The Root Causes of Code Growth in Genetic Programming

Matthew J. Streeter
Genetic Programming, Inc.
mjs@tmolp.com

ABSTRACT

This paper discusses the underlying pressures responsible for code growth in genetic programming, and shows how an understanding of these pressures can be used to use to eliminate code growth while simultaneously improving performance. We begin with a discussion of two distinct components of code growth and the extent to which each component is relevant in practice. We then define the concept of resilience in GP trees, and show that the buildup of resilience is essential for code growth. We present simple modifications to the selection procedures used by GP that eliminate bloat without hurting performance. Finally, we show that eliminating bloat can improve the performance of genetic programming by a factor that increases as the problem is scaled in difficulty.

1 Introduction

The problem of code growth in genetic programming is well documented [2,6-7,11-13]. In addition to making successive generations take longer and making the solutions produced by GP be larger than is necessary, it has been speculated that this growth essentially prevents long runs from being effective and limits the scalability of genetic programming [6]. For these reasons, there has been much research in the GP community both on theoretical explanations of code growth and on practical measures to prevent it. This paper seeks to provide a starting point for addressing both of these concerns. We provide evidence that code growth is indeed a protective mechanism, but that the means by which this protection may be achieved are more complex than those that have previously been hypothesized. We also show that simple changes to the selection scheme that are motivated by our investigation can actually eliminate code growth. Further, we show that eliminating code growth in this way leads to an improvement in performance by a factor that increases as the problem is scaled up.

Section 1.1 defines the terms used in this paper. Section 1.2 reviews existing theories of code growth. Section 1.3 discusses the problems and methodology. Section 2 describes two distinct components of code growth. Section 3 introduces the concept of resilience of program trees and presents an empirical study of resilience. Section 4 shows that the phenomenon of code growth requires phenotypically near-neutral crossovers. Section 5 discusses the effects of removing code growth on scalability. Section 6 is the conclusion.

1.1 Terminology and Background

In this paper we will define an *intron* as a subtree that can be deleted (i.e. replaced with a constant terminal) without changing the behavior of the overall program. For example, in the tree (+ (* X X) (- X X)), the subtree (- X X) would be an intron. *Inviable* code is code belonging to a subtree that can be replaced by any other subtree without changing the behavior of the overall program, e.g. (* (- X X) <*inviable*>). A

neutral crossover is a crossover that produces a child whose behavior is identical to that of the receiving parent, while a crossover is *disruptive* to the extent that it produces a large change in behavior. We will use the terms code growth and bloat interchangeably.

In analyzing code growth it will also be useful to define several distinct populations. The child population $c(n)$ is just the set of individuals that are members of generation n (i.e. what is usually called the "population"). The parent population $p(n)$ consists of N copies of each individual that participates N times as a parent of a member of $c(n)$. The grandparent population $g(n)$ consists of N copies of each individual that participates N times as a parent of a member of $c(n)$ who wins at least one tournament. We define $p_r(n)$ and $g_r(n)$ to be the subsets of $p(n)$ and $g(n)$ containing only those individuals that participated as receiving parents in a crossover, and $p_d(n)$ and $g_d(n)$ to be the subsets containing only donor parents. We will use absolute value bars to denote the average size of the individuals in a population, e.g. $|c(0)|$ indicates the average size of the individuals in generation 0.

Most explanations of code growth make the assumption that a child which is the result of a neutral or near-neutral crossover will in general be more likely to be fit than a child that results from a more disruptive crossover. The reason for this assumption is that it has been empirically shown that, once an evolutionary run has progressed beyond the first few generations, the vast majority of crossovers will produce offspring that are less fit than their parents [10]. Thus, children that are similar to their parents (who must be relatively fit in order to become parents) will be at advantage relative the majority of the offspring population.

1.2 Existing Theories of Code Growth

At the time of this writing there are four major theories of code growth, which we summarize in roughly the order they were originally proposed.

The Intron Theory: Perhaps the earliest theory of code growth concerns what in this paper is termed inviable code [1,8,9]. This theory is motivated by the idea that, since the majority of crossovers are destructive, individuals structured in such a way that they are likely to undergo neutral crossovers will be at an advantage relative to equally fit individuals which are not similarly structured. Trees with large amounts of inviable code are more likely to undergo a neutral crossover when selected as receiving parents, since any subtree inserted into the inviable code will have no affect on the program's behavior. Thus, the accumulation of larger and larger amounts of inviable code as a defense against crossover and mutation is a possible cause of code growth.

The Diffusion Theory: Langdon [3-5] has observed that for many problems the proportion of trees having a given fitness is constant as a function of size so long as the size exceeds some critical threshold. It follows that the absolute number of trees having a given fitness increases as a function of size in the same way as the total number of trees (i.e. exponentially in general). Based on this he has proposed a general explanation of bloat, which is that "any stochastic search technique … will tend to find the most common programs in the search space of the current best fitness" [4] and due to the properties mentioned above these most common programs are large.

The Theory of Removal Bias: Building on the observation that the children of an inviable node are always inviable, while the parents are not necessarily so, Soule has proven that inviable nodes on average occur deeper in trees than do nodes as a whole [11]. Thus, a crossover that is neutral due to removal of an inviable subtree will tend to remove a relatively small tree, but will insert in its place a tree of average size, so that children produced through neutral crossovers (which due to the generally destructive

nature of crossover will tend to be relatively fit) will tend to be larger than their parents, leading to an overall pattern of growth.

The Depth-Correlation Theory: Luke [6,7] has presented a two-part theory involving the relationship between the depth at which crossover occurs and the change in behavior that crossover produces in the offspring. Across a variety of problem domains, Luke has shown that there is a correlation between the depth at which the subtree removed by crossover occurs in the larger overall tree and the degree to which the child program's behavior is different from that of the receiving parent, with deeper removed subtrees corresponding to smaller changes in behavior. This suggests two mechanisms that may contribute to code growth. First, since fit children tend to be the result of crossovers that are relatively non-disruptive, they will tend to be the offspring of receiving parents in whom a deep crossover point was selected, but there will be no corresponding bias toward deep crossover points in the donor parent, thus tending to make fit children larger than their parents. Note that this part of the theory predicts the same phenomenon as removal bias, but assigns it a more general cause. The second part of the theory simply notes that, as receiving parents, larger trees will on average have subtrees removed at deeper points (and due to the correlation will therefore typically produce offspring less different from themselves), thus making size *in itself* a defense against genetic operators such as crossover and mutation.

1.3 Problems and Methodology

The problems studied in this paper lie in the symbolic regression domain, and use target functions that are polynomials of the form $x+x^2+...+x^n$. Such problems have previously been studied in connection with code growth by Langdon [4]. Fitness cases consist of 101 points uniformly spaced over the interval [0,1]. The function set is {+,-,%,*}, where % is a protected division operator that returns 1.0 upon division by zero, and the terminal set is {X, $\Re$}, where $\Re$ denotes the random numeric terminal. Tournament selection is used with a tournament size of 3. Population size is 1000. Unless otherwise noted, we use 100% crossover, a depth limit of 17, and a size limit of 500 points.

2 Components of Code Growth

Code growth occurs when the children in successive generations are consistently larger than the children in previous generations, i.e. $|c(n+1)|>|c(n)|$. It is known that crossover alone cannot increase the expected size of individuals in a population, so that on average $|p(n)|=|c(n)|$. It is also true on average that $|p(n)|=|p_r(n)|=|p_d(n)|$, since receiving and donor parents are selected in the same way. This means that the growth $|c(n+1)|-|c(n)|$ that occurs at generation n is on average equal to $|p_r(n+1)|-|p_r(n)|$.

Recalling that $g_r(n)$ is the subset of the receiving parents $p_r(n)$ whose children win at least one tournament ("the grandparent population"), we can write the difference $|p_r(n+1)|-|p_r(n)|$, as the sum of two terms: $|p_r(n+1)|-|g_r(n)|$ and $|g_r(n)|-|p_r(n)|$. The first term measures the extent to which the fit children (i.e. those who win tournaments) are larger than their parents, while the second term measures the extent to which the parents of fit children are larger than parents as a whole. By arguing that fit children will tend to be larger than their parents, both the theory of removal bias and the first part of the depth-correlation theory predict that the first term will be positive. By arguing that larger parents will be more likely to produce fit children, both the intron theory and the second part of the depth-correlation theory predict that the second term will be positive. Thus, by explicitly tracking the values of these two terms in an actual run, we can determine to what extent each of these two types of theories has the potential to explain code growth.

We tracked the values of these two terms over two sets of 20 runs of the degree 9 polynomial problem. The first set of runs used no size or depth limits and a run length of 50 generations, while the second set used a depth limit of 17 and run length of 300 generations. When running with depth limits, we use one-offspring crossover and retry the selection of both crossover points when the depth limits would be violated (in this case $|c(n)|<|p(n)|$ on average). Figures 1 and 2 show the average sizes of $c(n)$, $p_r(n)$, and $g_r(n)$ for 5 typical generations from each set of runs. For the runs with no depth limits, the average value of the term $|p(n+1)|-|g(n)|$ was 7.60 and the average value of the term $|g(n)|-|p(n)|$ was 12.5. Thus, for this problem it is reasonable to say that 37.9% of the growth was attributable to fit children being larger than their parents, while 62.1% was attributable to the parents of fit children being larger than parents as a whole. In the runs with depth limits, however, only the second term is positive on average, with the average values of the two terms being -1.47 and 2.24, respectively. It must be noted that Luke [6] has found that in symbolic regression problems there is a statistical bias toward fit children being the result of crossovers in which small subtrees were inserted (in addition to the bias toward small subtrees being removed), so that the relationships between these two terms observed in this problem are not necessarily reflective of the relationships that are typical for problems as a whole. Nevertheless, it appears that at least for the problems studied here the dominant cause of code growth is the second term, i.e. that the parents of fit children are larger than parents as a whole.

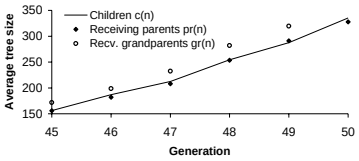

Figure 1. CPG graph, no size/depth limits.

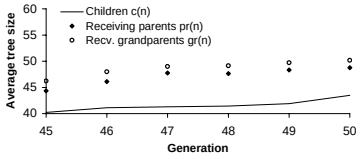

Figure 2. CPG graph, depth limit of 17.

3 Resilience in GP Trees

The key idea behind both the intron theory and the second part of the depth-correlation theory is that the receiving parents who produce offspring more similar to themselves will tend to be large, i.e. that it is large trees which will be most *resilient* in the face of crossover. In this section we present an empirical measure of resilience and investigate the relationship between size and resilience by applying our empirical measure to randomly generated trees.

Conceptually, the resilience of an individual is determined by the probability distribution over all possible values of the behavioral difference between parent and child (quantified in some application-specific way) that is associated with the application of a genetic operator to that individual. In empirically sampling this distribution, we have found that its mean value is often arbitrarily large, since there is in general no limit on the maximum value of behavioral change. For this reason, we instead characterize this distribution by its median value, using the resilience measure described below.

3.1 Our Measurement of Resilience

We measure the resilience of an individual by performing a large number of mutations on separate copies of the individual, and recording for each mutation the difference in behavior between the parent and the child. For the symbolic regression problems we are studying, we define the behavioral difference between two individuals as the average absolute difference between corresponding points on the curves produced by the two individuals. We define *vulnerability* as the median of the behavioral differences associated with the mutations, and *resilience* as -1 times vulnerability. For

all experiments reported here, we will use 101 subtree mutations to estimate resilience. Subtree mutations are performed using a 90% internal, 10% leaf weighted choice of subtree insertion points, and the inserted subtrees are created using the grow algorithm with a minimum depth of 1 and maximum depth of 5. Note that we define resilience in terms of mutation rather than crossover since using crossover would introduce a population-dependence that would make experiments on random trees considerably more complicated.

3.2 The Resilience of Randomly Generated Trees

Our first experiment with resilience was to measure the resilience of arbitrary random trees of various sizes. However, it soon became apparent that many of these trees would be very unlikely to appear in any later generation GP population. For example, the most resilient randomly generated trees had the form (* *<always-zero>* *<always-zero>*), and thus could only be meaningfully changed by a mutation that replaced the entire tree. At the other extreme, some trees performed computations involving final and intermediate values on the order of 10^{15}, which made them extremely non-resilient. Since neither of these two types of trees are likely to be fit with respect to any reasonable fitness function, we chose to narrow our sampling of randomly generated trees using a specific target curve — in this case the quartic polynomial $f(x) = x+x^2+x^3+x^4$. Specifically, any tree whose average difference over the interval [0,1] from this function was less than or equal to 0.5 was considered "fit", while any tree with more than this level of error was discarded as unfit.

1000 relatively fit trees of each size (3 through 31, odd sizes only) were generated, and their resiliences calculated as described in section 2.1 (note that there are no binary trees containing an even number of nodes). All trees were generated using the grow initialization method. Since the use of the grow algorithm introduces a certain shape bias to the trees which are generated, this experiment could admittedly be improved by using a ramped uniform initialization method [4].

Figure 3 shows the average and median vulnerability (-1 times resilience) of randomly generated trees as a function of their size. With the exception of one point on the curve for average resilience, both average and median vulnerability decrease monotonically as the tree size increases. Random trees of size 31, for example, on average had 61% of the vulnerability value of random trees of size 9. Figure 3 establishes that with respect to the given domain (symbolic regression), the given genetic operator (subtree mutation), and the given subset of all randomly generated trees which were actually used in the experiment (those which had an average difference of 0.5 or less from the quartic polynomial), large trees are on average less vulnerable (more resilient) than small trees.

It is also possible to study tree size as a function of resilience. Based on the same experiment described above, figure 4 shows average tree size as a function of resilience, where resilience is given as a percentile, i.e. the point above the label 90 on the horizontal axis denotes the average size of trees that were in the 90-91st percentile with respect to resilience. Figure 4 makes clear that in addition to large trees being on average more resilient than small trees, the most resilient trees are large on average.

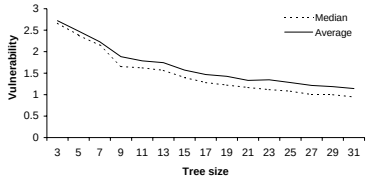

Figure 3. Vulnerability as a function of tree size. Larger trees are typically more resilient.

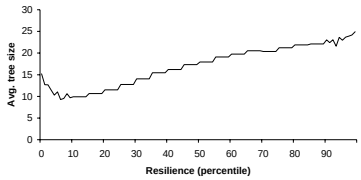

Figure 4. Average tree size by resilience percentile. Resilient trees are large.

3.3 Resilience in Actual GP Runs

A quantity that is closely related to the average vulnerability of trees in a GP population is the median of the difference between offspring and receiving parent behavior for all offspring created at a particular generation. For all problems discussed in this paper, two relationships concerning this quantity consistently hold: median behavioral change decreases during the lifetime of the run, and the median behavioral change for fit children is always lower than that for the child population as a whole. As an example, figure 5 illustrates the median behavioral change from the receiving parent for children $c(n)$ and for fit children $p(n+1)$ for a typical run against the 9th degree polynomial target function. Though the run described by figure 5 used 100% crossover, these two relationships also hold if subtree mutation is used as the sole genetic operator (though in this case behavioral change decreases considerably more slowly). Since the genetic operators do not change during the course of the run, this can only be the result of the population in one way or another becoming more resilient with respect to the genetic operators.

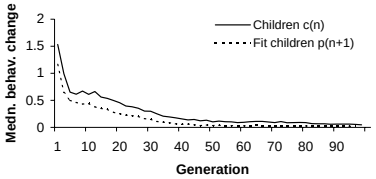

Figure 5. Median behavioral change in a typical run of degree 9 polynomial.

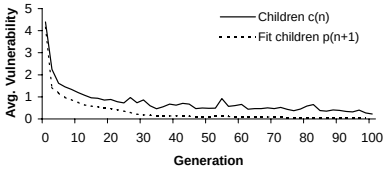

Figure 6. Average vulnerability in the run described by figure 5.

Figure 6 tracks the average vulnerability of individuals in the same run described by figure 5, as measured by explicitly applying our resilience measure to every individual in the population history. As shown in the figure, individuals become more resilient over time, and selected individuals are consistently more resilient on average than individuals as a whole. It is significant that individuals bred only through crossover also become more resilient with respect to subtree mutation. The fact that the children who are selected are consistently more resilient than children as a whole indicates that evolution is preferentially selecting resilient individuals (though it does not necessarily establish that the children's resilience is the cause of this preference). However, if we accept that evolution is in a sense seeking out the more resilient individuals, it follows from our analysis in section 2.2 that doing so would require it to generate larger and larger trees.

3.4 What Makes Trees Resilient

The above sections have quantified resilience on a per-individual basis; however, to understand the mechanisms by which resilience may be achieved it will be necessary to quantify it on a per-node basis. To do this, we define a per-node measure of vulnerability that is identical to the measure given above, except that in measuring the vulnerability of a node N we will only perform mutations where N is the insertion point for the randomly generated subtree. We refer to the set of per-node vulnerability values for a tree as its *vulnerability map*. As an example, the S-expression (* (- X X) (+ X X)) has a vulnerability map (* [2.15] (- [1.70] X [1.44] X [1.58]) (+ [0] X [0] X [0])), where each value in brackets is the vulnerability of the node whose identity is given by the symbol to the left of the brackets. Note that inviable nodes are indicated by zeroes in the vulnerability map. Through experiments both with individuals from actual GP runs and with hand-created trees, we have found that there are at least 4 distinct means by which low vulnerability can be achieved, which we summarize below. In all cases where examples are given, more dramatic examples could be

created using larger trees. Additionally, some examples depend on X being restricted to the interval [0,1], but examples exist that do not have this dependency.

Inviable code: Inviable code has an obvious affect on both the vulnerability map and on vulnerability. As illustrated above, inviable code is indicated by zero values in the vulnerability map. Since each per-node vulnerability can never be less than zero, the presence of inviable code can only make individuals more resilient.

Introns: Introns can be used to make trees more resilient. For example, in the tree (+ (+ (* X X) X) (- 0 0)), replacing the two zeroes with the subtree (* (* X X) (* X X)) lowers the overall vulnerability from 0.823 to 0.666, with the first occurrence of the inserted subtree having a vulnerability map (* [1.55] (* [0.408] X [0.444] X [0.314]) (* [0.458] X [0.283] X [0.478])), which contains a set of relatively small values. This use of introns is similar to depth attenuation (described below).

Arrangement of genetic material: Among equivalent trees of the same size and shape, the arrangement of specific symbols can affect resilience. For example, two trees which code for the expression x^2+x are (* (+ X 1) X) and (+ (* X X) X). The former expression has a vulnerability map of (* [2.35] (+ [1.04] X [.718] 1 [.754]) X [2.47]) and vulnerability of 1.04, while the latter has a vulnerability map of (+ [2.35] (* [1.64] X [0.718] X [0.788]) X [1.52]) and vulnerability of 0.865. In this case, the placement of * at the root of the former tree gives its rightmost leaf node a high vulnerability which accounts for most of the overall difference.

Depth-attenuation: It is relatively easy to create trees whose per-node vulnerability values decrease monotonically with depth. For example, the tree (* (* X X) (* X X)) has a vulnerability map (* [2.34] (* [0.567] X [0.39] X [0.418]) (* [0.467] X [0.323] X [0.242])) which has this property. This, in combination with cancelling or near-cancelling terms such as those in the example in the "Introns" section above, provides a simple way to create large trees that are highly resilient.

In actual GP runs that we have studied, the predominant way in which trees achieve resilience appears to be depth-attenuation. In looking at the vulnerability maps for later-generation trees, we consistently find that the vulnerability of deeper nodes tends to be lower than that of nodes near the root (though this is not consistently true for randomly generated trees). As an example, the following data is from an individual from generation 50 of a run against the degree 9 polynomial. The individual had a depth of 17, and its per-node vulnerability values averaged over depths 0 through 16 respectively were 2.89, 1.91, 2.71, 1.7, 0.649, 0.566, 0.257, 0.420, 0.309, 0.232, 0.203, 0.0501, 0.0488, 0.0368, 0.0329, 0.0299, and 0.0369. The individual's overall vulnerability was 0.335. Notice that the per-node vulnerabilities for depths 13 through 16 are approximately 1/10 this value.

4 A Selection Scheme that Eliminates Bloat

Our preceding analysis has suggested that code growth is a result of the tendency of GP to seek out trees that are more resilient. If this is the case, it should follow that preventing GP from evolving a population of more and more resilient individuals should eliminate bloat. One certain way to prevent the average behavioral change from falling to an arbitrarily low value is simply to introduce a certain *minimum behavioral change* (MBC) that individuals must undergo as a result of crossover or mutation in order to be eligible for selection. For example, if we specify an MBC of 0.05 and heavily penalize (i.e. assign infinite fitness to) any child which differs from its receiving parent by an value lower than this, then we would not expect the average behavioral change to drop much below 0.05. Note this often penalizes children that are *better* than their parents.

We want to emphasize that we do not necessarily think this is a good idea for a selection scheme. For one thing, it is extremely heavy-handed, imposing a strict penalty on individuals whose genetic material may be of value to the population. It may even penalize what would otherwise be the best-of-generation individual. It also eliminates the possibility of neutral walks, which are considered an important aspect of artificial evolution [14]. Nevertheless, the fact that this selection method limits the buildup of resilience means that using it will tell us something about our explanation of code growth.

We conducted experiments using 4 levels of MBC: 0.2, 0.1, 0.05, and 10^{-5}, in addition to a control experiment with no MBC. No size or depth limits were used in these experiments. Each experiment involved 20 separate runs against the degree 10 target polynomial, each lasting 300 generations, with the exception that the experiment with MBC of 10^{-5} and the control experiment were run for only 50 generations due to the extreme increase in tree size. The degree 10 polynomial was used to allow most or all of the growth to occur before the run solved. In practice we found for MBC values of 0.05, 0.1 and 0.2 that between 5% and 15% of the individuals were penalized in each generation.

As shown in figure 7, tree growth was lower for each successively higher value of MBC. Moreover, for MBC values of 0.1 and 0.2 the average tree size plateaus rather than continuing to increase. For these MBC values, we have continued individuals runs for up to 10,000 generations and have never seen any deviation from this plateau. Note that eliminating only neutral crossovers (MBC of 10^{-5}) did not dramatically reduce code growth, which is consistent with the results of Luke [6-7]. However, eliminating (phenotypically) near-neutral crossovers did eliminate code growth. Furthermore, as will be shown in section 5, eliminating code growth in this way actually improves performance independent of the obvious savings in execution time per generation.

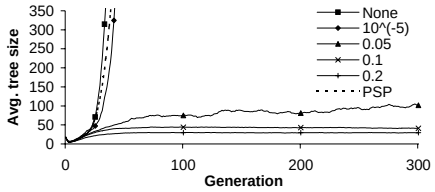

Figure 7. Average tree sizes for various levels of minimum behavioral change (MBC).

4.1 The Possibility of MBC as a Probabilistic Size Penalty

Since we have shown in section 2.2 that the more resilient trees are typically large, and since MBC tends to penalize resilient trees, the possibility exists that the effect of MBC on code growth is due to its penalizing of some critical fraction of the larger trees, rather than to any connection with the underlying causes of code growth. To test this possibility we recorded, for all 50 runs of the degree 9 polynomial using an MBC of 0.1, the average fraction of individuals of each size *S* that were penalized in each generation *N*. We then performed 20 additional runs in which, for every *S* and *N*, this same fraction of individuals were penalized, but the individuals to be penalized were selected at random from among individuals of the given size (rather than being selected based on behavioral change). The curve labeled PSP in Figure 7 shows the average tree size for these additional runs. As shown in the figure, the probabilistic size penalty used in these additional runs has only a very slight affect on code growth.

Thus, that an MBC of 0.1 did eliminate code growth depended critically on the fact that it was the *resilient* individuals that were specifically penalized.

5 Effects on Scalability

As mentioned in section 3, there are a number of reasons not to use MBC as a selection scheme, including its heavy-handedness, its ability to penalize good individuals, and its elimination of neutral walks. Nevertheless, it is worth studying the affect of this approach on performance for two reasons. First, doing so will give us at least a rough idea of the affect on performance of eliminating bloat. Second, the performance of this approach can act as a benchmark for any more sophisticated measures that are devised to eliminate bloat. We tested the performance of this approach using 6 symbolic regression problems with target functions of the form $x+x^2+x^3 + \ldots + x^n$ for $4 \le n \le 9$, using 50 runs for each problem. Figures 8 and 9 give the success probability curves associated with MBC and with standard GP selection, respectively. Table 1 summarizes the computational effort associated with each selection method, where I_A, G_A, and R_A denote the computational effort, optimum number of generations, and optimum number of runs, respectively for standard GP selection, and I_B, G_B, and R_B denote the corresponding quantities for MBC selection.

The affect of MBC on scalability in these problems is dramatic. With standard GP selection, the probability of finding a perfect solution increases rapidly up to a certain generation, and then plateaus. As the problem is scaled up in difficulty, the value at which the success probability plateaus becomes lower and lower. In contrast, with an MBC of 0.1, although the success probability curves rise more slowly as the problem is scaled up, they consistently attain a high final value. Indeed, with an MBC of 0.1 all 300 runs eventually succeed, with the longest run of the 9th degree polynomial succeeding at generation 276.

It must be noted that since MBC eliminates the buildup of resilience in addition to eliminating code growth (which we take as a symptom of the buildup of resilience), it is not entirely clear which of these features is responsible for the performance difference we observe. In other words, although we believe code growth happens because creating larger trees is a natural and easy way to build up resilience, it may still be possible to eliminate code growth while allowing the buildup of resilience to occur in other ways. At the moment, we do not have a way to know what the affect on performance of eliminating code growth without eliminating the buildup of resilience would be in these problems. But in any case, it appears there is untapped potential to increase the scalability of GP by taking the issues of resilience and code growth into account.

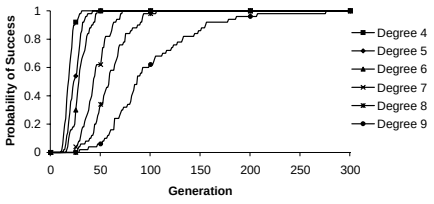

Figure 8. Success probabilities using MBC of 0.1. Probability of success continues to improve even late in the run.

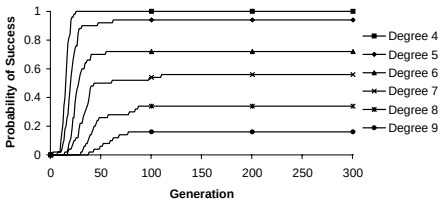

Figure 9. Success probabilities using standard GP selection. Probability of success plateaus at different levels as problem is scaled up.

Table 1. Comparison of computational effort for standard GP selection (A) and for MBC of 0.1 (B).

Degree	$I_A(M,i,z)$	G_A	R_A	$I_B(M,i,z)$	G_B	R_B	I_A/I_B
4	26,000	25	1	32,000	31	1	0.81
5	64,000	31	2	43,000	42	1	1.5
6	164,000	40	4	48,000	47	1	3.4
7	308,000	43	7	73,000	72	1	4.2
8	784,000	48	16	107,000	106	1	7.3
9	2,106,000	77	27	277,000	276	1	7.6

6 Conclusions

We have proposed an explanation of code growth based on the concept of resilience, and have shown that preventing the buildup of resilience also prevents code growth. Through random sampling of equal numbers of individuals of 15 different sizes, we have found that the most resilient individuals are larger than average. By monitoring resilience in actual runs, we have found that trees become more resilient over time and that selected individuals are consistently more resilient than individuals as a whole. We have also shown that by using a selection method that prevents the population from becoming resilient, we can eliminate code growth. All of this strongly suggests that code growth occurs as a side effect of the seeking out by evolution of resilient trees. Finally, we have shown that eliminating code growth in this way yields an improvement in performance that increases as the problem is scaled in difficulty.

References

1. T. Blickle and L. Thiele. Genetic programming and redundancy. In J. Hopf (ed.), *Genetic Algorithms Within the Framework of Evolutionary Computation*, Max-Planck-Institut fur Informatik: Saarbrucken, Germany, 1994, p 33-38.
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, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In *Advances in Genetic Programming III*, Cambridge, MA: The MIT Press, 1999, p 163-190.
4. W. B. Langdon. Size-fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95-119, 2000.
5. W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag; 2002.
6. S. Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD thesis, University of Maryland, College Park, 2000.
7. S. Luke. Code growth is not caused by introns. In Late-Breaking Papers, Proceedings of GECCO 2000, 2000, p 228-235.
8. N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In L. J. Eshelman (ed.), *Proc. Sixth Int. Conf. Genetic Algorithms*, Morgan Kaufmann, 1995, p 303-309.
9. P. Nordin and W. Banzhaf. Complexity compression and evolution. In L. J. Eshelman (ed.), *Prof. Sixth Int. Conf. Genetic Algorithms*, Morgan Kaufmann, 1995, p 310-317.
10. P. Nordin and F. Francone. Explicitly defined introns and destructive crossover in genetic programming. In P. Angeline and K.E. Kinnear Jr (eds.), *Advances in Genetic Programming II*, Cambridge, MA: The MIT Press, 1996, p 111-134.
11. T. Soule. *Code Growth in Genetic Programming*, PhD thesis, University of Idaho, 1998.
12. T. Soule and J. A. Foster. Removal bias: a new cause of code growth in tree-based evolutionary programming. In *ICEC 98: IEEE International Conf. on Evolutionary Computation*, IEEE Press, 1998, p 781-786.
13. T. Soule and R. B. Heckendorn. An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 3(3):283-309, 2002.
14. T. Yu and J. Miller. Finding needles in haystacks is not hard with neutrality. In Foster et al. (eds.), *Proceedings of EuroGP'2002*, Springer-Verlag, 2002, p 13-25.