A Survey and Comparison of Tree Generation Algorithms

Sean Luke
George Mason University
http://www.cs.gmu.edu/~sean/

Liviu Panait
George Mason University
http://www.cs.gmu.edu/~lpanait/

Abstract

This paper discusses and compares five major tree-generation algorithms for genetic programming, and their effects on fitness: RAMPED HALF-AND-HALF, PTC1, PTC2, RANDOM-BRANCH, and UNIFORM. The paper compares the performance of these algorithms on three genetic programming problems (11-Boolean Multiplexer, Artificial Ant, and Symbolic Regression), and discovers that the algorithms do not have a significant impact on fitness. Additional experimentation shows that tree size does have an important impact on fitness, and further that the ideal initial tree size is very different from that used in traditional GP.

1 INTRODUCTION

The issue of population initialization has received surprisingly little attention in the genetic programming literature. (Koza, 1992) established the GROW, FULL, and RAMPED HALF-AND-HALF algorithms, only a few papers have appeared on the subject, and the community by and large still uses the original Koza algorithms.

Some early work was concerned with algorithms similar to GROW but which operated on derivation grammars. (Whigham, 1995a,b, 1996) analyzed biases due to population initialization, among other factors, in grammatically-based genetic programming. (Geyer-Schulz, 1995) also devised similar techniques for dealing with tree grammars.

The first approximately uniform tree generation algorithm was RAND-TREE (Iba, 1996), which used Dyck words to choose uniformly from all possible tree structures of a given arity set and tree size. Afterwards the tree structure would be populated with nodes. (Bohm and Geyer-Schulz, 1996) then presented an exact uniform algorithm for choosing among all possible trees of a given function set.

Recent tree generation algorithms have focused on speed. (Chellapilla, 1997) devised RANDOMBRANCH, a simple algorithm which generated trees approximating a requested tree size. After demonstrating problems with the GROW algorithm, (Luke, 2000b) modifed GROW to produce PTC1 which guaranteed that generated trees would appear around an expected tree size. (Luke, 2000b) also presented PTC2 which randomly expanded the tree horizon to produce trees of approximately the requested size. All three of these algorithms are linear in tree size.

Both (Iba, 1996) and (Bohm and Geyer-Schulz, 1996) argued for the superiority of their algorithms over the Koza standard algorithms. (Whigham, 1995b) showed that biasing a grammar-based tree-generation algorithm could dramatically improve (or hurt) the success rate of genetic programming at solving a given domain, though such bias must be hand-tuned for the domain in question.

In contrast, this paper examines several algorithms to see if any of the existing algorithms appears to make much of a difference, or if tree size and other factors might be more significant.

2 THE ALGORITHMS

This paper compares five tree generation algorithms from the literature. These algorithms were chosen for their widely differing approaches to tree creation. The chief algorithm not in this comparison is RAND-TREE (Iba, 1996). This algorithm has been to some degree subsumed by a more recent algorithm (Bohm and Geyer-Schulz, 1996), which generates trees from a truly uniform distribution (the original unachieved goal of RAND-TREE).

The algorithms discussed in this paper are:

2.1 Ramped Half-And-Half and Related Algorithms

RAMPED HALF-AND-HALF is the traditional tree generation algorithm for genetic programming, popularized by

(Koza, 1992). RAMPED HALF-AND-HALF takes a tree depth range (commonly 2 to 6 – for this and future references, we define "depth" in terms of number of nodes, not number of edges). In other respects, the user has no control over the size or shape of the trees generated.

RAMPED HALF-AND-HALF first picks a random value within the depth range. Then with 1/2 probability it uses the GROW algorithm to generate the tree, passing it the chosen depth; otherwise it uses the FULL algorithm with the chosen depth.

GROW is very simple:

GROW(depth d, max depth D)
  Returns: a tree of depth $\leq D - d$
  If $d = D$, return a random terminal
  Else
○   Choose a random function or terminal f
    If f is a terminal, return f
    Else
      For each argument a of f,
        Fill a with GROW($d + 1$,D)
      Return f with filled arguments

GROW is started by passing in 0 for d, and the requested depth for D. FULL differs from GROW only in the line marked with a ○. On this line, FULL chooses a nonterminal function only, never a terminal. Thus FULL only creates full trees, and always of the requested depth.

Unlike other algorithms, because it does not have a size parameter, RAMPED HALF-AND-HALF does not have well-defined computational complexity in terms of size. FULL always generates trees up to the depth bound provided. As (Luke, 2000b) has shown, GROW without a depth bound may, depending on the function set, have an expected tree size of infinity.

2.2 PTC1

PTC1 (Luke, 2000b) is a modification of the GROW algorithm which is guaranteed to produce trees around a finite expected tree size. A simple version of PTC1 is described here. PTC1 takes a requested expected tree size and a maximum legal depth. PTC1 begins by computing p, the probability of choosing a nonterminal over a terminal in order to maintain the expected tree size E as:

$$p = \frac{1 - \frac{1}{E}}{\sum_{n \in N} \frac{1}{|N|} b_n}$$

where N is the set of all nonterminals and b_n is the arity of nonterminal n. This computation can be done once offline. Then the algorithm proceeds to create the tree:

PTC1(precomputed probability p, depth d, max depth D)
  Returns: a tree of depth $\leq D - d$
  If $d = D$, return a random terminal
  Else if a coin-toss of probability p is true,
    Choose a random nonterminal n
    For each argument a of n,
      Fill a with PTC1(p,$d + 1$,D)
    Return n with filled arguments
  Else return a random terminal

PTC1 is started by passing in p, 0 for d, and the maximum depth for D. PTC1's computational complexity is linear or nearly linear in expected tree size.

2.3 PTC2

PTC2 (Luke, 2000b) receives a requested tree size, and guarantees that it will return a tree no larger than that tree size, and no smaller than the size minus the maximum arity of any function in the function set. This algorithm works by increasing the tree horizon at randomly chosen points until it is sufficiently large. PTC2 in pseudocode is big, but a simple version of the algorithm can be easily described.

PTC2 takes a requested tree size S. If $S = 1$, it returns a random terminal. Otherwise it picks a random nonterminal as the root of the tree and decreases S by 1. PTC2 then puts each unfilled child slot of the nonterminal into a set H, representing the "horizon" of unfilled slots. It then enters the following loop:

1. If $S \leq |H|$, break from the loop.

2. Else remove a random slot from H. Fill the slot with a randomly chosen nonterminal. Decrease S by 1. Add to H every unfilled child slot of that nonterminal. Goto #1.

At this point, the total number of nonterminals in the tree, plus the number of slots in H, equals or barely exceeds the user-requested tree size. PTC2 finishes up by removing slots from H one by one and filling them with randomly chosen terminals, until H is exhausted. PTC2 then returns the tree.

PTC2's computational complexity is linear or nearly linear in the requested tree size.

2.4 RandomBranch

RANDOMBRANCH (Chellapilla, 1997) is another interesting tree-generation algorithm, which takes a requested tree size and guarantees a tree of that size or "somewhat smaller". RANDOMBRANCH is as follows:

Problem Domain	Algorithm	Parameter	Avg. Tree Size
11-Boolean Multiplexer	`RAMPED HALF-AND-HALF`	(No Parameter)	21.2
11-Boolean Multiplexer	`RANDOMBRANCH`	Max Size: 45	20.0
11-Boolean Multiplexer	`PTC1`	Expected Size: 9	20.9
11-Boolean Multiplexer	`PTC2`	Max Size: 40	21.4
11-Boolean Multiplexer	`UNIFORM-even`	Max Size: 42	21.8
11-Boolean Multiplexer	`UNIFORM-true`	Max Size: 21	20.9
Artificial Ant	`RAMPED HALF-AND-HALF`	(No Parameter)	36.9
Artificial Ant	`RANDOMBRANCH`	Max Size: 90	33.7
Artificial Ant	`PTC1`	Expected Size: 12	38.5
Artificial Ant	`PTC2`	Max Size: 67	35.3
Artificial Ant	`UNIFORM-even`	Max Size: 65	33.9
Artificial Ant	`UNIFORM-true`	Max Size: 37	36.8
Symbolic Regression	`RAMPED HALF-AND-HALF`	(No Parameter)	11.6
Symbolic Regression	`RANDOMBRANCH`	Max Size: 21	11.4
Symbolic Regression	`PTC1`	Expected Size: 4	10.9
Symbolic Regression	`PTC2`	Max Size: 18	11.1
Symbolic Regression	`UNIFORM-even`	Max Size: 19	11.2
Symbolic Regression	`UNIFORM-true`	Max Size: 11	10.8

Table 1: Tree Generation Parameters and Resultant Sizes

`RANDOMBRANCH`(requested size s)
    Returns: a tree of size $\leq s$
    If a nonterminal with arity $\leq s$ does not exist
        Return a random terminal
    Else
        Choose a random nonterminal n of arity $\leq s$
        Let b_n be the arity of n
        For each argument a of n,
            Fill a with `RANDOMBRANCH`($\lfloor \frac{s}{b_n} \rfloor$)
        Return n with filled arguments

Because `RANDOMBRANCH` evenly divides s up among the subtrees of a parent nonterminal, there are many trees that `RANDOMBRANCH` simply cannot produce by its very nature. This makes `RANDOMBRANCH` the most restrictive of the algorithms described here. `RANDOMBRANCH`'s computational complexity is linear or nearly linear in the requested tree size.

2.5 Uniform

`UNIFORM` is the name we give to the exact uniform tree generation algorithm given in (Bohm and Geyer-Schulz, 1996), who did not name it themselves. `UNIFORM` takes a single requested tree size, and guarantees that it will create a tree chosen *uniformly* from the full set of all possible trees of that size, given the function set. `UNIFORM` is too complex an algorithm to describe here except in general terms.

During tree-generation time, `UNIFORM`'s computational complexity is nearly linear in tree size. However, `UNIFORM` must first compute various tables offline, including a table of numbers of trees for all sizes up to some maximum feasibly requested tree sizes. Fortunately this daunting task can be done reasonably quickly with the help of dynamic programming.

During tree-generation time, `UNIFORM` picks a node selected from a distribution derived from its tables. If the node is a nonterminal, `UNIFORM` then assigns tree sizes to each child of the nonterminal. These sizes are also picked from distributions derived from its tables. `UNIFORM` then calls itself recursively for each child.

`UNIFORM` is a very large but otherwise elegant algorithm; but it comes at the cost of offline table-generation. Even with the help of dynamic programming, `UNIFORM`'s computational complexity is superlinear but polynomial.

3 FIRST EXPERIMENT

(Bohm and Geyer-Schulz, 1996) claimed that `UNIFORM` dramatically outperformed `RAMPED HALF-AND-HALF`, and argued that the reason for this was `RAMPED HALF-AND-HALF`'s highly non-uniform sampling of the initial program space. Does uniform sampling actually make a significant difference in the final outcome? To test this, the first experiment compares the fitness of `RAMPED HALF-AND-HALF`, `PTC1`, `PTC2`, `RANDOMBRANCH`, and two different versions of `UNIFORM` (`UNIFORM-true` and

UNIFORM-even, described later). It is our opinion that the "uniformity" of sampling among the five algorithms presented is approximately in the following order (from most uniform to least): UNIFORM (of course), PTC2, RAMPED HALF-AND-HALF, PTC1, RANDOMBRANCH.

The comparisons were done over three canonical genetic programming problem domains, 11-Boolean Multiplexer, Artificial Ant, and Symbolic Regression. Except for the tree generation algorithm used, these domains followed the parameters defined in (Koza, 1992), using tournament selection of size 7. The goal of 11-Boolean Multiplexer is to evolve a boolean function on eleven inputs which performs multiplexing on eight of those inputs with regard to the other three. The goal of the Artificial Ant problem is to evolve a simple robot ant algorithm which follows a trail of pellets, eating as many pellets as possible before time runs out. Symbolic Regression tries to evolve a symbolic mathematical expression which best fits a training set of data points.

To perform this experiment, we did 50 independent runs for each domain using the RAMPED HALF-AND-HALF algorithm to generate initial trees. From there we measured the mean initial tree size and calibrated the other algorithms to generate trees of approximately that size. This calibration is not as simple as it would seem at first. For example, PTC1 can be simply set to the mean value, and it should produce trees around that mean. However, an additional complicating factor is involved: duplicate rejection. Usually genetic programming rejects duplicate copies of the same individual, in order to guarantee that every initial individual is unique. Since there are fewer small trees than large ones, the likelihood of a small tree being a duplicate is correspondingly much larger. As a result, these algorithms will tend to produce significantly larger trees than would appear at first glance if, as was the case in this experiment, duplicate rejection is part of the mix. Hence some trial and error was necessary to establish the parameters required to produce individuals of approximately the same mean size as RAMPED HALF-AND-HALF. Those parameters are shown in Table 1.

In the PTC1 algorithm, the parameter of consequence is the expected mean tree size. For the other algorithms, the parameter is the "maximum tree size". For PTC2, RANDOMBRANCH, and UNIFORM-even, a tree is created by first selecting an integer from the range 1 to the maximum tree size inclusive. This integer is selected uniformly from this range. In UNIFORM-true however, the integer is selected according to a probability distribution defined by the number of trees of each size in the range. Since there are far more trees of size 10 than of 1 for example, 10 is chosen much more often than 1. For each remaining algorithm, 50 independent runs were performed with both problem domains. ECJ (Luke, 2000a) was the genetic programming system used.

Fisher LSD	Algorithm	Tukey
	PTC2	
	PTC1	
	RAMPED HALF-AND-HALF	
	UNIFORM-true	
	UNIFORM-even	
	RANDOMBRANCH	

Table 2: ANOVA Results for Symbolic Regression. Algorithms are in decreasing order by average over 50 runs of best fitness per run. Vertical lines indicate classes with statistically insignificant differences.

Figures 1 through 6 show the results for the various algorithms applied to 11-Boolean Multiplexer. Figures 8 through 13 show the results for the algorithms applied to Artificial Ant. As can be seen, the algorithms produce surprisingly similar results. ANOVAs at 0.05 performed on the algorithms for both the 11-Boolean Multiplexer problem and the Artificial Ant problem indicate that there is no statistically significant difference among any of them. For Symbolic Regression, an ANOVA indicated statistically significant differences. The post-hoc Fisher LSD and Tukey tests, shown in Figure 2, reveal that UNIFORM fares worse than all algorithms except RANDOMBRANCH!

4 SECOND EXPERIMENT

If uniformity provides no statistically significant advantage, what then accounts for the authors' claims of improvements in fitness? One critical issue might be average tree size. If reports in the literature were not careful to normalize for size differences (very easy given that RAMPED HALF-AND-HALF has no size parameters, and duplicate rejection causes unforseen effects) it is entirely possible that significant differences can arise.

The goal of the second experiment was to determine how much size matters. Using UNIFORM-even, we performed 30 independent runs each for the following maximum-size values: 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20, 25, 30, 40, 50, 60, 80, 100. The test problem domains were again 11-Boolean Multiplexer, Artificial Ant, and Symbolic Regression with two features modified. First, the population size was reduced from 500 (the standard in (Koza, 1992)) to 200, to speed up runtime. Second, the runs were only done for eight generations, rather than 50 (standard for (Koza, 1992)). The reasoning behind this is that after eight generations or so the evolutionary system has generally settled down after initial "bootstrapping" effects due to the tree-generation algorithm chosen.

Figures 7, 14, and 21 show the results of this experiment. The light gray dots represent each run. The dark gray dots represent the means of the 30 runs for each maximum-size value. Because of duplicate rejection, runs typically have mean initial tree sizes somewhat different from the values predicted by the provided maximum-size. Also note the apparent horizontal lines in the 11-Boolean Multiplexer data: this problem domain has the feature that certain discrete fitness values (multiples of 32) are much more common than others.

These graphs suggest that the optimal initial tree size for `UNIFORM-even` for both domains is somewhere around 10. Compare this to the standard tree sizes which occur due to `RAMPED HALF-AND-HALF`: 21.2 for 11-Boolean Multiplexer and 36.9 for Artificial Ant!

5 CONCLUSION

The tree generation algorithms presented provide a variety of advantages for GP researchers. But the evidence in this paper suggests that improved fitness results is probably not one of those advantages. Why then pick an algorithm over `RAMPED HALF-AND-HALF` then? There are several reasons. First, most new algorithms permit the user to *specify* the size desired. For certain applications, this may be a crucial feature, not the least because it allows the user to create a size distribution more likely to generate good initial individuals. Fighting bloat in subtree mutation also makes size-specification a desirable trait.

Second, some algorithms have special features which may be useful in different circumstances. For example, `PTC1` and `PTC2` have additional probabilistic features not described in the simplified forms in this paper. Both algorithms permit users to hand-tune exactly the likelihood of appearance of a given function in the population, for example.

The results in this paper were surprising. Uniformity appears to have little consequence in improving fitness. Certainly this area deserves more attention to see what additional features, besides mean tree size, *do* give evolution that extra push during the initialization phase. Lastly, while this paper discussed effects on *fitness*, it did not delve into the effects of these algorithms on *tree growth*, another critical element in the GP puzzle, and a worthwhile study in its own right.

Acknowledgements

The authors wish to thank Ken DeJong, Paul Wiegand, and Jeff Bassett for their considerable help and insight.

References

Walter Bohm and Andreas Geyer-Schulz. Exact uniform initialization for genetic programming. In Richard K. Belew and Michael Vose, editors, *Foundations of Genetic Algorithms IV*, pages 379–407, University of San Diego, CA, USA, 3–5 August 1996. Morgan Kaufmann.

Kumar Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, September 1997.

Andreas Geyer-Schulz. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, volume 3 of *Studies in Fuzziness*. Physica-Verlag, Heidelberg, 1995.

Hitoshi Iba. Random tree generation for genetic programming. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 144–153, Berlin, Germany, 22-26 September 1996. Springer Verlag.

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

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

Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions in Evolutionary Computation*, 4(3), 2000b.

P. A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 July 1995a.

P. A. Whigham. Inductive bias and genetic programming. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 461–466, Sheffield, UK, 12-14 September 1995b. IEE.

P. A. Whigham. Search bias, language bias, and genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

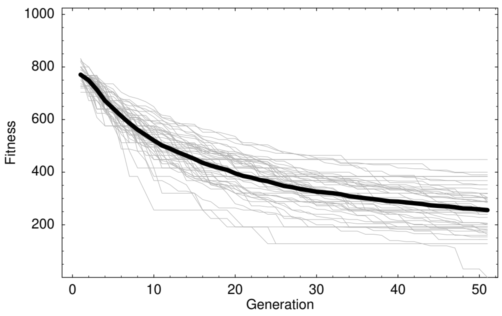

Figure 1: Generation vs. Fitness, `RAMPED HALF-AND-HALF`, 11-Boolean Multiplexer Domain

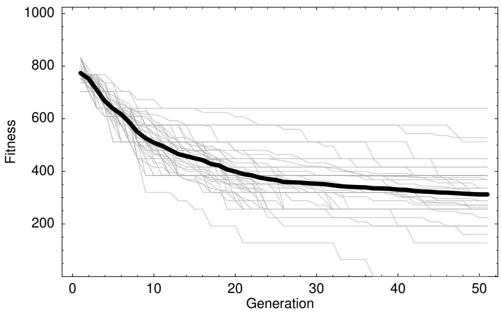

Figure 2: Generation vs. Fitness, `PTC1`, 11-Boolean Multiplexer Domain

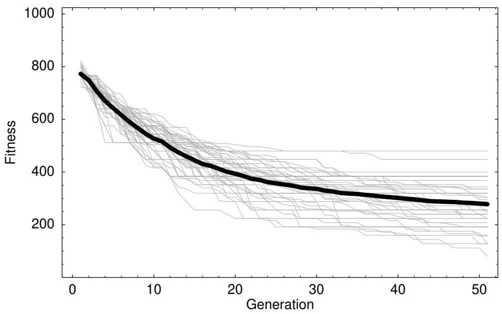

Figure 3: Generation vs. Fitness, `PTC2`, 11-Boolean Multiplexer Domain

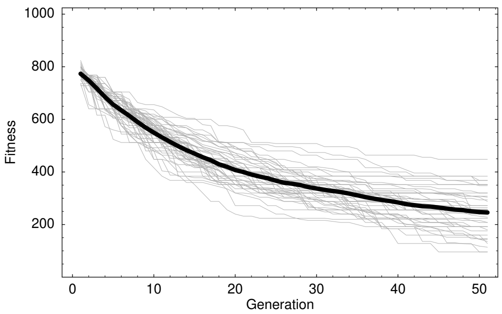

Figure 4: Generation vs. Fitness, `RANDOMBRANCH`, 11-Boolean Multiplexer Domain

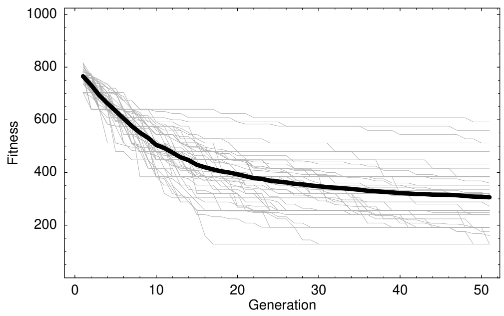

Figure 5: Generation vs. Fitness, `UNIFORM-even`, 11-Boolean Multiplexer Domain

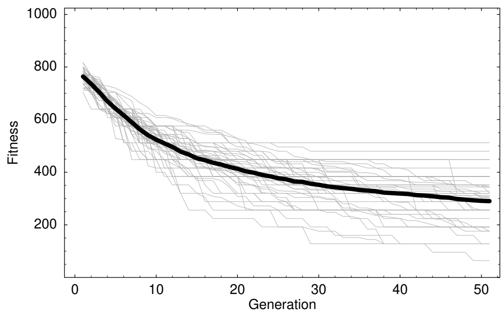

Figure 6: Generation vs. Fitness, `UNIFORM-true`, 11-Boolean Multiplexer Domain

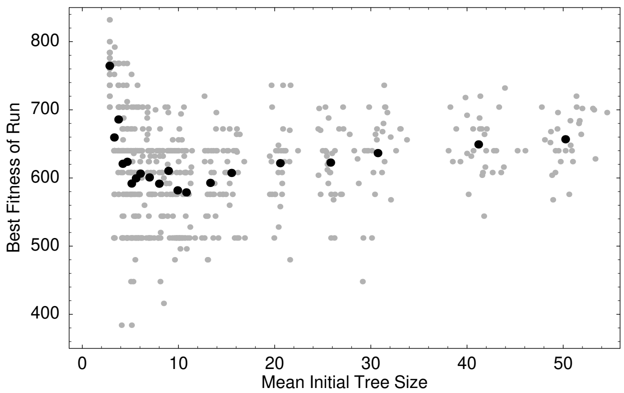

Figure 7: Mean Initial Tree Size vs. Fitness at Generation 8, 11-Boolean Multiplexer Domain

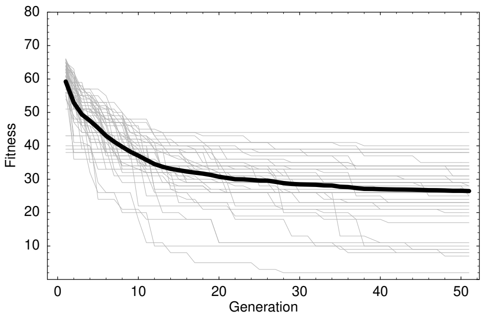

Figure 8: Generation vs. Fitness, `RAMPED HALF-AND-HALF`, Artificial Ant Domain

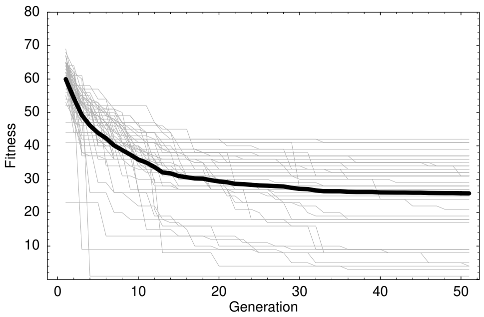

Figure 9: Generation vs. Fitness, `PTC1`, Artificial Ant Domain

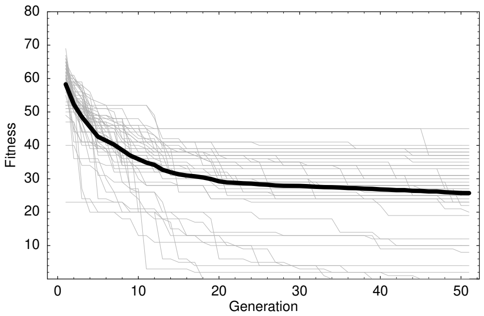

Figure 10: Generation vs. Fitness, `PTC2`, Artificial Ant Domain

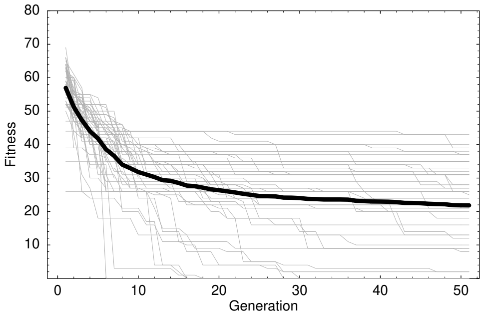

Figure 11: Generation vs. Fitness, `RANDOMBRANCH`, Artificial Ant Domain

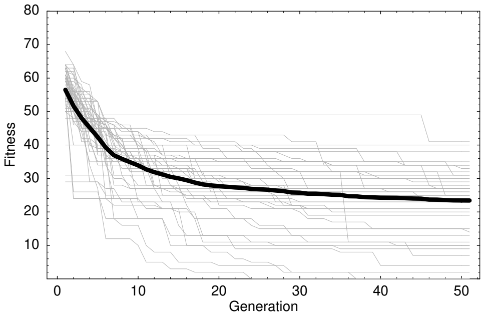

Figure 12: Generation vs. Fitness, `UNIFORM-even`, Artificial Ant Domain

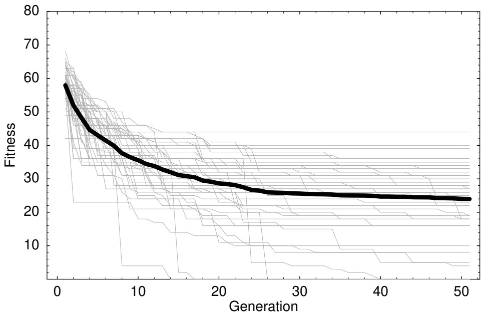

Figure 13: Generation vs. Fitness, `UNIFORM-true`, Artificial Ant Domain

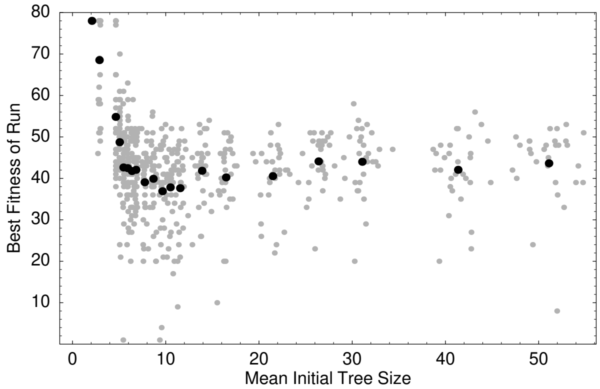

Figure 14: Mean Initial Tree Size vs. Fitness at Generation 8, Artificial Ant Domain

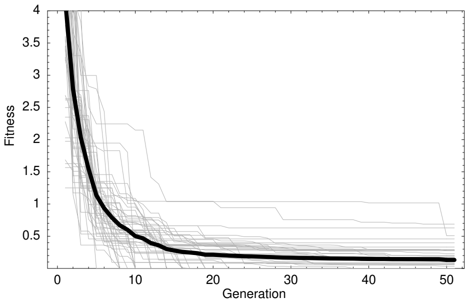

Figure 15: Generation vs. Fitness, `RAMPED HALF-AND-HALF`, Symbolic Regression Domain

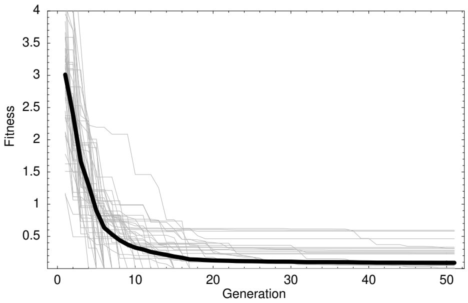

Figure 16: Generation vs. Fitness, `PTC1`, Symbolic Regression Domain

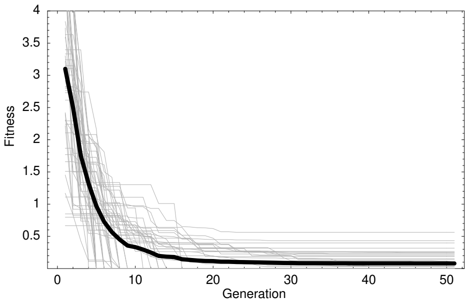

Figure 17: Generation vs. Fitness, `PTC2`, Symbolic Regression Domain

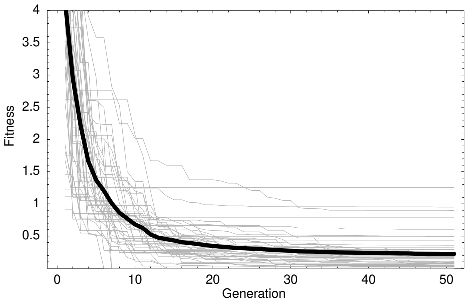

Figure 18: Generation vs. Fitness, `RANDOMBRANCH`, Symbolic Regression Domain

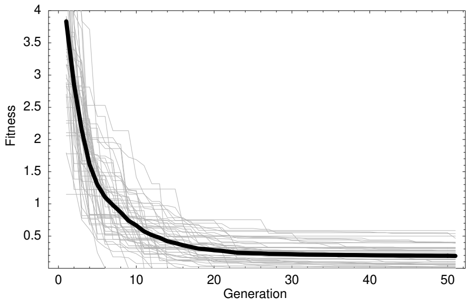

Figure 19: Generation vs. Fitness, `UNIFORM-even`, Symbolic Regression Domain

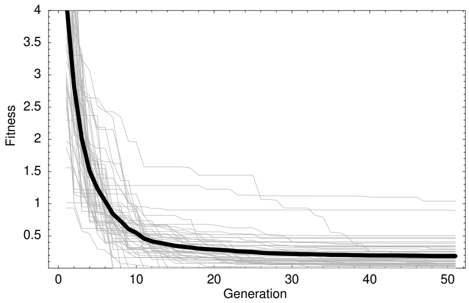

Figure 20: Generation vs. Fitness, `UNIFORM-true`, Symbolic Regression Domain

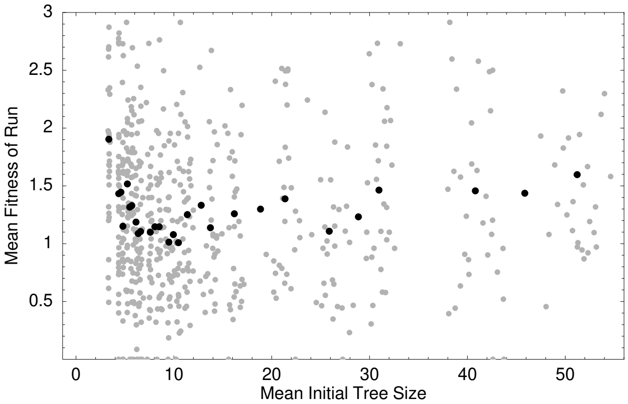

Figure 21: Mean Initial Tree Size vs. Fitness at Generation 8, Symbolic Regression Domain