Growth of Self-Canceling Code in Evolutionary Systems

Xue Zhong
Department of Mathematics
University of Idaho
Moscow, ID, 83844-1103
xue2500@uidaho.edu

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

ABSTRACT

This research examines the behavior of inoperative code (introns) in the evolution of genetically robust solutions. Genetically robust solutions are solutions that are less likely to be degraded by genetic operators, such as crossover. Previous work has shown that there is significant evolutionary pressure in favor of genetically robust solutions and that evolving programs adopt a number of strategies to increase genetic robustness, notably an increase in inoperative 'genes' (individual genetic units that don't influence fitness) and a preference for 'genes' with a relatively small effect on fitness.

Here we examine the role of genes that cancel each other out. We find that allowing such 'canceling genes' leads to an overall increase in the rate of code growth, both through the inclusion of self-canceling code and through a general increase in introns. Finally, we find that the evolution generally follows a two-step process. Initially the operative code evolves rapidly to achieve a (near) optimal fitness. Then, the inoperative code begins to evolve most rapidly to increase robustness. In an extreme case of a problem that can be solved with no operative genes, individuals evolve by losing all operative genes and then losing all inoperative genes.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms

Genetic robustness

Keywords

Bloat, code bloat, robustness, introns, exons

*Address correspondence to Dr. Soule. This publication was made possible by NIH Grant P20 RR16448 from the COBRE Program of the National Center for Research Resources

GECCO'06 July 8-12, 2006, Seattle, Washington, USA

1. INTRODUCTION

Genetic robustness is a measure of the invariance of fitness when an individual undergoes genetic changes [3]. Individuals with a smaller expected variance are more robust than individuals with a larger expected variance. The definition is based on fitness (rather than behavior, performance, phenotype, or another related measure) because in genetic programming (GP) it is typically the fitness of the offspring that determines the offspring's probability of survival.

Research has shown that there is significant evolutionary pressure to evolve genetically robust solutions and that this pressure has a significant effect on the course and trajectory of evolution. The most obvious and well documented effect is the phenomenon of code growth (or bloat) in genetic programming (GP) [5, 2, 13, 11, 22, 10, 13, 14, 12]. Recently it has been shown that the pressure for robustness also influences which genes dominate the evolutionary process. Given multiple genes with similar affects those genes that produce more resilient individuals will tend to replace genes that produce less resilient individuals [19, 24, 18, 1]. Finally, several experiments have shown that pressure for robustness may cause evolution to favor individuals with lower fitness, but higher robustness than individuals with higher fitness and lower robustness [19, 20].

All three of these effects (code growth, gene choice, and preference for less fit, but more robust, solutions) are significant to the GP practitioner. Thus, it is important to understand how pressure for robustness influences the course and trajectory of evolution. In particular, to understand the different strategies that are adopted by evolving individuals to increase their robustness. Because the most obvious and arguably the most significant effect is code bloat, considerable research has been devoted to understanding the causes of code bloat. Recent research has focused on the role of different types of genes and genetic structures in contributing to bloat. In this paper we examine the role of 'canceling genes', genes that cancel each other out in the evolution of robustness and bloat. Our goal is to better understand how different types of code contribute to bloat and thereby to develop representations that eliminate or at least reduce the biggest contributors.

2. BACKGROUND

Code bloat is a rapid increase in code size that does not result in fitness improvements. The extra code usually consists of introns (code that does not contribute to the program's fitness). Early research into the code bloat phenomenon focused on the role of introns - code that did not effect fit-

ness. However, Luke [10] has argued that introns themselves are not the cause of code growth. Smith and Harries have shown that growth can occur in code that does influence fitness (exons) if the exons only have a negligible effect on performance [17]. More recently, Soule has shown that code growth can occur even with exons that have a significant impact on the programs' fitness [18]. Research by Besetti and Soule has shown that in GP the number of different functions increases at significantly different rates [1]. For example, in a typically symbolic regression problem the number of division functions increased twice as rapidly as the next most common function (multiplication).

This research makes it clear that different types of code do contribute to code bloat. Thus, the next question is whether all types of code contribute equally. Several taxonomies of GP code have been previously proposed. Nordin, Francone, and Banzhaf introduced a taxonomy of intron types for their linear GP system with 5 code categories based on whether changing the given code could affect the program's behavior [14]. Smith and Harries adopted this taxonomy for tree structured GP [17]:

1. Type 1: changes to the code region cannot change the program's behavior for any input in the problem domain.

2. Type 2: changes to the code region cannot change the program's behavior for any input in the training cases.

3. Type 3: code does not contribute to fitness and replacing the code region with a no-op will not change the program's behavior for any input in the problem domain (replacing the code with something other than a no-op could change the behavior).

4. Type 4: code does not contribute to fitness and replacing the code region with a no-op will not change the program's behavior for any input in the training cases (replacing the code with something other than a no-op could change the behavior).

5. Type 5: code has a negligible effect on fitness.

Soule introduced a general taxonomy for code types [21]:

1. Operative code: code that effects fitness, changes to the code are likely to change fitness.

2. Inoperative code: code that does not effect fitness (code that could be replaced by a no-op without changing fitness), but that is likely to effect fitness if modified.

3. Inviable code: code that cannot effect fitness even if changed.

4. Viable code: code that could effect fitness if changed.

Note that under this taxonomy inviable code is a proper subset of inoperative code and operative code is a proper subset of viable code.

A typical example of Type 1 or inviable code is the code labeled *SUBSECTION* in the following code fragment:

IF(FALSE)THEN(*SUBSECTION*)

The subsection is inviable because it is never executed.

Table 1: Summary of the evolutionary algorithm parameters.

Objective	Target (see text) = 50
Integer values	0, 1, 4, -1, -4
Population Size	500
Crossover probability	0.9
Mutation probability	0
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 type	Constant (see text)

A typical example of type 3 or viable, but inoperative code, is the expression $+(Y-Y)$ in the following code fragment:

$$X + (Y - Y)$$

The section $+(Y-Y)$ has no effect on fitness because the two Y's cancel each other out. Replacing $+(Y-Y)$ with a no-op will not effect fitness, but any other change to the code section is likely to have an effect. A less typical example of this type of code is:

$$Y + (\ldots) - Y$$

Again the two Y's cancel each other out and do not contribute to fitness.

The protective value of Type 1 or inviable code is obvious; changes to this code will not effect fitness. Thus, it is a likely candidate for causing bloat. The protective value of Type 3 - viable, inoperative code - is less clear. Moving whole sections of this type of code will not effect fitness, e.g. inserting or removing $+Y-Y$ can't change the individual's fitness, but changes within a section could change the individual's fitness.

Research has shown that operative code, the code that actually contributes to fitness, is generally not a major contributor to bloat. Most bloated code consists of inoperative code including types 1, 2, 3, 4, and 5. The goal of this paper is to determine whether type 3 and 4 code (viable, inoperative code), contributes significantly to bloat. In particular there are two questions we would like to answer:

1. Does type 3 and 4 code (viable, inoperative) bloat?

2. If type 3 and 4 code bloats, does it increase the overall growth rate? I.e. is the bloating of type 1 and 2 code additive with bloating by type 1 and 2 (inviable) code?

Specifically we examine the role of canceling genes, e.g. +4 and -4 when used with addition.

3. EXPERIMENT

To answer the questions proposed above we need a GP system that gives the user control over the types of code that are possible. Soule designed an experiment that uses strings of variable length to examine the code growth of different code types [20, 19]. In that experiment, the goal was

to find a set of integers that sum to a given target value T. The allowed integers were 0, 1, 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 (the individual's value) and the target value, i.e. $fitness = |value - T|$. For example, the individual 10401 has value $1 + 0 + 4 + 0 + 1 = 6$ and fitness $|6 - T|$. Clearly a lower fitness is better.

The advantage of this very simple evolutionary representation is that the types of code are fixed. 1s and 4s are operative (exons) as they always affect fitness. 0s are inoperative (introns). If mutation is not used then 0s approximate inviable code (types 1 and 2) because most crossover operations will simply exchange 0s. (Note that for this system types 1 and 2 are equivalent, as are types 3 and 4, because there is only one fitness case.)

To approximate tree based crossover in this linear system a special form of crossover tailed for variable length strings known as constant crossover was used. In this crossover method, the length of crossed region l 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. 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, 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 parent's size.

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 [4, 16]. Larger branches are exponentially less likely to be chosen for crossover. In practice, GP usually leads to randomly shaped trees rather than full trees [15, 7]. However, the distribution of crossover points still heavily favors small branches [23] and using the 90/10 rule (choosing leaf nodes for crossover only 10% of the time) only slightly shifts the distribution towards larger branches [23]. Thus, the distribution of crossover sizes with constant crossover is comparable to those seen in tree based GP; both emphasize exchanging small branches.

To introduce the possible of type 3 code (inoperative, but viable) we only need to introduce two new (negative) integers: -1 and -4. The other parameters are given in Table 1.

4. RESULTS

Figure 1 shows the total code growth for the 'gene' sets (0, 1, 4) and (-4, -1, 0, 1, 4). The populations start with individuals of the same average size (i.e. individuals in both initial populations are generated using the same algorithm), but an overall increase in the rate of code growth is observed with the set (-4, -1, 0, 1, 4). Two factors contribute to the larger average size. First, the population with 'canceling genes' (-1 and -4) jumps from the (average) size of 30 to 50 within a few generations. Second, the overall growth rate

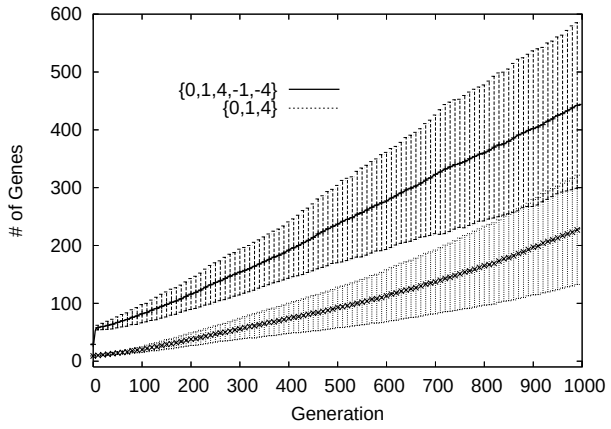

Figure 1: Comparison of code growth with and without canceling genes. Inclusion of -1s and -4s leads to an overall increase in the rate of code growth

when the canceling genes are included is more rapid. The first factor, the rapid initial jump, is easy to explain. Without canceling genes the average individual value in the initial population is roughly 50, the same as the target value. With canceling genes the average value in the initial population is zero, because an individual is equally likely to contain 1s or -1s and 4s or -4s. Thus, individuals that have above average numbers of positive genes, which tend to be larger overall, are selected and there is a rapid jump in average size. Once the target value is reached (within a few generations) this pressure is removed and size ceases to grow as rapidly. The second factor, steady growth that is more rapid than in the case where the canceling genes are absent has two possible causes. Either the inoperative pairs (1 and -1, and 4 and -4) are being preferentially included, increasing the overall growth rate, or the number of 0s is increasing more rapidly when -1 and -4 are included in the set of allowed values. These possibilities are explored further below.

Figure 3 show the average number of 0s, 1s, and 4s when canceling genes are not included and Figure 2 shows the average number of 0s and the average number of all 1s (positive and negative) and the average number of all 4s (positive and negative) when the canceling genes are included. Figure 3 shows a rapid increase in 0s and a replacement of 4s with multiple 1s (i.e. the number of 4s tends towards zero and the number of 1s increases towards the target value). Figure 2 shows a similar, but slightly more rapid, increase in the number of 0s. This difference is significant (Student's t-test, t = 3.48, P < 0.01) meaning that the inclusion of canceling genes significantly increases the rate of growth of 0s.

Figure 2 shows that the number of positive and negative 1s increases throughout all 1000 generations. The total number of positive and negative 1s in generation 1000 (Figure 2) is significantly greater the number of positive 1s alone (Figure 3) (Student's t-test, t = 24.6, P < 0.01). Similalry, the number of positive and negative 4s decreases much more slowly than the number of positive 4s when negative 4s are not included (Figure 3) (Student's t-test, t = 11.2, P < 0.01).

Thus, the more rapid growth when canceling genes are included does have two causes. One, inoperative code, rep-

resented by -1,1 pairs, increases significantly. Two, the amount of inviable code, represented by 0s, increases more rapidly. The first cause clearly shows that evolutionary systems will increase inoperative, but viable code. The second cause implies that there are synergistic relationships between different code types than can further increase growth rates.

Figure 4 shows the average number of each gene type (0, -4, -1, 1, 4). It shows almost parallel code growth between each pair of canceling genes (1 and -1, or 4 and -4). Close inspection shows that there is a slight increase in the gap between the small effect canceling genes (1 and -1) and a slight reduction of the gap between the large effect canceling genes (4 and -4). This appears to confirm that there is a preference for genes with a relatively small effect on fitness.

The affect of canceling genes on the evolutionary dynamics is further examined by excluding 0s from the 'gene' set. Figure 5 shows the average number of each gene type in this case. Comparing Figures 5 and 4 reveals a number of similarities and differences. In both figures the number of small effect genes (1s and -1s) increases rapidly, but the growth is significantly more rapid when 0s are not included (Student's t-test, $P < 0.01$ for both positive and negative 1s). This suggests that in the absence of inviable code the evolutionary process grows the available inoperative, but viable, code more rapidly to compensate. In the absences of 0s the number of 4s and -4s increases, but much less rapidly than for the 1s and -1s. The number of 4s and -4s in generation 1000 is significantly different depending on whether or not zeros are included (Student's t-test, $P < 0.01$ for both positive and negative 4s). Towards the end of the run the number of 4s and -4s does level off and it appears that given a longer run they might decrease. This further supports the hypothesis that the evolutionary process is favoring individuals using the small effect genes.

All the results above are from a target value of 50 (T=50). An interesting case to explore is T=0. In this case no operative genes are required to achieve the target value. We found that for T=0 solutions eventually lose all operative genes (rather than, for example, keeping canceling pairs). Even more interesting, after all operative genes are lost, all inoperative genes are also lost, resulting in individuals of size 0.

Figure 6 shows the number of 0s for three small target values (0, 1, and 5). In all three cases all operative genes are eventually lost. In the case of target values of 1 and 5 this means that the population converged on solutions that were sub-optimal (1 and 5 respectively - 0 is optimal). The number of 0s behaves differently across the three cases. The number of 0s either goes to 0, stabilizes at some small value, or grows, depending on the target value.

Two factors working together explain this phenomenon. First, short 'gene' sequences evolve during early generations. Under a small (absolute) target value, a sequence of a few operative genes is sufficient to achieve a high fitness. Thus, since it takes time to evolve robustness, most surviving sequences during the early generations are short. Second, in this paper the generic operation only involves crossover, there is no mutation. This makes an accidental loss of a certain 'gene' type in the population a permanent loss. As such permanent losses accumulate all operative genes are eventually lost.

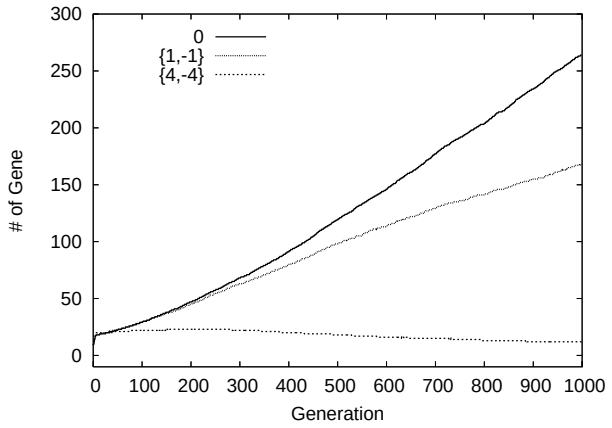

Figure 2: Average code size with canceling genes (paired). 0s increase most rapidly, but some inoperative pairs (1,-1) also increase; other inoperative pairs (4,-4) decrease slightly.

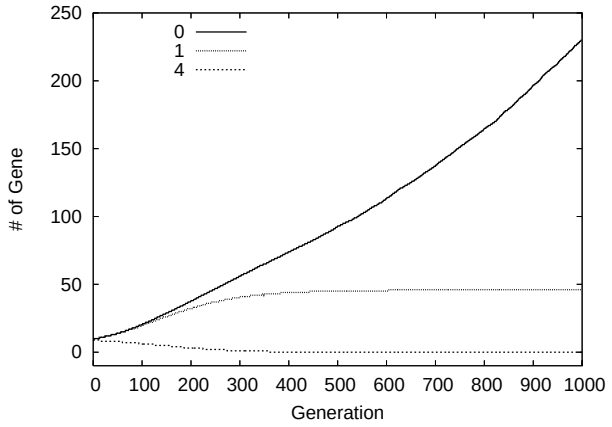

Figure 3: Average code size without canceling genes. The number of 0s exhibits the most rapid growth; the number of 1s rises to approximately 50 to achieve the target value (T=50), the number of 4s drops to near zero.

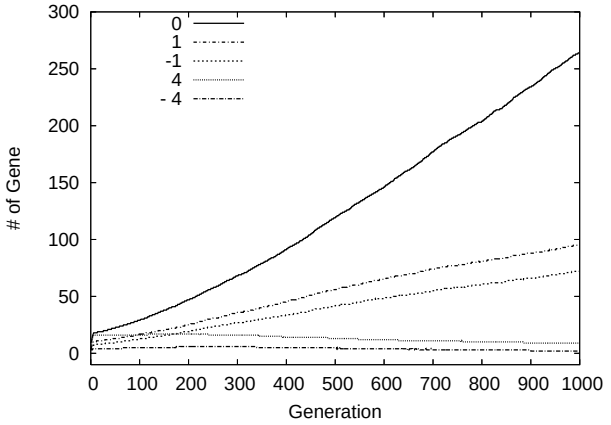

Figure 4: Average code growth with canceling genes (unpaired). Canceling genes (1 and -1, or 4 and -4) exhibit almost parallel growth. The slight enlargement of the gap between 1 and -1, and the slight reduction of the gap between 4 and -4 implies a preference for genes with a smaller effect on fitness (e.g. 1s over 4s).

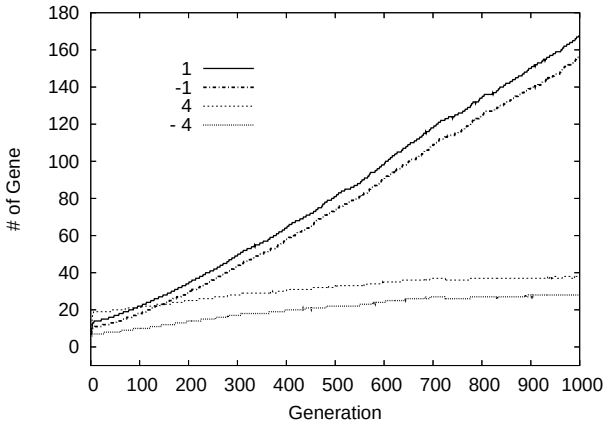

Figure 5: Average code growth with canceling genes, but without 0s. The number of each type of gene changes in parallel. There is fairly rapid growth in the number of 1,-1 pairs indicating that they are being used to generate code growth.

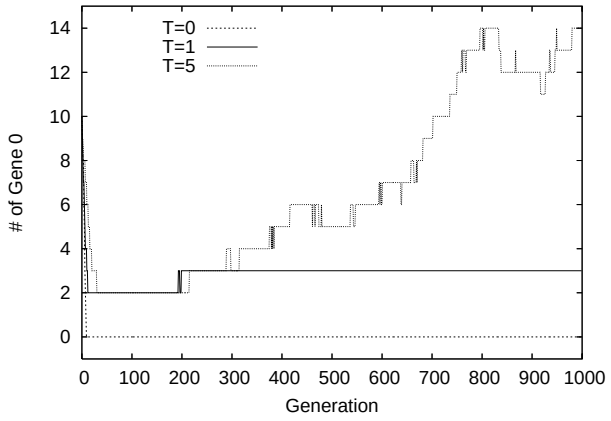

Figure 6: Number of 0s for small target values when all five gene types (-4, -1, 0, 1, 4) are included. In each case, the population converges to individuals with no operative genes (e.g. all 1s, -1s, 4s, or -4s are lost) (data not shown). For T=0, solutions evolves by first losing all operative genes and then losing all 0s; for T=1 or 5, evolved solutions lose all operative genes but maintain some 0s.

5. CONCLUSION

The results show that evolutionary systems will increase the overall amount of inoperative, but viable, code (represented in this research by pairs of genes whose effects cancel each other out: 1 and -1, and 4 and -4) in evolving individuals. Interesting the degree to which this occurs is dependent on the potential contribution of the genes to the overall fitness. Genes with a *s*maller potential contribution increase the most rapidly. Specifically, in these experiments the number of (1, -1) pairs increased quite rapidly, whereas the number of (4, -4) pairs actually decreased slightly.

The increase in inoperative pairs occurs even when inviable code (represented by 0s) was present, although the number of 0s increased more rapidly. Further, when inviable code was not present the inoperative pairs increased more rapidly than when it was present. Thus, the results indicate that the evolutionary process is using whatever type of code, inviable or inoperative, is available to increase code size.

Recently Langdon and Banzhaf showed that repeated sequences commonly evolve in both linear and tree based representations [9, 8]. For tree based representations they found many repeated subtrees that were either syntactically identical or syntactically different, but that produced correlated answers - suggesting semantic similarity. They conclude that 'where bloat is possible, GP will generally evolve programs containing copious repeated patterns' [9]. One assumption is that repeated patterns provide robustness via redundancy - if one copy of the structure is removed another copy is present to fulfill its role. Pressure for redundancy as a means to increase robustness has been observed by Krackauer and Plotkin [6].

Our results suggest an alternative explanation, that the repeated structures they observed were evolving to cancel each other out. For example, in the code $(X+Y)-(X+Y)$ the structure $(X+Y)$ is repeated, but the structure contributes to robustness by canceling itself out and producing

an inoperative code section, rather than acting redundant code. If this is the source of the code redundancies observed by Langdon and Banzhaf it suggests that evolution may lead to fractal-like code structures in which structures are recurrently constructed from canceling copies of canceling substructures, e.g., structures like $((X+Y)-(X+Y))-((X+Y)-(X+Y))$.

6. REFERENCES

[1] S. Besetti and T. Soule. Function choice, resiliency and growth in genetic programming. In *GECCO*, pages 1771–1772, 2005.

[2] T. Blickle and L. Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation*, pages 33 – 38. Saarbrucken, Germany: Max-Planck-Institut fur Informatik, 1994.

[3] A. G. M. DeVisser, J. Hermission, G. P. Wagner, L. A. Meyers, H. Bagheri-Chaichain, J. L. Blanchard, L. Chao, J. M. Cheverud, S. F. Elena, W. Fontana, G. Gibson, T. F. Hansen, D. Krakauer, R. C. Lewontin, C. Ofria, S. H. Rice, G. von Dassow, and A. Wagner. Perspective: Evolution and detection of genetic robustness. *Evolution*, 57:1959–1972, 2003.

[4] C. Igel and K. Chellapilla. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference 1999*, pages 1061–1068. Morgan Kaufmann, 1999.

[5] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press, 1992.

[6] D. C. Krackauer and J. B. Plotkin. Redundnacy, antiredundancy, and the robustness of genomes. *PNAS*, 99:1405–1409, 2002.

[7] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference 1999*, pages 1092–1097. Morgan Kaufmann, 1999.

[8] W. B. Langdon and W. Banzhaf. Repeated sequences in linear GP genomes. In M. Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 26 July 2004.

[9] W. B. Langdon and W. Banzhaf. Repeated patterns in tree genetic programming. In *EuroGP*, pages 190–202, 2005.

[10] S. Luke. Code growth is not caused by introns. In *Late Breaking Papers, Proceedings of the Genetic and Evolutionary Computation Conference 2000*, pages 228–235, 2000.

[11] N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 303–309. San Francisco, CA: Morgan Kaufmann, 1995.

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

[13] P. Nordin and W. Banzhaf. Complexity compression and evolution. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 310–317. San Francisco, CA: Morgan Kaufmann, 1995.

[14] P. Nordin, W. Banzhaf, and F. D. Francone. Introns in nature and in simulated structure evolution. In D. Lundh, B. Olsson, and A. Narayanan, editors, *Proceedings Bio-Computing and Emergent Computation*, pages 22–35. Springer, 1997.

[15] R. Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and mutation. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. R. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 278–285. San Francisco, CA: Morgan Kaufmann, 1997.

[16] J. P. Rosca and D. H. Ballard. Complexity drift in evolutionary computation with tree representations. Technical Report NRL5, University of Rochester, Rochester, New York, 1996.

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

[18] T. Soule. Exons and code growth in genetic programming. In J. A. Foster, E. Lutton, J. F. Miller, C. Ryan, and A. Tettamanzi, editors, *Genetic Programming, 5th European Conference, EuroGP 2002*, pages 142–151, 2002.

[19] T. Soule. Operator choice and the evolution of robust solutions. In R. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practice*, pages 257–270, 2003.

[20] T. Soule. Resilient individuals improve evolutionary search. *Artificial Life*, 12:1:17–34, 2006.

[21] T. Soule and J. A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *ICEC 98: IEEE International Conference on Evolutionary Computation 1998*, pages 781–786. IEEE Press, 1998.

[22] T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. R. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223. Cambridge, MA: MIT Press, 1996.

[23] T. Soule and R. Heckendorn. An analysis of the causes of code growth in genetic programmin. *Genetic Programming and Evolvable Machines*, 3:283–309, 2002.

[24] T. Soule, R. Heckendorn, and J. Shen. Solution stability in evolutonary computation. In I. Cicekli, N. K. Cicekli, and E. Gelenbe, editors, *Proceedings of the 17th International Symposium on Computer and Information Systems*, pages 237–241, 2002.