Exploiting Disruption Aversion to Control Code Bloat

Jason Stevens
University of Idaho
Moscow, Idaho, USA
stev0931@uidaho.edu

Robert B. Heckendorn
University of Idaho
Moscow, Idaho, USA
heckendo@uidaho.com

Terry Soule
University of Idaho
Moscow, Idaho, USA
tsoule@uidaho.com

ABSTRACT

The authors employ multiple crossovers as a novel natural extension to crossovers as a mixing operator. They use this as a framework to explore the ideas of code growth. Empirical support is given for popular theories for mechanisms of code growth. Three specific algorithms for multiple crossovers are compared with classic methods for performance in terms of fitness and genome size. The details of the performance of these algorithms is examined in detail for both practical value and theoretical implications. The authors conclude that multiple crossovers is a practical scheme for containing code growth without a significant loss of fitness.

Categories and Subject Descriptors

Genetic Programming []

General Terms

Algorithms, Experimentation, Performance, Theory

Keywords

code bloat, code growth, effective fitness

1. INTRODUCTION

Code growth (or bloat) is the tendency of variable-length individuals undergoing simulated evolution to grow progressively larger without corresponding improvements in fitness. The increase typically consists of genetic information that has little or no functional value. Code growth was originally observed in genetic programming models - hence the term code growth - but current research suggests that it can occur in any variable-length evolutionary model [7, 12]. Code growth can cause a number of serious problems for evolutionary optimization. Larger individuals often require significantly more memory for storage. They also use significantly more computational time for evaluation and manipulation of the data structures. Finally, changes to larger individuals (e.g. crossover and mutation) are less likely to have functional effect, potentially stalling the evolutionary process. Thus, fully understanding the causes of growth and finding effective and efficient solutions to prevent code growth that does not interfere with the discovery of better individuals is an important goal. This research hopes to make progress in these directions.

A leading explanation of code growth is that the growth occurs to protect individuals against the destructive effects of crossover, thereby making them more **resilient** [1,7,9]. By resilient individuals, we mean individuals whose children are more likely than average to have the same or better fitness than their parent. It has recently been suggested that code growth should only be evolutionarily favored when the increased growth actually leads to increased resiliency and further that it generally depends on how the variation operators (e.g. crossover and mutation) are applied [14]. In particular, it was hypothesized that, for operators applied with a per-individual probability, larger programs will be more resilient and growth will be encouraged; whereas, for operators applied with a per-site probability, larger programs will not be more resilient and code growth will not be encouraged [14]. As an example of this difference, crossover is typically applied with a probability (often 1) **per individual**, encouraging code growth, while mutation is typically applied with a probability **per site**. This means, even with per site mutation, code growth will occur.

The results presented in this paper have both theoretical and practical ramifications. Theoretically, they support the popular hypotheses that code growth does not occur when crossover is applied at a per-site basis (e.g. longer individuals are subjected to more crossovers), provided that mutation is also on a per-site basis. They support that code growth is protective and that, in some cases, the active code (exons) of the genome can be encouraged to decrease in size. From a practical standpoint, these results suggest that code growth can be controlled by changing the way in which the probability of applying the mixing operators is computed.

2. BACKGROUND

The tendency of programs generated using genetic programming (GP) to grow without corresponding increases in fitness (code bloat) is well documented in the GP literature [1,2,6,7,9,12,16,19]. Growth has also been demonstrated in non-tree based evolutionary paradigms [8–10,14]. Current research on code growth in GP strongly suggests that it will occur in any evolutionary technique which uses variable-size

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'05, June 25–29, 2005, Washington, DC, USA.
Copyright 2005 ACM 1-59593-010-8/05/0006 ...$5.00.

representations [7, 12] and Langdon has shown that growth can occur in non-population based search techniques [3].

In roughly-equivalent theories, Nordin and Banzhaf, McPhee and Miller, and Blickle and Thiele have argued that code growth occurs to protect programs against the destructive effects of crossover [1, 7, 9]. Very generally, the theory is that, as the ratio of the size of the functionally important regions (exons) to total genome length decreases, the probability that a crossover will affect (and in particular damage) functionally important regions also decreases. Several studies have confirmed that crossover is much more likely to decrease fitness than to increase fitness (destructive crossovers) [5, 11, 17, 19] and, in most evolutionary models, different regions within an individual's 'genome' have different functional importance. The extreme example is inviable regions [13, 15]: regions that, even if changed, can not have an effect on the individual's fitness. There is generally a limit to how small the functionally important regions of a genome can get without harming fitness. So, in variable-length individuals, the ratio is manipulated primarily by increasing the length of regions of the genome that are functionally less important.

However, a decrease in the ratio of the size of functionally important regions to total genomic length is only protective if the probability of changing any given site in the genome (e.g. bit in a binary GA, node in a tree based GP, etc.) is independent of the overall genome length. On the contrary, if the probability of changing a particular site is proportional to the length of the genome, then adding less important regions simply increases the overall probability of change.

In particular, in an evolutionary system, mutations are typically applied with a per-site probability. Thus, the probability of changing a functionally important site is independent of the overall length of the genome. It has been confirmed that mutations applied with a per-site probability do not encourage growth and can even discourage it [14]. However, if the number of mutations per individual is fixed, meaning that as the individual grows longer the probability of any specific site being mutated decreases, then growth is encouraged [17].

Similarly, in tree-based GP, crossover encourages growth because the probability of a node being affected by crossover decreases as the size of the tree increases. Each individual is subjected to one crossover per generation or iteration regardless of the individual's size and the average size of the crossed branches (i.e. the average number of nodes affected by crossover) is generally independent of the individual's size. It has been shown that, in models where the number of sites affected by crossover is a function of the individual's size, growth is not encouraged [14].

Previous experiments have explored adjusting mutation rates with size but have not adjusted crossover rates with size. If larger individuals would be subjected to more crossovers, eliminating the protective resilience of a large total genome size, code growth might not occur. Previous work reviewed above suggests that, in this case, growth will not be encouraged.

The experiments in this paper are designed to explore this hypothesis. In addition to examining a significant question regarding the causes of code growth, the results of these experiments may demonstrate a novel, and in some sense, more natural solution to the problem of code growth.

More specifically, the fact that code bloat may be a reaction to disruptive pressures suggests that we may be able to use this reaction to contain code bloat. Since it is not possible to remove the disruptive features of crossover and mutation, it might be possible to have the algorithm avoid bloated entries if disruption increases by length. Therefore, we propose to vary the intensity of disruption by varying the number of crossovers in order to control growth but remain with useful GP.

1. We hypothesize that, if we increase disruption with size, that the code will reach a balance between defense and allowable disruption.

2. Furthermore, we hypothesize that, if the solutions can be found in the set of solutions whose length produces only relatively low disruption, the algorithm performance will not be hurt and answer quality in terms of length will improve.

In the next section, we will explain the test problems and algorithms we will use. This will be followed by a section discussing our results for each problem and some general observations.

3. THE TEST PROBLEMS

To test our hypotheses, we use two test problems and four different approaches to increase the disruptive affects of crossover with size of the genome. The test problems use radically-different representations, but both allow variable-size genomes. The four approaches include three ways to increase disruption and a fourth as a control.

The first problem is the 0-1-4's problem [18]. This problem is an idealized problem designed to easily test the insertion of introns represented by 0's. The gene is a variable-length string of 0's, 1's and 4's. The fitness is how close the sum of the values in the string comes to a given fixed value. In our case, the value was 100.

The second problem is a general, tree-based GP function regression problem. This provides an opportunity for a practical demonstration of code growth control. This is a more practical test problem than the first and can, in conjunction with the first, be used to suggest that the behaviors we document represent a general rule rather than just an artifact of a specific problem.

In our experiments, we will try four different approaches to increase the disruptive pressure on the population (also referred to as destructive crossover). In all cases, but the control, the total number of crossovers occurring during the mixing phase between two selected parents is dependent on the sum of the sizes of the parents involved. No attempt is made to prevent the areas altered by previous crossovers from being selected for alteration again in the same pairing. While the use of multiple crossovers may seem strange at first, it is a natural extension of a single crossover and is similar to the way mutation is used. To some degree, this simulates the varying number of crossover points that may occur by random processes in biological crossover.

In the **HardEdge** approach, a size parameter P and a crossover parameter c are used. If the average size of the two parents is α then the **total** number of crossovers performed, N_c, is

$$N_c = \begin{cases} 1 & \text{if } \alpha \leq P \\ 1 + c & \text{otherwise} \end{cases}$$

For a pressure point $P = 30$ and $c = 2$, one crossover would be performed when the average size of the parents was less than or equal to 30, while 3 crossovers would be performed if the average size was larger. Table 1 shows the number of crossovers by average size for a typical set of parameters P and c.

In the **SoftEdge** approach, a size parameter called the granularity G and a constant c are used. The total number of crossovers performed is:

$$N_c = \begin{cases} 1 & \text{if } \alpha < G \\ c\lfloor \alpha/G \rfloor & \text{otherwise} \end{cases}$$

As the length increases for the 0-1-4's problem(and as the number of nodes increases for the symbolic regression problem), a steadily-increasing signal of disruption is applied.

The **MultiEdge** approach is a cross between the soft and HardEdge. Parameters G, P, and c are chosen. The total number of crossovers performed is:

$$N_c = \begin{cases} 1 & \text{if } \alpha \leq P \\ c\lfloor \alpha/G \rfloor & \text{otherwise} \end{cases}$$

Table 1 shows some typical values of N_c for various example parameter settings and crossover schemes. In all non-control cases, the disruptive pressure can be increased by reducing the size of parameters P and G, where applicable, and increasing the size of c.

Table 1: The Number of Crossovers, N_c, Dictated by HardEdge ($P = 40, c = 2$)**, SoftEdge** ($G = 20, c = 1$)**, MultiEdge** ($P = 60, G = 20, c = 2$)

Size	HardEdge	SoftEdge	MultiEdge
10	1	1	1
20	1	1	1
30	1	1	1
40	1	2	1
50	3	2	1
60	3	3	1
70	3	3	6
80	3	4	8
90	3	4	8
100	3	5	10

4. RESULTS

We first present the results from the experiments on the 0-1-4's problem. Then, we present the GP problem and compare them.

4.1 The 0-1-4's Problem

In this problem, we have a variable-length gene composed of a string of 0's, 1's and 4's. The fitness function is:

$$\text{fitness} \;= 100 - |100 - \sum_{x \in \text{gene}} x|$$

The goal is to maximize the fitness. The maximum fitness is 100. This problem has the nice feature that the 0's are clearly introns, the 4's are an optimal choice for compact code, and the 1's allow for smaller steps in fitness so that the loss of a 1 is not as bad as a loss of a 4. For a control, we ran a GA on the problem with a single 2-point crossover for each mating. The details of the algorithm are described in the tableau in Table 2. It is important to note that the size of the crossover segment size for constant crossover is strongly biased to short lengths. This makes the average size of the crossover essentially a small constant and makes the amount of disruption similar, within a constant, in size to that seen in tree-based crossover.

For all the graphs that follow, the median of the population was chosen as the measure. We believe that this best removes aspects of outliers and allowed us to collect quartile statistics. However, the use of mean instead does not alter the appearance of the graphs to any great degree.

Figure 2 shows both the median size and fitness of the populations averaged over 100 runs for the 0-1-4's problem. Our control cases correspond to the work of Soule et al. [18]. We find, as did Soule et al., that crossovers that averaged a constant size encouraged code growth, while crossovers whose crossover segment grew proportionally showed little code growth. We also see that the rapidly growing code could afford to maintain a fitness close to maximum throughout the population, while damage from the proportional crossover caused less than perfect fitness in the populations.

SoftEdge, HardEdge, and MultiEdge algorithms were also run on the same problem with all the same conditions as for constant crossover, except that multiple crossovers were performed as described in the previous section. This increases the damage done, while preserving the exchange of genetic material as the mechanism. The parameters control the strength of the size-relative disruptive pressure. Figure 2 shows that, in all three cases, the size is limited to just below the size boundary where the increased number of crossovers begins, although SoftEdge had the unexpected property that individuals would go beyond a soft edge when it was not required to achieve good fitness only to return to that edge at some later point. That is, individuals would often approach a soft edge, pass it, and suddenly come back to it later.

On average, all three methods performed statistically better at limiting size than the proportional crossover control algorithm. The corresponding fitness graph in the same figure shows that fitness for the three methods is, on average, better in the population than with the proportional crossover. The data supports this as a true difference (the error bars do not intersect as the graph approaches 100 generations for GP or 2000 generations for GA); however, error bars are not included so that the graphs can remain readable.

In most trials, we ran with sufficient pressure to significantly slow growth. For SoftEdge and MultiEdge, the growth was stopped by the first increase (edge) in the number of crossovers, but occasionally the population moved beyond the first edge to the second edge before being forced back to the first edge. The result is that some runs have populations with a median size that is between the first and second increase in the number of crossovers.

For the 0-1-4's problem, the punishment in terms of numbers of extra crossovers necessary to control code growth was small. A c value of 2 or more was effective for HardEdge, while a c value of 1 or more was effective for SoftEdge and MultiEdge. In both fitness and code size, SoftEdge was less effective than either HardEdge or MultiEdge.

One of the more surprising results of these experiments relates to the concept of resilience, or **effective fitness** [9]. Effective fitness is the idea that, under the pressures of destructive operators, individuals are more fit if they have a

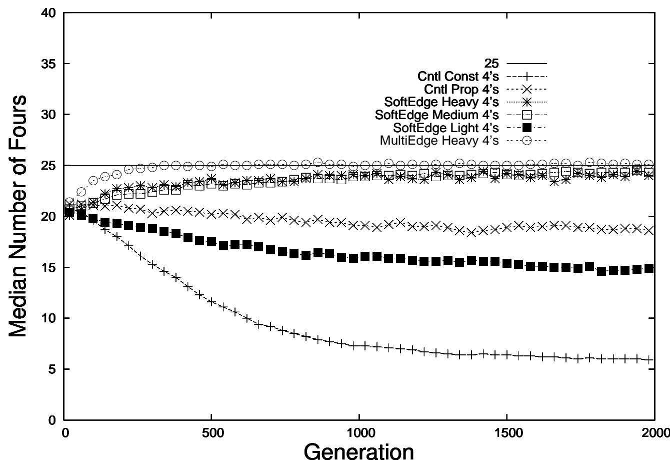

Figure 1: Average median (the average of the medians) number of 4's of individuals over 100 runs of the 0-1-4's problem. SoftEdge($G = 5, c = 1$), SoftEdge($G = 25, c = 1$), SoftEdge($G = 50, c = 1$), MultiEdge($P = 30, G = 20, c = 1$).

"higher chance of reproducing accurately" [4]. There are two processes that can improve effective fitness. The first is insertion of introns, or inactive code, which has a protective benefit for the exons. The second is the reduction in either the number of exons or the size of the active-code region. This is referred to as **active-code compression** [4]. In Figure 3, we see graphs of the average median number of 1s and 0s that make up the individuals in 100 sample run populations. Notice that, with moderate to heavy application of disruptive pressure, not only are the introns in the form of 0s removed, but so are the 1s which only increase the size of the active-code region of the individual. That is, we see a strong push for active code compression. Figure 1 shows the corresponding increase in 4s to create the needed fitness. All of this is in strong contrast to the fact that no such compression occurs for the control algorithms as we see in Figure 3.

We found that the parameters selected for destructive crossover are critical to the effectiveness of the GA. With parameters that apply too much pressure, we saw that the fitness is sacrificed to some degree in order to minimize size. With HardEdge, a pressure point set low to where individuals must pass it to achieve maximum fitness does not retard code bloat, since individuals simply pass the edge in order to achieve optimum fitness (once the hard edge about the pressure point is passed, there are no more edges to retard code bloat). Fortunately, the range of parameters that work well for this problem is large. Even using less-than-optimal parameters, we still see good results.

We also experimented with increasing the destructive crossover pressure by changing the parameters gradually during execution. We found the effects most encouraging. By gradually increasing pressure, we were able to completely remove all introns (0s) from populations when poor choices were made for initial parameters. From these results, we believe that destructive crossover may have practical applications for compressing solutions in addition to retarding code bloat.

We have seen in this section that with the 0-1-4's problem not only is code growth severely retarded while maintaining fitness, but that active code compression is observed. This problem had nice diagnostic features, but will this work in a classic genetic programming environment? In the next section, we pick a problem whose encoding and optimization algorithm are quite different to see if these observations might have general applicability.

4.2 The GP Problem

For this problem, we use a symbolic regression problem. The goal is to use a tree based GP to estimate a function based on a set of sample points. The function we targeted was $f(x) = x^2 - 3x + 2$ with sample points $\mathcal{S} = \{-4,\ -3,\ -2,\ -1,\ 0,\ 1,\ 2,\ 3,\ 4\}$. Each gene maps directly to an estimating function, $\widehat{f}$ over the same domain as the target function, f. The fitness is defined as:

$$\text{fitness} \;= \sum_{x \in \mathcal{S}} (f(x) - \widehat{f}(x))^2$$

The goal is to minimize the fitness. The minimum fitness is 0. Double-precision constants were used and crossover was the only evolutionary operator. We believe that it is the lack of mutation that accounts for why the GP was not able to do better for fitness (without mutations, the GP must manipulate available high-precision, non-integer constants to approximate integer constants).

Our control algorithm is a tree-based GP with a population size of 64. Crossover is a subtree exchange where the root of the subtree is uniformly selected from all nodes but the root. It is important to see that, in the majority of cases, the subtree is very small. This is comparable to the strong bias for short crossover segments in our constant crossover algorithm in the previous problem. The details of the GP implementation can be found in the tableau in Table 3. Finally, for practical reasons, we control runaway code bloat with a $64,000$ node allocation limit for the population. This is an average of 1000 nodes per individual. We will indicate in the text where this affects our results.

In Figure 4, we see that the control algorithm using a single crossover operation results in the expected runaway code growth. The SoftEdge algorithm, which produces a mild but ever-increasing resistance to code growth, seems to actually **accelerate** code growth! We noticed many occasions where light but increasing pressure appears to accelerates growth. We will investigate this more thoroughly in future work. The apparent flattening of these two curves in later generations is due to the node allocation clipping. In fact these curves, when unbounded, rapidly stall the program with huge trees.

For HardEdge and MultiEdge we see that code growth is contained and asymptotic to the edge imposed by the $P = 150$ for these tests. The accompanying fitness graph shows that, except for SoftEdge, the average median of the population is the same. Error bars would reinforce this idea, but were left off for clarity. The performance of SoftEdge is not as good on average.

Picking a strong enough disruptive pressure was important in getting the code growth contained in GP. This is demonstrated in Figure 5(Right). Here, we see the HardEdge algorithm applied with varying degrees of punishment. The more rapid the rise in number of crossovers with size, the stronger the disruptive pressure. Both algorithms which had insufficient pressure demonstrated strong growth. (The flattening of the highest curves is again due to the node allocation limit.) The curves for medium and heavy destruc-

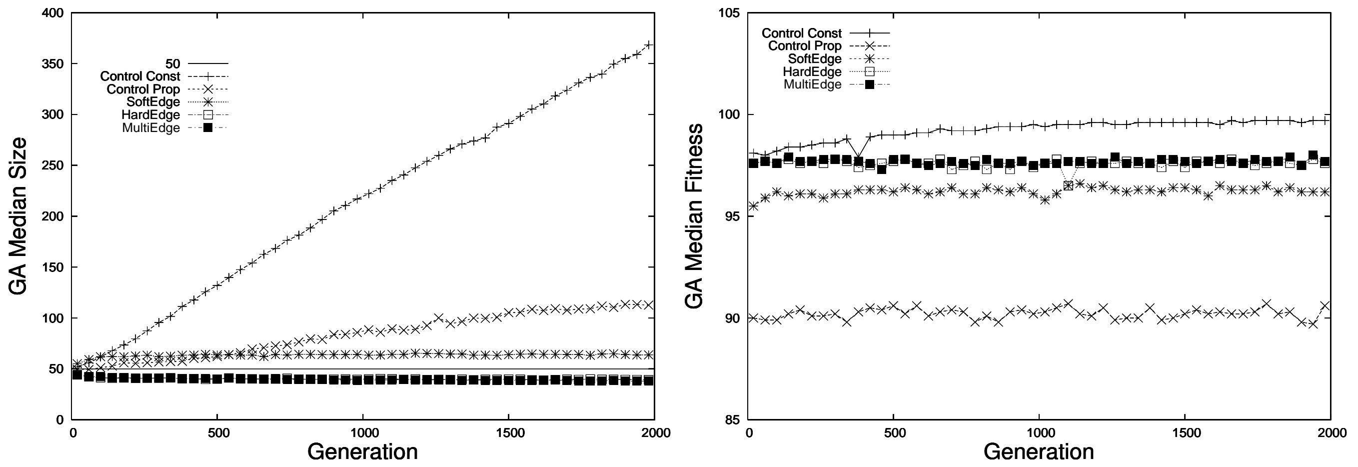

Figure 2: Average median size and fitness over 100 runs of individuals for the 0-1-4's problem. SoftEdge($G = 50, c = 1$), HardEdge($P = 50, c = 4$), MultiEdge($P = 50, G = 25, c = 1$).

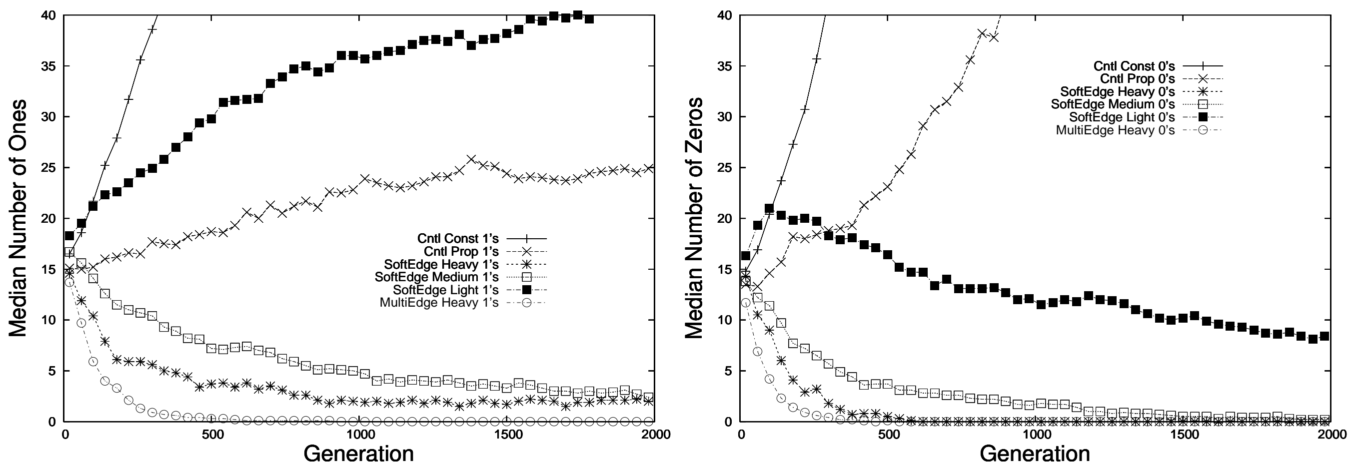

Figure 3: Average median number of ones (Left) and zeros (Right) of individuals over 100 runs of the 0-1-4's problem. SoftEdge($G = 5, c = 1$), SoftEdge($G = 25, c = 1$), SoftEdge($G = 50, c = 1$), MultiEdge($P = 30, G = 20, c = 1$).

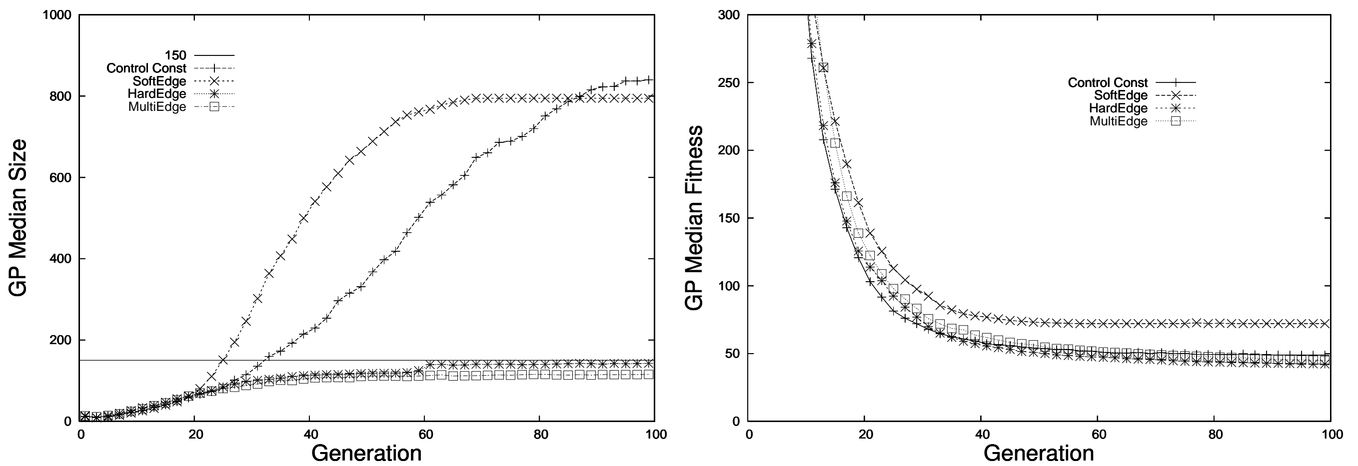

Figure 4: Average median size and fitness of individuals over 100 runs of the function regression problem. SoftEdge($G = 100, c = 1$), HardEdge($P = 150, c = 14$), HardEdge($P = 150, G = 15, c = 1$). Note that average median fitness is approximately average best fitness.

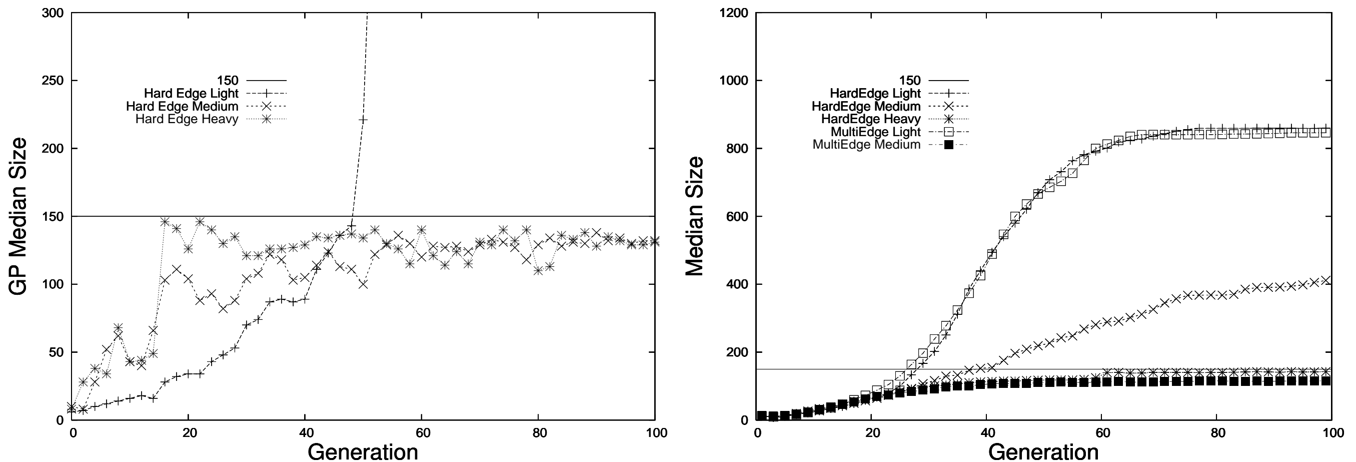

Figure 5: Left: Average median size of individuals in the population over 100 example runs of HardEdge Light($P = 150, c = 4$), HardEdge Medium ($P = 150, c = 9$), HardEdge Heavy ($P = 150, c = 14$). Right: Average median size of the population over 100 runs under various disruptive pressures. All with a $P = 150$. HardEdge Light ($P = 150, c = 4$), HardEdge Medium ($P = 150, c = 9$), HardEdge Heavy ($P = 150, c = 14$), MultiEdge Light ($P = 150, G = 50, c = 4$), MultiEdge Medium ($P = 150, G = 15, c = 9$).

tive pressure, as described by the parameters in the caption for the figure, are clearly bounded by the edge induced by $P = 150$.

An example of several individual runs that exemplify the size curves of light, medium, and heavy disruptive pressure is presented in 5(Left). In Figure 5(Left), we see in this one example that applying too little pressure may result in excessive growth. When enough pressure is applied, growth is contained. In general for this problem, it was found that a $G \leq P/10$ was required for MultiEdge to control the growth. As in the 0-1-4's problem if too little pressure is applied, this may actually **encourage** code growth! We suspect this is an adaptive protection mechanism.

For GP, SoftEdge did not supply a strong enough signal to halt the growth of the trees. We speculate that SoftEdge is prone to providing a signal that essentially herds the genomes to larger sizes. We are investigating the details of this accelerated growth mechanism.

Error bars also demonstrate that, in Figure 4, the fitness for HardEdge is better than the control and the fitness for SoftEdge is worse than the control. These results were unexpected and should be studied further. This may indicate that the parameters selected for destructive crossover could be manipulated to increase the fitness achieved.

The parameters selected for the destructive crossover are, once again, critical. Unlike the GA, we observed accelerated code bloat with parameters that applied too little pressure. There was a "sweet spot," where the pressure was not too low as to cause accelerated code bloat, but not high enough to sacrifice fitness or GP performance.

4.3 General Observations

Although these graphs powerfully demonstrate the control of the size of individuals in the 0-1-4's problem, the choices of parameters cannot be made completely arbitrarily and are dependent on the destructiveness of the crossover operators. For HardEdge, a P set too low for individuals to achieve an optimum fitness resulted in the individuals passing P in size and increased code bloat. For SoftEdge, a G set too low also resulted in increased code bloat. For MultiEdge, problems only occurred when both P and G were set too low. A c value of 10 or more was required for HardEdge without elitism while more pressure seemed to be required for elitism (another unexpected observation).

In general, it appeared from our experiments that the GP problem required more pressure than the 0-1-4's problem. We speculate that this may be because of two factors. First, the number of nodes disrupted by GP crossover was less than half of the number of loci disrupted by the GA crossover. And second, the hierarchical nature of the GP tree tended to better protect the phenotype from crossover affects than the set-like representation in the GA.

5. CONCLUSIONS

We used two very different problems: a variable-length linear genome and a tree GP to explore the use of multiple crossovers as a natural means to contain code growth. We compared three similar multicrossover algorithms to a pair of control algorithms testing for both size and fitness of the populations.

Our experiments support the hypothesis that code growth is a protective response to disruption of crossover. They support the idea that code growth can be controlled by adjusting the number of crossovers that occur in a mixing operation. They demonstrated that, in some cases, active code compression can be encouraged. We showed that, with sufficient disruptive pressure, individuals can be encouraged to remain smaller than a given size, but that, if insufficient disruptive pressure is applied, it can actually encourage code growth. Additionally, the experiments seem to indicate that better fitness (as well as worse fitness) can be achieved in special circumstances using destructive crossover.

Of the three new algorithms, MultiEdge and HardEdge implementations appear to be the best choices for containment of code growth. These implementations have been able

Table 2: 0s1s4s Parameters

Representation	Strings of length n composed of 0s, 1s, and 4s.
Recombination	**constant crossover:** 2-Point crossover with a crossover with a random segment length from the distribution: 50% chance of length 2, 25% chance of length 4, 12.5% chance of length 8, etc.) **proportional crossover:** 2-Point crossover with a crossover segment length proportional to string length
Recombination Probability	100%
Mutation Probability	No mutation
Parent selection	Uniform random
Survival selection	Truncation selection, a new population of 48 was generated and the best 16 selected from the 48. No elitism.
Population Size	16
Initialization	Random. Size uniformly distributed from 10 to 20

Table 3: GP Regression

Representation	Binary trees
Terminals	Constant values (double precision), the variable x
Operators	$+, -, *, /$
Recombination	Nodes selected uniformly beneath the root.
Recombination Probability	100%
Mutation	None
Parent selection	Tournament, size 5
Survival selection	Generational
Population Size	64
Initialization	Random (random depth, number of nodes, and node distribution in the tree)

to consistently eliminate code bloat with proper values for P, G, and c without harming the fitness.

This work has a lot of potential as a practical means of code growth containment without sacrificing performance; however, there were also many unexpected observations, such as the accelerated code bloat seen in the GP for SoftEdge.

This research provides many opportunities for future work. Key questions for future study include:

1. How does destructive crossover compare with other methods that control code bloat (in particular, parsimony pressure)?

2. By expanding the set of test problems, can we confirm the general conclusions we are attempting to make?

3. How is fitness affected (and how can it be affected) by destructive crossover? Can destructive crossover be used to improve the fitness in a population?

4. How is robustness affected for solutions that have been compressed by destructive crossover?

5. Why wasn't SoftEdge effective for GP?

6. What is the range of good (or effective) parameters for an arbitrary problem? (Is it going to be prohibitively difficult to select good parameters for an arbitrary problem?)

7. How does the relative size of P vs. the smallest good solution affect performance for HardEdge?

8. Why did GP with elitism appear to require more pressure (insufficient testing was conducted to be certain that elitism does require more pressure)?

9. Can we evolve the parameters for destructive crossover?

In general, our goal will be to continue to examine the mechanisms of both containment and the unexpected accelerated code growth to better understand how code disruption affects evolutionary processes.

6. ACKNOWLEDGMENTS

This publication was made possible by NIH Grant Number P20 RR16448 from the COBRE Program of the National Center for Research Resources.

7. REFERENCES

[1] 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.

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

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

[4] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, Berlin, Germany, 1998.

[5] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming III*, pages 163–190. Cambridge, MA: The MIT Press, 1999.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In P. Angeline and J. Kenneth E. Kinnear, editors, *Advances in Genetic Programming II*, pages 111 – 134. Cambridge, MA: The MIT Press, 1996.

[12] T. Soule. *Code Growth in Genetic Programming*. PhD thesis, University of Idaho, University of Idaho, 1998.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] M. J. Streeter. The root causes of code growth in genetic programming. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Genetic Programming, 6th European Conference, EuroGP 2003*, pages 443–454, 2002.