Modification Point Depth and Genome Growth in Genetic Programming

Sean Luke sean@cs.gmu.edu
Department of Computer Science, George Mason University, 4400 University Drive
MS# 4A5, Fairfax, VA 22030, USA

Abstract

The evolutionary computation community has shown increasing interest in arbitrary-length representations, particularly in the field of genetic programming. A serious stumbling block to the scalability of such representations has been *bloat*: uncontrolled genome growth during an evolutionary run. Bloat appears across the evolutionary computation spectrum, but genetic programming has given it by far the largest attention. Most genetic programming models explain this phenomenon as a result of the growth of *introns*, areas in an individual which serve no functional purpose. This paper presents evidence which directly contradicts intron theories. The paper then uses data drawn from this evidence to propose a new model of genome growth. In this model, bloat in genetic programming is a function of the mean depth of the modification (crossover or mutation) point. Points far from the root are correspondingly less likely to hurt the child's survivability in the next generation. The modification point is in turn strongly correlated to average parent tree size and to removed subtree size, both of which are directly linked to the size of the resulting child.

Keywords

Introns, Inviable Code, Code Bloat, Genetic Programming, Crossover Point

1 Introduction

In evolutionary computation, *bloat* is the tendency for arbitrary-length representations to grow in size as a run progresses, without the justification of a corresponding improvement in fitness. Bloating slows the evolutionary search process, consumes memory, and can hamper effective breeding. This produces a sort of Zeno's paradox, slowing successive generations by so much that it places a cap on the useful runtime of the given evolutionary computation technique.

As arbitrary-length representations have become more common, particularly in genetic programming (GP), bloat has received an increasingly large amount of attention. This attention has come both in techniques for preventing or lessening bloat, and in theoretical explanations for its existence, without which such prevention techniques are merely ad-hoc. The GP literature has yielded four bloating models: three of these four models focus on the existence of *introns*, regions of code which can be trivially simplified without affecting function. The fourth theory attempts to describe bloating phenomena only in general terms, without providing a specific functional explanation.

However, the experimental methodology used in the intron theory literature to date has so far not been arranged to falsify the theories. This paper presents falsifiable experimental data which, as it turns out, contradicts the functional explanations given in the intron theories.

©2004 by the Massachusetts Institute of Technology

Through analysis of this data, the paper then presents a new theory of bloating as a function of tree depth selection bias: the deeper the tree, the more survivable its children. This is due to a strong relationship between the depth of a subtree modification (crossover, mutation) and the impact that modification has on the individual's performance. Modifications close to the root generally make dramatic changes to an individual, whereas modifications far from the root have lesser effect. Trees with more deep nodes thus are more likely to survive highly damaging modifications. Deeper nodes are correlated prevalent in large, and tend to root smaller subtrees; these subsequent effects in turn produce a bias towards larger children.

The paper is arranged as follows. First, it presents a survey of arbitrary-length representations in the evolutionary computation literature, and a quick description of genetic programming. Then it discusses bloat, ways of combatting it, and genetic programming theories explaining the bloat phenomenon. Following this, it presents experiments which cast doubt on existing theories, then proposes a depth-based theory of bloating in genetic programming.

2 Arbitrary-length Representations

Arbitrary-length representations have been with evolutionary computation since its inception. At the same time that fixed-size genetic algorithms and evolution strategies were being developed to handle parameter vectors, evolutionary programming sought to search for arbitrary-sized finite-state automata (Fogel et al., 1966), applying operators to add and delete edges and vertices. Smith (1980) evolved arbitrary-sized rule-based programs which learned problems such as maze navigation and poker betting. This work eventually contributed to the "Pitt approach" to learning rule systems (De Jong, 1989), in which each individual in the population is a classification rule set of arbitrary length.

Other interest in arbitrary-length genomes has sprung from attempts to improve on GA's fixed-length vector genotypes while still evolving for a fixed number of parameter settings. Best known in this area is the *messy genetic algorithm* (Goldberg et al., 1993), which evolved lists of $\langle parameter, value\rangle$ pairs. This work was later extended to the *gene expression messy genetic algorithm* (GEMGA) (Kargupta, 1996). Some theoretical work has also treated genomes as unordered sets of objects (Radcliffe and George, 1993).

Another impetus for abitrary-length genomes has been interest in using a more "DNA-like" genome to solve optimization problems. Genomes in such approaches have typically been long strands of DNA-like codons, often even using a four-letter A,T,G,C alphabet. Although the genomes are formally fixed in size, they are usually very long, and genes are randomly dispersed throughout the genome, delimited by start and stop codon sequences. This means that the number of genes expressed in the genome can be of any size, and genes can be any length (up to a point). This approach has been used to evolve neural networks (Fullmer and Miikkulainen, 1991) and to attack problems in evolutionary robotics (Jakobi, 1995). Wu and Lindsay (1995) examined the dynamics of non-coding segments (parts of the strands that did not define genes) in these kinds of genomes. Burke et al. (1998) continued this work, adding more ideas from biology, including multiple reading frames (allowing genes to overlap) and homologous crossover (recombination at points where codons are most similar). Unfortunately, most work in this area has been exploratory; few papers have compared such approaches empirically to other EC work in nontrivial domains. One exception to this trend ((Luke et al., 1999)) borrowed similar ideas from cellular biology, but es-

chewed the notion of using DNA codon strings. Instead, a genome was used which consisted of an arbitrary-sized set of genes, each tagged with a real-valued locus between 0 and 1 which placed them on the chromosome. The genes' rules were then mapped to state transitions in finite-state automata and the technique evolved FSAs to do language induction on a popular benchmark, with good results compared to the existing literature.

One fertile area for arbitrary-length representations has been in attempts to evolve graphs and networks. Early attempts at evolving neural networks fixed the network topology and evolved the weights as values in the genome (Collins and Jefferson, 1991), usually represented with a simple fixed-length vector. But later approaches used representations to directly encode arbitrary numbers of graph vertices and edges in the genome (Fullmer and Miikkulainen, 1991; Lindgren et al., 1992; Angeline et al., 1994). Still later techniques attempted to use morphological rules to describe a neural network. Kitano (1990) was the first publication to attempt to evolve graph structures using sets of rules for "building" a graph, rather than explicitly stating the vertices and edges. Boers et al. (1993) also "grew" networks, using Lindenmeyer systems operating on graph vertices and edges.

By far the largest literature on arbitrary-length genomes has been in the area of genetic programming. GP is concerned with evolving actual symbolic computer functions, a problem domain which lends itself naturally to arbitrary-length representation. The lion's share of genetic programming work has focused on parse trees representing Lisp s-expressions (Koza and Rice, 1991), but many GP researchers have taken other tacks, such as representing a computer program as a list of machine instructions (also known as *linear GP*, see for example (Stoffel and Spector, 1996; Nordin, 1997; Banzhaf et al., 1998)). Teller (1998) represented programs as cyclic call-graphs. Some work has also been done with directed acyclic graphs of functions (Keijzer, 1996; Handley, 1994). Last, cellular encoding, a field related to GP, evolves trees to describe graphs and networks. Cellular encoding has been used for neural networks (Gruau, 1992), finite state automata (Brave, 1996; Luke, 2000b), and electrical circuits (Koza et al., 1997b,a; Jones and Joines, 1999).

3 Genetic Programming

The form of genetic programming discussed in this paper operates over Lisp s-expressions, and was popularized by John Koza (1992). Though genetic programming prints its individuals as s-expressions, it represents them internally as trees of labeled nodes. Leaf nodes in the tree are known as *terminals* and nonleaf nodes are known as *nonterminals*. The mapping from tree to s-expression is straightforward: a terminal is represented as a Lisp atom, and a nonleaf node is represented as a Lisp function or macro whose arguments are subtree children to that node. That is, a nonterminal named "foo" appears in Lisp form as (foo *arg [arg*]*), where *arg* are subexpressions, one for each child subtree of the foo node. Traditionally, a terminal node named "bar" always appears in Lisp s-expression form as bar (without parentheses).

Except where detailed later in this paper, all the experiments presented here follow the parameters laid out in Koza's *Genetic Programming* (1992). Genetic programming commonly uses its own idiosyncratic tree-generation algorithms, GROW and FULL, to generate initial inidividuals. Selection is done through tournament selection with a tournament size of seven. Breeding is performed either through direct copying (10% of the time) or through *subtree crossover*, whereby two parents swap an arbitrary subtree.

Three common GP problem domains were chosen for inclusion in these exper-

Function Syntax	Arity	Description
(+ *i j*)	2	Returns $i + j$
(− *i j*)	2	Returns $i - j$
(∗ *i j*)	2	Returns ij
(% *i j*)	2	If j is 0, returns 1, else returns $\frac{i}{j}$
(sin *i*)	1	Returns sin i
(cos *i*)	1	Returns cos i
(exp *i*)	1	Returns e^i
(rlog *i*)	1	If j is 0, returns 0, else returns log $\lvert i \rvert$
x	0	Returns the value of the independent variable (x).
ERCs	0	(*Optional*) Random numerical constants chosen from floating-point values from -1 to 1 inclusive.

Table 1: Genetic Programming Function Set for the Symbolic Regression Domain

iments. *Eleven-Bit Boolean Multiplexer* is the most difficult problem, and describes a search through a discrete solution space to find a working 3-in, 8-out multiplexer. *Symbolic Regression* uses a floating-point solution space, searching for a specific mathematical equation. *Six-Bit Boolean Multiplexer* is a relatively easy problem for GP to solve, and represents a search through a discrete solution space to find a working 2-in, 4-out multiplexer.

What follows is a quick overview of these three problem domains.

3.1 Symbolic Regression

Symbolic Regression is the canonical example domain for genetic programming. The object of symbolic regression is to find a symbolic function which best fits a set of data points of the form $\langle x, y \rangle$ in the real cartesian plane. Symbolic Regression differs from classic regression methods in statistics in that the function set includes transcendental functions (sine, cosine, natural logarithm).

The benchmark problem for Symbolic Regression has 20 random data points to fit. The x values of the points are picked at random; these form the "independent variables" in the data set. Their corresponding y values are computed from the benchmark function $y = x^4 + x^3 + x^2 + x$. GP individuals will try to fit to this function. Table 1 shows Symbolic Regression's function set. In some versions of Symbolic Regression, random numerical constants are added to the function set.

Symbolic Regression assesses fitness as follows. For each data point $\langle x_i, y_i \rangle$, the independent variable (the value that the terminal x will return) is set to x_i. The individual's tree is then evaluated and the result is stored in r_i. The fitness is $\sum_{i=1}^{n} |(y_i - r_i)|$, lower values are fitter. A standardized fitness of 0 represents a perfect match. An example ideal individual is:

Result: (+ (* x (* (+ x (* x x)) x)) (* (+ x (cos (- x x))) x)).

3.2 6- and 11-Bit Multiplexer

The objective of the 6-Bit and 11-Bit Multiplexer problems is to find a boolean function which performs multiplexing over a 2-bit or 3-bit address.

In 6-Bit Multiplexer, there are two boolean-valued address variables (A0 and A1) and four corresponding boolean-valued data variables (D0, D1, D2, D3). The 6-Bit Multiplexer function must return the value of the data variable at the address described

Function Syntax	Arity	Description
(and *i j*)	2	Returns $i \cap j$
(or *i j*)	2	Returns $i \cup j$
(not *i*)	1	Returns $\neg i$
(if *test then else*)	3	If *test* is true, then *then* is returned, else *else* is returned.
a0, a1	0 each	Return the values of variables A0 and A1 respectively.
a2	0	(*11-Multiplexer only*) Returns the value of variable A2.
d0, d1, d2, d3	0 each	Return the values of variables D0, D1, D2, and D3 respectively.
d4, d5, d6, d7	0 each	(*11-Multiplexer only*) Return the values of variables D4, D5, D6, and D7 respectively.

Table 2: Genetic Programming Function Set for the 6- and 11-Bit Multiplexer Domains

by the binary values of A0 and A1. For example, if A1 is true and A0 is false, the address is 2 (binary 10), and in this case, the optimal individual must return the value stored in D2. Since there are six boolean variables altogether, there are 64 permutations of these variables and hence 64 test cases.

11-Bit Multiplexer is similar, except that it has three address variables (A0, A1, A2), and thus has eight data variables (D0, D1, D2, D3, D4, D5, D6, D7). In this case, there are eleven variables altogether, and so there are 2048 test cases. Table 2 shows the 6-Bit and 11-Bit Multiplexer function sets.

Both 11-Bit and 6-Bit Multiplexer assesses fitness as follows. For each test case, the data and address variables are set to return that test case's permutation of boolean values, and the individual's tree is then evaluated. The individual's fitness (again, lower values are fitter) is the number of test cases for which the individual returned the wrong value for the data variable expected, given the current setting of the address variables.

An example of an ideal 6-Bit Multiplexer individual is:

Result: (if a1 (not (not (if a0 d3 d2))) (if (not (and (if a0 a1 d0) (and d1 d2))) (if (not (if a1 a0 d1)) (if a0 a1 d0) (or (or a1 d0) (or (if d2 d0 d2) (and d1 a0)))) (if (or d1 a1) (if (not (if a1 d3 (if d3 d1 d1))) (and (and d2 a1) (not a0)) (or (or a1 d0) (or (if d2 d0 d2) (and d1 a0)))) (and (not d0) (and a0 d1)))))

Though this is a much larger individual than the Symbolic Regression individual shown, in fact 6-Bit Multiplexer is a very simple problem for Genetic Programming to solve. 11-Bit Multiplexer is significantly more difficult than the other two.

4 The Race Against Bloat

Bloating appears to be a problem across the spectrum of arbitrary-length representations. For example, Burke et al. (1998) discussed bloating problems in DNA-like codon strings. Smith (1980) noted bloat as a serious impediment to evolving Pitt-approach rulesets. Similar problems have also been noted when using Pitt-approach rule systems to control micro-air vehicles (Bassett and De Jong, 2000). However, most bloat research has been in genetic programming. Bloat is a major bugaboo both for tree-based GP genomes (for example (Koza, 1992; Blickle and Thiele, 1994; McPhee and Miller, 1995; Angeline, 1998; Langdon et al., 1999; Lanzi, 1999)) and for linear GP genomes (Banzhaf et al., 1998). Bloat is also an impediment to cellular encoding (Luke, 2000b).

There are two basic causes of bloat. First, bloat may arise from an interaction between the representation and the breeding operators, causing individuals to grow on average independent of selective pressure. For example, Rowe and McPhee (2001) specified fixed-point distributions to which list-type representations would converge under several different genetic operators with random selection. Subtree mutation causes similar asymptotic bloating independent of selection (Luke, 2000b). Second and more commonly, bloat is driven by selection in combination with the representation and breeding operators. This bloat is, to quote Bill Langdon, a phenomenon of "survival of the fattest". In the process of selecting for fitness, the system's dynamics also shift the population towards larger and larger individuals. This paper focuses on this second kind of bloat.

Fighting Bloat The typical approach to fighting bloat is *parsimony pressure*, which takes into consideration the individual's size as part of its fitness assessment. Some approaches apply a constant weighted size penalty (for example (Smith, 1980; Soule et al., 1996; Bassett and De Jong, 2000)), while others add parsimony pressure adaptively in response to growth metrics in the individual and in the population (Iba et al., 1994; Blickle, 1996). A recent variation on this approach is to treat fitness and parsimony as independent objectives, and use fitness based on Pareto dominance ranking to evolve for both of them at the same time (Bleuler et al., 2001; de Jong et al., 2001).

Another approach is to simply limit the maximum size of the individual. For example, much work in GP follows the technique used by Koza (1992), which restricts modification operators to produce new trees of depth less than 17. Unfortunately, size restrictions can have unforseen consequences. In GP, for example, tree depth limits have been shown to have unfortunate effects when most trees in the population reach the limit (Gathercole and Ross, 1996).

In GP bloat can also be somewhat countered through careful choice of the modification operator. In tree-based GP, two common operators include *subtree crossover*, which swaps subtrees among individuals, and *subtree mutation*, where subtrees in an individual are replaced with randomly-generated subtrees. Subtree crossover has been indicted as major source of bloat when compared to mutation and other methods (Luke, 2000b; Angeline, 1998). However, choosing mutation is not a panacea, however, as it too can cause bloating, though to a lesser degree (Luke, 2000b; Langdon and Poli, 1997c).

Another GP method is *explicitly defined introns* (Nordin et al., 1996; Smith and Harries, 1998; Blickle, 1996; Angeline, 1996). The idea here is to allow the inclusion of special nodes which adapt the likelihood of subtree crossover or mutation at specific positions in the tree. The final major GP method is *code editing*: simplifying and optimizing GP individuals' parse trees. Some work ((Soule et al., 1996; Blickle, 1996; Iba and Terao, 2000)) reports strong results with this approach. However, Haynes (1998) warns that editing can lead to premature convergence in the evolutionary system.

5 Genetic Programming and Bloat

Because GP operates over arbitrary-sized parse trees of Lisp s-expressions, it is strongly susceptable to the bloating phenomenon. The exact cause of GP code bloat is still not well understood, but much of the code bloat literature in genetic programming has focused on so-called *introns* — extraneous regions of code in an individual which neither add nor detract from its fitness, because they are ignored or do nothing. Like so many other terms in evolutionary computation, "introns" is borrowed from genetics,

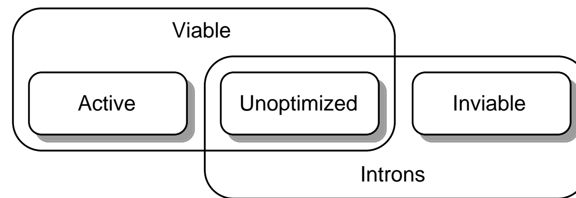


Figure 1: A Venn Diagram of Labels Used to Describe Various Kinds of Code

referring to areas in DNA genes which do not contribute to the final protein or RNA endproduct.

Angeline (1994) is the first researcher to explicitly identify genetic programming introns and give them their nickname. Angeline defined introns as areas of code that "are unnecessary since they can be removed from the program without altering the solution the program represents". The term "introns" has been used in at least three different ways, which I will denote as *introns*, *inviable code*, and *unoptimized code*.

1. **Introns**. Areas of code which do not contribute to an individual's function.

2. **Inviable Code**. Code regions which cannot be replaced by anything that can possibly contribute to the individual's function. This is a specific subset of #1. Such code is typically associated with an *invalidator*, a structure elsewhere in the individual which is responsible for nullifying the intron's effect. It is possible for two invalidators to make each other inviable. I call this "co-inviable code". Ther term "inviable code" originates with (Soule and Foster, 1998; Langdon et al., 1999)), but has been called other things: "absolute introns" (Banzhaf et al., 1998), "redundant nodes" (Blickle and Thiele, 1994; Blickle, 1996), "ineffective code" (Rosca, 1996), "type 1 introns" and "type 2 introns" (Nordin et al., 1995), and "syntactic introns" (Angeline, 1998).

3. **Unoptimized Code**. Code regions do not contribute to an individual's function, but can be replaced with code which does contribute. This consists of all introns which are not inviable code. Langdon et al. (1999) call these regions "inoperative code", though they unfortunately restrict their term in such a way as to exclude plausible candidates such as redundant parent nodes in a tree, as in (not (not (not (not *foo*)))), or redundant subtrees, as in (and d1 d1). Nordin et al. (1995) similarly make this exclusion, calling them "type 3" and "type 4" introns. Unoptimized code has also gone by "inert subexpressions" (Tackett, 1994) and "semantic introns" (Angeline, 1998).

In this paper, code which is not part of an intron will be termed "active". Both active and unoptimized code will be termed "viable". Figure 1 illustrates this with a Venn diagram. See Appendix A for specific examples of inviable and unoptimized code.

5.1 Theories of Code Bloat

Angeline (1994) argued that individuals with large numbers of introns stood a better chance of surviving crossover intact, but he viewed this feature as beneficial to the evolutionary process. Tackett (1994), citing personal communication from Andrew Sin-

gleton in 1993, also noted this argument, though Tackett himself disagreed with it. Subsequent literature, however, has argued strongly that introns are a chief culprit in code bloat, and thus are not desirable.

Generally speaking, there are three overlapping theories which implicate introns in code bloat: hitchhiking, defense against crossover, and removal bias.

5.1.1 Hitchhiking

The first intron theory, *hitchhiking* (Tackett, 1994) says that if introns (the hitchhikers) are attached to parents of "important" active code (though Tackett did not use the term, this is effectively building blocks), then crossover which preserves this active code is likely to take some of these introns along with it and thus propagate introns throughout the population. Tackett noted that if the evolutionary computation system was more selective, code growth would increase. He argued that with the increase in selectivity, the value of important building blocks also increased, and as these building blocks were replicated more rapidly through the population, introns hitchhiking on these building blocks would spread all the faster.

5.1.2 Defense Against Crossover

In the second theory, *defense against crossover*, introns are propagated because they act to make it difficult to destroy an individual, by increasing the number of crossover points which have no effect on the individual. This gives rise to *neutral crossover*, where the child is identical in function (and fitness) to its parent. Despite its name, this theory is equally applicable to a wide variety of non-crossover tree-manipulation mechanisms, including subtree mutation.

This general theory has been cited, in one way or another, by a large chunk of literature: (Blickle and Thiele, 1994; McPhee and Miller, 1995; Nordin and Banzhaf, 1995; Rosca, 1996; Blickle, 1996; Rosca, 1997; Banzhaf et al., 1998; Langdon et al., 1999; Andre and Teller, 1996). The chief theoretical support for defense against crossover was outlined by Blickle and Thiele (1994) and Nordin and Banzhaf (1995). The argument is roughly as follows: let s_i be the expected number of times an individual i will be selected from a given generation to undergo crossover or reproduction. Let p_c be the probability that crossover will be the breeding mechanism. Let C_{ai} be the number of crossover points in the individual (its *absolute complexity*), and let C_{ei} be the number of crossover points which might result in something other than neutral crossover (its *effective complexity*). Let d_i be the probability that individual i will be damaged by crossing over in non-neutral crossover points. The probability of damage by crossing over in neutral points is 0. Then the expected number of children n_i of individual i that are better than or equal to i in fitness is:

$$n_i = s_i(1 - p_c(\frac{C_{ei}}{C_{ai}}d_i + \frac{1 - C_{ei}}{C_{ai}}0)) = s_i(1 - p_c\frac{C_{ei}}{C_{ai}}d_i)$$

From this equation both Blickle and Thiele (1994) and Nordin and Banzhaf (1995) argue that there are two factors influencing whether a parent will produce many highly competitive children. The first and more obvious factor is the fitness of the parent itself. But another important factor is the probability that crossover of a parent will destroy the child. Initially crossover is fairly likely to be constructive, so fitness is the predominant factor. But as an evolutionary run progresses, individuals become fitter and finding a better solution becomes rarer; usually crosssover results in worse children, often far worse children. In this latter situation, the most survivable individuals are the ones which prevent crossover from having much effect. This gives rise to parents with

lots of intron material, thus a small ratio of effective to absolute complexity, as a way of hampering crossover's modification ability.

It is notable, however, that this equation is only applicable to inviable code when applied to tree-based GP. Indeed, one surprising feature of the defense-against-crossover literature is that while most proponents of this theory argue that *introns* increase the number of ineffective crossover points in an individual, in reality the lion's share of experiment and all theoretical justification have dealt solely with inviable code for tree-based GP.

While for linear string GP systems all kinds of introns can decrease effective complexity (as noted in work in AIMGP (Nordin and Banzhaf, 1995; Banzhaf et al., 1998)), in tree-based GP, effective complexity can only be reduced through inviable code (Blickle and Thiele, 1994). Similarly, Blickle and Thiele (1994); Blickle (1996) uses only inviable code in his experimental examples, as does Rosca (1996). Some of the literature ((Langdon, 2000; Rosca, 1997)) simply defines introns to be inviable code. Other experiments ((Langdon et al., 1999)) support the theory in abstract but are not specific enough to distinguish between inviable and unoptimized code.

In summary, while the defense against crossover theory is typically stated in terms of introns in general, proofs of the theory consistently deal with inviable code only, and experiments with the theory generally focus on inviable code in tree-based genomes. I believe it is fair to say that, at least for its history in the tree-based genetic programming literature so far, defense against crossover is an inviable code theory.

5.1.3 Removal Bias

A recent third theory, *removal bias*, eschews unoptimized code entirely and focuses solely on inviable code (Soule and Foster, 1998; Langdon et al., 1999). In this theory, the presence of inviable subtrees provides safe harbors for tree growth. The theory first proposes that if an individual contains inviable subtrees, it is more likely to survive if it performs modifications (crossover, mutation) within these subtrees. In some sense this is similar to defense against crossover. But removal bias goes further, suggesting a direct functional linkage between inviable code and bloat. In order to guarantee preservation of the individual, the subtree removed during modification must be no larger than the inviable subtree area; hence there is a bias towards removing small subtrees from the individual. But the *replacing* subtree has no such bias at all — it can be any size and still have no effect on the individual. From this the theory predicts that inviable subtrees will grow without bound.

The primary evidence for the removal bias theory is a form of hillclimbing originally proposed by O'Reilly (1995) which rejects crossover results that decrease the fitness of an individual (or more strongly, do not increase the fitness of the individual). If crossover fails this standard, then the parent is replicated into the new population in lieu of the failed child. By rejecting crossover which decreases fitness, tree growth is slowed. By rejecting crossover which does not increase fitness, tree growth is slowed even further. The claim made by the removal bias literature ((Soule and Foster, 1998; Langdon et al., 1999)) is that the difference in growth rates between these techniques is due to the rejection of any crossover events which occur in inviable code areas (and thus do not change the individual's fitness). Langdon and Poli (1998) use a related technique and report similar results.

The primary flaw in this argument is that this technique also acts to replicate large numbers of parents into future generations. Further, rejecting children with no increase in fitness serves to replicate many more parents than simply rejecting children which

decrease fitness. If some (unknown) code bloat force is causing children to be larger than their parents, then this technique may be doing little more than artificially dampening code growth by filling the population with parents and ancestors, which are generally smaller then their descendents.

5.1.4 Fitness Causes Bloat

The chief non-intron bloat theory in genetic programming, *fitness causes bloat*, is due to Langdon and Poli, and has been applied in various ways in (Langdon and Poli, 1997b,c; Langdon, 1998; Langdon and Poli, 1997a; Langdon et al., 1999; Langdon, 2000). According to this theory, in the space of programs, there are generally many more large-sized highly-fit trees than there are small-sized ones, if only because there are many more large trees than small ones overall. If genetic programming searched uniformly in this space, average tree size would be expected to remain constant. But genetic programming runs start out with small trees initially, and so code bloat might be explained simply as the system moving towards equilibrium.

This theory is supported with an argument borrowed from Price's Covariance and Selection Theorem (Price, 1970) from population genetics. Fitness-causes-bloat adapts this theorem to suggest a relationship between how selective the system is and the growth in tree size, taking a cue from Tackett (1994). When the evolutionary system suddenly chooses individuals at random (selectivity drops to 0), trees begin to shrink rather than grow. Langdon further argues that this code growth is not exponential but subquadratic in the number of evaluations (Langdon, 2000).

The general notions behind this theory are promising, though I feel there are problems with its specifics. While it may be true that there are generally many more large-sized high-fitness trees than small-sized ones, it is also likely that there are many *many* more large-sized *low*-fitness trees than small-sized ones. What matters here is not the ratio of numbers but of percentage. Also, by trying to stay as general as possible, the theory disregards the functional breeding relationship between small trees and large ones: certain large trees are much more likely than others to be bred from a given small tree. In saying that there are generally more highly-fit large trees than small ones, the model cannot predict that a *given* population trajectory is likely to find more highly-fit large trees than small ones. Given EC's finite population sizes, and the tendency of some representations to converge, a functional explanation is important.

6 Experiments

The two most prevalent intron theories of code growth (defense against crossover and removal bias) both rely on a similar thesis: that crossover in inviable subtree areas is the driving force behind tree growth in general. In this paper I present what I believe to be the first experimental evidence against this claim. The experiments do the surprisingly obvious: deny individuals the ability to cross over in inviable areas.

Blickle and Thiele (1994) proposed exactly this, calling it *marking*: identifying inviable code areas and disallowing crossover within those regions. Unfortunately, no code growth results were presented, and Blickle later wrote off the technique, stating "the marking method only avoids redundant crossover sites but does not address the bloating phenomenon directly as it leaves the redundant subtrees unchanged" (Blickle, 1996). In fact, marking does significantly decrease the amount of inviable code. But what is surprising is that marking does not appear to affect tree growth, even in inviable code-heavy domains.

The following experiments compare results with and without marking for the

Symbolic Regression, 11-Bit Multiplexer, and 6-Bit Multiplexer domains. The experimental methodology is as follows. The only breeding mechanism used was one-child crossover: two parents are picked from the population and crossed over to produce two children. However, while the first child is placed into the next generation, the second child is discarded. Each experiment consisted of 50 random runs. Experimental runs lasted 64 generations, or until an ideal solution was found, using a population of 512, and 7-tournament selection. Rather than using the traditional ad-hoc node-selection scheme (picking terminals 10% of the time), node selection was uniform. However, additional experiments run the traditional scheme have yielded very similar results. Additionally, no limit was placed on the size or depth of trees. Symbolic Regression did not use ephemeral random constants. ECJ was the genetic programming system used (Luke, 2000a). From these experiments was collected a variety of statistics, including:

- Mean number of nodes of individuals in the run.
- Mean tree depths in the run.
- Best fitness so far in the run.
- Number of individuals identical in fitness to their parents (neutral crossovers).

6.1 Multiplexer

For each Multiplexer domain, two sets of runs were performed, each with fifty independent runs. In the first set, runs were performed normally as described earlier. In the second set, the crossover point in the first parent was specially chosen through marking: a node was picked at random from the set of viable nodes in the individual. Since the child of the second parent was discarded, this meant that all children would be generated from a crossover point chosen from among viable nodes only.

Table 4 shows many common Multiplexer introns and their causes. One nice feature of Multiplexer is that it is possible, if expensive, to identify all inviable code. The most common forms of inviable code in Multiplexer are operations with true or false, such as (or *returns-true inviable*). In a less common form, parents and parents' siblings work together to eliminate the effect of inviable code. A simple example of this is (and (not a0) (and a0 *inviable*)). For 11-Bit Multiplexer, these less common forms are so rare that it is unlikely a single one will occur during a run. For 6-Bit Multiplexer, the less common forms are also rare, though they occur more often.

Results Tree growth results from these experiments are shown in Figures 3 and 4. After denying the ability to cross over in inviable regions, code growth in 6-Bit and 11-Bit Multiplexer *increased*. However, these increases were not statistically significant (using an ANOVA with an alpha value of 0.05). Nonetheless, they were not the expected *decreases* predicted by the intron theories. Tree depth similarly increased.

Inviable code results from the experiments are shown in Figure 7. One unexpected result was that 11-Bit Multiplexer has very little inviable code! When inviable code was restricted, the number of inviable nodes, as a percent of each individual, dropped dramatically for the 6-Multiplexer problem. The (negligible) rise in inviable code in 11-Multiplexer was also eliminated upon restriction. Note that the growth in neutral crossovers was unchanged. This is expected: an unusual feature of the Multiplexer domains is the very high likelihood of performing crossover which, while dramatically changing the functional semantics of an individual, does not change its overall score.

Inviable Code Examples	
0 as invalidator	($*$ 0 *inviable*)
	(% 0 *inviable*)
$\pm\infty$ or *NaN* as invalidator	($*$ $\pm\infty$*-or-NaN inviable*)
	(% $\pm\infty$*-or-NaN inviable*)
	(+ $\pm\infty$*-or-NaN inviable*)
	(- $\pm\infty$*-or-NaN inviable*)
Decimation	(rlog (+ (sin *inviable*) (exp (exp *returns-about-6*))))
Co-inviable code	($*$ 0 $\pm\infty$)
	(+ *NaN* $\pm\infty$)
	(% 0 0)
Invalidator Examples	
Creating 0	(- x x)
	(rlog (% x x))
Creating $\pm\infty$	(exp (exp (exp (exp (% x x)))))
Creating *NaN*	(sin $\pm\infty$)
Unoptimized Code Examples	
Redundant Subtrees	(+ (- x x) (rlog (% x x)))
Redundant Parents	(rlog (exp (rlog (exp (rlog x)))))

Table 3: Intron Examples for the Symbolic Regression Domain

Fitness change was negligible, except in 6-Bit Multiplexer, which found perfect scores much more easily without marking.

6.2 Symbolic Regression

Table 3 shows common Symbolic Regression introns and their causes. Symbolic Regression inviable code takes three forms: multiplication or division by zero, multiplication, division, addition or subtraction with infinity/NaN, and decimation.

Multiplying and dividing by zero is by far the most common cause of inviable code in Symbolic Regression: for example, ($*$ *always-returns-zero inviable*). Less obvious is how to achieve operations involving infinity or NaN. Structures returning infinity can be achieved with successive calls to exp, as in (exp (exp (exp (exp (exp *foo*))))). Structures returning NaN can be achieved with (sin *infinity*). However, infinity and NaN dominate the return value of an individual; as such when they appear, they give the individual the worst possible fitness, and so cannot propagate.

Decimation is more insidious. In decimation, parent nodes work together to eliminate the effect of a child by dropping its contribution below the precision of the data type. An example of decimated inviable code for the Java double type is (rlog (+ (sin *inviable*) (exp (exp *6*)))). While most forms of decimation are nearly impossible to identify directly, their effect can be ascertained indirectly by tracking the growth of neutral crossovers for which decimation can be the only possible culprit. In this regression experiment, such tracking determined that decimation never occurred prior to generation 15, and rarely occurred prior to generation 25.

Symbolic Regression differs from Multiplexer in that it is extremely unlikely that crossover of semantically different subtrees (that is, subtrees which return different values) will result in identical fitness, except within inviable code regions. This means that restricting semantically-identical crossover can have a significant impact on the

Inviable Code Examples	
true as invalidator	(or *true inviable*)
	(if *true executed inviable*)
false as invalidator	(and *false inviable*)
	(if *false inviable executed*)
Invalidation of test	(if *inviable* a0 a0)
Conspired invalidation	(and (not a0) (and a0 *inviable*))
	(or d0 (or *inviable* (not d0)))
Co-inviable code	(or *inviable-returns-true inviable-returns-true*)
	(and *inviable-returns-false inviable-returns-false*)
Invalidator Examples	
Creating true	(not (and a0 (not a0)))
Creating false	(and d1 (not d1))
Unoptimized Code Examples	
Redundant Subtrees	(and a0 (not a0))
	(or d0 (and d0 d0))
Redundant Parents	(not (not (not (not a0))))

Table 4: Intron Examples for the 6- and 11-Multiplexer Domains

number of neutral crossovers.

In order to gauge the effects of introns on tree growth in Symbolic Regression, four sets of runs were performed, each with fifty independent runs. In the first set, as usual, runs were performed unencumbered. In the second set, the crossover point for the first parent was chosen through marking. Note that this marking does not eliminate inviable code due to decimation.

In the third set, the crossover point for the first parent was chosen through marking, and the system rejected semantically identical crossover points. This means that once the system had selected two individuals for crossover, the system would try 500 times to find two semantically dissimilar subtrees to cross over. It would reject semantically identical subtrees such as x and (+ (- x x) x). If it failed 500 times (which only occurred in the rare case when both individuals were simply the terminal x), the first parent was replicated in lieu of its child. The goal of this set of runs was to eliminate any possibility that unoptimized code could be "increasing" neutral crossover points through semantically-identical crossover, which will be discussed more later. After applying these restrictions, there are only two possible causes for neutral crossover left: crossover inside decimated inviable code, and crossover between two trees consisting of the single terminal x (very rare after generation 7).

In the final set, the first parent was chosen through marking, the system rejected semantically identical crossover points, and decimated inviable code was controlled through a special technique: disallowing an individual to be selected if it had the same fitness as its parent. This final check effectively eliminated all neutral crossovers due to decimation. Note that this is different from the pseudo-hillclimbing technique used in the removal-bias literature: parents are not promoted in lieu of an identical-fitness child. Instead, the child is simply marked "bad" in the population and is not permitted to breed.

Results Tree growth results from all four experiment sets are shown in Figures 5 and 6. In all three restriction approaches, tree growth increased. Using an ANOVA with Fisher LSD at 0.05, this difference was statistically significant. Using an ANOVA with Tukey at 0.05, the tree growth increase was statistically significant in the third and fourth sets. Again, the surprising result is that the expected hypothesis (that tree growth would decrease) was not fulfilled. Tree depth similarly increased. Changes in fitness were again negligible. Inviable code results from the four experiments is shown in Figure 8. Note that the number of inviable nodes, as a percentage of the individual, dropped as experiments became increasingly draconian in their restrictions.

6.3 Discussion

Proponents of intron theories point to the increase in inviable nodes, neutral crossovers, and tree growth and suggest that the correlation among them is in fact causation: inviable code growth is driving tree growth. But consider the reverse relationship. Associated with each inviable subtree is an invalidator, a chunk of code responsible for making the subtree inviable. For example, in ($*$ 0 *inviable*), 0 is the invalidator. If invalidators were uniformly but randomly distributed, their effect on large trees would be much higher than on small trees, since in large trees there is a higher probability that an invalidator is proportionally closer to the root. Thus if trees grow but invalidator distribution remains constant, then the percentage of inviable code would be expected to grow naturally towards 100%.

As shown in Figures 7 and 8, invalidators generally remain constant throughout the run, as a percentage of the total individual. When inviable code crossover was permitted in the 6-Multiplexer and Symbolic Regression domains, the inviable percentage of an individual rose. 11-Multiplexer, as it turns out, rarely had invalidators, resulting in only a negligible rise in inviable code.

With crossover only in viable code regions, the likelihood increases that an ancestor of inviable code is chosen for crossover, thus swapping out the entire inviable tree; or that the invalidator itself is swapped out, thus making the inviable code viable once again. Thus one should expect the percentage of inviable code to drop significantly when denied crossover. Figures 7 and 8 show an unexpectedly strong drop: inviable code is nearly held at to a constant percentage.

Figure 6 shows an interesting result: in the harshest Symbolic Regression experiment, individuals with neutral crossover (at that point caused only by decimation) were immediately terminated, yet decimation events continued to rise, as indicated by the rise in neutral crossovers. This suggests that growth in decimated inviable code, when it occurs, is also caused by overall tree growth, rather than the other way around.

One theoretically possible source of tree growth is an increase in neutral crossover due to unoptimized code propagation. The idea here is that perhaps many unoptimized code subtrees are semantically identical, and so their propagation increases the likelihood that crossover might accidentally trade two such subtrees, resulting in neutral crossover; however in the experiments in Figure 6, such propagation was entirely eliminated yet code continued to grow unabated.

7 Analysis

If not introns, then what is causing these trees to grow? Additional statistics gathered from these results reveal interesting trends. For crossover events in the previous experiments, the following data was gathered from the first parent:

- The depth of the crossover point chosen.

- The size of the parent.
- The size of the subtree removed.
- The size of the subtree inserted.
- The number of times the child was later selected for crossover (its *child survivability*). Note that this is *not* the fitness of the child; it is roughly equivalent to the quality of the child relative to peers in its generation.

From this data was selected five generations' slices of data to view: 4, 8, 16, 32, and 64. Each data slice is labeled by the generation in which survivability data was gathered for each child, that is, when each child's children were evaluated. For example, "generation 4" means that in generation 2 the original parent was evaluated, in generation 3 its child was evaluated, and in generation 4 the child's children were evaluated.

If there is a positive relationship between child survivability and some factor in code bloat, then the population should be expected to move towards increasing that factor. The goal of this analysis is to compare candidate factors for tree bloat (parent size, crossover point depth, size of removed and inserted subtrees) against their effect on child survivability, with or without marking. Additionally, crossover point depth is compared against parent tree size and against removed subtree size to gauge relationships between them.

For each set of runs, nine subtables are presented showing regression analysis of the data. Tables 5 and 6 show analysis of Symbolic Regression with and without inviable code restricted. Tables 7 and 8 show analysis of 11-Multiplexer with and without inviable code restricted. Tables 9 and 10 show analysis of 6-Multiplexer with and without inviable code restricted.

For each set of runs, the first subtable presents multiple regression results for child survivability as a function of crossover point depth, parent tree size, removed subtree size, and inserted subtree size. The second subtable presents multiple regression results for child survivability as a function of crossover point depth, the log of parent tree size, the log of removed subtree size, and the log of inserted subtree size. The reason for the second table is to make certain that crossover point depth doesn't get an unfair advantage, if tree size is typically quadratic in depth. Both of these tables regress with a Poisson distribution and logarithmic link function, because child survivability more closely followed a Poisson curve than a normal curve. The tables are scaled with the square root of Pearson's χ^2/DoF.

The third and fourth tables present simple regression results for parent tree size as a function of tree depth, and removed subtree size as a function of tree depth, respectively. These two tables regress with a normal distribution and a linear link function, and are scaled by maximum likelihood. The final two tables in turn give simple regression results for child survivability as a function of parent tree size and of removed subtree size, showing these two factors independently rather than jointly as in the multiple regression tables. These last two tables regress with a Poisson distribution and logarithmic link function, and are scaled with the square root of Pearson's χ^2/DoF.

Those estimates whose alpha values exceed 0.05 (the test value used) are marked with asterisks. These estimates are statistically insignificant. Because of the high number of observations used (typically about 25,000), if an estimate's alpha value did not exceed 0.05, it was almost always less than 0.01 and usually much less than 0.0001. The deviance per degree of freedom prior to scaling is also given. As can be seen, the data

is overdispersed (it is usually much more or much less than 1.0). After scaling, the deviance was consistently very close to 1.0.

There are two caveats which must be taken into consideration. First, because this is a post-hoc analysis of the run data, the data slices are dependent from generation to generation. This means that conservative analysis of this data would take into consideration only the results for a *single* generation, as the other results are strongly tied to that generation. I suggest considering either generation 8 or 16, because generation 4's results are probably too reflective of initialization phenomena (its evaluation occurred only in generation 2), and generations later than 16 are probably not statistically relevant for 6-Multiplexer due to the drop in sample size. Second, crossover point depth, parent size, and removed subtree size are multicollinear, that is, there is a high correlation among them — deeper crossover point depths are related to larger parents and smaller removed subtrees.

Figures 9, 10, and 11 are provided to help visualize the data. For brevity, only generation 16 is shown, though other plots are similar; for a complete collection of plots, see (Luke, 2000b).

7.1 Results

Five features appear to be almost invariant. They are:

1. The strongest relationship with survivability is almost always its positive relationship with crossover point depth, even when other factors are placed on a log scale.

2. There is consistently a positive relationship between parent tree size and child survivability.

3. There is consistently a negative relationship between removed subtree size and survivability. But interestingly, this remains so even after marking. This effect is always stronger than that of inserted subtree size, with or without marking.

4. There is consistently a positive relationship between crossover point depth and parent tree size.

5. There is consistently a negative relationship between crossover point depth and removed subtree size.

Surprisingly, in the Symbolic Regression domain there is *also* a consistent negative relationship between inserted subtree size and child survivability. However, in the Multiplexer domains, there is only a weak positive relationship between inserted subtree size and child survivability. One oddity in the multiple regression tables (the first and second for each domain) is that when crossover point depth has a stronger estimate than parent tree size, the partial estimate for parent tree size goes negative. This doesn't mean that parent tree size is negatively correlated with child survivability, however. It is likely due to parent tree size being strongly multicollinear with crossover point depth.

8 A Depth-based Theory of Tree Growth

The results presented suggest that removed subtree size and parent tree size only have an indirect effect on child survivability in that all three are correlated with crossover point depth. They in turn are also correlated with child tree size, completing the chain

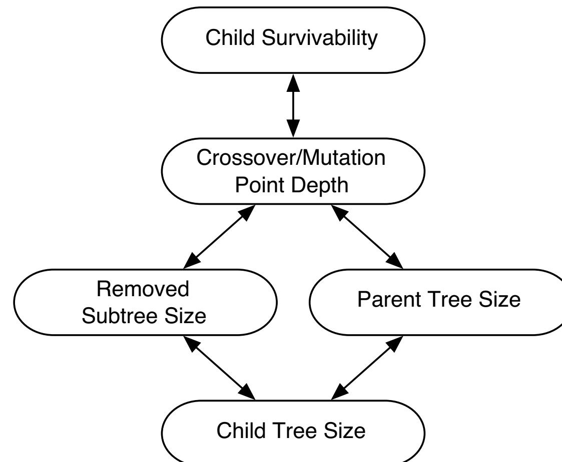


Figure 2: Correlative relationships between features during genetic programming evolution.

shown in Figure 2. The following model describes a functional relationship which explains this strong correlation between crossover point depth and child survivability.

In many evolutionary computation techniques, some components contribute more to the fitness of an individual than other components do. Due to peculiarities in the breeding operators in these techniques, components added to an individual are likely to be added in such a way that they have less contribution, and thus have relatively less effect on fitness. But *removing* some components from an individual runs the risk of removing those components with significant contribution, and thus it can have a dramatic effect on fitness. As an evolutionary computation run progresses, "changing fitness" increasingly means destroying fitness — and thus there is a fitness bias towards adding components versus removing components.

Unfortunately, the intron theories of code bloat have oversimplified this scenario, partitioning individuals' components into those which *have* an effect, and those which have *no* effect. There is no recognition of the notion that effects *differ in magnitude.* But the world is not this simple, and particularly not in genetic programmning.

In GP some nodes are more important than others. Removing some subtrees results in dramatic changes to an individual, while removing other subtrees results in minor changes, or in the case of inviable code, inconsequential changes.

As discussed in the previous analysis, one strong predictor of node "importance" is its depth within the tree. The reason for this seems straightforward: an s-expression is evaluated starting at the root, and the return value is obtained from the root. Tree nodes near the root often have a dramatic effect on the s-expression's operation, whereas tree nodes far away from the root have proportionally less effect, because their return values are decimated or they are rarely chosen from among a web of ancestor nodes' if-then constructs. Thus changing a tree node near the root can dramatically modify an individual, but changing a tree node far from the root has a correspondingly lower likelihood of modifying the individual by much.

Ordinarily this might be considered a good feature of genetic programming. The principle of *causality* (Rechenberg, 1973; Rosca and Ballard, 1995) argues that for evolu-

tionary computation it is good to have the magnitude of genotypical change positively correlated with the magnitude in phenotypical (and fitness) change. Since deep nodes often root small subtrees, depth seems likely to be related with the amount of genotypical change. Igel and Chellapilla (1999) analyzed causality in genetic programming both with the depth of the crossover point and with a metric of difference between the inserted and swapped out trees (called the *edit distance*), and found that while depth had a strong causal relationship in an individual, the relationship for edit distance was more tenuous.

However, this relationship between node depth and survivability can also explain tree bloat. Note that causality only relates the *magnitude* of change, not its direction (for better or for worse). But it is generally believed that in most common forms of genetic programming, modifying the individual is destructive most of the time (see (Teller, 1996; Nordin and Banzhaf, 1995)). If a decrease in node depth decreases the magnitude of change, and usually change is for the worse, then the expected survivability of a child is better if deep nodes are chosen for crossover or mutation points.

Such a bias towards node depth might have two effects on tree growth. First, it can promote larger parents, which have deeper crossover or mutation points. These larger parents in turn tend to produce larger children. Second, deeper-rooted removed subtrees are more likely to be small, but the inserted subtrees have no such size bias (a generalization of the removal bias theory).

One expected result of this theory is the emergence of *pseudoinviable code*, subtree areas for which crossover has a very low (but nonzero) probability of effecting an individual in a significant ways. The deeper a node is in the tree, the higher likelihood its rooted subtree will be forced into pseudoinviable code by its parents and ancestors. Since crossover in a pseudoinviable code region is unlikely to modify the individual by much, it is at one end of the gamut of strength of relationship between tree depth and survivability. Further, large trees would be expected to have correspondingly larger amounts of pseudoinviable code, just as they are expected to have larger amounts of inviable code. Both Blickle and Nordin *et al* noted the possible existence of pseudoinviable code (Blickle, 1996; Nordin et al., 1995). Smith and Harries (Smith and Harries, 1998) have recently performed experiments suggesting a strong relationship between pseudoinviable code and code bloat. True inviable code may be viewed as simply the most extreme end of the pseudoinviable spectrum.

Other effects might be explained by this theory. McPhee's inc-dec and Smith and Harries's R2 experiments both present contrived but important examples of code growth due to unoptimized code, but *not* due to neutral crossover — rather, trees seem to be simply filling up with junk (McPhee and Miller, 1995; Smith and Harries, 1998). As it turns out these experiments both appear to be arranged in such a way that deeper crossover points are less likely to cause destruction of the individual; however this effect deserves more examination.

Last, Langdon (2000) has noted that while tree depth appears to increase linearly during the course of GP runs, node size tends to increase subquadratically. This can be explained in two ways: either tree depth is driving tree size, or tree size is driving tree depth. Since depth is increasing linearly with the progression of an evolutionary run, the first of these two explanations seems the most intuitive.

9 Conclusions

Previous theories of code growth have suggested that the phenomenon is due to the propagation of inviable code. But the experiments in this paper contradict this prevail-

ing wisdom. After denying inviable code the ability to propagate, code growth actually increases. Further, the uniform distribution of node invalidators throughout the evolutionary run suggests that in fact not only is code growth not due to inviable code, but that it's the other way around. An increase in inviable code is an expected natural result of code growth.

Statistics gathered from the experimental work in this study suggest a more general explanation of code bloat: in evolutionary computation representations, some components are more important to the survivability of the individual than other components are. As it turns out, in GP this node-importance is strongly linked to node depth. Deeper nodes are generally less important than ones closer to the root. This bias in tree depth can result in a preference for larger parents (which beget larger children) and also a preference for smaller removed subtrees (with no corresponding preference for the size of replacing subtrees).

While genetic programming has the largest share of bloat papers, it turns out that bloating is a serious problem for a number of other less-studied representations, from neural networks to Pitt-approach rule systems. The depth-based node analysis presented in this paper is of course representation specific. But I believe that correlations between overall component significance and bloating in individuals might explain phenomena found in these other representations as well, and hope to explore this in future work.

A An Intron Bestiary

What follows are interesting examples of inviable code and unoptimized code events. This catalog is hardly complete; but it illustrates the most common cases.

A.1 Inviable Code

Inviable code is code which cannot contribute to an individual, no matter what it consists of. Thus modification of this code, through subtree crossover or mutation, cannot change the individual's fitness assessment in any way. Two common reasons for inviable code are: because it is never executed; or because its return value is ignored. Inviable code is often due to the presence of an *invalidator*, a chunk of code responsible for nullifying the inviable code.

Sibling Interference This is the most common inviable code form, where an invalidator forces a sibling's return value to be unused or causes it to be left unexecuted. Some examples: (∗ (- x x) *inviable*), (or (or x (not x)) *inviable*), (if (and x (not x)) *inviable always-executed*), (if *inviable* x x).

Co-inviable Code This unusual special case of sibling-interference inviable code deserves mentioning. Here two invalidators effectively force each other to be inviable code. For example in (∗ (- x x) (rlog (% x x))), both (- x x) and (rlog (% x x)) return 0.

Propagated Inviable Code A single invalidator sets off a chain of inviable-code events. For example in (∗ *inviable2* (∗ (- x x) *inviable1*)), *inviable1* is nullified by the invalidator (- x x) which returns 0, and *inviable2* is in turn nullified by the propagated invalidator (∗ (- x x) *inviable1*) which also returns 0. (- x x) is called a *base invalidator*.

Denial of Action In some domains, code can be made effectively inviable by tying its viability to some event in the evaluation period which never occurs. For example, in the code snippet (if-food-ahead *inviable always-executed*), the inviable code *could* be executed

if there was ever food ahead, but as it turns out every time this food test was evaluated, there is never food ahead.

Delay Inviable code can occur because by the time it is reached it is unneeded, typically because the individual's evaluation has ended. Such inviable code is often highly dependent on execution order in the tree. For example, in some problems the program is permitted to run only for some n time steps; if the tree consists of more than n commands, then further nodes in the tree are ignored.

Remote Causes Inviable code may occur because an earlier-executed remote invalidator caused the code to be inviable when it is reached later in execution. This occurs with domains which read and write to a global memory. For example, a subtree executed early on may execute (set-internal-state 1); a later-executed code snippet might read (if (= 1 (get-internal-state)) *always-executed inviable*).

Decimation Parents and parents' siblings conspire to eliminate the effect of an inviable subtree's return value by dropping it below the effective precision of the computer system's data type. For example: (rlog (+ (sin *inviable*) (exp (exp 6)))) will return the same value (about 403.43) regardless of the value of the inviable subtree, when using the Java double type.

Conspired Invalidation Parents and parents' siblings conspire to eliminate the effect of an inviable subtree's return value by moving it beyond some boundary condition. For example, if my-log were defined to return 0 for any negative number, (my-log (- (sin *inviable*) 1.0)) would always return 0. Another example occurs infrequently in the Multiplexer domains: (and (not a0) (and *inviable* a0)).

Redundant Tests Nested if-statements can make it impossible for a subtree to ever be evaluated. For example, (if-food-ahead *evaluated-when-true* (if-food-ahead *inviable evaluated-when-false*))

Uncalled Automatically Defined Functions and Macros Some Genetic Programming problems rely on trees with *automatically defined functions* (ADFs) or *macros* (ADMs) which are called as subfunctions of the main tree. If an ADF/ADM tree is never in fact called by the main program tree, then that ADF/ADM tree is inviable code.

A.2 Unoptimized Code

Unoptimized code is code which is excessively redundant in form and can be intelligently cleaned up without changing the operation of the individual. However, arbitrary modification of unoptimized code can change the individual's operation. Thus unoptimized code is viable.

Mutually Exclusive Parents A chain of parents which together form the identity function on the data passed through them and eliminate each others' side effects when executed. For example, (not (not *foo*)) or (turn-left-then (turn-right-then *foo*)).

Redundant Subtrees Two subtrees return or perform exactly the same thing, and so can be condensed into a single subtree. For example, (and (or x1 x2) (or x1 x2)).

Mutually Exclusive Subtrees Two subtrees cancel each others' side effects. As in (prog2 (prog2 turn-left turn-left) (prog2 turn-right turn-right)).

Identity Conditions The return value of a subtree is worthless because it converts its parent into the identity function and eliminates any side effects. Examples: (+ *foo* 0), (∗ *foo* 1), (set-internal-state (get-internal-state)), or (and *foo bar*), where *bar* always returns true.

Other Unoptimized Code A variety of messy code structures can be cleaned up through copy propagation, constant folding, common subexpression elimination, and other code optimization procedures. For example: (+ 4.2 (+ 3.5 *foo*)). Strictly speaking this is not unoptimized code as formally defined in this paper, since all elements in the subtree do contribute, in some sense, to the function of the individual.

References

Andre, D. and Teller, A. (1996). A study in program response and the negative effects of introns in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 12–20, Stanford University, CA, USA. MIT Press.

Angeline, P. J. (1994). Genetic programming and emergent intelligence. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, pages 75–98. MIT Press.

Angeline, P. J. (1996). Two self-adaptive crossover operators for genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, pages 89–110. MIT Press, Cambridge, MA, USA.

Angeline, P. J. (1998). Subtree crossover causes bloat. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 745–752, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, pages 54–65.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann.

Bassett, J. K. and De Jong, K. A. (2000). Evolving behaviors for cooperating agents. In Ras, Z. W. and Ohsuga, S., editors, *Proceedings of the Twelth International Symposium on Methodologies for Intelligent Systems*, pages 157–165. Springer-Verlag.

Bleuler, S., Brack, M., Thiele, L., and Zitzler, E. (2001). Multiobjective genetic programming: Reducing bloat using SPEA2. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 536–543, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea. IEEE Press.

Blickle, T. (1996). *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich.

Blickle, T. and Thiele, L. (1994). Genetic programming and redundancy. In Hopf, J., editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Saarbrücken,Germany. Max-Planck-Institut für Informatik.

Boers, E. J. W., Kuiper, H., M., B. L., and Sprinkhuizen-Kuyper, I. G. (1993). Designing modular artificial networks. In Wijshoff, H. A., editor, *Proceedings of Computing Science in The Netherlands*, pages 154–159, Amsterdam. SION, Stichting Mathematisch Centrum.

Brave, S. (1996). Evolving deterministic finite automata using cellular encoding. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 39–44, Stanford University, CA, USA. MIT Press.

Burke, D., De Jong, K., Grefenstette, J., Ramsey, C., and Wu, A. (1998). Putting more genetics into genetic algorithms. *Evolutionary Computation*, 6(4).

Collins, R. and Jefferson, D. (1991). An artificial neural network representation for artificial organisms. In Schwefel, H. P. and Mäanner, R., editors, *Parallel Problem Solving from Nature*, pages 269–263, Berlin. Springer-Verlag.

de Jong, E. D., Watson, R. A., and Pollack, J. B. (2001). Reducing bloat and promoting diversity using multi-objective methods. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 11–18, San Francisco, California, USA. Morgan Kaufmann.

De Jong, K. (1989). Using genetic algorithms to learn task programs: the pitt approach. *Machine Learning*, 2(2-3).

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley, New York.

Fullmer, B. and Miikkulainen, R. (1991). Using marker-based genetic encoding of neural networks to evolve finite-state behavior. In *Proceedings of the First European Conference on Artificial Life (ECAL-91), Paris*.

Gathercole, C. and Ross, P. (1996). An adverse interaction between crossover and restricted tree depth in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA. MIT Press.

Goldberg, D. E., Deb, K., Kargupta, H., and Harik, G. (1993). Rapid, accuate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth Annual International Conference of Genetic Algorithms (ICGA-93)*, pages 56–64.

Gruau, F. (1992). Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In Schaffer, J. D. and Whitley, D., editors, *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN92)*, pages 55–74. The IEEE Computer Society Press.

Handley, S. (1994). On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 154–159, Orlando, Florida, USA. IEEE Press.

Haynes, T. (1998). Collective adaptation: The exchange of coding segments. *Evolutionary Computation*, 6(4):311–338.

Iba, H., de Garis, H., and Sato, T. (1994). Genetic programming using a minimum description length principle. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, pages 265–284. MIT Press.

Iba, H. and Terao, M. (2000). Controlling effective introns for multi-agent learning by genetic programming. In Whitley, D., Goldberg, D., Cantú-Paz, E., Spector, L., Parmee, I., and Beyer, H.-G., editors, *GECCO-2000: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 419–426. Morgan Kaufmann, San Fransisco.

Igel, C. and Chellapilla, K. (1999). Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1061–1068, Orlando, Florida, USA. Morgan Kaufmann.

Jakobi, N. (1995). Harnessing morphogenesis. In *International Conference on Information Processing in Cells and Tissues*.

Jones, E. A. and Joines, W. T. (1999). Genetic design of electronic circuits. In Brave, S. and Wu, A. S., editors, *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pages 125–133, Orlando, Florida, USA.

Kargupta, H. (1996). The gene expression messy genetic algorithm. In *Proceedings of the IEEE International Conference on Evolutionary Computation*.

Keijzer, M. (1996). Efficiently representing populations in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, pages 259–278. MIT Press, Cambridge, MA, USA.

Kitano, H. (1990). Designing neural networks using a genetic algorithm with a graph generation system. *Complex Systems*, 4:461–476.

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

Koza, J. R., Bennett III, F. H., Hutchings, J. L., Bade, S. L., Keane, M. A., and Andre, D. (1997a). Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate arrays. In Higuchi, T., editor, *Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence*, pages 27–32, Nagoya.

Koza, J. R., Bennett III, F. H., Lohn, J., Dunlap, F., Keane, M. A., and Andre, D. (1997b). Automated synthesis of computational circuits using genetic programming. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, pages 447–452, Indianapolis. IEEE Press.

Koza, J. R. and Rice, J. P. (1991). Genetic generation of both the weights and architecture for a neural network. In *International Joint Conference on Neural Networks, IJCNN-91*, volume 2, pages 397–404, Washington State Convention and Trade Center, Seattle, WA, USA. IEEE Computer Society Press.

Langdon, W. B. (1998). The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, Alaska, USA. IEEE Press.

Langdon, W. B. (2000). Quadratic bloat in genetic programming. In Whitley, D., Goldberg, D., Cantú-Paz, E., Spector, L., Parmee, I., and Beyer, H.-G., editors, *GECCO-2000: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 451–458, San Fransisco. Morgan Kaufmann.

Langdon, W. B. and Poli, R. (1997a). An analysis of the MAX problem in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 222–230, Stanford University, CA, USA. Morgan Kaufmann.

Langdon, W. B. and Poli, R. (1997b). Fitness causes bloat. In Chawdhry, P. K., Roy, R., and Pant, R. K., editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London.

Langdon, W. B. and Poli, R. (1997c). Fitness causes bloat: Mutation. In Koza, J., editor, *Late Breaking Papers at the GP-97 Conference*, pages 132–140, Stanford, CA, USA. Stanford Bookstore.

Langdon, W. B. and Poli, R. (1998). Genetic programming bloat with dynamic fitness. In Banzhaf, W., Poli, R., Schoenauer, M., and Fogarty, T. C., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 96–112, Paris. Springer-Verlag.

Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1999). The evolution of size and shape. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, pages 163–190. MIT Press, Cambridge, MA, USA.

Lanzi, P. L. (1999). Extending the representation of classifier conditions part ii: from messy coding to s-expressions. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 345–352, Orlando, Florida, USA. Morgan Kaufmann.

Lindgren, K., Nilsson, A., Nordahl, M. G., and Råde, I. (1992). Regular language inference using evolving neural networks. In Schaffer, J. D. and Whitley, D., editors, *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN92)*, pages 55–74, Los alamitos. The IEEE Computer Society Press.

Luke, S. (2000a). ECJ: A Java-based evolutionary computation and genetic programming system. Available at http://www.cs.umd.edu/projects/plus/ec/ecj/.

Luke, S. (2000b). *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD thesis, Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA.

Luke, S., Hamahashi, S., and Kitano, H. (1999). "Genetic" programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1098–1105, Orlando, Florida, USA. Morgan Kaufmann.

McPhee, N. F. and Miller, J. D. (1995). Accurate replication in genetic programming. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA. Morgan Kaufmann.

Nordin, P. (1997). *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik.

Nordin, P. and Banzhaf, W. (1995). Complexity compression and evolution. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA. Morgan Kaufmann.

Nordin, P., Francone, F., and Banzhaf, W. (1995). Explicitly defined introns and destructive crossover in genetic programming. In Rosca, J. P., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, California, USA.

Nordin, P., Francone, F., and Banzhaf, W. (1996). Explicitly defined introns and destructive crossover in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, pages 111–134. MIT Press, Cambridge, MA, USA.

O'Reilly, U.-M. (1995). *An Analysis of Genetic Programming*. PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada.

Price, G. R. (1970). Selection and covariance. *Nature*, 227:520–521.

Radcliffe, N. and George, F. (1993). A study in set recombination. In *Proceedings of the Fifth Annual International Conference of Genetic Algorithms (ICGA-93)*, pages 22–30.

Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen evolution ("Evolution strategy: the optimization of technical systems according to the principles of biological evolution")*. Frommann–Holzboog Verlag, Stuttgart.

Rosca, J. (1996). Generality versus size in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387, Stanford University, CA, USA. MIT Press.

Rosca, J. and Ballard, D. H. (1995). Causality in genetic programming. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 256–263, Pittsburgh, PA, USA. Morgan Kaufmann.

Rosca, J. P. (1997). Analysis of complexity drift in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA. Morgan Kaufmann.

Rowe, J. E. and McPhee, N. F. (2001). The effects of crossover and mutation operators on variable length linear structures. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 535–542, San Francisco, California, USA. Morgan Kaufmann.

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

Smith, S. F. (1980). *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, Computer Science Department, University of Pittsburgh.

Soule, T. and Foster, J. A. (1998). Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA. IEEE Press.

Soule, T., Foster, J. A., and Dickinson, J. (1996). Code growth in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA. MIT Press.

Stoffel, K. and Spector, L. (1996). High-performance, parallel, stack-based genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Stanford University, CA, USA. MIT Press.

Tackett, W. A. (1994). *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems.

Teller, A. (1996). Evolving programmers: The co-evolution of intelligent recombination operators. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, pages 45–68. MIT Press, Cambridge, MA, USA.

Teller, A. (1998). *Algorithm Evolution with Internal Reinforcement for Signal Understanding*. PhD thesis, School of Computer Science, Carnegie Mellon University.

Wu, A. and Lindsay, R. (1995). Empirical studies of the genetic algorithm with non-coding segments. *Evolutionary Computation*, 3(2).

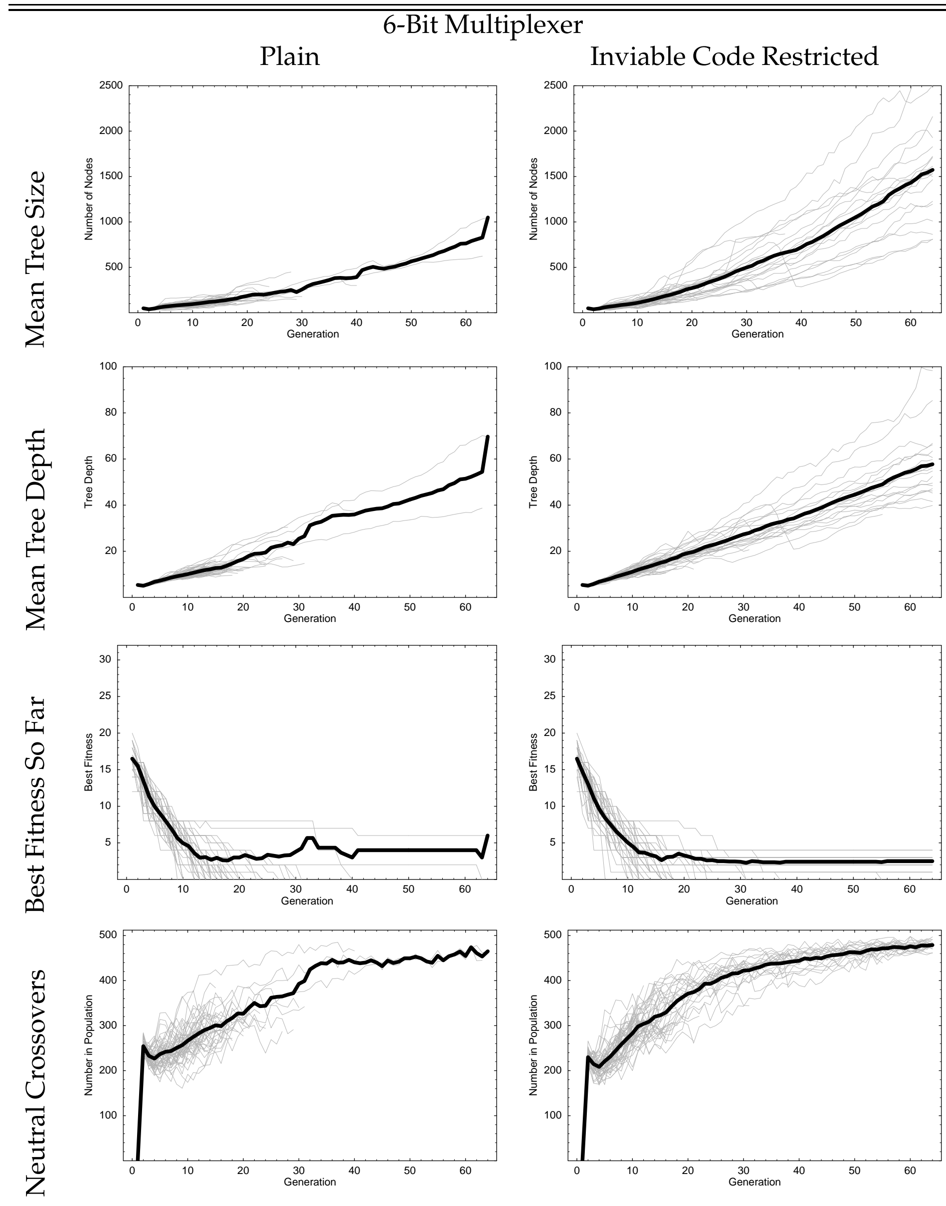


Figure 3: Node Statistics for 50 Runs in the 6-Bit Multiplexer Domain.

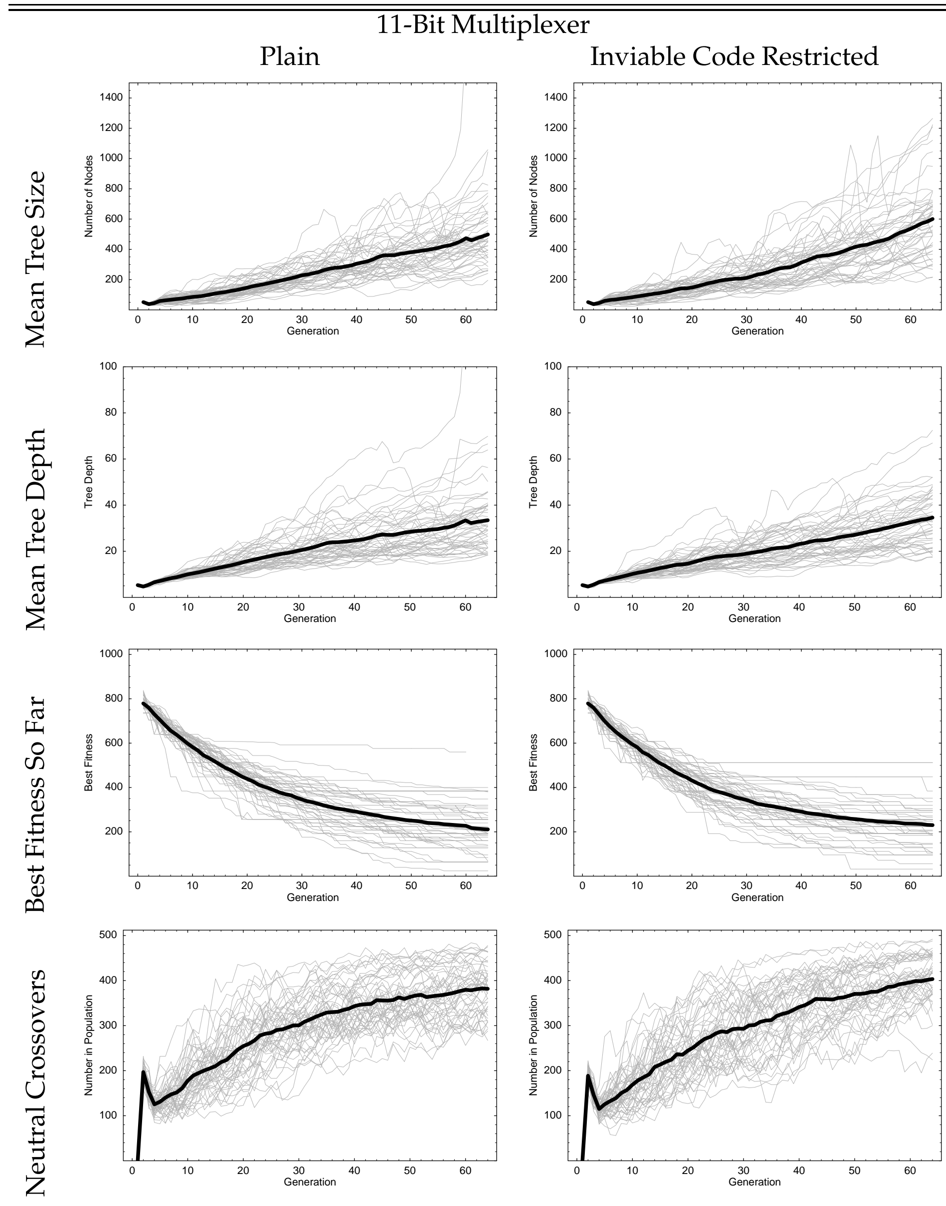


Figure 4: Node Statistics for 50 Runs in the 11-Bit Multiplexer Domain

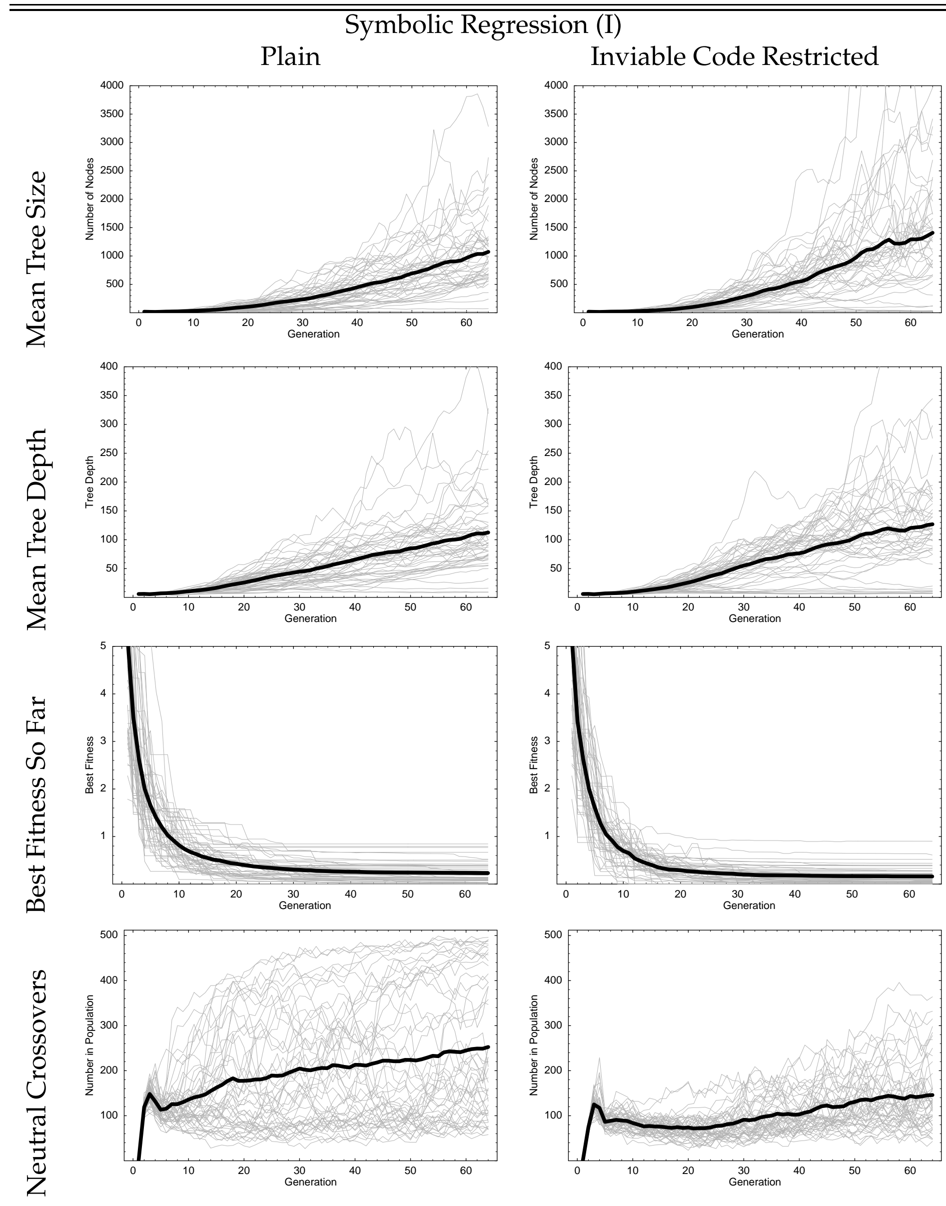


Figure 5: Node Statistics for 50 Runs in the Symbolic Regression Domain (Part I)

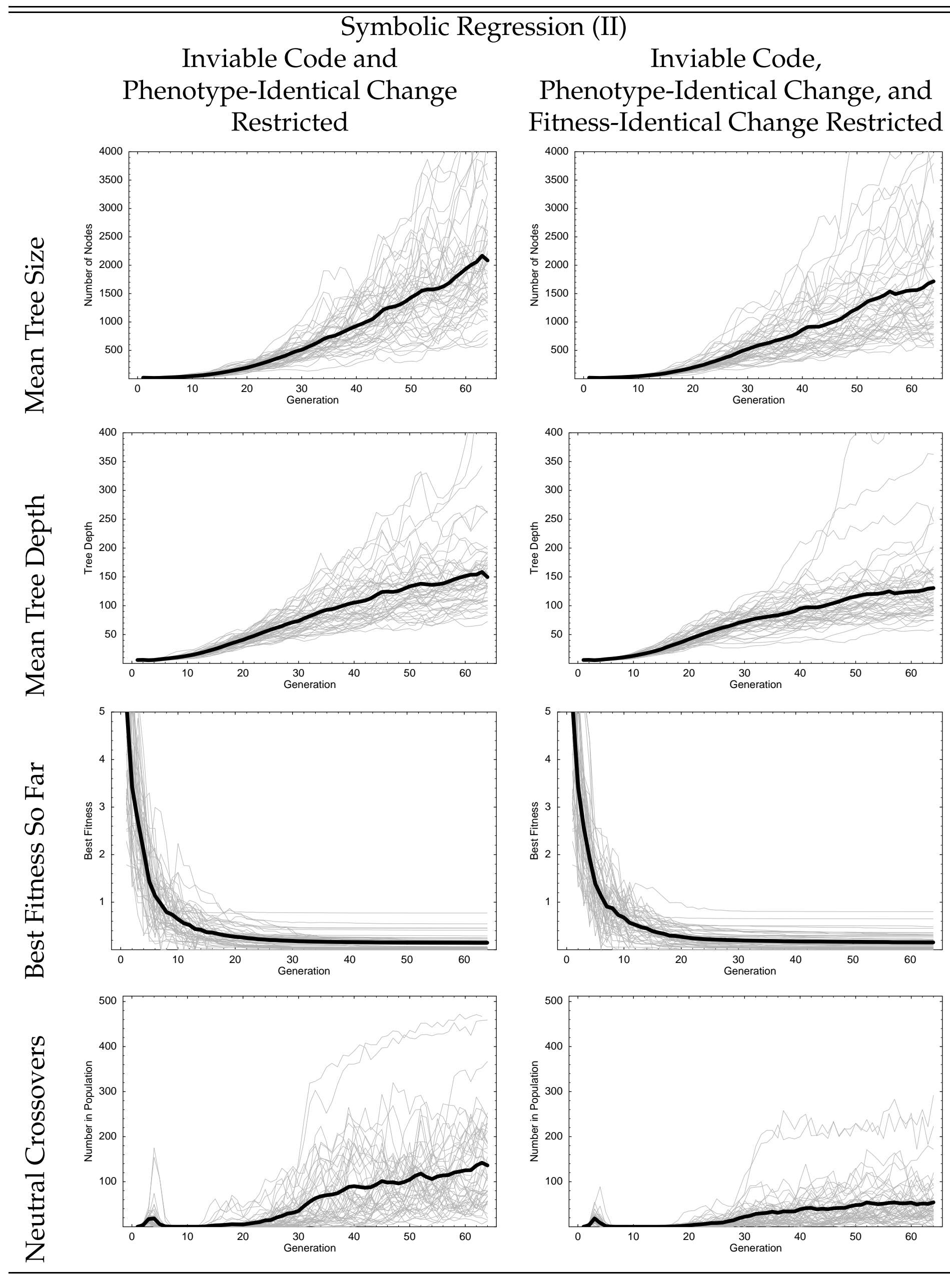


Figure 6: Node Statistics for 50 Runs in the Symbolic Regression Domain (Part II)

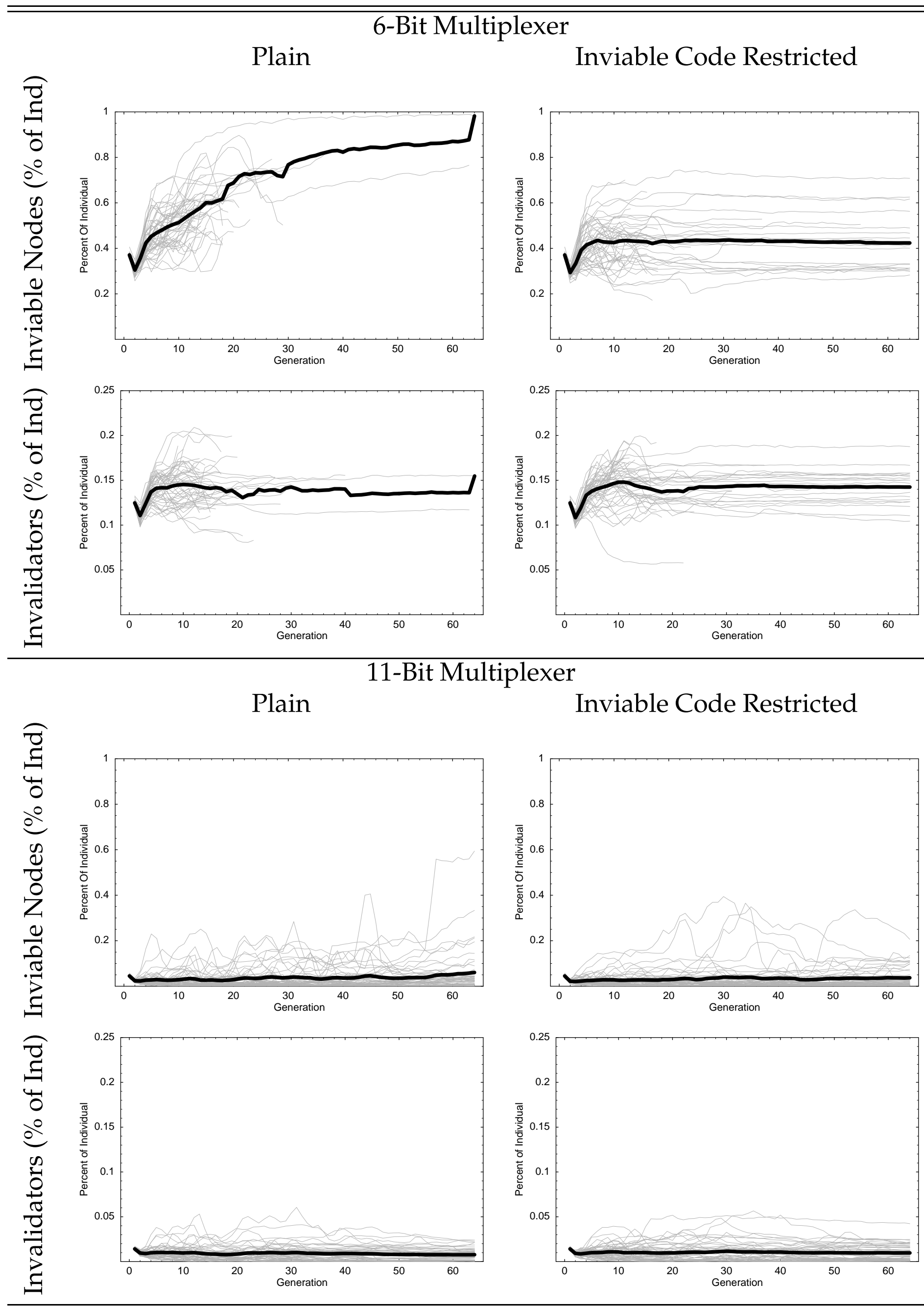


Figure 7: Inviable Code Statistics for 50 Runs in the 6-Bit and 11-Bit Multiplexer Domains.

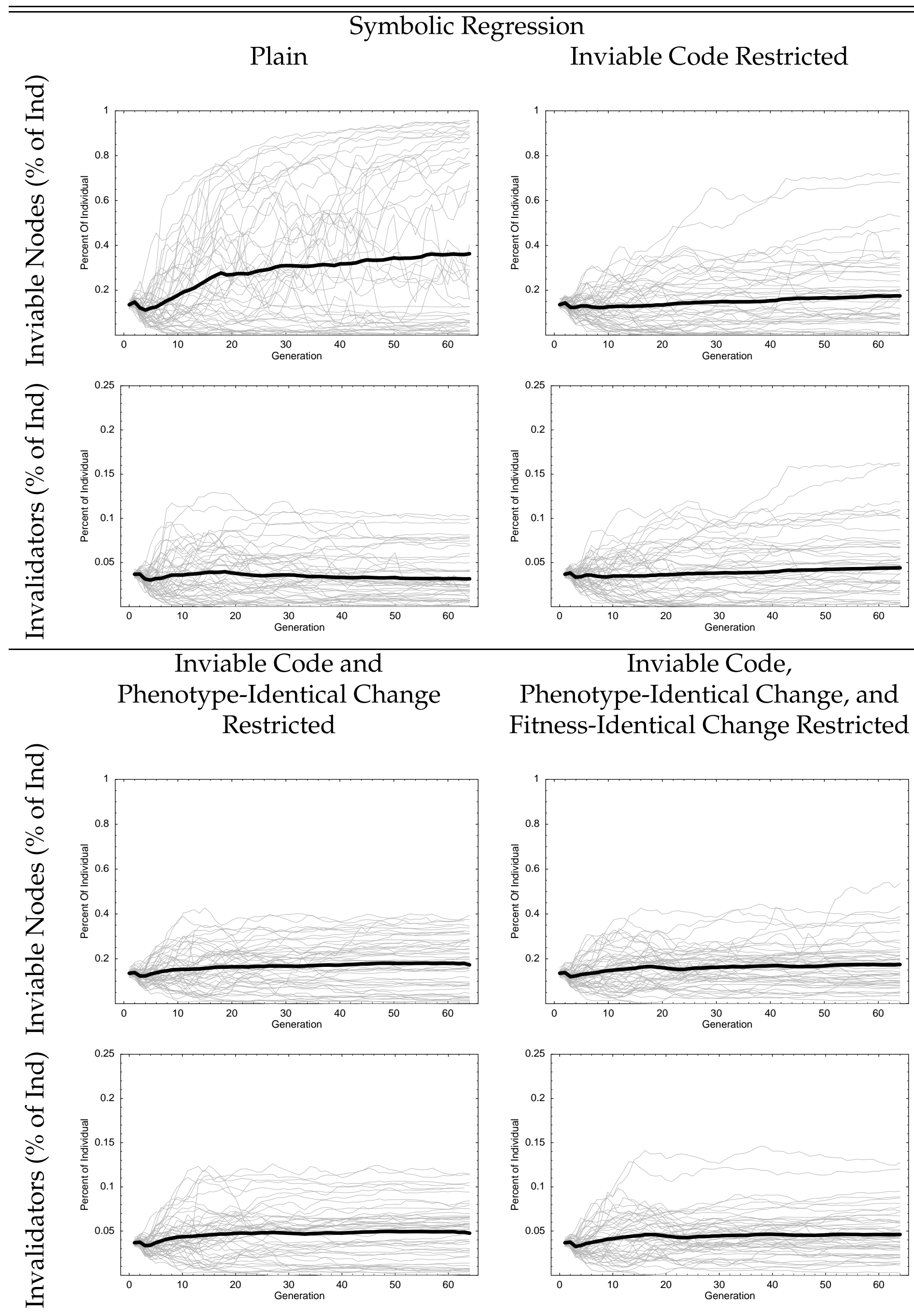
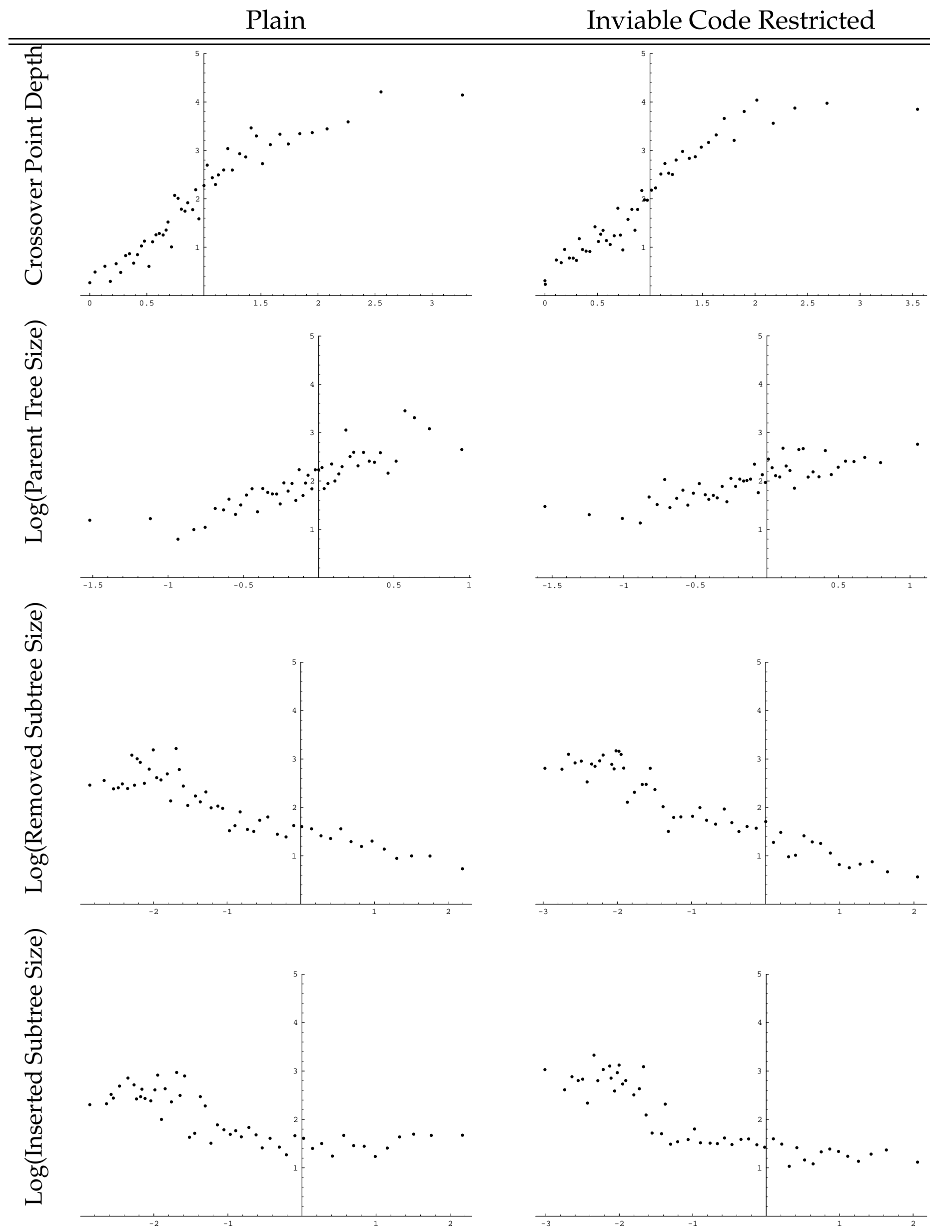
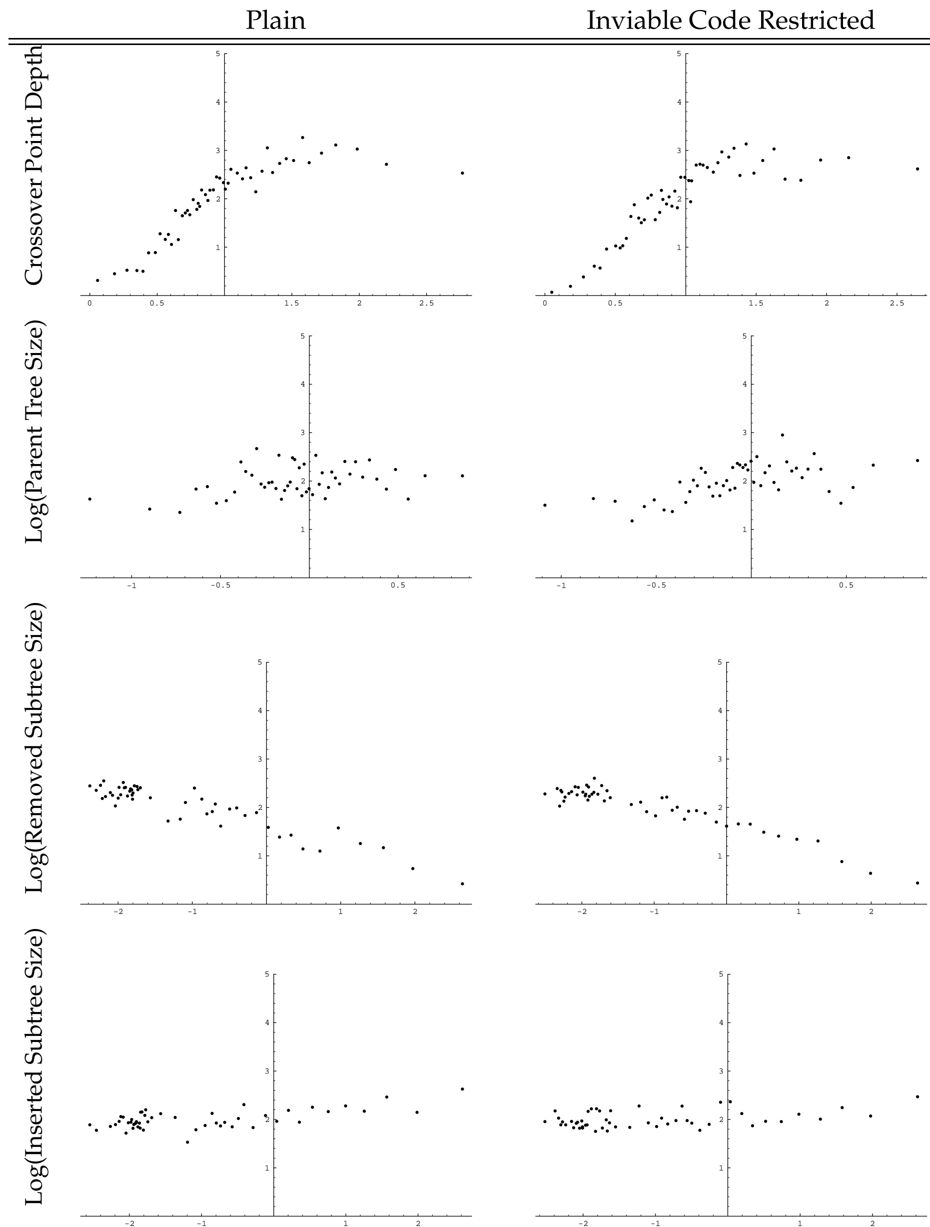


Figure 8: Inviable Code Statistics for 50 Runs in the Symbolic Regression Domain.



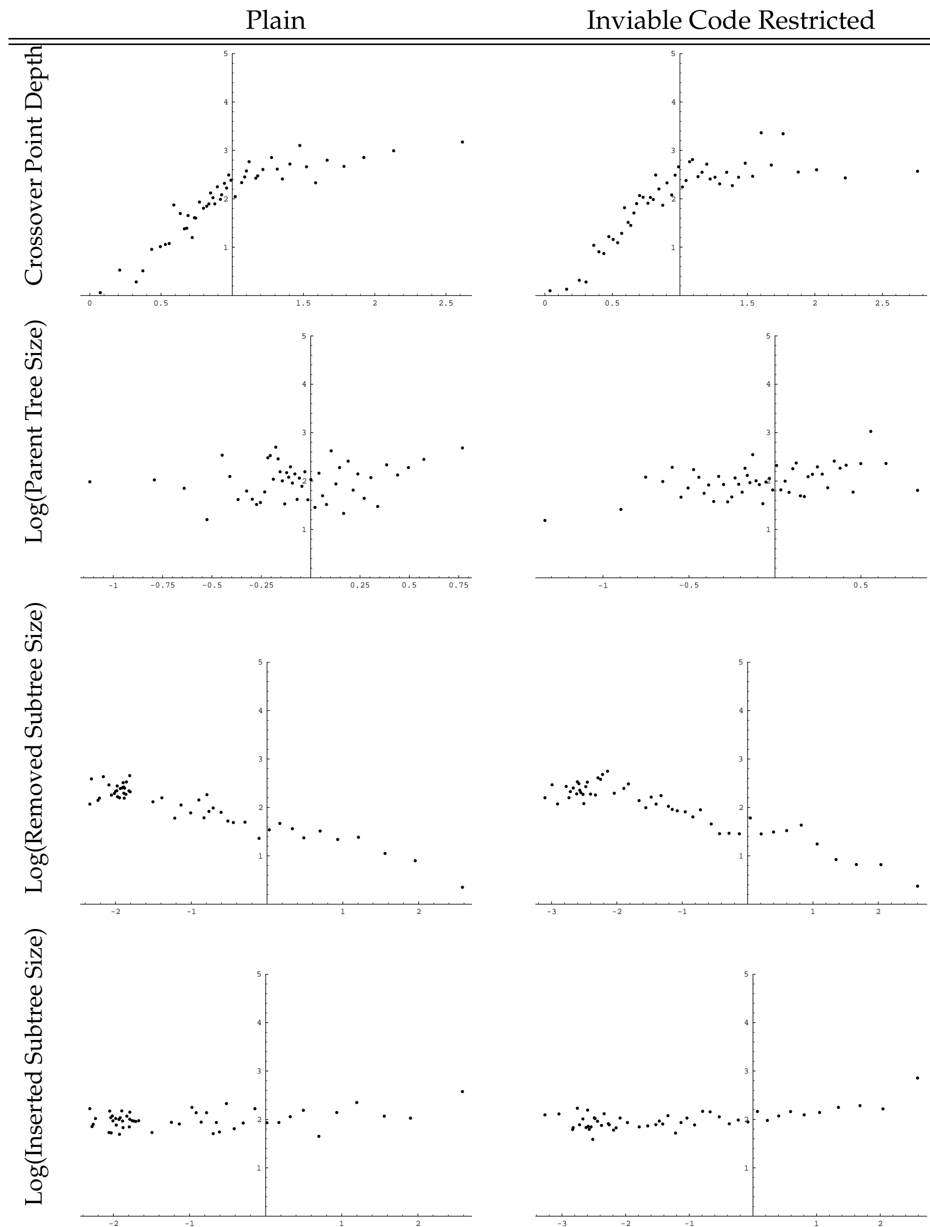
Legend. Child survivability (Y axis) is the number of times the child is selected in the next generation. The crossover statistics (X axis) were first normalized by dividing by the mean. All crossover statistics except crossover point depth were then transformed to a log scale. Data was sorted and divided evenly into 50 bins by the crossover statistics variable. Points show the mean of each bin.

Figure 9: Relationship between four crossover statistics and child survivability for the Symbolic Regression domain, Generation 16, with or without permitting inviable code.



Legend. Child survivability (Y axis) is the number of times the child is selected in the next generation. The crossover statistics (X axis) were first normalized by dividing by the mean. All crossover statistics except crossover point depth were then transformed to a log scale. Data was sorted and divided evenly into 50 bins by the crossover statistics variable. Points show the mean of each bin.

Figure 10: Relationship between four crossover statistics and child survivability for the 11-Multiplexer domain, Generation 16, with or without permitting inviable code.



Legend. Child survivability (Y axis) is the number of times the child is selected in the next generation. The crossover statistics (X axis) were first normalized by dividing by the mean. All crossover statistics except crossover point depth were then transformed to a log scale. Data was sorted and divided evenly into 50 bins by the crossover statistics variable. Points show the mean of each bin.

Figure 11: Relationship between four crossover statistics and child survivability for the 6-Multiplexer domain, Generation 16, with or without permitting inviable code.

Multiple Regression of Child Survivability by Depth, Parent Size, Removed Size, Inserted Size

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.1556	0.1438	-0.1481	-0.0671	0.5673	2.5249	4.7766
8	0.4381	-0.1031	-0.3130	-0.1895	0.6953	2.5954	4.3241
16	0.5521	*-0.0016	-0.1379	-0.0856	0.2393	2.3894	4.1211
32	0.4953	-0.2495	-0.1024	-0.0413	0.5091	2.2728	3.9419
64	0.4940	-0.2505	-0.0458	-0.0102	0.4548	2.1539	3.6412

Multiple Regression of Child Survivability by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	*-0.0267	0.2467	-0.3306	-0.1730	0.5209	2.5304	4.6923
8	0.3593	*0.0249	-0.3715	-0.3140	-0.2328	2.4805	4.1844
16	0.5047	0.1144	-0.1811	-0.1596	-0.2547	2.3718	4.0341
32	0.4603	-0.2027	-0.1151	-0.0894	-0.1884	2.2398	3.9076
64	0.4637	-0.1719	-0.0497	-0.0385	*-0.0240	2.1594	3.6451

Regression of Removed Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	-0.4196	0.4401	0.6013	0.3615
8	-0.5767	0.5275	0.4975	0.2475
16	-0.6683	0.6285	0.3994	0.1596
32	-0.5983	0.5530	0.4602	0.2118
64	-0.8792	0.6350	0.3985	0.1588

Regression of Parent Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	0.5598	1.4181	1.2474	1.5562
8	0.4724	1.5763	1.2953	1.6778
16	0.3715	1.6683	1.6326	2.6655
32	0.4470	1.5983	2.2798	5.1979
64	0.3650	1.8792	2.9464	8.6818

Regression of Child Survivability by Removed Size

Gen	Removed	Intercept	Scale	Prescaled Dev/DF
4	-0.1562	0.8333	2.5491	4.8236
8	-0.4359	1.0207	2.6725	4.5604
16	-0.2166	*0.8623	2.4556	4.2020
32	-0.1467	0.8018	2.3111	4.0427
64	-0.0711	0.7480	2.1785	3.6854

Regression of Child Survivability by Parent Size

Gen	Parent	Intercept	Scale	Prescaled Dev/DF
4	0.1964	0.4864	2.5256	4.8236
8	0.1902	0.4960	2.5049	4.5604
16	0.4466	0.2222	2.4453	4.2020
32	0.1297	0.5608	2.3134	4.0427
64	0.1081	0.5838	2.1907	3.6854

Legend. Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table 5: Exploratory Regression Analysis of the Symbolic Regression Domain

Multiple Regression of Child Survivability by Depth, Parent Size, Removed Size, Inserted Size

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.1852	0.1104	-0.1153	-0.0304	0.5038	2.4793	4.7501
8	0.3745	-0.1022	-0.2140	-0.1825	0.7016	2.6143	4.5322
16	0.5354	-0.1691	-0.2463	-0.1884	0.5757	2.5345	4.2825
32	0.5448	-0.2539	-0.1559	-0.0504	0.4959	2.4962	4.4755
64	0.5411	-0.3333	-0.1243	-0.0266	0.5431	3.6327	4.5287

Multiple Regression of Child Survivability by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.0432	0.2126	-0.2688	-0.1269	0.4752	2.4923	4.6827
8	0.2859	*0.0326	-0.3346	-0.3024	-0.1183	2.5732	4.3687
16	0.4931	-0.0801	-0.2429	-0.2409	-0.4383	2.4546	4.1566
32	0.4713	-0.1536	-0.1192	-0.0824	-0.2128	2.4493	4.4731
64	0.3858	-0.1967	-0.0836	-0.0593	-0.0720	2.4329	4.5663

Regression of Removed Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	-0.3678	0.4869	0.6396	0.4091
8	-0.4633	0.5559	0.5659	0.3203
16	-0.5221	0.6192	0.4438	0.1970
32	-0.6351	0.5593	0.4396	0.1933
64	-0.8571	0.5276	0.4626	0.2140

Regression of Parent Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	0.5128	1.3668	1.2846	1.6503
8	0.4441	1.4632	1.3062	1.7063
16	0.3808	1.5221	1.4837	2.2015
32	0.4407	1.6351	2.0503	4.2041
64	0.4724	1.8571	2.7484	7.5544

Regression of Child Survivability by Removed Size

Gen	Removed	Intercept	Scale	Prescaled Dev/DF
4	-0.1310	0.8113	2.5291	4.7757
8	-0.3078	0.9397	2.6339	4.7164
16	-0.3496	0.9487	2.5758	4.5311
32	-0.2108	0.8423	2.5156	4.6262
64	-0.1648	0.8008	2.6458	4.6847

Regression of Child Survivability by Parent Size

Gen	Parent	Intercept	Scale	Prescaled Dev/DF
4	0.1845	0.4980	2.5090	4.7757
8	0.1057	0.5850	2.5552	4.7164
16	0.2617	0.4215	2.5606	4.5311
32	0.1652	0.5238	2.5113	4.6262
64	0.0387	0.6542	2.4830	4.6847

Legend. Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table 6: Exploratory Regression Analysis of the Symbolic Regression Domain with Inviable Code Restricted

Multiple Regression of Child Survivability by Depth, Parent Size, Removed Size, Inserted Size

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.5861	0.1121	-0.0398	0.0246	-0.0920	2.5291	4.5488
8	0.4123	-0.1078	-0.1912	0.0186	0.4810	2.6392	4.7157
16	0.5441	-0.2494	-0.1076	0.0230	0.4093	2.5156	4.5584
32	0.6007	-0.3066	-0.0558	0.0080	0.3795	2.3085	4.0362
64	0.4074	-0.2146	-0.0399	0.0045	0.4992	1.7202	2.5757

Multiple Regression of Child Survivability by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.3234	0.2303	-0.1362	0.0655	0.3482	2.5313	4.5120
8	0.3871	-0.0521	-0.2092	0.0391	0.1029	2.5338	4.7306
16	0.5069	-0.1820	-0.1497	0.0494	0.0052	2.5002	4.5631
32	0.5804	-0.2442	-0.0749	0.0129	*-0.0456	2.3121	4.0489
64	0.4049	-0.2110	-0.0549	0.0096	*0.1672	1.7253	2.5829

Regression of Removed Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	-0.9333	0.1310	0.6794	0.4616
8	-1.9229	0.6544	0.4121	0.1698
16	-1.3266	0.6905	0.3522	0.1241
32	-1.2333	0.7647	0.3140	0.0986
64	-1.0214	0.7370	0.3406	0.1160

Regression of Parent Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	0.8690	1.9333	1.9655	3.8634
8	0.3456	2.9228	1.8325	3.3583
16	0.3095	2.3266	2.2633	5.1227
32	0.2353	2.2333	2.6967	7.2726
64	0.2630	2.0214	3.3274	11.0727

Regression of Child Survivability by Removed Size

Gen	Removed	Intercept	Scale	Prescaled Dev/DF
4	-0.1180	0.7919	2.5423	4.5681
8	-0.2623	0.8801	2.9196	4.8046
16	-0.1638	0.8123	2.5472	4.6313
32	-0.0943	0.7644	2.3266	4.0798
64	-0.0568	0.7372	1.7279	2.6056

Regression of Child Survivability by Parent Size

Gen	Parent	Intercept	Scale	Prescaled Dev/DF
4	0.3317	0.3172	2.5400	4.5681
8	*0.0427	0.6502	2.5471	4.8046
16	0.0978	0.5947	2.5107	4.6313
32	0.1269	0.5653	2.3316	4.0798
64	0.0813	0.6114	1.7373	2.6056

Legend. Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table 7: Exploratory Regression Analysis of the 11-Multiplexer Domain

Multiple Regression of Child Survivability by Depth, Parent Size, Removed Size, Inserted Size

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.6142	0.1166	-0.0278	0.0229	-0.1389	2.5311	4.5308
8	0.5249	-0.1686	-0.1463	0.0242	0.3794	2.5263	4.6687
16	0.4855	*-0.0550	-0.1137	0.0161	0.2904	2.4919	4.4912
32	0.5279	-0.2069	-0.0876	0.0113	0.3813	2.3063	4.0891
64	0.3863	-0.1271	-0.0262	0.0048	0.4279	1.5313	2.2049

Multiple Regression of Child Survivability by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.3581	0.2324	-0.1158	0.0552	0.3156	2.5323	4.4940
8	0.4694	-0.0828	-0.1688	0.0535	0.0514	*2.5199	4.6876
16	0.4638	*0.0085	-0.1364	0.0296	0.0574	*2.4609	4.5007
32	0.5182	-0.1822	-0.1082	0.0236	-0.0058	*2.2999	4.0986
64	0.3775	-0.0902	-0.0405	0.0093	0.2307	1.5350	2.2102

Regression of Removed Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	-0.9358	0.1403	0.6802	0.4627
8	-1.8267	0.5608	0.4471	0.1999
16	-1.5359	0.7067	0.3537	0.1251
32	-1.3886	0.7328	0.3229	0.1043
64	-1.0578	0.7406	0.3402	0.1157

Regression of Parent Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	0.8596	1.9346	1.9473	3.7922
8	0.4392	2.8268	1.9132	3.6607
16	0.2933	2.5359	2.2467	5.0479
32	0.2672	2.3886	2.7406	7.5116
64	0.2594	2.0578	3.5214	12.4015

Regression of Child Survivability by Removed Size

Gen	Removed	Intercept	Scale	Prescaled Dev/DF
4	-0.1050	0.7821	2.5550	4.5492
8	-0.2377	0.8631	2.5900	4.7481
16	-0.1674	0.8137	2.5761	4.5432
32	-0.1254	0.7818	2.3253	4.1436
64	-0.0390	0.7249	1.5400	2.2196

Regression of Child Survivability by Parent Size

Gen	Parent	Intercept	Scale	Prescaled Dev/DF
4	0.3416	0.3051	2.5394	4.5492
8	0.0641	0.6285	2.5426	4.7481
16	0.2156	0.4740	2.4790	4.5432
32	0.1520	0.5397	2.3234	4.1436
64	0.1137	0.5786	1.5427	2.2196

Legend. Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table 8: Exploratory Regression Analysis of the 11-Multiplexer Domain with Inviable Code Restricted

Multiple Regression of Child Survivability by Depth, Parent Size, Removed Size, Inserted Size

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.5116	0.1859	-0.0523	0.0155	-0.0573	*2.5139	4.5375
8	0.4647	-0.0611	-0.2028	0.0174	0.3778	2.5124	4.5441
16	0.5388	-0.2001	-0.1094	0.0201	0.3732	2.2129	3.7820
32	0.3184	*0.0054	-0.0741	*-0.0062	0.4066	1.1964	1.6350
64	0.1812	*-0.0404	*-0.0239	*-0.0069	0.5729	1.1427	1.4647

Multiple Regression of Child Survivability by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.2670	0.2778	-0.1408	0.0370	0.3954	2.5195	4.4990
8	0.4367	*-0.0046	-0.2166	0.0415	0.0349	*2.4849	4.5585
16	0.5270	-0.2066	-0.1421	0.0341	-0.0182	*2.2104	3.7851
32	0.3295	*0.0067	-0.0647	*-0.0094	0.2120	1.1997	1.6559
64	0.1982	*-0.0652	*-0.0199	*-0.0191	0.3860	1.1435	1.4699

Regression of Removed Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	-1.1544	0.1919	0.6789	0.4609
8	-2.0979	0.7001	0.4269	0.1822
16	-1.5757	0.7283	0.3136	0.0984
32	-1.0197	0.6739	0.4188	0.1757
64	-0.8889	0.5181	0.4527	0.2058

Regression of Parent Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	0.8081	2.1546	1.9051	3.6297
8	0.2999	3.0979	1.9453	3.7844
16	0.2717	2.5757	2.2518	5.0715
32	0.3261	2.0197	2.8418	8.0866
64	0.4819	1.8889	3.6929	13.6913

Regression of Child Survivability by Removed Size

Gen	Removed	Intercept	Scale	Prescaled Dev/DF
4	-0.1222	0.7955	2.5341	4.5744
8	-0.2859	0.8870	2.5813	4.6319
16	-0.1664	0.8137	2.2360	3.8440
32	-0.0931	0.7621	1.2114	1.6701
64	-0.0300	0.7182	1.1386	1.4685

Regression of Child Survivability by Parent Size

Gen	Parent	Intercept	Scale	Prescaled Dev/DF
4	0.3478	0.3020	2.5127	4.5744
8	0.0805	0.6119	2.5005	4.6319
16	0.1431	0.5488	2.2333	3.8440
32	0.1890	0.5001	1.2181	1.6701
64	*0.0709	0.6215	1.1400	1.4685

Legend. Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table 9: Exploratory Regression Analysis of the 6-Multiplexer Domain

Multiple Regression of Child Survivability by Depth, Parent Size, Removed Size, Inserted Size

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	0.3015	0.2096	-0.0987	0.0402	0.1621	2.5107	4.5705
8	0.4823	-0.1285	-0.1643	0.0340	0.3750	2.5061	4.4502
16	0.4299	-0.1032	-0.1315	0.0250	0.3973	2.1435	3.6314
32	0.3006	-0.1207	-0.0388	0.0022	*0.5274	1.2459	1.7070
64	0.1119	*-0.0141	-0.0096	-0.0004	*0.6017	1.0760	1.3338

Multiple Regression of Child Survivability by Depth, Log(Parent Size), Log(Removed Size), Log(Inserted Size)

Gen	Depth	Parent	Removed	Inserted	Intercept	Scale	Prescaled Dev/DF
4	*0.0002	0.3357	-0.2323	0.0819	0.6526	2.5026	4.5083
8	0.4533	-0.0761	-0.1677	0.0624	0.0469	*2.4468	4.4621
16	0.4051	*-0.0514	-0.1427	0.0411	0.0894	2.1368	3.6399
32	0.2912	-0.0902	-0.0438	0.0046	*0.3039	1.2469	1.7163
64	0.1069	*-0.0004	-0.0168	-0.0023	*0.5364	1.0757	1.3341

Regression of Removed Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	-1.0538	0.3131	0.7042	0.4960
8	-1.5924	0.6657	0.4458	0.1988
16	-1.3793	0.6800	0.3625	0.1315
32	-1.0265	0.6501	0.3980	0.1584
64	-0.6048	0.5823	0.4631	0.2145

Regression of Parent Size by Depth

Gen	Depth	Intercept	Scale	Prescaled Dev/DF
4	0.6869	2.0538	1.8672	3.4866
8	0.3343	2.5915	1.8532	3.4345
16	0.3200	2.3793	2.2845	5.2196
32	0.3499	2.0265	2.9996	8.9995
64	0.4177	1.6048	3.7426	14.0102

Regression of Child Survivability by Removed Size

Gen	Removed	Intercept	Scale	Prescaled Dev/DF
4	-0.1406	0.8073	2.5371	4.6397
8	-0.2555	0.8708	2.6376	4.5416
16	-0.1799	0.8178	2.1570	3.7110
32	-0.0546	0.7373	1.2483	1.7288
64	-0.0122	0.7044	1.0758	1.3357

Regression of Child Survivability by Parent Size

Gen	Parent	Intercept	Scale	Prescaled Dev/DF
4	0.3027	0.3566	2.5167	4.6397
8	0.0787	0.6138	2.4773	4.5416
16	0.1638	0.5272	2.1557	3.7110
32	0.0655	0.6272	1.2510	1.7288
64	0.0579	0.6347	1.0755	1.3357

Legend. Asterisks indicate statistically insignificant results (with an alpha value greater than 0.05). *Prescaled Dev/DF* indicates the deviance of the model, divided by the degrees of freedom, prior to scaling (at which time it approaches 1.0). *Intercept* and *Scale* are constants in the model function.

Table 10: Exploratory Regression Analysis of the 6-Multiplexer Domain with Inviable Code Restricted