The underlying cause of bloat is still open to debate. What is universally agreed upon, however, is that bloat occurs and often has detrimental effects on the improvement in fitness in genetic programming runs. In addition, it clearly slows down genetic programming runs by consuming CPU cycles and large amounts of memory.

4 New Operators

We studied four variations of the genetic operators (two variations of mutation, two of crossover), in addition to the naive operators described above.

Node Selection, a method described in (Koza, 1992), chooses an internal node 90% of the time and a leaf node 10% of the time for either mutation or crossover. Node selection was implemented both for mutation and crossover in PushGP.

"Size Fair" crossover and mutation operators are operators that on average produce children of the same size as their parents. The size fair mutation operator we use is identical to the 50%-150% operator described in (Langdon, 1998; Langdon, et al. 1999), except that it produces mutations of length $\ell \pm \frac{\ell}{4}$ instead of $\ell \pm \frac{\ell}{2}$, where ℓ is the number of points in the subtree to be mutated.[2] The size distribution of the replacement

[1] The random code generator is described in (Spector and Robinson, 2002).

[2] The fraction $\frac{\ell}{4}$ was chosen arbitrarily, prior to reading Langdon's work. We assume the specific fraction has little effect on performance.

subtrees (and thus the resulting children) is uniform.

Our new crossover operator, Fair Crossover, differs from the size fair crossover operator described in (Langdon, 1999). Langdon's operator selects the first crossover point at random from Parent 1. The size (ℓ) of the subtree at the first crossover point is calculated, and the lengths of all subtrees in Parent 2 are also calculated. All subtrees from Parent 2 whose size is larger than $1+2\ell$ are excluded. This limits the amount by which the child can increase in size to $1+\ell$ larger than its parent. For the remaining subtrees, the number that are smaller, the same size and larger than ℓ are each counted, along with the mean size difference for the larger and smaller subtrees. A roulette wheel is used to select the size of the subtree to be crossed over. The selection method is biased using calculated mean size differences such that on average there is no change in program size after crossover is performed.

With our new Fair Crossover operator, the first crossover point is selected at random from Parent 1, and the length of the subtree at that point is measured. Then a randomly selected subtree from Parent 2 is measured. If its length is within the range $\ell \pm \frac{\ell}{4}$ (where ℓ is the length of the subtree from the first parent), the subtree from Parent 2 replaces the subtree in Parent 1. If not, another subtree is randomly selected from Parent 2, and the test is repeated. If no subtrees are found within the range $\ell \pm \frac{\ell}{4}$ after n attempts, the subtree whose size was closest to $\ell \pm \frac{\ell}{4}$ is used in crossover. The size distribution of replacement subtrees is dependent on the parents, and may not be uniform.[3]

For the experiments described in this paper, Fair Crossover would perform 20 retries before giving up and using the subtree with the closest length. We will call this a "punt". In the 300 runs on symbolic regression of a sextic polynomial that used Fair Crossover, the operator punted just over 80% of the time. This means that if 2250 crossovers were performed, the operator only found replacement subtrees within the length $\ell \pm \frac{\ell}{4}$ about 450 times. However, since the subtree whose size is *closest* to $\ell \pm \frac{\ell}{4}$ is used after 20 tries, Fair Crossover still has an effect close to that of a size fair operator. In the same 300 runs, replacement subtrees found by Fair Crossover were on average only 0.8 points larger than the original subtree. Given the low bloat observed when Fair Crossover is used, it appears that while Fair Crossover may not be perfectly size fair, it is quite close.

[3]Fair Crossover was used instead of Langdon's size fair crossover operator because it was simpler to implement.

```
- push-base-type:
      dup, pop, swap, rep, =, set, get, convert,
      pull, pulldup, noop
  - number: +, -, *, /, >, <
    - integer: pull, pulldup, /
  - boolean: not, and, or
  - expression:
      quote, car, cdr, cons, list, append, subst,
      container, length, size, atom, null, nth,
      nthcdr, member, position, contains, insert,
      extract, instructions, replace-atoms,
      discrepancy
    - code: do, do*, if, map
```

Figure 1: Push function set used for the PushGP runs.

5 Results

All combinations of crossover and mutation operators were used in sets of 100 independent genetic programming runs on 3 different problems: Sextic Regression, Even-5 Parity, and 6-Bit Multiplexor.

For all problems, the population size was 5000, the program size ceiling 50 points, and the runs were limited to 50 generations. The size of mutant subtrees added by the Naive Mutation operator was limited to 10 points. The operator rates were 45% crossover, 45% mutation and 10% straight reproduction. The tournament size was 7. The function set is listed in Figure 1. See (Spector and Robinson, 2002) additional information on the Push functions.

Computational Effort was computed in the standard way, as described by Koza on pages 99 through 103 of (Koza, 1994). To summarize briefly, one conducts a large number of runs with the same parameters (except random seeds) and begins by calculating $P(M, i)$, the cumulative probability of success by generation i using a population of size M. For each generation i this is simply the total number of runs that succeeded on or before the ith generation, divided by the total number of runs conducted. From $P(M, i)$ one can calculate $I(M, i, z)$, the number of individuals that must be processed to produce a solution by generation i with probability greater than z. Following the convention in the literature we use a value of z=99%. $I(M, i, z)$ can be calculated using the following formula:

$$I(M, i, z) = M * (i + 1) * \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil$$

The more steeply the graph of $I(M, i, z)$ falls, and the lower its minimum, the better the genetic programming system is performing. Koza defines the minimum of $I(M, i, z)$ as the "computational effort" re-

Table 1: Results for symbolic regression of $x^6 - 2x^4 + x^2$, sorted by computational effort.

Crossover Method	Mutation Method	Successful Runs	Average Solution Size	Average Size Limit Replications (Gen. 25)	Average Size Limit Replications (Gen. 49)	Computational Effort
Fair	Node Sel	93/100	31.29	363.00	715.71	450000
Fair	Naive	85/100	32.66	965.76	1456.47	480000
Node Sel	Fair	87/100	39.29	725.46	948.71	495000
Naive	Node Sel	88/100	37.10	941.20	1216.17	540000
Fair	Fair	88/100	21.77	10.19	64.85	540000
Naive	Fair	87/100	33.63	362.70	587.21	585000
Naive	Naive	71/100	38.17	1519.24	1920.52	800000
Node Sel	Node Sel	76/100	40.58	1328.78	1576.00	820000
Node Sel	Naive	61/100	42.36	2024.82	2280.97	960000

Table 2: Results for Even-5 Parity, sorted by computational effort.

Crossover Method	Mutation Method	Successful Runs	Average Solution Size	Average Size Limit Replications (Gen. 25)	Average Size Limit Replications (Gen. 49)	Computational Effort
Fair	Naive	100/100	34.85	318.50	*	240000
Naive	Naive	98/100	36.62	1201.22	850.00	250000
Fair	Node Sel	99/100	29.32	29.69	281.00	270000
Naive	Node Sel	99/100	36.06	413.65	386.00	280000
Node Sel	Naive	100/100	41.58	1659.48	*	290000
Naive	Fair	96/100	29.56	214.06	258.60	310000
Node Sel	Fair	96/100	31.75	342.15	793.50	320000
Fair	Fair	97/100	20.99	8.23	0.00	330000
Node Sel	Node Sel	98/100	37.89	657.51	1016.00	350000

* All runs completed before 49th Generation

Table 3: Results for 6-Bit Multiplexor, sorted by computational effort.

Crossover Method	Mutation Method	Successful Runs	Average Solution Size	Average Size Limit Replications (Gen. 25)	Average Size Limit Replications (Gen. 49)	Computational Effort
Fair	Fair	30/100	19.80	0.46	28.56	1870000
Fair	Node Sel	36/100	27.58	71.41	428.67	1885000
Naive	Fair	32/100	27.53	127.00	410.82	2080000
Naive	Node Sel	26/100	30.96	389.41	749.47	2520000
Fair	Naive	26/100	32.27	623.75	1388.20	2635000
Node Sel	Naive	23/100	37.57	1375.40	1725.29	2835000
Node Sel	Fair	26/100	27.96	325.13	673.92	3120000
Naive	Naive	26/100	37.92	972.08	1519.34	3200000
Node Sel	Node Sel	18/100	31.11	697.06	1014.76	4320000

quired to solve the problem. Computational effort is not a perfect measure (see, for example, (Luke and Panait, 2002)) but we believe it is sufficient for the modest uses to which it is put here.

5.1 Symbolic Regression of $x^6 - 2x^4 + x^2$

As shown in Table 1, the combination of Fair Crossover and Node Selection Mutation yielded the most solutions, least computational effort and the second-most parsimonious solution sizes. The combination of Fair Mutation and Fair Crossover yielded the second most solutions (tied with Naive Crossover and Node Selection mutation) and the most parsimonious ones as well (by nearly 10 points), but scored in the middle of the field in terms of computational effort. Notable also is that the operators causing the greatest amount of bloat (and thus the greatest number of replications due to hitting the size ceiling) finished in the last three spots in terms of solutions found and computational effort.

5.2 Even-5 Parity

Even-5 Parity is a fairly easy problem for PushGP to solve, as shown by the high number of solutions found. Interestingly, one of the least successful combinations from the regression runs, Naive Crossover with Naive Mutation, scored just behind Fair Crossover with Naive Mutation in terms of computational effort. Again, Fair Crossover with Fair Mutation found the most parsimonious solutions by nearly 10 points, and kept replications down to almost nothing, but scored next to last in terms of computational effort.

5.3 6-Bit Multiplexor

When applied to the 6-bit Multiplexor problem different operators performed best. The pairing of size fair mutation and crossover found the third most solutions with the least computational effort. The size fair operators also found the most parsimonious solutions, beating the next best pair of operators by almost 8 points. The rest of the top performers all had performed well in previous runs. Performing particularly poorly was the pairing of Node Selection mutation and crossover, which found the fewest solutions and required the most computational effort.

6 Discussion

The efficacy of the size fair operators in controlling bloat and in producing parsimonious solutions is clear from the data. Certainly for the cases in which fair mutation and fair crossover were used together the improvements in these measures were dramatic. Additionally, in many cases the use of just one size fair operator, in conjunction with a non-size-fair operator, seems to confer advantages.

It should come as no surprise that it is impossible to declare one operator or combination of operators as clearly being better than the rest with respect to the computational effort required to find a solution. However, we do note that all of the runs with size fair operators performed at least reasonably well; the use of size fair operators does not appear to be detrimental with respect to this measure. We also note that the better combinations often included one size fair operator and one non-size-fair operator. One could speculate that size fair operators, used by themselves, slow the genetic programming system in its progress to larger areas of the search space (where solutions are more plentiful) thus increasing the time it takes to find solutions. If so then one might further speculate that the judicious mixing of non-size-fair operators, which can have more dramatic impacts on program size, with size fair operators would be the best way to encourage robust problem solving performance. More research would be required to confirm or falsify these speculations.

7 Conclusions

The size fair operators examined in this work appear to control bloat well and to encourage the production of parsimonious solutions without negative impacts on the computational effort required to find a solution. This is important because unchecked bloat limits the applicability of genetic programming by requiring exorbitant computational resources, and because naive approaches to bloat control can change the system's evolutionary dynamics in ways that make it harder to find solutions. Solution parsimony is also important because it simplifies the work of humans who must interpret the output of genetic programming systems, and because more parsimonious solutions may in some cases also be more general.

This work was conducted using the PushGP system which is similar to traditional genetic programming systems in some ways but different from them in others. The reported work extends Langdon's earlier work, demonstrating that the idea of size fair operators has utility across a broader range of program representations. The obvious next step is to repeat this study, using Langdon's operators and the new size fair crossover operator that we have developed, in a more traditional genetic programming system. If they per-

form as well in such a follow-up study, controlling bloat and producing parsimonious solutions without sacrificing the problem-solving capacity of the system, then we would recommend their wide-spread adoption.

Acknowledgments

Thanks are due to Benjamin Lefstein for observations and suggestions, and to Alan Robinson for keeping the cluster computer running.

This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30502-00-2-0611. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

This research was also made possible by generous funding from Hampshire College to the Institute for Computational Intelligence at Hampshire College.

References

Angeline, P. 1994. Genetic Programming and Emergent Intelligence. In *Advances in Genetic Programming*. MIT Press, pp. 75-98.

Banzhaf, W., P. Nordin, R. E. Keller and F. D. Francone. 1998. Genetic Programming: An Introduction. Morgan Kaufmann Publishers.

D'haeseleer, P. 1994. Context Preserving Crossover in Genetic Programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press, pp. 256-261.

Francone, F. D., M. Conrads, W. Banzhaf, and P. Nordin. 1999. Homologous Crossover in Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*. Morgan Kaufmann, pp. 1021 - 1026.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection,* MIT Press, 1992.

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

Langdon, W. B. 1999. Size Fair and Homologous Tree Genetic Programming Crossovers. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99*. Morgan Kaufmann, pp. 1092 - 1097.

Langdon, W. B., T. Soule, R. Poli, and J. A. Foster. 1999. The evolution of size and shape. In *Advances in Genetic Programming 3*. MIT Press, Chapter 8, pages 163 - 190.

Langdon, W. B. 1998. The Evolution of Size in Variable Length Representations. In *1998 IEEE International Conference on Evolutionary Computation*. IEEE Press, pp. 633 - 638.

Langdon, W. B. and R. Poli. 1997. Fitness Causes Bloat. In *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London.

Luke, S. 2000. Code Growth Is Not Caused By Introns. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*. IEEE Press, pp. 228-235.

Luke, S., and L. Panait. 2002. Is the Perfect the Enemy of the Good? In Langdon, W. B., et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002*. San Francisco, CA: Morgan Kaufmann Publishers.

Nordin, P. and W. Banzhaf. 1995. Complexity compression and evolution. In *Genetic Algorithms: Proceedings of the Sixth International Conference*. Morgan Kaufmann Publishers.

Poli, R. and W. B. Langdon. 1997. Genetic Programming with One-Point Crossover. In *Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London, pp. 180-189.

Robinson, A. 2001. "Genetic Programming: Theory, Implementation, and the Evolution of Unconstrained Solutions," Hampshire College Division III (senior) thesis.
http://hampshire.edu/lspector/robinson-div3.pdf.

Soule, T. and J. A. Foster. 1998. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*. IEEE Press, pp. 781-186.

Spector, L. 2001. Autoconstructive Evolution: Push, PushGP, and Pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*. Morgan Kaufmann Publishers.

Spector, L., and A. Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push

Programming Language. In *Genetic Programming and Evolvable Machines*. Vol. 3, No. 1, pp. 7–40.