On the Behavioral Diversity of Random Programs

Moshe Looks
Department of Computer Science and Engineering
Washington University in St. Louis
Saint Louis, MO 63130, USA
moshe@metacog.org

ABSTRACT

Generating a random sampling of program trees with specified function and terminal sets is the initial step of many program evolution systems. I present a theoretical and experimental analysis of the expected distribution of uniformly sampled programs, guided by algorithmic information theory. This analysis demonstrates that increasing the sample size is often an inefficient means of increasing the overall diversity of program behaviors (outputs). A novel sampling scheme (*semantic sampling*) is proposed that exploits semantics to heuristically increase behavioral diversity. An important property of the scheme is that no calls of the problem-specific fitness function are required. Its effectiveness at increasing behavioral diversity is demonstrated empirically for Boolean formulae. Furthermore, it is found to lead to statistically significant improvements in performance for genetic programming on parity and multiplexer problems.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming – *Program synthesis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Design, Experimentation

Keywords

Empirical Study, Heuristics, Optimization

1. INTRODUCTION

"Duplicate individuals in the initial random generation are unproductive deadwood; they waste computational resources and undesirably reduce the genetic diversity of the population." *John Koza*, *Genetic Programming* [7]

A fundamental aspect of evolutionary optimization is variation of solutions and selection according to differential fitness. Variation without differential fitness is referred to as *neutrality.* In domains such as program evolution, where solutions are transformed into some phenotype before fitness evaluation takes place, we can distinguish *selective neutrality* (identical fitness) from the generally stronger notion of *behavioral neutrality* (identical behavior). For example, the Boolean formulae *and(x, x)* and *or(x, x)* exhibit behavioral neutrality, whereas *and(x, x)* and *or(x, y)* do not, although the may be assigned the same score according to some fitness function (selective neutrality).

After evolutionary search has progressed somewhat, a set of behaviorally neutral programs in a population may arguably encode useful information regarding the exploration of surrounding regions of the space, partial immunity from disruptive operations such as subtree crossover, etc. Certain kinds of neutrality have indeed been shown to enhance the performance of evolutionary algorithms [19]. In the initial generation, however, the population consists entirely of a random sampling of programs. Here, we may safely consider behavioral neutrality undesirable; all things being equal, the more unique behaviors that get sampled, the better.

It is thus of interest to investigate how diverse random samplings of program space are, behaviorally, when drawn according to different sampling methodologies. We shall see that behavioral diversity is in some cases quite lacking. I propose and validate a novel heuristic to remedy this situation, that attempts to generate a sample with maximal behaviorally diversity (i.e., all programs corresponding to unique behaviors). This goes well beyond the common practice of eliminating syntactic duplicates in the initial population as suggested by Koza [7].

2. THE DISTRIBUTION OF BEHAVIORS

Previous studies [7, 10, 5, 17] have found that program spaces, when sampled according to various schemes, compute a highly skewed distribution of functions ("simpler" functions are overrepresented). Langdon and Poli's work in particular contains numerous results and discussion, encompassing a number of specific domains, and programs-in-general as well [10]. Furthermore, this skewed distribution cannot be avoided in general by sampling larger or smaller programs – convergence results can be obtained across program spaces satisfying fairly general requirements, indicating that the distribution of behaviors as a function of program size remains essentially fixed past a certain size [10].

These results can be qualitatively understood as corollar-

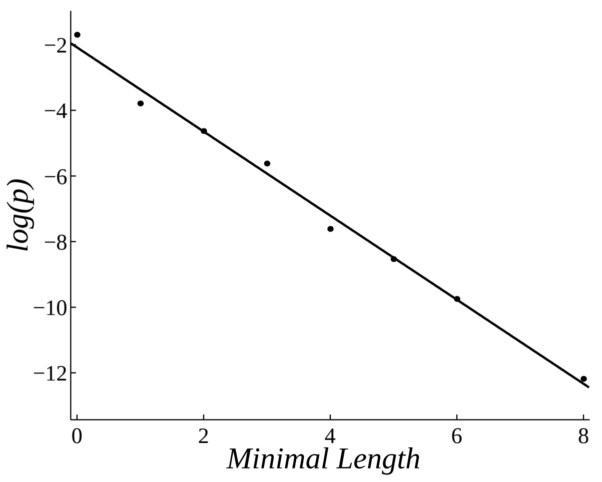

Figure 1: Minimal formula length measured in literals vs. log-of-density, in the space of ternary Boolean formulae with one hundred literals. Density (p) is the proportion of formulae sampled with a given minimal length. The solid line is a regression fit, $log(p) \approx -2.083 - 1.281 \cdot Minimal\ Length$.

ies of: (1) the incompressibility of minimal programs [3]; and (2) that an output (i.e., behavior) x's algorithmic probability may be effectively approximated by 2^{-l}, where l measures (in bits) the length of the minimal program computing x [16]. This value l may be considered x's *intrinsic complexity* in the given program space, corresponding to Kolmogorov complexity for Turing-complete program spaces.

2.1 Experiments

Boolean formulae in particular follow this general relationship (i.e., a geometric distribution), as illustrated in Figure 1. This shows the distribution of formulae with one hundred literals,[1] grouped by minimal program length (the number of literals in the shortest corresponding formula). Because computing the minimal program length for functions of high arity is prohibitively expensive, data are limited to ternary functions (i.e., the terminal set contains three variables). A sampling of a million programs was used. All ternary functions except for parity are represented; *AND/OR/NOT* formulae computing 3-parity are too rare to regularly appear in a random sampling of a million formulae (their minimal formulae have ten literals). Note that tautology and contradiction are displayed as containing zero literals.

The graph in Figure 1 fits the theory well because for formula with one hundred literals the *limiting distribution* for ternary Boolean formulae has effectively been reached, and because one million samples is sufficient to compute reasonably accurate probabilities for all but the rarest of behaviors (i.e., the parity functions). What are the consequences for smaller sample sizes and samples containing smaller programs? Consider how many unique behaviors we can expect in a sample of m programs of a particular size (k) in a particular space (for Boolean formulae this will grow as a function of the arity). In particular, we will consider the number of unique behaviors as a fraction of the total number of programs sampled (henceforth the *behavioral diversity* of a sample).

[1]Sampled by uniformly randomly selecting a binary tree with one hundred leaves and labeling the leaves with random literals, and the internal nodes with either *AND* or *OR*, chosen uniformly randomly.

The graph on the left in Figure 2 shows that as the sample size m increases, behavioral diversity decreases, which is to be expected given the heavy skew towards programs with short minimal lengths outlined above. An observation based on this result is that simply increasing our (syntactic) sample size is often an inefficient method for achieving behavioral diversity. The graph on the right in Figure 2 shows the same qualitative effect as a function of formula size (k). However, program space grows so quickly that in practice this can only really be observed for very small k. For $k = 10$ for instance, the behavioral diversity for a random sample is nearly identical to the distribution for $k = 100$ (and in fact to that for $k = 2000$ as well).

A final feature to consider in these graphs is the ubiquitous S-shaped curve as the arity (n) increases. The left side of the transition ($n \leq 2$) can be understood as the regime where behavioral diversity is effectively bounded by the number of unique behaviors in the space (2^{2^n}). The right side of the transition (around $n \geq 8$), correspondingly, is the regime where the number of unique behaviors in the space dominates distributional skew and sample size, leading to high behavioral diversity. A future area of interest is to consider the ubiquity of these phase transitions in program spaces and their relationship to problem difficulty and the underlying combinatorics of the space, as has been successfully achieved for optimization problems such as Boolean satisfiability [15] and the traveling salesman [4].

2.2 Discussion

A consequence of a highly skewed program distribution is that the likelihood of generating "complex" behaviors in a random sampling, when complex is quantified in terms of minimal length, decreases exponentially with the target complexity. This holds for sampling schemes where all programs generated have the same size (as above and in [9, 2]), as well as when program size is varied (as in the common ramped-half-and-half scheme [7], and in the ramped uniform scheme proposed in [9]).

This may help explain the results of Luke and Panait [13], who found little difference in the performance of genetic programming when initialized with five different sampling schemes. The critical difference between their study and prior investigations was that average program size, which presumably affected the dynamics of subtree crossover, was controlled for. This normalization across initialization methods is of course also carried out for the comparative analyses presented later in this paper.

I hypothesize that increasing behavioral diversity and controlling the distribution of program complexities in the initial population can improve the performance of program evolution. But how to test this? One simple strategy for increasing behavioral diversity is to simply discard programs leading to duplicate behaviors, much as duplicate programs are often discarded. Three drawbacks of this scheme are: (1) the behaviors of all programs must be computed and indexed for comparison (e.g., via hashing), which adds a problem-dependent computational overhead; (2) the num-

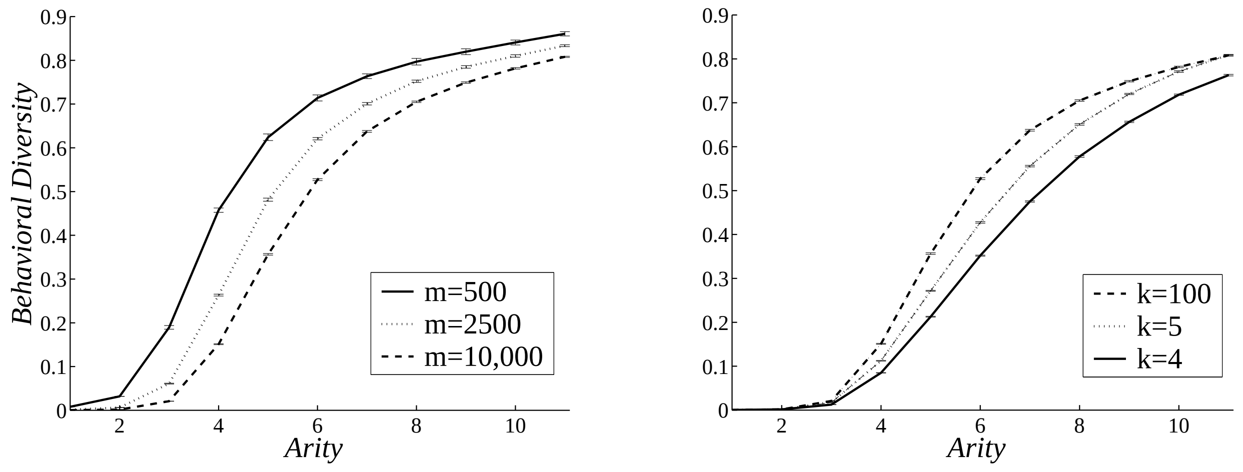

Figure 2: The proportion of unique behaviors (Boolean functions) in a given sample of programs (Boolean formulae), as a function of the arity of the function space. This can be seen to vary based on the sample size m (left), and formula size measured in literals, k (right). Formula size for the left graph is one hundred literals, and sample size for the right graph is ten thousand. Error bars are 95% confidence intervals based on thirty independent trials.

ber of programs that will need to be generated and tested may be quite large; and (3) there is still no direct control over the distribution of program complexities. In the next section I propose a new heuristic for sampling programs that addresses all of these difficulties.

3. A NEW SAMPLING HEURISTIC

The goal of the new "semantic sampling" approach described below is to approximate a uniform distribution of minimal programs by length. That is, given a maximum program size n and a sample size N, we want to generate around N/n unique minimal programs for each size between one and n. Such a hypothetical distribution is in a sense optimal: (1) it has total behavioral diversity (by definition); and (2) if the number of minimal programs of size l grows exponentially (as it does for general programs), selection will be in accordance with algorithmic probability (i.e., geometric with respect to l) [16].

3.1 Algorithm

The problem with directly implementing the scheme outline above is that minimal programs are often incomputable, and at best very expensive to find. As an approximation, a heuristic program reduction (simplification) mechanism will be assumed; programs that cannot be reduced will be considered minimal. An idea that immediately presents itself is to generate random programs according to some syntactic mechanism, reduce them, and then add them to the sample (if they are not duplicates). This is more tractable, but may still take a very long time, if many duplicates are generated. Furthermore, there are no guarantees regarding the distribution of program lengths.

Semantic sampling exploits the heuristic that a program of s created by combining several smaller reduced random programs of sizes adding up to about s is more likely to be irreducible than a completely random program of size s. To avoid the inefficiency of discarding programs that do not match the target size, dynamic programming is used to cache unused program fragments.

The pseudocode shown in Algorithm 1 is based on the uniform random tree sampling procedure described in [1]. This uses a precomputed table of the probabilities of different tree shapes for a given size s. A tree shape is a tuple $(s_1, s_2, ..., s_a)$ indicating that the arity of the root node is a, with the ith subtree containing s_i nodes, such that $\sum_{i=1}^{a} s_i = s$.

For very small programs, complete enumeration is used; a constant C bounds the maximum number of programs that will be enumerated, and should be at least an order of magnitude greater than n/N. This ensures that there are at least n/N irreducible programs for sizes for which enumeration does not occur, avoiding the possibility of an infinite loop.[2] In the experiments described below, $C = 10,000$. Note that the cache does not store duplicate programs.

After $Generate(N, n)$ has been called, the cache will contain all minimal programs up to the largest size where there are less than C programs total. For every larger size up to n, it will contain at least N/n unique, irreducible programs. Programs are synthesized in *Generate* from largest to smallest to avoid wasted effort; if we sample a program of size n that reduces to size $n-1$, there is no reason to discard it. Thus in practice, it is generating the largest programs that is the most computationally costly.

The worst case bound for *Generate* is quite bad, because for arbitrary programming languages and simplification routines, there is no guarantee that a program randomly composed of irreducible subprograms will be anywhere near irreducible. In practice however, this is quite often the case. In all of the experiments described below involving genetic programming, for instance, GP runtimes dominated the runtimes for semantic sampling required to generate the ini-

[2]In the actual implementation, there is a cutoff after repeated failures to guarantee termination. This is omitted for simplicity from the pseudocode.

Algorithm 1 Semantic Sampling

```
procedure GENERATE($N, n$)
    $minSize \leftarrow 1$
    while $ProgramCount(minSize) < C$ and
            $minSize \leq n$ do
        generate all programs of size $minSize$
        reduce them
        add those still of size $minSize$ to the cache
        $++minSize$
    end while
    for $programSize \leftarrow n$ down to $minSize$ do
        while cache contains $< N/n$ programs
                of size $programSize$ do
            insert $Sample(programSize)$ into cache
        end while
    end for
end procedure

procedure PROGRAMCOUNT($c$)
    return total number of possible programs of size $c$
end procedure

procedure SAMPLE($s$)
    initialize a new program cache $tmp$
    loop
        $(s_1, s_2, ..., s_a) \leftarrow$ random tree shape with $s$ nodes
        $p \leftarrow$ an empty program
        $root(p) \leftarrow$ a random function of arity $a$
        for $i \leftarrow 1$ to $a$ do
            append $GetProgram(s_i)$ as a child of $root(p)$
        end for
        if $p$ is irreducible then
            move all items from $tmp$ to the main cache
            return $p$
        end if
        add all of $root(p)$'s children to $tmp$
    end loop
end procedure

procedure GETPROGRAM($s$)
    if all reduced programs of size $s$ are cached then
        return a random program of size $s$ from the cache
    else if cache contains programs of size $s$ then
        $p \leftarrow$ a random program of size $s$ from the cache
        erase $p$ from the cache
        return $p$
    end if
    return $Sample(s)$
end procedure
```

tial populations. Specifically, a C++ implementation of semantic sampling for Boolean formulae with the reduction rules described below takes around a second to generate 500 unique random formulae on a modern (circa 2006) PC, and around twenty second to generate 10,000 (i.e., the scaling is about linear).

Calls to $ProgramCount(c)$ simply return the total number of program trees of size c. For example, given two binary functions and three terminals $ProgramCount(c) = 3^c \cdot 2^{c-1} \cdot Catalan(c-1)$, assuming program size is here measured by the number of terminals (which is reasonable for binary trees). The Catalan numbers are an integer sequence, $Catalan(n) = (2n)!/((n+1)!n!)$, corresponding to the number of binary trees with $n+1$ leaves (among other things).

The *Sample* procedure generates a new random program of a specified size. Along the way, it may generate new random programs of smaller sizes as well, which are added to the cache (if they are not already there). An important property of *Sample* is that even though a common cache is used across successive calls, each program that is actually sampled is *independent* of the others, because any subprograms that get composed into larger programs are themselves discarded. *GetProgram* is an auxiliary procedure used by *Sample* to generate subprograms (by recursively calling *Sample*, if necessary). Subprograms that are obtained from *GetProgram* and lead to a reducible overall program are put into a local cache (*tmp*) rather than being returned directly to the main cache. This keeps them from being immediately resampled, which can lead to pathological cases where the same small set of subprograms is endlessly recombined.

3.2 Simplification

Before applying semantic sampling, it is necessary to select domains and define simplification procedures for reducing programs. The domain studied in this paper is Boolean formulae with the basis $\{AND, OR, NOT\}$.

Holman has developed an "elegant normal form" (ENF) [6] for the simplification of Boolean formulae that is both computationally efficient to derive and heuristically effective. Holman reports for instance on experiments involving randomly generated Boolean formulae with between one hundred and five hundred literals, where 99% of the formulae required fewer than 10,000 atomic operations to reduce to ENF, and retained fewer than 2% of their original literals. Formulae in ENF use the basis $\{AND, OR, NOT\}$. To define ENF, it is convenient to introduce some terminology:[3]

- The *guard set* of an internal node is all of its children that are literals, and the guard set of a literal is itself.
- A *branch set* is the union of all of the guard sets of conjunctions and literals on the shortest path between some leaf and the root.
- The *dominant set* of a node is the union of all of the guard sets of nodes on the shortest path between the node and the root (excluding the node itself).

Consider the formula on the left in Figure 3. The AND node in the lower right's guard set is $\{x_3, x_6\}$, and its dominant set is $\{x_1, x_2, x_3, x_7\}$. The branch set for the literal

[3]For clarity, these definitions are slightly different from those presented in [6].

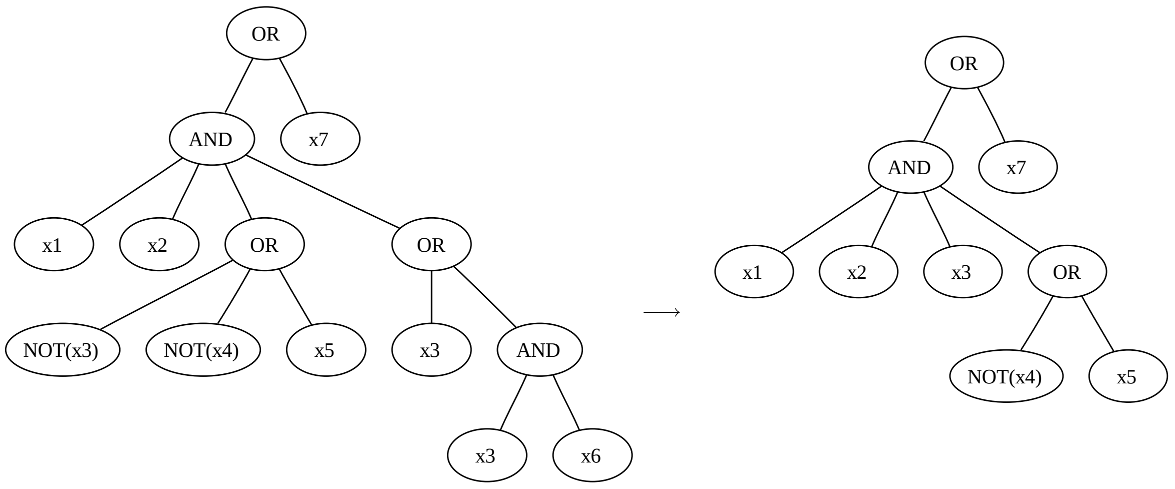

Figure 3: A redundant Boolean formula (left) and its equivalent in hierarchical normal form (right).

x_5 (in the center) is $\{x_1, x_2, x_5\}$. A formula is in ENF, as defined by Holman, if all of the following hold:

1. Negation appears only in literals.
2. Levels of conjunction and disjunction alternate.
3. No conjunction or disjunction has both a literal and its negation, or multiple copies of the same literal, as children.
4. No branch set contains a literal and its negation.
5. The intersection of all of the children of any disjunction's guard sets is empty.
6. The intersection of any conjunction's guard set and dominant set is empty.

Thus, the formula in Figure 3 on the left is not in ENF, because the intersection of the *OR* node in the lower right's children's guard sets is non-empty – it contains x_3 (condition 5). Holman [6] presents and algorithm reducing any formula to ENF that executes in $O(n \cdot min(n, k))$, where n is the arity of the space, and k is the number of literals in the formula. Essentially, this procedure consists of a set of eight reduction rules that are executed iteratively over the entire formula until no further reductions are possible. I have extended Holman's ENF and the corresponding reduction procedure to obey the following additional constraints:

7 The intersection of the guard sets of the children of a conjunction is empty – this corresponds to item 5 above, for a conjunction-of-disjunctions rather than a disjunction-of-conjunctions.

8 No node's guard set is a subset of any of it's siblings' guard sets.

9 For any pair of siblings' guard sets having the form $\{x\} \cup S_1$ and $\{NOT(x)\} \cup S_2$, where S_1 and S_2 are sets of literals, no *third* sibling's guard set is a subset of $S_1 \cup S_2$.

The rationale behind the the first of these additions of is merely symmetry – there appears to be no reason to reduce redundancy in conjunctions-of-disjunctions (item 4) and not in disjunctions-of-conjunctions (item 7). The latter two additions were chosen based on experimentation with small hand-crafted formulae. Empirically, their addition can reduce random formulae further than ENF alone about 7% of the time, by an average of around three literals, for formulae with one hundred literals.[4] From a computational complexity standpoint, the reductions needed to implement items 8 and 9 (searching all pairs of siblings for matching literals and their negations, then searching for subsets) add an additional multiplicative term that is quadratic in the maximum arity of any conjunction or disjunction. Empirically however, the impact on runtimes relative to the reduction to ENF is negligible.[5]

3.3 Experiments

The first experiment to run is to compute the behavioral diversity of programs generated via semantic sampling. Results are shown in Figure 4 for a target average program size of twenty; qualitatively similar results are achieved for smaller target program sizes (leading to somewhat less diversity at lower arities) and for larger target program sizes (leading to somewhat more diversity at lower arities). As can be seen, behavioral diversity for semantic sampling follows a qualitatively different pattern than for uniform random sampling (compare to Figure 2). It is nearly perfect for arities of six and above, and up to several times higher than for comparable uniform random samplings for arities four and five.

Now that we have verified that semantic sampling can effectively generate behaviorally diverse samples, the next step is to test the hypothesis that increased behavioral di-

[4]Tested on ten thousand random formulae generated as in the last chapter with arity ten. With arity five, about 6% of formulae reduced further than for ENF alone.

[5]An important implementation detail is to reduce formulae to Holman's ENF *before* searching through all pairs of siblings, otherwise there can be a significant increase (typically around 50%) in reduction times for large formulae.

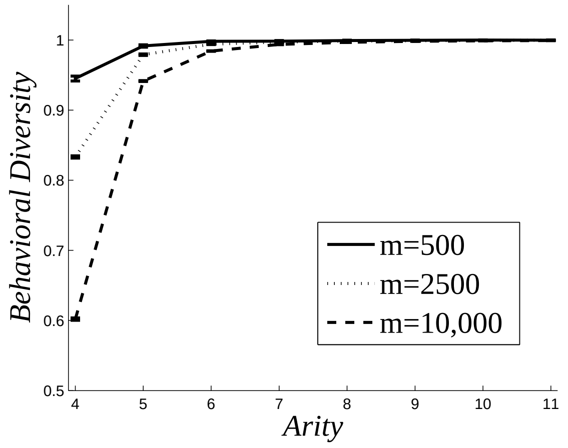

Figure 4: The proportion of unique behaviors (Boolean functions) in a given sample of programs (Boolean formulae), generated by semantic sampling. This varies based on the sample size (m). Error bars are 95% confidence intervals based on thirty independent trials.

versity in the initial sample can improve the performance of program evolution systems. Classic genetic programming as described by Koza [7] is used for this. Experiments were carried out with the the lil-gp system (version 1.1., *http://garage.cse.msu.edu/software/lil-gp/*). Size-seven tournaments were used for selection, with 90% crossover and 10% elitism (the defaults used in [8]). The maximum number of generations was set to fifty, the maximum number of nodes allows was 1000, and the maximum allowed depth was 17 (all standard default values). Two hundred independent runs were carried out with population sizes of 500, 4000, and 10,000 for every problem.

The control was ramped half-and-half with a depth ramp of 2 to 6 and duplicate rejection. In the comparative experiments of [13], this configuration is compared to four other program sampling schemes for generating the initial population for genetic programming applied to three problems; Boolean 11-multiplexer, artificial ant, and symbolic regression. None of the other methods were found to lead to (statistically significant) better results than ramped half-and-half on any of these problems (no statistically significant differences at all were observed for the first two problems).

The average tree sizes generated by ramped half-and-half were found empirically to be between 20 and 23 for all experiments (somewhat lower for smaller populations and lower arities). This is in agreement with the average sizes found in [13]. Semantic sampling was configured (by setting n, the maximum program size) to generate a uniform distribution of program sizes (starting from a single node), such that the average tree size remained within ± 2 of the average size for ramped half-and-half on all experiments.

The problems considered were Boolean 6-multiplexer,[6] 11-multiplexer, 4-parity, and 5-parity. Note that the function set is somewhat different from that commonly used in the literature; the *same* set, $\{AND, OR, NOT\}$, is used for all of the Boolean problems. This change is not expected to qualitatively affect the comparative results, while simplifying the implementation considerably.

Results are shown in Table 1. Fitness is normalized uniformly to fall in $[0, 1]$ on all problems (higher is better). So on 4-parity, for example, which has $2^4 = 32$ test cases, a program with 24 hits has fitness $24/32 = 0.75$. Success rate and computational effort are point measures (computed over the complete set of runs), and hence cannot be used as a basis for claims of statistical significance. The second column from the right, showing mean fitness after ten generations with 95% confidence intervals, indicates that using semantic sampling to generate the initial sampling boosts GP's performance, for all problems and population sizes tested. The results of the last column confirm the observation in [13] that GP dynamics tend to obliterate the effects of the initial sampling method in later generations – the differences that remain are of smaller magnitudes and not generally significant.

4. CONCLUSIONS

The origin of this paper was the question of how to sample a random complex program, given that most large random programs are not complex (when complexity is quantified in terms of minimal program length). A prior study that influenced this work was Luke and Panait [13], which demonstrated little difference in performance for genetic programming across the most popular sampling heuristics for generating the initial population (ramped half-and-half, PCT1 and PCT2 [12], and two uniform tree generation schemes). In the same paper, the average initial tree size is shown to have a marked effect on performance; this is used as an argument in favor of sampling heuristics where the distribution of tree sizes can be easily and precisely controlled. Semantic sampling falls under this heading; above very small sizes, any number of trees of any desired size may be generated (provided irreducible trees of the target size in fact exist, which was not an issue in any of my experiments).

Based on the experimental results, semantic sampling may be considered most appropriate for applications where good performance is desired quickly, after relatively few generations of evolution. Because there is some overhead involved in comparison to other tree sampling methods, it may not be appropriate for applications where fitness evaluation is extremely fast. The good results demonstrated for Boolean formulae should be expected to transfer to the symbolic regression domain, where simplification systems are also quite well developed. Since many GP systems already include simplification mechanisms for analyzing the final results, implementing semantic sampling should not be very difficult.

A further domain where semantic sampling should be explored is general (Turing-complete) programs. The distribution of such programs can tend towards *no programs halting* (see [11] for a detailed study and analysis). A semantic sampling procedure for generating initial populations could heuristically eliminate many non-halting programs, and approximate a uniform (with respect to complexity) distribution of halting programs. This could accordingly be expected to have a significant positive impact on performance.

Further work hybridizing semantic sampling with algebraic simplification during evolution [18], and/or canonical form functions [14], is of interest. The resulting GP system

[6]As this is a very easy problem for GP to solve, results are only shown for a population size of 500.

Table 1: Experiments comparing ramped half-and-half (rhh) and semantic sampling as the initialization method for GP. The two rightmost columns show mean best fitness with 95% confidence after 10 and 50 generations of evolution, respectively. When one method leads to improved performance (with statistical significance), the better result is shown in boldface.

Experimental Configuration		Population Size	Success Rate	Computational Effort	Mean Best Fitness After 10	Mean Best Fitness After 50
6-multiplexer	rhh	500	81.5%	68,000	0.894±0.005	0.989±0.004
	semantic	500	91%	40,500	**0.941±0.004**	**0.998±0.002**
11-multiplexer	rhh	500	0%	NA	0.711±0.002	0.847±0.006
	semantic	500	0%	NA	**0.722±0.003**	**0.863±0.006**
	rhh	4000	23%	3,672,000	0.75±0.002	0.972±0.004
	semantic	4000	32%	2,448,000	**0.764±0.002**	0.977±0.003
	rhh	10,000	61%	2,550,000	0.764±0.002	0.992±0.002
	semantic	10,000	77%	1,960,000	**0.78±0.002**	0.995±0.002
4-parity	rhh	500	20%	495,000	0.797±0.006	0.915±0.008
	semantic	500	23%	437,000	**0.833±0.005**	0.921±0.008
	rhh	4000	85%	416,000	0.874±0.004	0.990±0.003
	semantic	4000	95%	200,000	**0.906±0.005**	**0.997±0.002**
	rhh	10,000	100%	270,000	0.898±0.005	1.000±0.000
	semantic	10,000	99.5%	320,000	**0.938±0.004**	1.000±0.001
5-parity	rhh	500	0%	NA	0.664±0.004	**0.789±0.007**
	semantic	500	0%	NA	**0.680±0.003**	0.774±0.006
	rhh	4000	3%	NA	0.706±0.003	0.908±0.007
	semantic	4000	0.5%	NA	**0.726±0.003**	0.9±0.006
	rhh	10,000	15.5%	14,000,000	0.727±0.003	0.945±0.006
	semantic	10,000	12%	18,870,000	**0.742±0.003**	0.945±0.005

will be expected to gain a significant edge by more effectively sampling from a smaller overall search space. Another possibility to consider is using the semantic sampling methodology to produce a more intelligent mutation operator, to direct variation along semantically meaningful lines.

5. REFERENCES

[1] L. Alonso and R. Schott. *Random Generation of Trees.* Kluwer Academic, 1995.

[2] W. Bohm and A. Geyer-Schulz. Exact uniform initialization for genetic programming. In *Foundations of Genetic Algorithms*, 1996.

[3] G. J. Chaitin. *Algorithmic Information Theory.* Cambridge University Press, 1987.

[4] I. P. Gent and T. Walsh. The TSP phase transition. *Artificial Intelligence*, 1996.

[5] S. Gustafson, E. K. Burke, and G. Kendall. Sampling of unique structures and behaviours in genetic programming. In *European Conference on Genetic Programming*, 2004.

[6] C. Holman. *Elements of an Expert System for Determining the Satisfiability of General Boolean Expressions.* PhD thesis, Northwestern University, 1990.

[7] J. R. Koza. *Genetic Programming.* MIT Press, 1992.

[8] J. R. Koza. *Genetic Programming II.* MIT Press, 1994.

[9] W. B. Langdon. Size fair and homologous tree crossovers for genetic programming. *Genetic Programming and Evolvable Machines*, 2000.

[10] W. B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[11] W. B. Langdon and R. Poli. The halting probability in von Neumann architectures. In *European Conference on Genetic Programming*, 2006.

[12] S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 2000.

[13] S. Luke and L. Panait. A survey and comparison of tree generation algorithms. In *Genetic and Evolutionary Computation Conference*, 2001.

[14] T. McConaghy and G. Gielen. Canonical form functions as a simple means for genetic programming to evolve human-interpretable functions. In *Genetic and Evolutionary Computation Conference*, 2006.

[15] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic ‘phase transitions’. *Nature*, 1999.

[16] R. Solomonoff. A formal theory of inductive inference. *Information and Control*, 1964.

[17] M. Tomassini, L. Vanneschi, P. Collard, and M. Clergue. A study of fitness distance correlation as a difficulty measure in genetic programming. *Evolutionary Computation*, 2005.

[18] P. Wong and M. Zhang. Algebraic simplification of GP programs during evolution. In *Genetic and Evolutionary Computation Conference*, 2006.

[19] T. Yu and J. Miller. Neutrality and the evolvability of Boolean function landscape. In *European Conference on Genetic Programming*, 2001.