Scalable Estimation-of-Distribution Program Evolution

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

ABSTRACT

I present a new estimation-of-distribution approach to program evolution where distributions are not estimated over the entire space of programs. Rather, a novel representation-building procedure that exploits domain knowledge is used to dynamically select program subspaces for estimation over. This leads to a system of demes consisting of alternative representations (i.e. program subspaces) that are maintained simultaneously and managed by the overall system. Meta-optimizing semantic evolutionary search (MOSES), a program evolution system based on this approach, is described, and its representation-building subcomponent is analyzed in depth. Experimental results are also provided for the overall MOSES procedure that demonstrate good scalability.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming – *Program synthesis*

General Terms

Algorithms, Design, Experimentation

Keywords

Empirical Study, Heuristics, Optimization, Representations

1. REPRESENTATIONS AND MODELS

Current approaches to program evolution based on the estimation of distributions may be divided into three categories: probabilistic models based on prototype trees with variables corresponding to program symbols, models based on grammar induction, and models with linear encodings that are mapped to programs.

The archetypal prototype tree approach to program evolution is probabilistic incremental program evolution (PIPE) [18]. PIPE is based on a rooted schema model of programs trees. All trees in the population are aligned with a probabilistic model with a fixed topology (the prototype tree). All subtrees are similarly aligned according to their absolute position in the tree (e.g., "third argument of second argument of the root node"). To generate new solutions, PIPE independently samples from the distribution of functions and terminals at each node in the prototype tree. If 30% of the programs in the population have root nodes consisting of the function $+$, and 70% have the function $*$, then trees rooted in $+$ will be created with probability 0.3 and $*$ with probability 0.7, independent of the remainder of their contents. The probabilistic model is thus univariate (no interactions between variables).

There are two basic questions that a probabilistic modeling approach to program evolution must answer: (1) "How are programs represented and program subcomponents defined?"; and (2) "What kinds of interactions are possible between subcomponents?". The former determines how programs are represented, and the latter what sort of probabilistic model is constructed on top of this representation.

Based on this outlook, two primary limitations of the PIPE paradigm are the absolute position addressing system based on rooted-tree schemata (inadequate representation) and assumption of independence between nodes (inadequate modeling). Both of these are manifest to varying degrees in most program learning problems. Regarding the former, consider that $f(x)$ and $0.99 \cdot f(x)$, while nearly identical behaviorally, may share no rooted-tree schemata at all. Regarding the latter, many program evolution problems have been demonstrated empirically to contain significant across-node dependencies, for example in [8, 20]. Note however that these two limitations are linked; an inadequate representation make it difficult to correctly model the structure of program spaces (a point we shall return to later).

Several extensions of PIPE have been developed. Hierarchical PIPE (hPIPE) [19] adds hierarchical instructions, which can bias sampling towards programs with a particular overall organization (e.g., a linear combination of nonlinear elements), and skip nodes, which allow nodes to contain instructions for expressing only one of their child subprograms, ignoring the others. Estimation-of-distribution programming (EDP) [24] uses a Bayesian network to represent dependency relationships between nodes, with a fixed topology; nodes are dependent on their parent and left sibling. In [25], EDP is hybridized with genetic programming (GP). Extended compact GP (ECGP) [20] similarly extends PIPE to a non-univariate model, in this case a marginal product model that is relearned after each generation.

These extensions all address one or both of the problematic assumptions of rooted-tree schema and independence mentioned above. The hPIPE does not strictly adhere to a rooted-tree schema model, because skip nodes allow schemata to manifest themselves in multiple program positions. Through different mechanisms, EDP and hPIPE's hierarchical instructions both relax the univariate assumption of PIPE via prior knowledge. Hybridizing GP with EDP (i.e., generating some new program via subtree crossover) allows rooted-tree schemata to "migrate" to other positions. The ECGP's probabilistic modeling eliminates the assumption of independence (or fixed dependencies) between nodes.

A number of EDAs for program evolution have been developed based on grammar models. Program evolution with explicit learning (PEEL) [21] models programs distributions with a stochastic context-free grammar (SCFG) extended to allow rules to be tagged with depth and location restrictions. Models are learned by incrementally splitting existing rules into sub-rules. Grammar model-based program evolution [22] and grammar-transformation-based EDA for GP [1] also learn SCFG models. Grammar induction methods may generically be said to represent program subcomponents locally (i.e., in terms of subtrees which may appear anywhere in a program), although refinements such as depth and location restrictions added to PEEL allow absolute-position constraints to be expressed. In order to model non-local interactions (e.g., between the content in distant program positions) context-dependent features must be introduced.

Two program evolution systems have been developed based on linear encodings that are mapped to programs, both based on the Bayesian optimization algorithm (BOA) [14]. BOA programming [10] is based on a probabilistic model (dynamically learned Bayesian networks with local structure [13]) with two sets of variables: one models the symbol content of programs (as in PIPE), and the other models program structure (tree shape). Schemata are not rooted in absolute positions, but neither are they completely relative, as the structure-description language is constrained. Bayesian automatic programming (BAP) [17] evolves fixed-length integer vectors which are optimized by the BOA (dynamically learned Bayesian network model [14]). The vectors are mapped to programs for evaluation, as in grammatical evolution [11]; integers are treated as indexing the rules in a context-free grammar defining the programming language.

1.1 A New Approach

There are clearly program learning problems that can benefit from non-univariate modeling procedures, because program spaces almost always exhibit strong dependencies across variables. Beyond this, how expressive models should to be for program-learning tasks of interest is an open question. For example, the ECGP models non-overlapping building blocks, while the BAP models overlapping but non-hierarchical building blocks, and BOA programming models overlapping hierarchical building blocks.

A more fundamental question, which I believe should be answered first, is how programs and program subcomponents are best represented for probabilistic modeling over. It may be that one of the representational approaches described above will emerge as facilitating greater "evolvability" than the others, leading to better performance on test problems. However, I claim that on a fundamental level, all programmatic representations are "the same" in the sense of having similar scaling behavior as problems become more difficult and languages become more expressive. Formalizations of this claim can be proven from a standpoint of computational complexity, learning theory, and algorithmic information theory (e.g., see Langdon and Poli's work on limiting distributions in program spaces [8]).

I would go further however and argue that in all but the simplest languages, the improvement to be had by leveraging domain knowledge and dynamically adjusting the representation *within* a program space (the approach herein) will be dramatically greater than the improvement from shifting *across* spaces (e.g., encoding programs in lists vs. trees, or learning absolute position vs. relative position dependencies). This is because, in contrast to general optimization, the semantics of program evaluation provide valuable prior knowledge applicable across broad ranges of problems.

2. MOSES

Meta-optimizing semantic evolutionary search (MOSES) is an estimation-of-distribution program evolution system based on this new approach, and distinguished by two key mechanisms: (1) exploiting semantics (what programs actually mean) to restrict and direct search; and (2) limiting the recombination of programs to occur within bounded subspaces (constructed on the basis of program semantics). As we shall see, this leads to superior performance and scalability in comparison to current purely syntactic techniques (local search, genetic programming, etc.). Furthermore, the evolved programs do not suffer from any kind of "bloating".

Recombination in MOSES occurs within parameterized program subspaces called *representations*. A specialized *representation-building* process is used which heuristically exploits semantics (e.g., $\forall x,\ x + 0 \rightarrow x$) to create meaningful parameters to vary, based on some *exemplar program*. This parameter variation is directed by the hierarchical BOA (hBOA) [13], an advanced estimation-of-distribution algorithm that dynamically learns problem decompositions encoded as Bayesian networks with local structure.

A population of programs associated with a common representation is a *deme*, and a set of demes (together spanning an arbitrary region of program space in a patchwork fashion) is a *metapopulation*. MOSES operates on a metapopulation, adaptively creating, removing, and allocating optimization effort to demes. Deme management is the second fundamental *meta* aspect of MOSES, after representation-building; it essentially corresponds to the problem of allocating computational resources among competing representations (i.e., programmatic organizational schemes).

A basic sketch of MOSES is as follows:

1. Construct an initial representation of very small programs (i.e., with the empty program as the exemplar) and use it to generate an initial random sampling. Add this deme to the metapopulation.

2. Select a deme from the metapopulation and iteratively update its sample, as follows:

(a) Select some promising programs from the deme's existing sample to use for modeling, according to the scoring function. Ties in the scoring function are broken by preferring smaller programs.

(b) Considering the promising programs as collections of parameter settings, generate new collections of parameter settings by applying hBOA optimization.

(c) Convert the new collections of parameter settings into their corresponding programs, evaluate their scores, and integrate them into the deme's sample, replacing less promising programs.

3. For each of the new programs that meet the criteria for creating a new deme, if any:

(a) Construct a new representation centered around the program (the deme's exemplar), and use it to generate a *new* random sampling of programs, producing a new deme.

(b) Integrate the new deme into the metapopulation, possibly displacing less promising demes.

4. Repeat from step 2.

Representation-building MOSES consists of three basic steps: reduction to normal form, neighborhood enumeration, and neighborhood reduction. In reduction to normal form, programs are heuristically simplified to eliminate redundancy. Neighborhood enumeration attempts to find possible transformations that correspond to behaviorally nearby variations on the source program. In neighborhood reduction, redundant transformations are heuristically culled to reach a more independent set. These will now be described in depth (for Boolean formulae); other details of MOSES (e.g., the deme creation criterion) are omitted for brevity, and may be found in [9]. Note that MOSES does not examine programs in the "neighborhood" one-by-one as in local search, but searches the space of *combinations* of neighborhood transformations (exponentially larger).

2.1 Reduction to Normal Form

In the domain of Boolean formulae, Holman has developed an "elegant normal form" (ENF) [4] for simplification that is both efficient to derive and heuristically effective. He reports for instance on experiments involving randomly generated Boolean formulae with hundreds of 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 $\{AND, OR, NOT\}$. In contrast to conjunctive normal form, ENF preserves formulae's hierarchical structure. To define ENF, let's introduce some terminology:[1]

- 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 1. 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 x_5 (in the center) is $\{x_1, x_2, x_5\}$. A formula is in ENF if:

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.

[1]For clarity, the definitions are slightly different than in [4].

Thus, the formula on the left in Figure 1 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). An efficient procedure for reducing any formula to ENF consisting of a set of eight reduction rules that are executed iteratively over the entire formula until no further reductions are possible appears in [4]. I have extended the definition 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 its 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 $S_1 \cup S_2$.

What is to be gained by reducing programs to a hierarchical normal form? Consider the relationship between syntactic distance (tree-edit distance in the case of programs) and semantic distance (distance in *behavior*, what programs actually do when evaluated): this may be quantified in terms of fitness-distance correlation measures. These can be effective predictors of algorithm performance [23], although they are not the whole story (cf. [5]). This can be easily studied in the domain of Boolean formulae, where the semantic distance between two formulae is simply the Hamming distance between their corresponding truth-tables. A neighborhood structure for defining syntactic distance may be derived from the following edit operations on formulae: (1) replacement of one literal with another; (2) removal of an operator and all but one of its children, which replaces it; and (3) the insertion of an operator above an existing subformula, along with a second argument consisting of a new literal.

The *edit distance* between two formulae may now be defined in terms of these operations by the minimal number of nodes that must be modified, removed, or inserted to obtain one formula from the other. We can define the syntactic neighborhood of a formula by considering all replacement operations (a distance of one), and all removals and insertions with a distance of two (i.e., only involving the removal or insertion of a single operator and a single literal).

Five thousand *syntactically and semantically unique* formulae were generated each for arities five and ten, and all pairwise distances computed. The resultant distributions are shown for arity five in Figure 2 on the left (results for arity ten are qualitatively similar). Syntactic distance is normalized to fall in $[0, 1]$. Density is normalized based on semantic distance (i.e., the density of every vertical slice sums to one). Data are binned into a 20×20 mesh, and

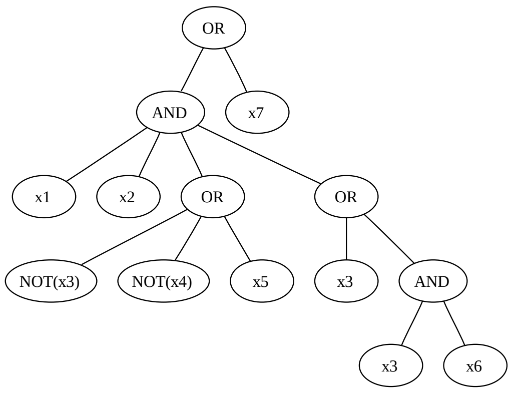


Figure 1: A redundant Boolean formula (left) and its equivalent in hierarchical normal form (right).

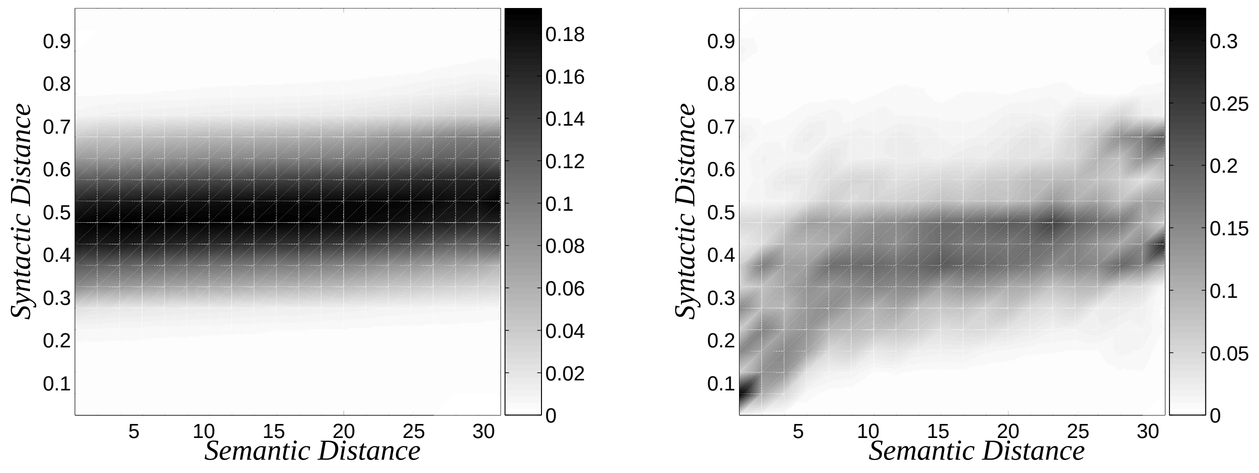


Figure 2: Pairwise distribution of program edit distance (syntactic) vs. truth-table Hamming distance (semantic) for random formulae with arity five, before (left) and after (right) reduction to ENF.

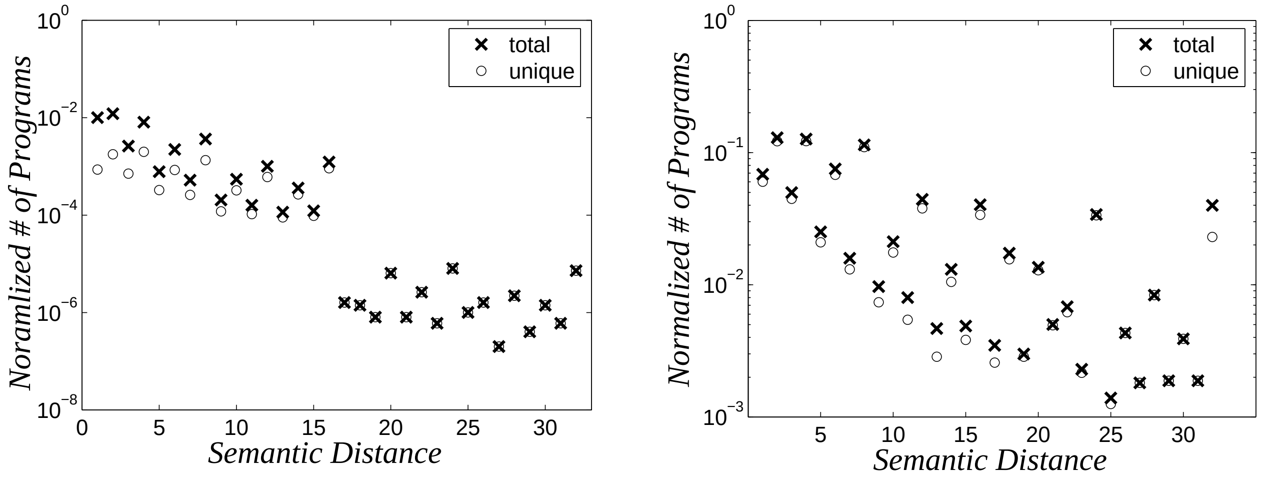


Figure 3: Distribution of behaviors and unique behaviors as a proportion of total neighborhood size for random formulae with arity five, before (left) and after (right) reduction to ENF.

interpolation is used to generate smooth figures. Identical computations were performed after reduction to ENF (Figure 2, right); the pairwise semantic distances will thus be identical, whereas the edit distances may change significantly. Edit distance is normalized based on division by the sum of the two formulae's sizes (this is to make comparisons of distances between formulae of different sizes meaningful).

For the original formulae, there is little variation in the distribution based on semantic distance; the syntactic space of large programs is globally nearly uniform with respect to semantic distance. Consequently, semantically similar programs are not, in a statistical sense, grouped any closer together than those of average similarity. After normalization, the results for both arities show a marked correlation between semantic and syntactic distance for small distances (signified by the diagonal sloping of the distribution towards the origin). Syntactic distances are now being computed based on symbols that actually shape the formula's behavior, and meaningless structural distinctions such *AND(x, AND(y, z))* vs. *AND(AND(x, y), z)* have been discarded.

2.2 Neighborhood Enumeration & Reduction

Given some neighborhood structure (e.g., all programs within some edit distance of the source), a straightforward way to do neighborhood reduction is to exploit the reduction to normal form outlined above; the number of symbols in the normal form of a program can be used as a heuristic approximation for its minimal length. If the reduction to normal form of the program resulting from some variation significantly decreases its size, it can be assumed to be a source of redundancy, and hence eliminated from consideration. The transformation to a slightly smaller program is typically a meaningful change to make, but a large reduction in complexity will rarely be useful (and if so, can be accomplished through a combination of smaller changes). At the end of this process, we will be left with a neighborhood defining a subspace of programs centered around a particular point in program space and heuristically centered around the corresponding point in behavior space as well.

Assume that this representation-building process creates effective neighborhoods based on a set of transformations within a constant distance from the exemplar program. There is still no guarantee that difficult problems will be solvable by recombining these small steps. Ideally, the step size should be adaptive, and increase as search progresses, when it becomes more difficult to find nearby improvements. Unfortunately, this methodology will lead to a combinatorial explosion without sophisticated safeguards.

The simplest remedy, taken in the current MOSES implementation, is to resort to subsampling. For the transformations described above involving the set of n literals, an additional $O(n)$ transformations will be considered involving the insertion of subformulae rather than single nodes. These will be based on $\lceil\sqrt{n}\rceil$ randomly generated formulae in ENF containing two literals, $\lceil\sqrt{n}\rceil$ with three literals, ..., and $\lceil\sqrt{n}\rceil$ with $\lceil\sqrt{n}\rceil$ literals, for a total of $(\lceil\sqrt{n}\rceil)^2 \leq n + 2\sqrt{n}$ additional transformations.

These random formulae are generated independently for each possible insertion. The asymptotic complexity of the sampling procedure is not affected. The motivation behind sampling from this distribution is to avoid overlap with existing transformations, and bias search towards uniform sampling by distance as the distance from the exemplar program increases (while the number of programs at a given distance grow exponentially).

Further background knowledge of Boolean formulae is exploited by controlling how the neighborhood space is parameterized. Having a subformula and its negation (e.g., x and $not(x)$) as children of the same *AND* or *OR* node will always lead to a contradiction or a tautology. Thus, instead of creating two separate binary parameters for the insertion/removal of a subformula and its negation, a single ternary parameter (present / absent / negated) is created.

Figure 3 left shows the distribution of behaviors and *unique* behaviors among the syntactic neighbors of random formulae,[2] as a proportion of the total neighborhood size. Only neighbors which are behaviorally different from the source formula are shown. The data are based on one thousand random formulae with one hundred literals. Most local perturbations of large random formulae have no effect – 96% of them for arity five, and 91% for arity ten (the drop is a corollary of the increase in the percentage of unique behaviors in a sample as the arity increases). This is a consequence of such formulae typically being highly redundant – e.g., a large formula that computes a tautology will likely remain a tautology when randomly perturbed (similarly a large *sub*formula which computes a tautology, etc.).

Considering the distribution of the remaining 4% of neighbors, note that the density of neighboring formulae with a behavioral distance of one or two is an order of magnitude greater than the density of unique behaviors. Not only is relatively little of the syntactic variation aligned with the semantic dimensions we would like to explore, but this little bit exhibits massive redundancy! This is yet another consequence of the skewed distributions present in program spaces (cf. [8]). The drop in density after the halfway point of semantic distance is due to symmetries in the space.

Figure 3 right shows the same computations, using the representation-building process described above to define the neighborhood. In contrast to the results for syntactic neighborhood structures, only 10% of the perturbations in representation-building neighborhoods have no effect (for both arities five and ten). There is also a significantly greater proportion of close neighbors – around an order of magnitude's difference. Furthermore, when the greater proportion of behaviorally unique neighbors attained via representation-building is taken into account, this rises to around two orders of magnitude.

3. EXPERIMENTAL RESULTS

Results are now presented for Boolean formula learning problems (even-parity and multiplexer), and JOIN expression mechanism problems (ORDER and TRAP). All of these problems are tuneably difficult based on adjusting their arities. Parameters for MOSES and their settings may be found in Table 1; none were varied across any of the problems studied herein. The extensive theory and practice of "competent optimization" [3, 12] allow reasonable settings to be straightforwardly extrapolated to MOSES.

3.1 Parity and Multiplexer

Parity formulae with the basis $\{AND,\ OR,\ NOT\}$ are difficult for evolutionary systems to learn, due to two main factors. Firstly, even-parity formulae are extremely rare in

[2]With arity 5; results for arity 10 are qualitatively similar.

Table 1: Parameters settings for MOSES.

Description	Value
population size (N) is $c{\cdot}n^{1.05}$ (derived in [12]), where n is the size of the representation in bits	$c = 5$
window size (w) for restricted tournament replacement in the hBOA, which implements niching	$w = min(0.05N, n)$, based on [12]
tournament size (selection pressure) used for model-building in the hBOA	2, based on [12]
complexity penalty for hBOA model-building (to avoid overfitting)	$log_2(N)$, derived in [12]
a deme is terminated after t_1 generations of hBOA, or t_2 generations with no improvement in the best score	$t_1 = n$, $t_2 = \lceil\sqrt{N/w}\rceil$, based on [15]

formula-space (their minimal formula size grows quadratically with the arity). Secondly, parity has a uniform structure (all variables are symmetrical), such that most formulae (in fact, any formula that does not contain all of the variables) will compute functions that are correct for exactly half of the cases; there is thus no gradient information to be obtained from these formulae. They are well-studied both theoretically and experimentally (cf. [8, 6]).

The easiest way to improve the performance of evolutionary learning for parity problems is to introduce some mechanism allowing the basis to be transformed (such as automatically defined functions [7]), or simply using a different basis (e.g., containing the equality or exclusive-or function); this eliminates the first source of difficulty. A method introduced by Poli and Page, sub-machine code GP with smooth uniform crossover [16], uses a representation which allows Boolean formulae to be varied in very small steps (by making changes to a single output of a binary function). This method eliminates the first source of difficulty, while additionally providing increased gradient information. These modifications, unsurprisingly, can indeed accelerate parity learning. However, part of what is of interest in the original problem formulation is no longer being studied, namely the question of how to evolve large programs with limited gradient information. Comparative results are thus only presented herein for the original formulation ([16] summarizes the results for transformed problem variants as well).

Multiplexer functions are defined to have arity $k + 2^k$, where k is a positive number. The first k arguments are address bits, and the remaining 2^k define an addressing space. With the ternary Boolean *if* function in the basis, multiplexers may be computed quite compactly, and learned fairly easily by evolutionary systems (c.f. [6]). Restricting the basis to {*AND, OR, NOT*} makes them more difficult. To obtain comparative results for multiplexers, GP was run for 51 generations with a population of 4,000, size-7 tournament selection, 90% crossover, and 10% elitism.[3] Results for parity are taken from the literature (see [2, 6] for settings).

To study the effects of probabilistic model-building on the performance of MOSES (as opposed to representation-building and deme management), experiments were also carried out assuming no interactions between variables. This configuration will henceforth be referred to as "univariate MOSES" (uMOSES). Fifty independent runs of MOSES and uMOSES were executed for parity problems with arities 3, 4, and 5, and multiplexer problems with arities 6 and 11. Ten independent runs were executed for 6-parity (due to the high computational cost). For 3-parity, 4-parity, and the multiplexer problems, an optimal solution was found in all runs. For 5-parity, runs were terminated after one million fitness evaluations (96% success rate for MOSES, 86% for uMOSES) and for 6-parity after eight million fitness evaluations (30% success rate for MOSES, 10% for uMOSES).

[3]Results for GP on multiplexers with various parameter settings are available in the literature (cf. [6, 7]) and are qualitatively similar to those here.

Table 2: Computational effort for $p = .99$.

Method	Computational Effort (in thousands)			
	3-parity	4-parity	5-parity	6-parity
uMOSES	6	73	2,403	342,280
EP [2]	29	182	2,100	no solutions
GP [7]	96	384	6,528	no solutions
MOSES	5	72	1,581	100,490

Method	Computational Effort	
	6-multiplexer	11-multiplexer
uMOSES (without *if*)	20,768	377,305
GP (with *if*)	43,600	794,000
GP (without *if*)	65,200	3,128,000
MOSES (without *if*)	14,065	350,276

Comparative computational effort is shown in Tables 2.[4] MOSES achieves the best performance of all techniques listed. Furthermore, MOSES does not "bloat" solutions on this or any other problems studied; the average size of best result for GP on the 11-multiplexer, for example, was 6.7 larger than for MOSES. Even after reduction to ENF, these solutions were still on average 2.8 times larger. Scaling still appears exponential for parity, which is expected to be unavoidable for this problem formulation, on theoretical grounds (cf. [8]). The effort figures for 6-parity should be considered rough approximations; additional runs would need to be carried out to obtain more accurate figures. The relative costs of modeling vs. fitness evaluation vary; for 5-parity for example (with 2^5 test cases), they are approximately equal, while for the 11-multiplexer (with 2^{11}), fitness evaluation strongly dominates.

3.2 ORDER and TRAP

The program space for the JOIN expression mechanism consists of a single binary function, *JOIN*, and set of terminals, $\{X_1, \bar{X}_1, X_2, \bar{X}_2, ..., X_n, \bar{X}_n\}$. As with Boolean formulae, the behavioral (output) space is fixed-length and binary; for JOIN however it grows linearly with the number of terminals rather than exponentially. To evaluate, program trees are parsed from left to right without regard for hierarchical structure. If a terminal X_i appears before its complement $\bar{X}_i$ in the parse, the ith bit of the output is 1, otherwise 0. For example, the $n = 4$ JOIN program

$$JOIN(JOIN(\bar{X}_4, X_2), JOIN(JOIN(X_3, X_3), X_4)$$

expresses X_2 and X_3, producing the output 0110.

How should problems in the JOIN domain be represented by MOSES? In principle, the entire expression mechanism could be dispensed with, and programs transformed into a

[4]A non-zero success rate for 6-parity might be achieved for EP or GP with larger populations and longer runs (EP used a population of 500 and 250 generations [2], while GP used a population of 16,000 and 51 generations [7]).

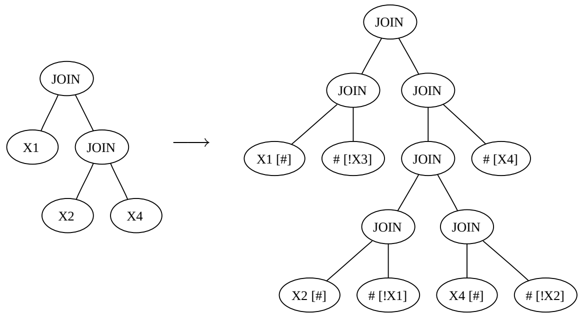


Figure 4: A JOIN exemplar program (left) and corresponding representation built by MOSES (right).

representation isomorphic to their outputs. This would not lead to very interesting results, however – MOSES would essentially reduce to the hBOA, which is already known to solve the bit-string equivalents of ORDER and TRAP. Instead we will "play along", and not exploit our knowledge; no reduction rules will be used.

Unlike junctors in the Boolean formula domain, the output of the JOIN function depends on the order of its arguments. Thus, there must be program transformations that consider inserting new nodes both to the left and to the right of existing nodes. As with formulae, a random sampling procedure is used to construct neighborhoods. Given that the behavioral space for JOIN is exponentially smaller, the number of insertions considered is scaled accordingly (i.e., logarithmic in the arity of the space rather than linear).

To create k neighborhood parameters corresponding to insertions at a particular program location, a random JOIN tree with $k+1$ leaves is generated. One of these leaves is randomly chosen and replaced with a copy of the existing subtree at the target location, which is subsequently replaced with the new tree. In other words, every possible subtree is spliced out of the tree and replaced with a new subtree containing it and k new leaves; $k = [log_2(n)] - 1$ is used herein. Each of the new leaves will be a binary parameter in the representation. For a tree with l leaves, this will lead to $2k(l-1)$ parameters, since trees are binary. Additionally, removal of existing leaves may be considered (l more parameters). A possible representation for an $n = 4$ program is shown in Figure 4; nodes containing parameters have their values in the exemplar listed first, followed the other possible value, '#' means no node, and '!' means complement.

ORDER is the simplest scoring function for the JOIN domain; it sums the number of 1s in the output, and thus has no interactions between variables. TRAP computes a bit-string concatenated deceptive traps function on the output bits (each trap is of order k, and designed to lead greedy local search away from the global optimum). Here traps of order three and signal difference of 0.25 are used, as in [20] (see this reference for a detailed definition and description). MOSES and uMOSES were both executed on ORDER and TRAP for 50 independent runs with varying ns (following [20]), with 100% success rates. The average number of evaluations required to find an optimal solution is shown in Figure 3.2. On ORDER, a univariate model gives better performance as n grows. This effect is statistically significant for $n = 40$; the average number of evaluations with 95% confidence is $24,201 \pm 1153$ for uMOSES, and $27,551 \pm 2010$ for MOSES. For TRAP, however, a univariate model performs *worse* as n grows. This effect is also significant for $n = 36$ $(288,653 \pm 22,314$ vs. $247,497 \pm 17,439)$ and for $n = 42$ $(471,155 \pm 37,237$ vs. $399,335 \pm 31,963)$.

Based on theory this is not surprising; simpler models achieve better performance when complex ones are not needed to correctly decompose the problem, and worse when they are. The results for MOSES and uMOSES on ORDER are comparable to those presented in [20] for GP and ECGP, which both appear to scale polynomially with n; for $n = 40$ the average number of evaluations required for both GP and ECGP is between 20,000 and 30,000. On TRAP, the performance of MOSES appears to be roughly comparable to and perhaps slightly better than that of the ECGP [20], where the average number of evaluations is 300,000-400,000 for $n = 36$, and 400,000-500,000 for $n = 42$.[5]

While not as strong as MOSES, uMOSES performs much better on TRAP than other methods without dependency learning (e.g., GP was not found to solve TRAP with $n > 24$). This is presumably because iterative resampling centered around existing good solutions when new demes are created allowed uMOSES to act as a stochastic hillclimber (which can solve the analogous bit-string TRAP problems better than a GA or a univariate model, but not as well as a dependency-learning approach). This should be testable by running a stochastic hillclimber on TRAP and seeing if it also outperforms GP. Also, MOSES and uMOSES may be applied to larger problems with an increased trap size and decreased signal difference, which is expected to increase the performance gap between MOSES and uMOSES.

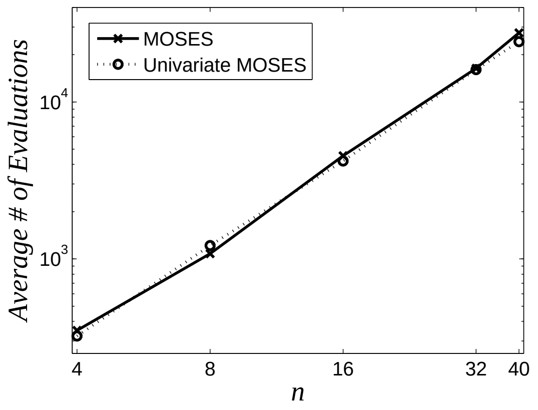
4. CONCLUSIONS

I have shown how two key mechanisms enhance the scalability of program evolution: (1) exploiting semantics (what programs actually mean) to restrict and direct search; and (2) limiting the recombination of programs to occur within bounded subspaces (constructed on the basis of program semantics). These mechanism are novel, and go well beyond previous GP work (e.g., on island models and incorporating expression simplification into evolution).

Existing models of problem difficulty for program evolution are typically based on generalizations of bit-string schema theory, where schema correspond to various syntactically defined fragments of program trees, quantified over the entire program space (see Poli and Langdon [8] for a review of these efforts). There are at present no refinements allowing one to specify, for instance, that *AND(AND(x, y), z)* and *AND(y, AND(x,z))* are in a sense "the same" schema. They are thus expected to give overly pessimistic estimates of problem difficulty in many cases, by not taking program semantics into account (assuming one has an evolutionary algorithm than can exploit semantics).

In order to take semantics and possible local fitness-distance correlation into account, I propose the development of a bipartite model of problem difficulty. Local problem difficulty can be quantified over bounded program subspaces, using methods developed for optimization over fixed-length strings [3]. The theory of global problem difficulty will quantify difficulty in learning *across* bounded program subspaces;

[5]The figures for GP and ECGP are for the number of evaluations required to converge to a score no more than one less than the optimum. Hence, actual comparative performance for MOSES is better than indicated.



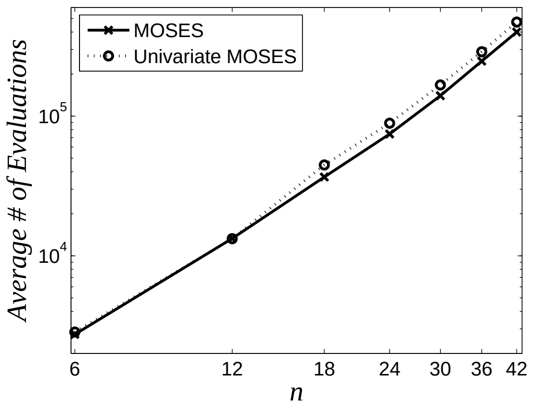


Figure 5: Scalability of MOSES and univariate MOSES or ORDER (left) and TRAP (right).

assuming one has an optimizer that can reliably discover optima in subspaces with a given structure, when and how can one discover optima over the entire program space? The answer will be based on quantifying global program space properties: neutrality, multimodality, etc.

5. REFERENCES

[1] P. A. N. Bosman and E. D. de Jong. Learning probabilistic tree grammars for genetic programming. In *Parallel Problem Solving from Nature*, 2004.

[2] K. Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1997.

[3] D. E. Goldberg. *The Design of Innovation*. Kluwer Academic, 2002.

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

[5] K. E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *IEEE World Conference on Computational Intelligence*, 1994.

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

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

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

[9] M. Looks. *Competent Program Evolution*. PhD thesis, Washington University in St. Louis, 2006.

[10] M. Looks, B. Goertzel, and C. Pennachin. Learning computer programs with the Bayesian optimization algorithm. In *Genetic and Evolutionary Computation Conference*, 2005.

[11] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 2001.

[12] M. Pelikan. *Hierarchical Bayesian Optimization Algorithm*. Springer, 2005.

[13] M. Pelikan and D. E. Goldberg. A hierarchy machine: Learning to optimize from nature and humans. *Complexity*, 2003.

[14] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. Linkage problem, distribution estimation, and Bayesian networks. *Evolutionary Computation*, 2002.

[15] M. Pelikan and T. Lin. Parameter-less hierarchical BOA. In *Genetic and Evolutionary Computation Conference*, 2004.

[16] R. Poli and J. Page. Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic Programming and Evolvable Machines*, 2000.

[17] E. N. Regolin and A. R. T. Pozo. Bayesian automatic programming. In *European Conference on Genetic Programming*, 2005.

[18] R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 1997.

[19] R. P. Salustowicz and J. Schmidhuber. H-pipe: Facilitating hierarchical program evolution through skip nodes. Technical report, IDSIA, 1998.

[20] K. Sastry and D. E. Goldberg. Probabilistic model building and competent genetic programming. In R. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practice*. Kluwer Academic, 2003.

[21] Y. Shan, R. I. McKay, H. A. Abbass, and D. Essam. Program evolution with explicit learning: a new framework for program automatic synthesis. In *Congress on Evolutionary Computation*, 2003.

[22] Y. Shan, R. I. McKay, R. Baxter, H. Abbass, D. Essam, and H. X. Nguyen. Grammar model-based program evolution. In *Congress on Evolutionary Computation*, 2004.

[23] 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.

[24] K. Yanai and H. Iba. Estimation of distribution programming based on Bayesian network. In *Congress on Evolutionary Computation*, 2003.

[25] K. Yanai and H. Iba. Program evolution by integrating EDP and GP. In *Genetic and Evolutionary Computation Conference*, 2004.