Code Growth Is Not Caused by Introns

Sean Luke
Department of Computer Science
University of Maryland
seanl@cs.umd.edu
http://www.cs.umd.edu/users/seanl/

Abstract

Genetic programming trees have a strong tendency to grow rapidly and relatively independent of fitness, a serious flaw which has received considerable attention in the genetic programming literature. Much of this literature has implicated *introns*, subtree structures with no effect on the an individual's fitness assessment. The propagation of *inviable code*, a certain kind of intron, has been especially linked to tree growth. However this paper presents evidence which shows that denying inviable code the opportunity to propagate actually increases tree growth. The paper argues that rather than causing tree growth, a rise in inviable code is in fact an expected result of tree growth. Lastly, this paper proposes a more general theory of growth for which introns are merely a symptom.

1 INTRODUCTION

An unforseen result of genetic programming's tree-based chromosome is *bloat*, the uncontrolled growth in the size of individuals over the course of a run. This phenomenon has been observed many times (for examples, see [7, 4, 2]) for both subtree crossover and various kinds of mutation. Whereas a fixed-length chromosome permits an evolutionary run to last as long as time permits, genetic programing's tree bloat effectively puts a time limit on its search, by slowing down both the breeding and evaluation of trees. Bloating also fills memory, leading to swapping and memory exhaustion. Lastly, bloating slows the search process by making it less likely to find good subtrees to modify. In a very real sense, genetic programming is a race against time, to find the best solution possible before bloat puts a stop to the search.

2 INTRONS

Much of the code bloat literature in genetic programming revolves around *introns*, extraneous regions in an individual which neither add nor detract from its fitness, because they are ignored or do nothing. This notion of introns stems from microbiology. In real chromosomes, genes consist both of regions which are expressed in final protein end-products, and regions which are edited out somewhere in the translation and transcription process. The expressed areas of these genes are known as *exons*, whereas the ignored regions are introns.

[1] defined introns in genetic programming as areas of code that "are unnecessary since they can be removed from the program without altering the solution the program represents". This general definition has been used in a variety of ways throughout the code bloat literature, and so it is very important to provide a specific definition. The two most common uses of the term are:

1. Introns are areas of code which can be trivially simplified without modifying the individual's operation.
2. (A subset of #1) Introns are subtrees which cannot be replaced by anything that can possibly change the individual's operation. Such code is associated with an *invalidator*, a structure elsewhere in the individual which is responsible for nullifying the intron's effect.

Definition #1 is more common in the literature and so I will use it as the meaning of "intron". For definition #2 I adopt the term "inviable code". For those introns which are not inviable code, I will use the term "unoptimized code".

2.1 INTRONS AND THEORIES OF CODE BLOAT

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

2.1.1 Hitchhiking

The earliest theory, *hitchhiking* [18] says that if introns (the hitchhikers) are attached to parents of "important" code, 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.

2.1.2 Defense Against Crossover

In the second theory, *defense against crossover*, introns take a more active role in driving code bloat. Here 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. The makes common *neutral crossover*, whereby crossover has no effect on the individual's fitness. This general theory has been cited, in one way or another, by a large chunk of literature: [4, 11, 10, 14, 3, 2, 8]. 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 nearly all theoretical justification have dealt solely with *inviable code*. While for tree-based genetic programming their theoretical support appears applicable to inviable code only, [11, 2] perform some experiments with introns in general, but only in a non-tree genetic programming style for which unoptimized code presents unique problems. As such, I think it is fair to say that, for most of its experimental and theoretical foundation in the tree-based genetic programming literature, defense-against-crossover is primarily an inviable code theory.

2.1.3 Removal Bias

A recent third theory, *removal bias*, eschews unoptimized code entirely and focuses solely on inviable code [16, 8]. The theory first proposes that if an individual contains inviable subtrees, it is more likely to survive if it performs its modifications within these subtrees. In this sense it is similar to defense against crossover. But removal bias suggests an actual mechanism which prefers larger trees within these inviable subtree areas. 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 penalty for removing large subtrees from the individual, but no such penalty for inserting large subtrees.

2.1.4 A Non-Intron Code Bloat Theory

The chief non-intron bloat theory [8] is *diffusion*. According to this theory, in the space of all possible programs, there are generally many more large-sized highly-fit trees than there are small-sized ones. Since genetic programming starts out with artificially small trees, code bloat can be described simply as the system moving towards equilibrium.

2.2 TECHNIQUES FOR CONTROLLING BLOAT

The most common method for controlling code bloat is by simply restricting trees to be a certain size or depth. [7] has popularized setting a maximum tree depth at 17. Unfortunately this technique has been shown to have negative effects when most trees reach the limit [5]. The second most common method is *parsimony pressure*: adding a tree size penalty as part of the individual's fitness assessment. While some approaches use a linear or constant function for parsimony pressure (for example [17]), others add parsimony pressure adaptively in response to tree growth metrics such as the amount of introns, for example [3]. A third method is *code editing*: physically deleting introns in programs. [17, 3] report strong results with this approach. Still another method is the inclusion of *explicitly defined introns* ([12, 15, 3], special nodes which increase the likelihood of crossover at specific positions in the tree.

Lastly some researchers (notably [13]) have suggested a form of hillclimbing: rejecting crossover results which decrease the fitness of an individual (or more strongly, do not increase its fitness). If crossover fails this test, then the parent is replicated into the new population in lieu of the child. Recently [16, 8] have used this mechanism to argue for the removal bias theory of tree bloat, arguing that the particular success of the strong form in countering code growth is due to its rejection of any crossover events which occur in inviable code areas. One weakness in this argument is that this technique also acts to replicate large numbers of parents into future populations. If some unrelated 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 would be generally smaller than their descendants.

3 EXPERIMENTS

The two most prevelant intron theories of code growth (defense against crossover and removal bias) both rely on a similar thesis: that crossover in inviable subtree areas is a 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.

[4, 3] proposed exactly this, calling it *marking*: identifying inviable code areas and disallowing crossover within those regions. Unfortunately, no code growth results were presented, and [3] 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." 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.

To test this I performed experiments in three canonical genetic programming domains: Symbolic Regression, 11-Bit Multiplexer, and 6-Bit Multiplexer. All experimental runs lasted 64 generations, or until an ideal solution was found, using a population of 512, and 7-tournament selection. Subtree crossover was the sole breeding mechanism used: rather than using the traditional node-selection scheme (picking terminals 10% of the time), node selection was uniform. Additionally, unlike in may previous experiments, *no limit* was placed on either the size or depth of crossed-over trees. These restrictions were lifted in order to better guage the true, unbiased dynamics of subtree modification with respect to tree growth. Crossover was performed as follows: two children are picked from the population and crossed over. However, while the first child is placed into the next generation, the second child is discarded. In all other respects, the runs follow the domain descriptions given in [7], without ephemeral random constants. ECJ was the genetic programming system used [9].

3.1 MULTIPLEXER

One nice feature of Multiplexer is that it is possible, if expensive, to identify all inviable code, for example (and (not a0) (and a0 *inviable*)). For 6-Multiplexer, inviable code is very common. Because of the increased size and complexity of its function set, inviable code is much less common in the 11-Multiplexer problem.

For each Multiplexer domain, two sets of runs were performed, each with fifty independent runs. In the first set, runs were performed as described above. 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.

The results of these experiments, shown in Figures 1 through 4, were surprising. After denying the ability to cross over in inviable regions, code growth in 11-Multiplexer *increased* to 120%. In 6-Multiplexer, code growth increased to 150%. Tree depth similarly increased. At the same time, the number of inviable nodes, as a percent of each individual, dropped dramatically. 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 neutral crossover which, while dramatically changing the functional semantics of an individual, does not change its overall score.

3.2 SYMBOLIC REGRESSION

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 intron*). 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, parents 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 crossovers with no change in fitness 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 occured 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 number of neutral crossovers.

In order to gauge the effects of introns on tree growth in Symbolic Regression, two sets of runs were performed, each with fifty independent runs. In the first set, as usual, runs were performed unemcumbered. In the second set, three special crossover restrictions were performed. First, the crossover point for the first parent was chosen through marking. All inviable code was marked except for undiscoverable decimated code. Second, once it had selected two individuals for crossover, the system would try 500 times to find two semantically dissimilar subtrees to cross over: that is, it would reject semantically identical subtrees such as x and (+ (- x x) x). If it failed 500 times (which rarely occurred, and only if both individuals were simply the nonterminal x), the first parent was replicated in lieu of

its child. Applying this restriction, plus rejecting inviable code crossover, meant that there were only two possible causes left for neutral crossover: crossing over two trees simply consisting of the terminal x (rare), and inviable code due to unmarked decimation. This made possible the third crossover restriction: countering the effects of unmarked decimation by disallowing an individual to be selected if it had the same fitness as its parent (caused by the arrival of decimation).

The result is shown in Figures 5 and 6. Note that the inviable code growth figures only indicate the growth in markable inviable code. Once again, tree growth increased, while inviable code decreased. Though not shown here for lack of space, similar experiments were performed with only marking, and with marking plus rejecting semantically identical crossover, with similar growth results.

4 DISCUSSION

Proponents of intron theories point to the increase in inviable nodes, neutral crossvers, and tree growth and suggest that the correlation among them is in fact causation: inviable code growth is driving tree growth. But these experiments suggest a more likely 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 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 should grow naturally towards 100%. As each figure shows, invalidators generally remain constant throughout the run. Additionally, Figure 6 shows that even though individuals with neutral crossover (caused by decimation) were immediately terminated, decimation events continued to rise. This suggests that growth in decimated inviable code is also driven by overall tree growth. In other words, these experiments suggest that *tree growth is the cause of inviable code growth*.

Another possible source of tree growth is a possible increase in neutral crossover due to unoptimized code propagation. However the Symbolic Regression experiment shows that such propagation can be entirely eliminated and code will still grow.

If not introns, then what is causing these trees to grow? I propose a more general mechanism behind tree growth. As [6] have shown, for many genetic programming domains there is a strong inverse correlation between the depth of the crossover point in a tree and its effect on fitness. For populations with high fitness, this suggests a relationship between the depth of a crossover point and its likelihood to destroy the individual. This bias towards deeper crossover points has two effects on tree growth. First, it promotes the use of larger trees, which have deeper crossover points. Second, deeper-rooted subtrees are more likely to be small, which biases the search towards tree growth in a generalization of the removal bias theory.

Inviable code growth is a natural result of this theory. Another expected result is *pseudoinviable code*, subtrees for which crossover has a low probability of effecting an individual in a significant way. Both [3] and [12] note the possible existence of pseudoinviable code. [15] have recently performed experiments suggesting a strong relationship between pseudoinviable code and code bloat.

Other effects might be explained by this theory. [10]'s inc-dec and [15]'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. 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.

The theory also suggests why restricted 6-Multiplexer was so much less likely to find a solution. 6-Multiplexer is a very simple domain with no formal subpotima, and so it benefits from an incremental strategy. Crossing over in deeper subtrees lowers the likelihood of changing an individual significantly. But one effect of restricting inviable subtree crossover is that subtrees closer to the root have a higher likelihood of being chosen, resulting in more radical semantic jumps; in effect, this underdampens the search.

5 CONCLUSIONS

Inviable code plays a central part in much of the code bloat literature, with unoptimized code a distant second. Relatively little work has been done on pseudoinviable code and other less-absolute forms of code bloat. And all experiments to date have suggested a strong positive relationship between introns and code bloat. This paper presents evidence to the contrary, showing that when inviable code crossover is restricted, code bloat continues unabated, and even increases. The paper then introduces a more abstract mechanism behind code bloat to explain these surprising results, suggesting that the cause of bloat is both more general and more complicated than previously thought.

Acknowledgements

This research was supported in part by grants from ONR (N00014-J-91-1451), ARPA (N00014-94-1090, DAST-95-C003, F30602-93-C-0039), and ARL (DAAH049610297).

References

[1] P. J. Angeline. Genetic programming and emergent intelligence. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

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

[3] T. Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Nov. 1996.

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

[5] C. Gathercole and P. Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[6] C. Igel and K. Chellapilla. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1061–1068, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

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

[8] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.

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

[10] N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[11] P. Nordin and W. Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[12] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, California, USA, 9 July 1995.

[13] U.-M. O'Reilly. *An Analysis of Genetic Programming*. PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada, 22 Sept. 1995.

[14] J. Rosca. Generality versus size in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

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

[16] T. Soule and J. A. Foster. 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, 5-9 May 1998. IEEE Press.

[17] T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

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

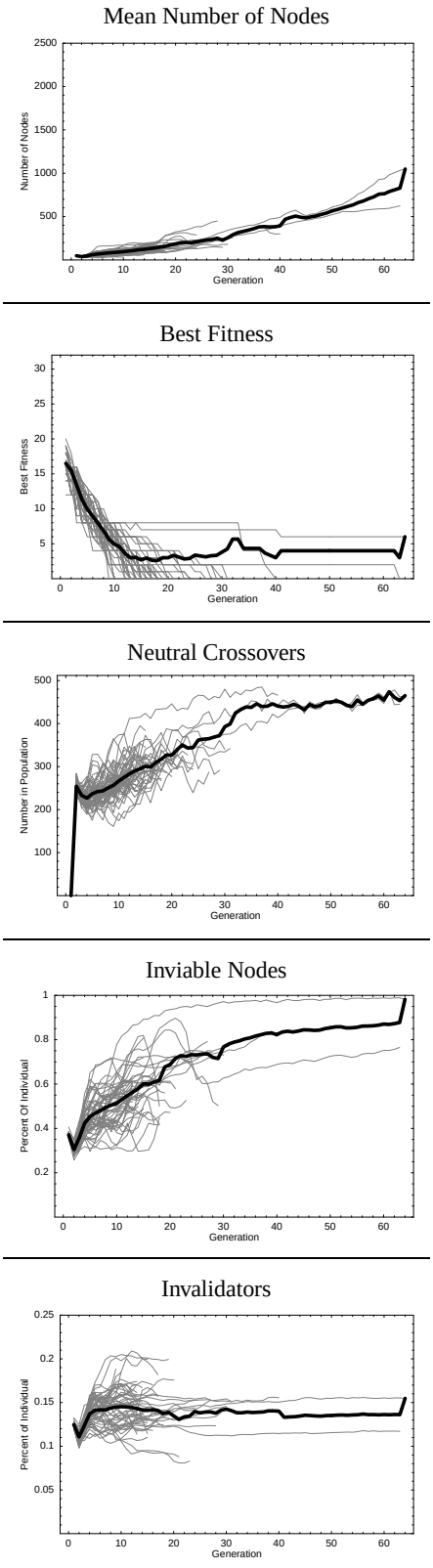

Figure 1: 6-Bit Multiplexer

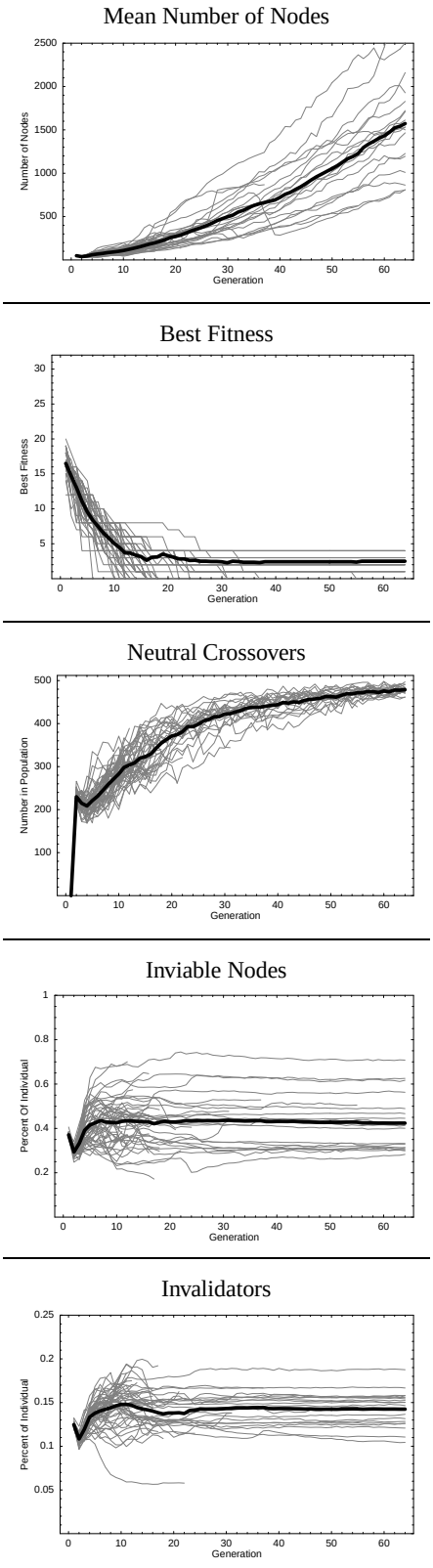

Figure 2: 6-Bit Multiplexer with Restricted Crossover

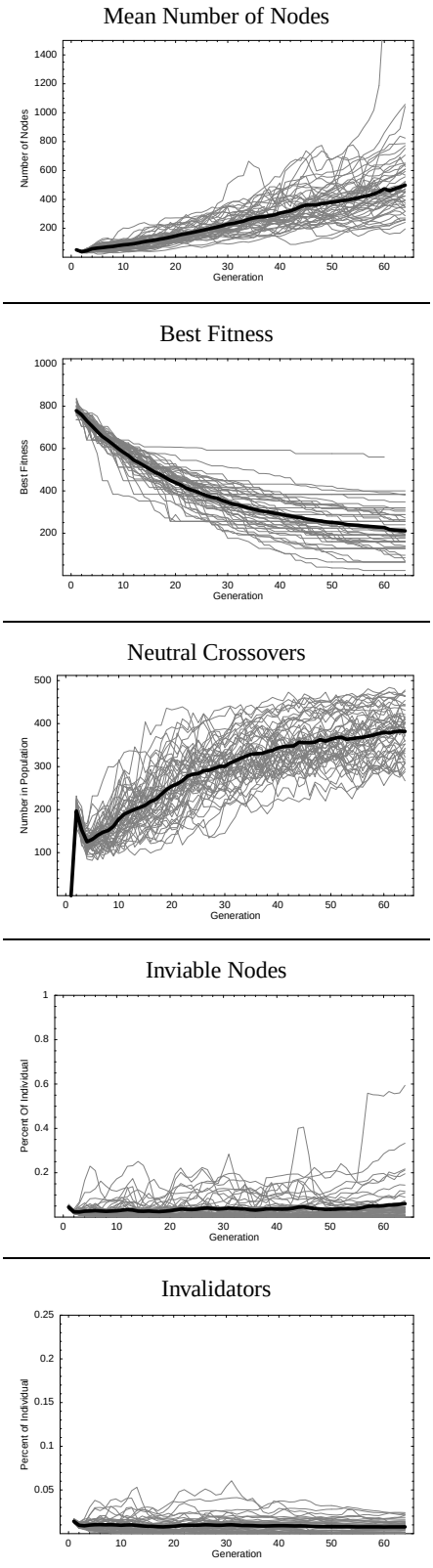

Figure 3: 11-Bit Multiplexer

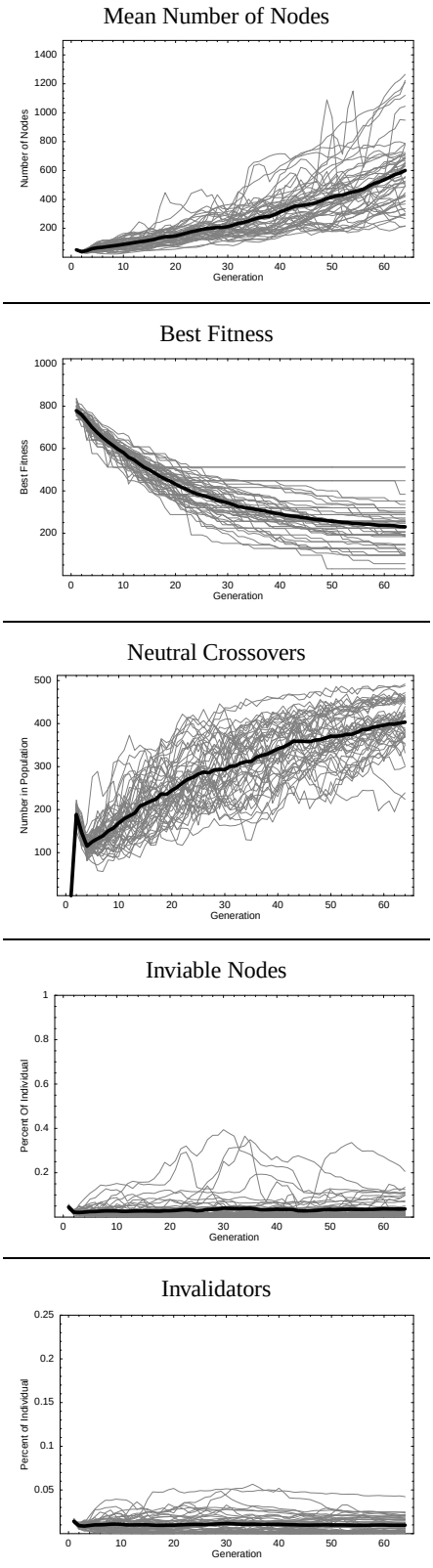

Figure 4: 11-Bit Multiplexer with Restricted Crossover

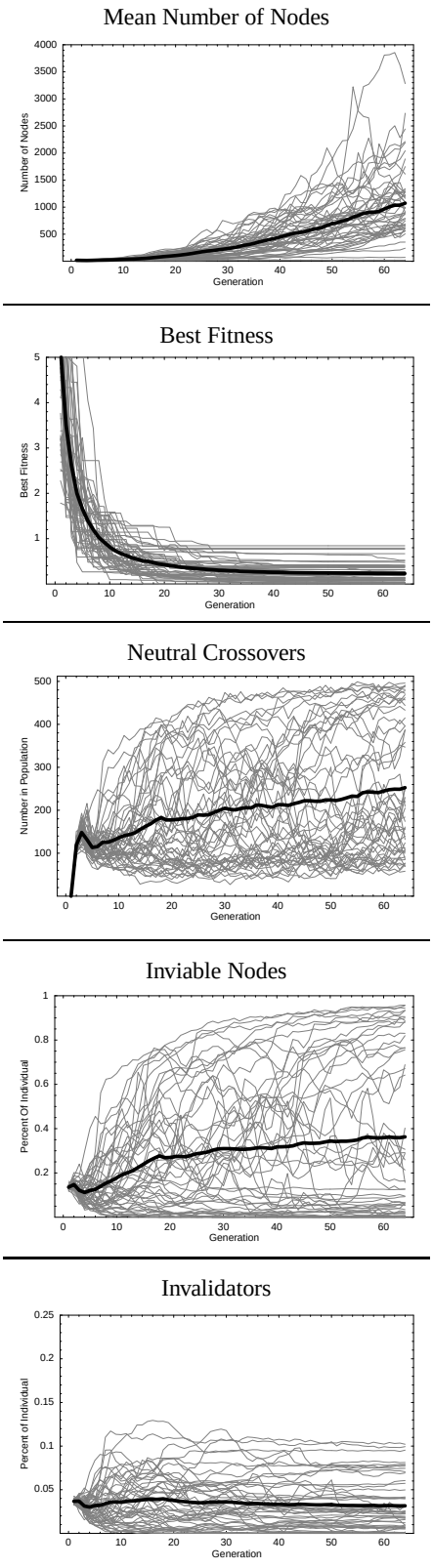

Figure 5: Symbolic Regression

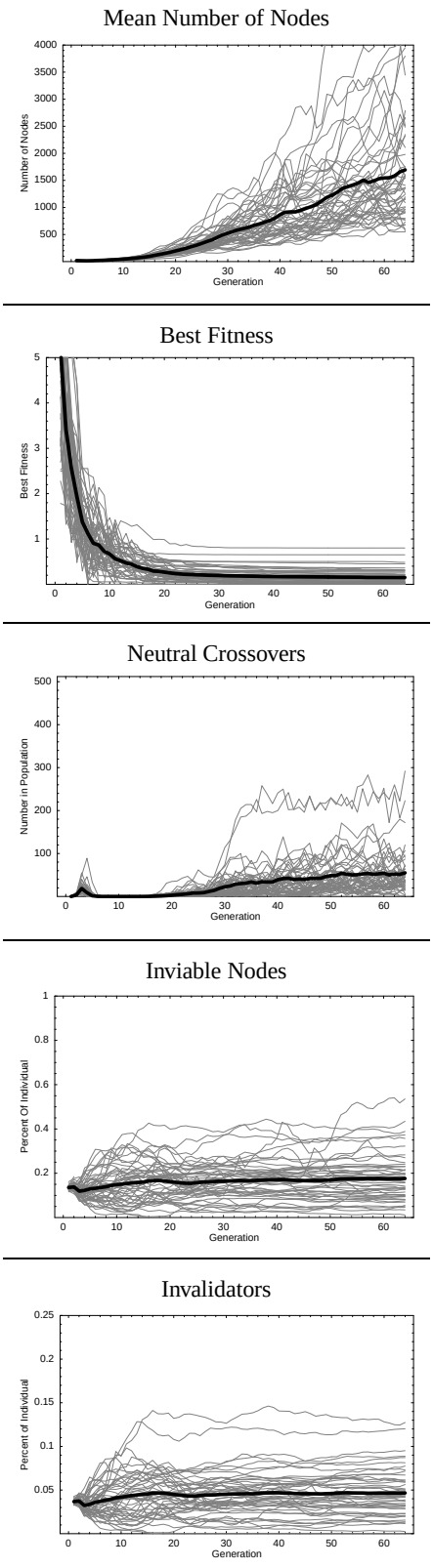

Figure 6: Symbolic Regression with Restricted Crossover