Chapter 16

OPERATOR CHOICE AND THE EVOLUTION OF ROBUST SOLUTIONS

Terence Soule
Department of Computer Science
University of Idaho, Moscow, ID, 83844-1010
tsoule@cs.uidaho.edu

Abstract

This research demonstrates that evolutionary pressure favoring *robust* solutions has a significant impact on the evolutionary process. More robust solutions are solutions that are less likely to be degraded by the genetic operators. This pressure for robust solutions can be used to explain a number of evolutionary behaviors. The experiments examine the effect of different types and rates of genetic operators on the evolution of robust solutions. Previously robustness was observed to occur through an increase in inoperative genes (introns). This work shows that alternative strategies to increase robustness can evolve. The results also show that different genetic operators lead to different strategies for improving robustness. These results can be useful in designing genetic operators to encourage particular evolutionary behaviors.

Keywords: Code growth, code bloat, operators, introns, design, robust solutions

1. Introduction and Background

The goal of this paper is to demonstrate that pressure for robust solutions is a general phenomenon effecting more than just tree-based GP and that growth is not the only evolutionary response to this pressure. The results show that evolution may favor any of several different strategies to improve robustness and that the strategy adopted depends significantly on the operators involved.

It is worth emphasizing that in this discussion robustness refers only to resistance to change from variation operators such as crossover and mutation. In a different context robustness might refer to other factors such as resistance to noise or to changing environments. Further, pressure for robustness is not

the *only* factor influencing evolution. Clearly selection produces considerable pressure for more fit solutions and there may be other, as yet unidentified, pressures as well.

In this paper the robustness (or stability) of a solution is a measure of the solution's average change in fitness under genetic operations such as crossover and mutation. It has been theorized that there is significant evolutionary pressure in favor of solutions that are more robust (Soule, 2002; Soule, 2002; Streeter, 2002). The most outstanding evidence of pressure towards stability is the phenomenon of code growth (or bloat) in genetic programming (GP) (Koza, 1992; Blickle and Thiele, 1994; Nordin and Banzhaf, 1995; McPhee and Miller, 1995; Soule, 1996; Soule, 1998; Luke, 2000c; Nordin and Banzhaf, 1995; Nordin *et al.*, 1997; Nordin, 1997). Code bloat is a rapid increase in code size that does not result in fitness improvements. The extra code usually consists of inoperative code or introns (code that does not contribute to the program's fitness). It is generally accepted that in large part programs generated with GP grow as a means of protecting the useful code within good solutions against the potentially negative effects of crossover. By adding introns the useful code (commonly known as operative code or exons) is less likely to be affected by crossover.

Luke has argued that introns themselves are not the cause of code growth (Luke, 2000c). Smith and Harries have shown that growth can occur in code that does influence fitness if the code has only a negligible effect on performance (Smith and Harries, 1998). The author has recently shown that code growth can also occur in exons that have a significant effect on fitness (Soule, 2002; Soule, 2002). It has also been shown that some forms of mutation can encourage code growth (Langdon *et al.*, 1999; Soule and Heckendorn, 2002). Finally, theoretical and experimental research strongly suggests that code growth is a factor in any evolutionary system using a variable sized representation (McPhee and Miller, 1995; Soule, 1998; Langdon, 1997).

Whereas early code growth research focused on tree based GP, introns and crossover, these recent results point to a more general phenomenon relating to any evolutionary system with variable length representations, any genetic operators and both introns and exons. This supports the hypothesis that code growth in GP is only one symptom of a general underlying evolutionary pressure favoring robust individuals. To avoid the focus on tree based representations a linear representation is used in these experiments.

The results presented here show that different operators create pressure for different forms of solutions. In particular, two versions of crossover are compared: with one version the pressure for robust solutions leads to growth and with the other version it does not. It is also shown that with some genetic operators a reduction in solution size is favored to make solutions more robust.

Finally, the results suggest rules for determining how a particular operator will influence the evolutionary process by favoring robust solutions. The rules will be particularly useful to a practitioner designing a new representation and operators for a novel problem. These rules will make it possible for the practitioner to begin to predict how the new operators will influence the evolutionary process and thus will be useful in designing evolutionary algorithms and operators that guide the evolutionary process in specific directions.

Other Causes of Growth

In addition to protective growth, two other causes of growth in GP have also been proposed: *removal bias* and *drift*. Because growth is an important strategy for evolving robustness these additional causes need to be considered in examining the data presented here.

Removal bias hypothesizes that growth occurs in part because the size of the region removed during crossover is much more significant in determining the offspring fitness than is the size of the added region (Soule and Foster, 1998). In particular, offspring created by removing a small region and adding a larger region (such offspring will be larger than their parents) are much more likely to maintain their fitness than offspring created by removing a large region and adding a small region (such offspring will be smaller than their parents). Thus, in the selection step the larger than average individuals are more likely to survive. Experimental data has confirmed that removal bias occurs and is a significant factor in GP growth (Soule and Heckendorn, 2002; Luke, 2000c). Theoretical analysis suggests that removal bias will be most significant when inviable and 'near' inviable regions (regions that have no effect or a negligible effect on fitness when modified) are common.

The drift theory of code growth is based on the structure of the search spaces. It has been experimentally observed that for many problems the number of solutions of a given fitness that are larger than a given size is much greater than the number of solutions with the same fitness that are smaller than the given size (Langdon *et al.*, 1999; Langdon, 1999). E.g. given a solution of fitness X and size Y there are many more solutions with fitness X that are larger than Y than that are smaller than Y.

Because larger programs are more common it has been proposed that an unbiased search is more likely to find larger solutions of a given fitness than it is to find smaller solutions of that fitness, simply because there are more larger solutions within the search space to be found. The general idea is that "... any stochastic search technique, such as GP, will tend to find the most common programs in the search space of the current best fitness" (Langdon, 1999) and for any given fitness, larger programs are more common.

It's worth noting that these three views of growth (protection, removal bias or drift) are not mutually exclusive. Nor would proving that any one of them is correct invalidate the general notion of selective pressure for robust solutions. In fact, as noted above, the protective view of growth can be taken as a special case of the general pressure for robustness.

2. Experiment

A very simple problem is used to examine the importance of code and operator types on a variable length, linear evolutionary algorithm. The goal is to find a set of integers that sum to a given target value T. The allowed integers are 0, 1 and 4. Individuals are variable length strings consisting of those three integers.

The fitness of an individual is the absolute value of the difference between the sum of the integers and the target value, i.e. fitness = |sum - T|. For the individual:

$$00141001104 \tag{16.1}$$

the sum is

$$0+0+1+4+1+0+0+1+1+0+4=12 \tag{16.2}$$

and the fitness is

$$|12-T|. \tag{16.3}$$

Clearly, a lower fitness is better.

Note that for these experiments there are effectively three types of "genes." Genes with a large effect on fitness: 4's, genes with a smaller effect on fitness: 1's and genes with no effect on fitness (inoperative genes): 0's. This problem is similar to those used in previous experiments, although a linear rather than a tree-based representation is used here (Soule, 2002; Soule, 2002).

The evolutionary algorithm is generational, with a population size of 500, elitism of 2 and is run for 1000 generations. The genetic operators, crossover and mutation, are described below. Other details of the evolutionary algorithm are shown in Table 16.1.

Crossover

Two-point crossover is used. Because the size of the individuals are variable the two crossover points are chosen independently in each parent. Thus, in general the sizes of the exchanged regions will be different. E.g.

$$\begin{array}{cc} \text{Parents} & \text{Offspring} \\ 00|14|1001104 \rightarrow & 00|0041|1001104 \end{array}$$

$$0140|0041|04 \rightarrow 0140|14|04$$

Two methods of picking the crossover points are used. In the first method both crossover points in an individual are chosen randomly. This is referred to as *proportional crossover* because the length of the exchanged region is proportional to the length of the parent. In general, for two randomly chosen crossover points 25% of each parent will be selected for crossover. This method is analogous to typical GA two-point crossover.

In second method the length l of the crossed region is chosen according to the following algorithm:

$l = 2$
While($l < L/2$ AND random real < 0.5)
$l = l * 2$

where L is the length of the parent individual. Thus, the size of the region selected for crossover is 2 "genes" long 50% of the time, 4 genes long 25% of the time, 8 genes long 12.5% of the time, etc. Thus, crossing short regions is very common and crossing longer regions happens infrequently. Once the length of the crossed region is generated, the left-hand crossover point is randomly selected and the right-hand point is l beyond the left-hand point. This form of crossover is referred to as *constant crossover* because the distribution of lengths of the crossed region is constant, regardless of the parents' sizes.

Constant crossover is analogous to crossover in tree-based GP. In standard tree-based GP crossover a random point is chosen for crossover. For full binary trees this results in an average crossover branch consisting of two nodes regardless of the tree size (Rosca and Ballard, 1996a; Soule and Foster, 1997). Larger branches are exponentially less likely to be chosen for crossover. In practice, GP usually leads to randomly shaped trees rather than full trees (Poli and Langdon, 1997; Langdon, 1999). However, the distribution of crossover points still heavily favors small branches (Soule and Heckendorn, 2002) and using the 90/10 rule (choosing leaf nodes for crossover only 10% of the time) only slightly shifts the distribution towards larger branches (Soule and Heckendorn, 2002). Thus, the distribution of crossover sizes with constant crossover is comparable to those seen in tree-based GP; both emphasize exchanging small branches.

Mutation

For a typical, fixed-length GA the mutation rate is proportional to the inverse of the length of the individuals. However, with a variable-length representation the inverse of the length is not the same for all members of the population. Thus, several fixed mutation rates are chosen (0, 0.0001, 0.001, 0.005, 0.01) and compared.

Table 16.1. Summary of the evolutionary algorithm parameters.

Objective	Find integers whose sum is T
Integer values	0, 1, 4
Population Size	500
Crossover Probability	0.9
Mutation Probability	0, 0.0001, 0.001, 0.005 or 0.01
Selection	3 member tournament
Run Time	1000 Generations
Maximum Size	None
Elitism	2 copies of the best individual are preserved
Initial Population	Random individuals of length 5 to 59
Number of trials	200
Crossover	Proportional and Constant (see text)

3. Results

Three separate sets of results are presented in this paper. First, the effect of the crossover operator on size is examined. It is shown that the different types of crossover have distinctly different effects on size. Next, the evolved number of gene's (0's, 1's and 4's) is examined. These results show that the pressure for robustness favors 1's over 4's. Finally, the effect of incorporating mutation is examined.

Growth

We begin by examining growth, as growth is the best understood strategy for increasing robustness. Figure 16.1 shows the average size (length) of individuals evolved using proportional and constant crossover with no mutation. Growth is quite pronounced with constant crossover and non-existent with proportional crossover.

The explanation for these very different results seems fairly straightforward. With proportional crossover the two crossover points are chosen randomly. Thus, as noted previously, the exchanged regions of the chromosome are proportional to the overall lengths of the chromosomes. The protective effect of growth observed with tree-based GP doesn't apply because adding inoperative code (0's) simply causes a larger region of the chromosome to be exchanged during crossover. Thus, the probability of effecting operative code (1's and 4's) doesn't change when inoperative code (0's) is added.

In contrast, with constant crossover the expected size of the exchanged regions is constant regardless of the chromosome's overall length. Thus, in this case additional inoperative code does decrease the probability of effecting operative code. This is exactly what is observed in GP, where the average size of the exchanged branches during crossover does not grow proportionally with the overall tree size and rapid growth is observed.

These results explain why growth has not been a significant problem in most versions of evolutionary computation using variable length, linear represen-

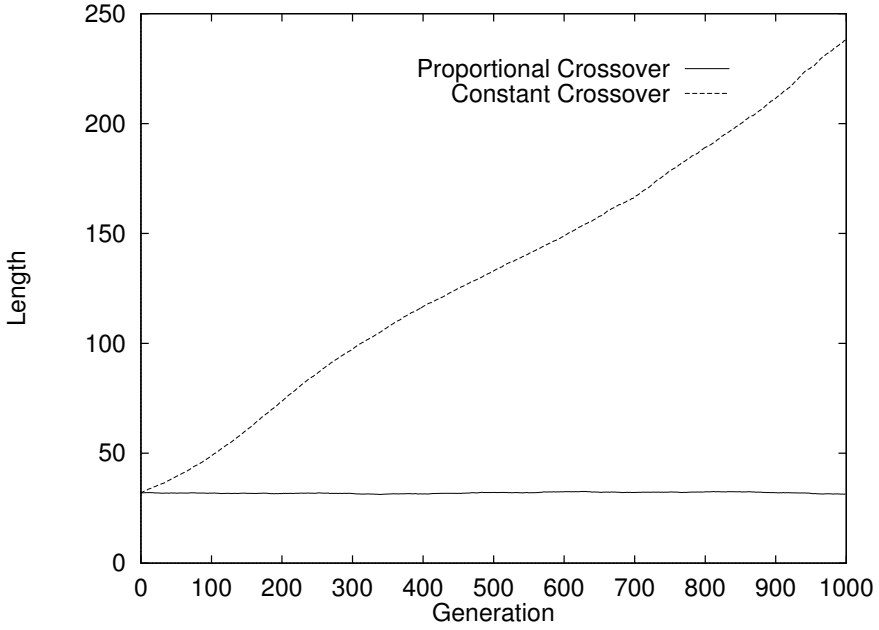

Figure 16.1. Average size (length) for individuals evolved using proportional and constant crossover.

tations. Typical variable length, linear representations use the proportional version of crossover, where the crossover points are picked randomly with a uniform distribution. As these results demonstrate, this method of picking crossover points does not encourage growth.

It is worth considering the other two proposed causes of growth: removal bias and drift. Removal bias is not expected to cause growth in either of the trials described above. The primary assumption of the removal bias hypothesis is that growth occurs because of a bias regarding the size of the removed region that does not apply to the added region during crossover. Normally this bias occurs because there is a probability that the added region will fall within an inviable region of the parent it is being added to. (For example, the added section may be within an "if(false)" branch or may be multiplied by zero.) However, with the current encoding there are no inviable regions; every added 1 or 4 has an effect. Thus, adding a large region is more likely to add 1's and 4's and thus effect fitness just as removing a large region is more likely to effect fitness by removing 1's or 4's. With no bias between removed and added regions removal bias does not apply to these results.

In contrast, drift should occur with either type of crossover. The primary assumption of the drift hypothesis is that there are more larger solutions of a given fitness than there are smaller solutions of the same fitness. Thus, the

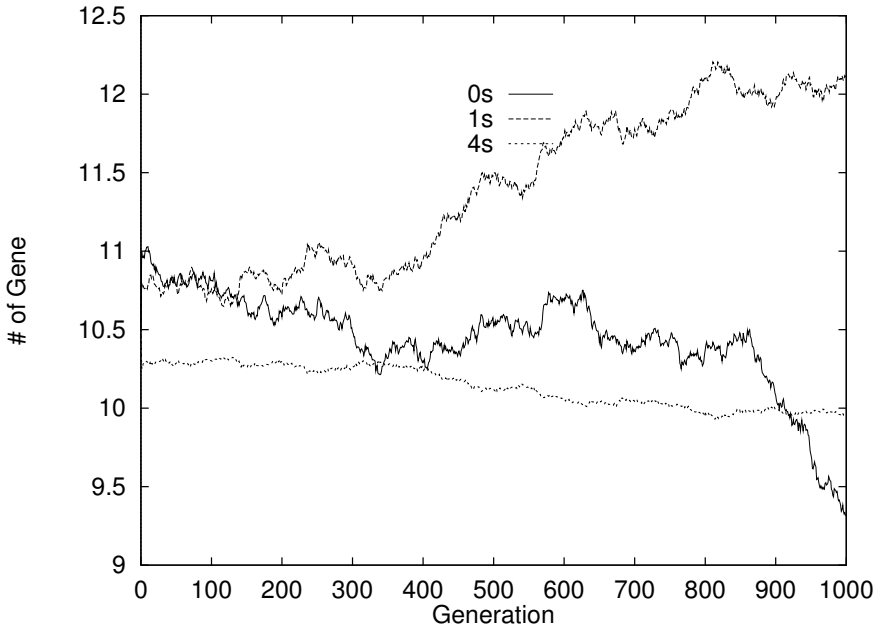

Figure 16.2. Number of 'genes' (0's, 1's or 4's) of each type when using proportional crossover. The number of 1's increases slowly, with a corresponding decrease in 4's.

search naturally stumbles upon the larger solutions and growth occurs. For this experiment this assumption certainly holds. (Consider a solution such as 010401, an infinite number of larger solutions of the same fitness can be made by adding 0's and, to a lesser extent, by replacing 4's with 1's. However, there are only a few smaller solutions with the same fitness.) The fact that growth does not occur with proportional crossover casts considerable doubt upon the drift hypothesis.

Gene Choice

Although proportional crossover does not lead to growth, the pressure for robust solutions still influences evolution. gene choice Given that the size of the crossed regions is proportional to the length of the chromosome a more robust solution can still be achieved by limiting the importance of the "genes" exchanged during crossover. Clearly 1's represent a less important gene than 4's as an extra or missing 1 has a much less significant effect on fitness than an extra or missing 4. Thus, we expect to see an increase in the number of 1's and a decrease in the number of 4's.

Figure 16.2 shows the numbers of 0's, 1's and 4's when proportional crossover is used. As predicted there is a (slight) increase in the number of 1's and a decrease in the number of 4's. Although this difference is small, on average there

are two more 1's than 4's in the final generation, the difference in the final generation is significant at the 1% level (Student's 2-tailed, t-test, $t = 2.6$). This shows that there is pressure for more robust solutions with proportional crossover. The system is simply using a different (non-growth based) strategy to increase robustness.

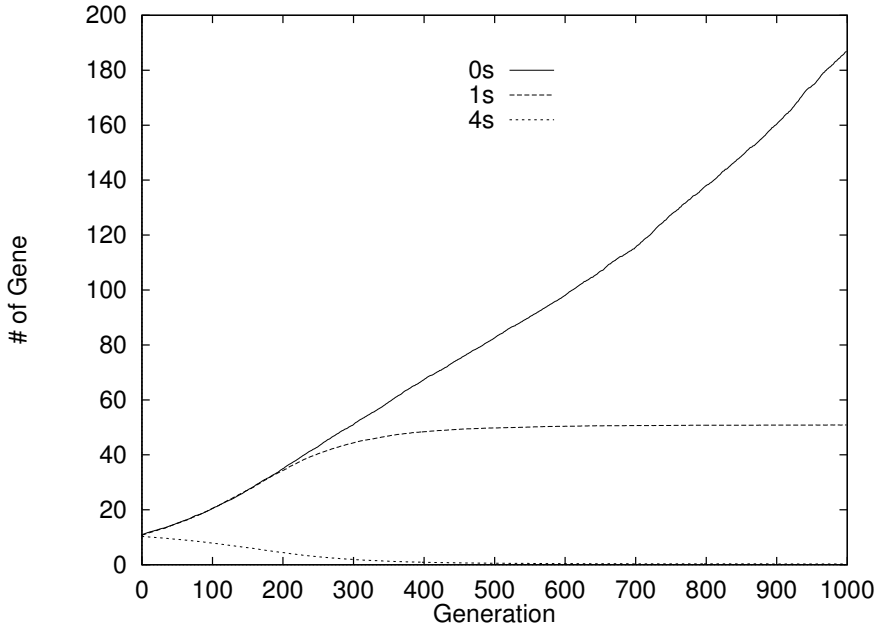

Figure 16.3. Number of "genes" (0's, 1's or 4's) of each type when using constant crossover. The number of 1's increases, with a corresponding decrease in 4's.

In fact the same robustness strategy is also adopted with constant crossover. Figure 16.3 shows the number of each type of genes with constant crossover; the results are much more pronounced than with proportional crossover. There is a pronounced shift away from 4's and toward 1's. In fact, by the final generation several of the trials showed no 4's at all. The reason for the shift from 4's to 1's is the same as with proportional crossover-individuals with relatively more 1's are more robust with respect to crossover. Possibly there is stronger pressure for the shift with constant crossover. Alternatively, constant crossover or the growth it creates may make it "easier" to exchange 1's for 4's; the pressure is the same in both cases, but the shift occurs more readily with the larger individuals.

As expected Figure 16.3 shows that the growth observed with constant crossover is primarily created by an increase in 0's. Thus, with constant crossover two evolutionary strategies are adopted to increase robustness, grow by adding 0's and increase the stability of the operative code by shifting from 4's to 1's.

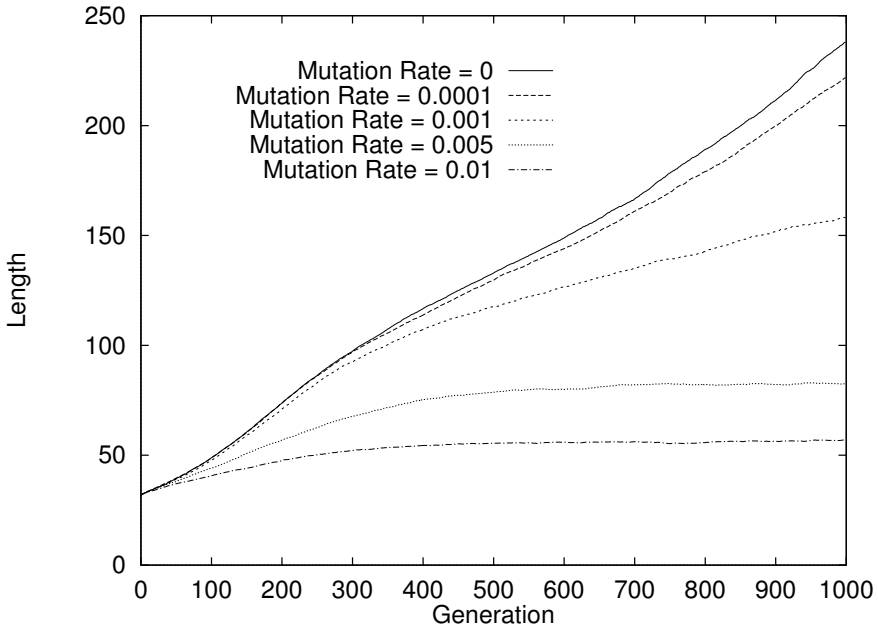

Figure 16.4. Average size (length) for individuals evolved using constant crossover with a variety of different mutation rates.

Mutation and Reduced Size

The previous experiments looked only at the effect of crossover. Next we examine the influence of mutation when constant crossover is used. Because the length of the individuals varies using a fixed mutation rate of 1/length is not possible. Instead several mutation rates are tried: 0.0001, 0.001, 0.005, 0.01. Each of these are the chance per "gene" of a mutation occurring. Figure 16.4 shows the results of these experiments on the growth rate of the evolving individuals. Table 16.2 shows the average sizes and standard deviations for the final generation. These differences are all significant at the 1% level.

Clearly the higher mutation rates limit growth. Again this can be explained in terms of robustness. Because the mutation rate is per gene (or per site), additional genes make the occurrence of a mutation event more likely. Because there are no inviable genes every mutation has an effect on fitness. Thus a short individual (one with fewer genes) is less likely to have a mutation event and is more robust with regards to mutation.

This increased robustness with regards to mutation for shorter individuals has to be balanced against the greater robustness for longer individuals with regards to crossover. Thus, as the mutation rate increases the ideal length for maximum robustness decreases. These results are particularly significant in

Table 16.2. Average sizes and standard deviations in the final generation (1000) when using constant crossover with a variety of different mutation rates.

Mutation Rate	Average Size	Std. Dev.
0.0	238.0	126.5
0.0001	221.9	90.36
0.001	158.2	38.20
0.005	82.50	7.967
0.01	56.93	5.253

that they appear to be the first time that pressure for robustness has been shown to lead to a smaller overall solution length.

4. Discussion and Conclusions

These results further support the idea the pressure for robustness is an important factor in the evolutionary process. The evolutionary process must balance this pressure for more robust solutions with the pressure for more fit solutions introduced by selection. Previous research has shown that robustness is often increased via the growth of inoperative and/or inviable code. The present results show that other evolutionary strategies are also adopted to increase robustness. In particular, we observed that increased robustness is achieved through gene choice, in this case replacing 4's with 1's. Robustness also increased by reducing the overall chromosome length (attenuation) in some cases.

Based on the results presented here it is possible to make predictions regarding the effects of different types and frequencies of operations with regards to "code" growth and gene selection. In general, changes that increase robustness (decrease the probability of fitness changes) will be favored.

These results show that increased growth will lead to increased robustness (and thus be evolutionarily favored) with respect to a given operator *if* that operator is applied with a per individual, and not a per site, probability. (Where "site" means nodes in a tree representation or loci in a linear representation.) I.e., when the operator is applied to individuals uniformly regardless of size. In these cases adding additional sites decreases the probability of any specific site being effected.

Examples of per individual, rather than per site operations (which encourage growth), include: GP crossover (one crossover per individual and the size of the crossed branches is not directly proportional to the number of nodes in the tree), mutation that is applied to a fixed number of sites per individual regardless of the individuals length, and the constant crossover operation described in this paper.

Examples of per site operations (which do not encourage growth) include: mutations with a per site probability (e.g. standard GA mutation) and the proportional two-point crossover described above (as individuals get longer-have more sites-the number of sites changed by crossover increases proportionally).

Even in cases where the operators are designed or chosen so that they will not encourage growth to increase robustness operators may have more subtle effects. E.g. these experiments demonstrated that pressure for robustness can effect "gene" choice.

The rules and examples presented in this paper can be used to predict the effect of novel operators on the evolutionary process and thus can improve our ability to design useful genetic operators. This is particularly beneficial to a practitioner who must design a novel representation and associated operators to solve a new problem. The rules described above can be used to design operators that guide evolution in a desired direction.

References

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

Igel C. and Chellapilla K. (1999). Investigating the Influence of Depth and Degree of Genotypic Change on Fitness. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pp 1061-1068, Orlando, Florida. Morgan Kaufmann.

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

Langdon, W. B. (1997). Fitness causes bloat: Simulated annealing, hill climbing and populations. Technical Report CSRP-97-22, The University of Birmingham, Birmingham, UK.

Langdon, W. B. (1999). Size fair and homologous tree genetic programming crossovers. In *Proceedings of the Genetic and Evolutionary Computation Conference 1999*, Banzhaf, W. *et al.* (Eds.), pp. 1092–1097. Morgan Kaufmann.

Langdon, W. B., Soule, T., Poli, R. and Foster, J. A. (1999). The evolution of size and shape. In *Advances in Genetic Programming III*, Spector, L. *et al.* (Eds.), pp. 163–190. Cambridge, MA: The MIT Press.

Luke, S. (2000c). Code growth is not caused by introns. In *Late Breaking Papers, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pp. 228–235.

McPhee, N. F. and Miller, J. D. (1995). Accurate replication in genetic programming. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, Eshelman, L. J. (Ed.), pp. 303–309. Morgan Kaufmann.

Nordin, P. (1997). *Evolutionary Program Induction of Binary Machine Code and its Application*. Muenster: Krehl Verlag.

Nordin, P. and Banzhaf, W. (1995). Complexity compression and evolution. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, Eshelman, L. J. (Ed.), pp. 310–317. San Francisco, CA: Morgan Kaufmann.

Nordin, P., Banzhaf, W. and Francone, F. D. (1997). Introns in nature and in simulated structure evolution. In *Proceedings Bio-Computing and Emergent Computation*, Lundh, D., Olsson, B., and Narayanan, A. (Eds.), pp. 22–35. Springer.

Poli, R. and Langdon, W. B. (1997). A new schema theory for genetic programming with one-point crossover and mutation. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J Koza *et al.* (Eds.), pp 278–285. San Francisco, CA: Morgan Kaufmann.

Rosca, J. P. and Ballard, D. H. (1996a). Complexity drift in evolutionary computation with tree representations. Technical Report NRL5, University of Rochester, Computer Science Department, Rochester, NY, USA.

Smith, P. and Harries, K. (1998). Code growth, explicitly defined introns, and alternative selection schemes. *Evolutionary Computation 6(4):* 339–360.

Soule, T. (1998). *Code Growth in Genetic Programming*. PhD thesis, University of Idaho, University of Idaho.

Soule, T. (2002). Exons and code growth in genetic programming. In *Genetic Programming, 5th European Conference, EuroGP 2002*, J. Foster *et al.* (Eds.), pp. 142–151.

Soule, T. and Foster, J. A. (1997). Code size and depth flows in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza *et al.* (Eds.), pp. 313–320. San Francisco, CA: Morgan Kaufmann.

Soule T. and Foster J. A. (1998). Removal Bias: a New Cause of Code Growth in Tree Based Evolutionary Programming. In *IEEE International Conference on Evolutionary Computation,* pp. 178-186, Anchorage, Alaska, IEEE Press.

Soule, T. and Heckendorn, R. B. (2002). An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines 3(3):* 283–309.

Soule, T., Foster, J. A. and Dickinson, J. (1996). Code growth in genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. Koza *et al.* (Eds.), pp. 215–223. Cambridge, MA: MIT Press.

Soule, T., Heckendorn, R. and Shen, J. (2002). Solution stability in evolutonary computation. In *Proceedings of the 17th International Symposium on Computer and Information Systems*, I. Cicekli *et al.* (Eds.), pp. 237–241.

Streeter, M. J. (2002). The root causes of code growth in genetic programming. In *Genetic Programming, 6th European Conference, EuroGP 2003*, C. Ryan *et al.* (Eds.), pp. 443–454.