Program Evolution for General Intelligence

Moshe LOOKS (moshe@metacog.org)
Washington University in St. Louis,
Department of Computer Science
and
Science Applications International Corporation (SAIC),
Integrated Intelligence Solutions Operation
and
Novamente LLC

Abstract. A program evolution component is proposed for integrative artificial general intelligence. The system's deployment is intended to be comparable, on Marr's level of computational theory, to evolutionary mechanisms in human thought. The challenges of program evolution are described, along with the requirements for a program evolution system to be competent - solving hard problems quickly, accurately, and reliably. Meta-optimizing semantic evolutionary search (MOSES) is proposed to fulfill these requirements.

1. Background and Motivation

"At every step the design logic of brains is a Darwinian logic: overproduction, variation, competition, selection … it should not come as a surprise that this same logic is also the basis for the normal millisecond-by-millisecond information processing that continues to adapt neural software to the world." *Terrence Deacon* [1]

In David Marr's seminal decomposition, any information-processing system may be understood at three nearly independent levels: (1) *computational theory*, a description of the problems the system attempts to solve; (2) *representations and algorithms*; and (3) *implementation*, the physical instantiation of the system's representations and algorithms [2]. What might the subsystems of human cognition look like on the level of computational theory?[1]

I propose that evolutionary learning be considered as one of these subsystems. Major subfields of both cognitive science and AI are concerned with evolutionary learning processes. One motivator for this concern is an attempt in both fields to augment purely local techniques such as Hebbian (associative) learning with more global methods, which try to make large leaps to find answers far removed from existing knowledge. This is a form of evolutionary learning [4], which Edelman [5] has presented as "Neural Darwinism", and Calvin and Bickerton [6, 7] as the notion of mind as a "Darwin Machine".

It is known that the immune system adapts via a form of evolutionary learning, and Edelman [5] has proposed that the brain does so as well, evolving new "neuronal maps", patterns of neural connection and activity spanning numerous neuronal clusters that are highly "fit" in the sense of contributing usefully to system goals. Edelman and

[1] Parts of this section are adapted from [3], which elaborates on the utility of emulating human cognition on the level of computational theory.

his colleagues have run computer simulations showing that Hebbian-like neuronal dynamics, if properly tuned, can give rise to evolution-like dynamics on the neuronal map level ("neuronal group selection").

Recently Deacon [1] has articulated ways in which, during neurodevelopment, difference computations compete with each other (e.g., to determine which brain regions are responsible for motor control). More generally, he posits a kind of continuous flux as control shifts between competing brain regions, again, based on high-level "cognitive demand" [1, p. 457]. Similarly, Calvin and Bickerton [6, 7] have given plausible neural mechanisms ("Darwin Machines") for synthesizing short "programs". These programs are for tasks such as rock throwing and sentence generation, which are represented as coherent firing patterns in the cerebral cortex. A population of such patterns, competing for neurocomputational territory, replicates with variations, under selection pressure to conform to background knowledge and constraints.

In summary, a system is needed that can recombine existing solutions in a non-local synthetic fashion, learning nested and sequential structures, and incorporate background knowledge (e.g. previously learned routines). I propose a particular kind of *program evolution* to satisfy these goals.

1.1. Evolutionary Learning

There is a long history in AI of applying evolution-derived methods to practical problem-solving; the original genetic algorithm [4], initially a theoretical model, has been adapted successfully to a wide variety of applications [8]. The methodology, similar to the Darwin Machines mentioned above, is applied as follows: (1) generate a random population of solutions to a problem; (2) evaluate the solutions in the population using a predefined *scoring function*; (3) select solutions from the population proportionate to their scores, and recombine/mutate them to generate a new population; (4) go to step 2. Holland's paradigm has been adapted from the case of fixed-length strings to the evolution of variable-sized and shaped trees (typically Lisp symbolic expressions), which in principle can represent arbitrary computer programs [9, 10].

Recently, replacements-for/extensions-of the genetic algorithm have been developed (for fixed-length strings) which may be described as *estimation-of-distribution* algorithms (see [11] for an overview). These methods, which outperform genetic algorithms and related techniques across a range of problems, maintain centralized probabilistic models of the population learned with sophisticated datamining techniques. One of the most powerful of these methods is the *Bayesian optimization algorithm* (BOA) [12].

The basic steps of the BOA are: (1) generate a random population of solutions to a problem; (2) evaluate the solutions in the population using a predefined *scoring function*; (3) from the promising solutions in the population, learn a generative model; (4) create new solutions using the model, and merge them into the existing population; (4) go to step 2. The neurological implausibility of this sort of algorithm is readily apparent – yet recall that we are attempting to emulate human cognition on the level of computational theory, not implementation, or even representations and algorithms.

Figure 1: The structure of OneMax, a paradigmatic *separable* optimization problem.

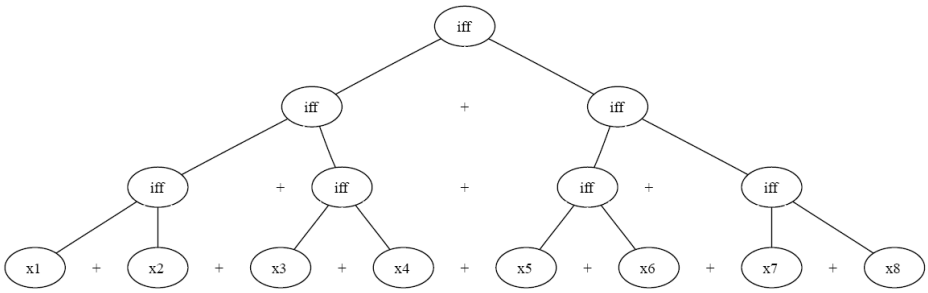


Figure 2: The structure of hierarchical if-and-only-if [16], a paradigmatic nearly decomposable optimization problem.

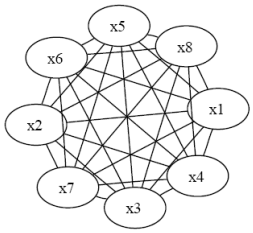


Figure 3: The structure of an intractable optimization problem, such as a uniform random scoring function, where changing the assignment of any variable results in a chaotic change in the overall score.

Fundamentally, the BOA and its ilk (the *competent* adaptive optimization algorithms) differ from classic selectorecombinative search by attempting to dynamically learn a problem decomposition, in terms of the variables that have been pre-specified. The BOA represents this decomposition as a Bayesian network (directed acyclic graph with the variables as nodes, and an edge from *x* to *y* indicating that *y* is probabilistically dependent on *x*). An extension, the hierarchical Bayesian optimization algorithm (hBOA) [12], uses a Bayesian network with local structure [13] to more accurately represent hierarchical dependency relationships. The BOA and hBOA are scalable and robust to noise across the range of nearly decomposable functions [12, 13]. They are also effective, empirically, on real-world problems with unknown decompositions, which may or may not be effectively representable by the algorithms; robust, high-quality results have been obtained for Ising spin glasses and MaxSAT [14], as well as a real-world telecommunication problem [15].

1.2.Representation-Building

"A representation is a formal system making explicit certain entities or types of information, together with a specification of how the system does this." *David Marr* [2]

In an ideally encoded optimization problem, all prespecified variables would exhibit complete *separability*, and could be optimized independently (Figure 1). Problems with hierarchical dependency structure (Figure 2) cannot be encoded this way, but are still tractable by dynamically learning the problem decomposition (as the BOA and hBOA do). For complex problems with interacting subcomponents, finding an accurate problem decomposition is often tantamount to finding a solution. In an idealized run of a competent optimization algorithm, the problem decomposition evolves along with the set of solutions being considered, with parallel convergence to the correct decomposition and the global solution optima. However, this is certainly contingent on the existence of some compact[2] and reasonably correct decomposition in the space (of decompositions, not solutions) being searched.

Difficulty arises when no such decomposition exists (Figure 3), or when a more effective decomposition exists that cannot be formulated as a probabilistic model over representational parameters. Accordingly, one may extend current approaches via either: (1) a more general modeling language for expressing problem decompositions; or (2) *additional mechanisms* that modify the representations on which modeling operates (introducing additional inductive bias). I focus here on the latter – the former would appear to require qualitatively more computational capacity than will be available in the near future. If one ignores this constraint, such a "universal" approach to general problem-solving is indeed possible [17, 18].

I refer to these additional mechanisms as representation-building because they serve the same purpose as the pre-representational mechanisms employed (typically by humans) in setting up an optimization problem – to present an optimization algorithm with the salient parameters needed to build effective problem decompositions and vary solutions along meaningful dimensions.

A secondary source of human effort in encoding problems to be solved is crafting an effective scoring function. The primary focus of this paper is issues surrounding the representation of solutions, rather than how solutions are scored, which may be more fruitfully addressed, I believe, in the context of integrative systems (cf. [19]).

1.3.Program Learning

An optimization problem may be defined as follows: a solution space $\boldsymbol{S}$ is specified, together with some scoring function on solutions, where "solving the problem" corresponds to discovering a solution in $\boldsymbol{S}$ with a sufficiently high score. Let's define program learning as follows: given a program space $\boldsymbol{P}$, a behavior space $\boldsymbol{B}$, an execution function $\boldsymbol{exec : P \rightarrow B}$, and a scoring function on *behaviors*, "solving the problem" corresponds to discovering a program $\boldsymbol{p}$ in $\boldsymbol{P}$ whose corresponding behavior, $\boldsymbol{exec(p)}$, has a sufficiently high score.

This extended formalism can of course be entirely vacuous – the behavior space could be identical to the program space, and the execution function simply identity, allowing any optimization problem to be cast as a problem of program learning. The utility of this specification arises when we make interesting assumptions regarding the program and behavior spaces, and the execution and scoring functions (the additional inductive bias mentioned above):

[2] The decomposition must be compact because in practice only a fairly small sampling of solutions may be evaluated (relative to the size of the total space) at a time, and the search mechanism for exploring decomposition-space is greedy and local. This is in also accordance with the general notion of learning corresponding to compression [17].

- **Open-endedness** – ***P*** has a natural "program size" measure – programs may be enumerated from smallest to largest, and there is no obvious problem-independent upper bound on program size.
- **Over-representation** – ***exec*** often maps many programs to the same behavior.
- **Compositional hierarchy** – programs themselves have an intrinsic hierarchical organization, and may contain subprograms which are themselves members of ***P*** or some related program space. This provides a natural family of distance measures on programs, in terms of the the number and type of compositions / decompositions needed to transform one program into another (i.e., edit distance).
- **Chaotic Execution** – very similar programs (as conceptualized in the previous item) may have very different behaviors.

Precise mathematical definitions could be given for all of these properties but would provide little insight – it is more instructive to simply note their ubiquity in symbolic representations; human programming languages (LISP, C, etc.), Boolean and real-valued formulae, pattern-matching systems, automata, and many more. The crux of this line of thought is that the combination of these four factors conspires to *scramble* scoring functions – even if the mapping from behaviors to scores is separable or nearly decomposable, the complex[3] program space and chaotic execution function will often quickly lead to intractability as problem size grows. These properties are not superficial inconveniences that can be circumvented by some particularly clever encoding. On the contrary, they are the essential characteristics that give programs the power to compress knowledge and generalize correctly, in contrast to flat, inert representations such as lookup tables (see Baum [20] for a full treatment of this line of argument).

The consequences of this particular kind of complexity, together with the fact that most program spaces of interest are combinatorially very large, might lead one to believe that competent program evolution is impossible. Not so: program learning tasks of interest have a compact structure[4] – they are not "needle in haystack" problems or uncorrelated fitness landscapes, although they can certainly be encoded as such. The most one can definitively state is that algorithm ***foo***, methodology ***bar,*** or representation ***baz*** is unsuitable for expressing and exploiting the regularities that occur across interesting program spaces. Some of these regularities are as follows:

- Simplicity prior – our prior assigns greater probability mass to smaller programs.
- Simplicity preference – given two programs mapping to the same behavior, we prefer the smaller program (this can be seen as a secondary scoring function).
- Behavioral decomposability – the mapping between behaviors and scores is separable or nearly decomposable. Relatedly, scores are more than scalars – there is a partial ordering corresponding to behavioral dominance, where one behavior dominates another if it exhibits a strict superset of the latter's desideratum, according to the scoring function.[5] This partial order will never contradict the total ordering of scalar scores.

[3] Here "complex" means open-ended, over-representing, and hierarchical.
[4] Otherwise, humans could not write programs significantly more compact than lookup tables.
[5] For example, in supervised classification one rule dominates another if it correctly classifies all of the items that second rule classifies correctly, as well as some which the second rule gets wrong.

- White box execution – the mechanism of program execution is known a priori, and remains constant across many problems.

How these regularities may be exploited via representation-building, in conjunction with the probabilistic modeling that takes place in a competent optimization algorithm such as the BOA or hBOA, will be discussed in the next section. Another fundamental regularity of great interest for artificial general intelligence, but beyond the scope of this paper, is patterns across related problems that may be solvable with similar programs (e.g., involving common modules).

2. Competent Program Evolution

In this section I present *meta-optimizing semantic evolutionary search* (MOSES), a system for competent program evolution. Based on the viewpoint developed in the previous section, MOSES is designed around the central and unprecedented capability of competent optimization algorithms such as the hBOA, to generate new solutions that simultaneously combine sets of promising assignments from previous solutions according to a dynamically learned problem decomposition. The novel aspects of MOSES described herein are built around this core to exploit the unique properties of program learning problems. This facilitates effective problem decomposition (and thus competent optimization).

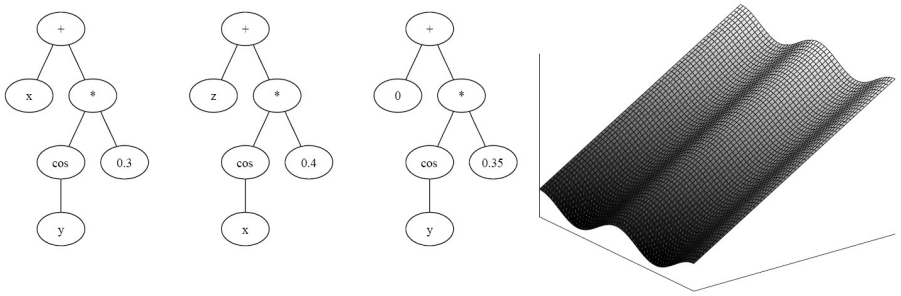


Figure 4: Three simple program trees encoding real-valued expressions, with identical structures and node-by-node semantics (left), and the corresponding behavioral pattern (right); the horizontal axes correspond to variation of arguments (***x***, ***y***, and/or ***z***), with the vertical axis showing the corresponding variation of output.

2.1.Statics

The basic goal of MOSES is to exploit the regularities in program spaces outlined in the previous section, most critically *behavioral decomposability* and *white box execution*, to dynamically construct representations that limit and transform the program space being searched into a relevant subspace with a compact problem decomposition. These representations will evolve as the search progresses.

2.1.1.An Example

Let's start with an easy example. What knobs (meaningful parameters to vary) exist for the family of programs depicted in Figure 4 on the left? We can assume, in accordance with the principle of white box execution, that all symbols have their standard mathematical interpretations, and that ***x***, ***y***, and ***z*** are real-valued variables.

In this case, all three programs correspond to variations on the behavior represented graphically on the right in the figure. Based on the principle of behavioral decomposability, good knobs should express plausible evolutionary variation and recombination of features in behavior space, regardless of the nature of the corresponding changes in program space. It's worth repeating once more that this goal cannot be meaningfully addressed on a syntactic level - it requires us to leverage background knowledge of what the symbols in our vocabulary (***cos***, +, ***0.35***, etc.) actually *mean*.

A good set of knobs will also be *orthogonal*. Since we are searching through the space of combinations of knob settings (not a single change at a time, but a set of changes), any knob whose effects are equivalent to another knob or combination of knobs is undesirable.[6] Correspondingly, our set of knobs should *span* all of the given programs (i.e., be able to represent them as various knob settings).

A small *basis* for these programs could be the 3-dimensional parameter space, $x_1 \in \{x, z, 0\}$ (left argument of the root node), $x_2 \in \{y, x\}$ (argument of ***cos***), and $x_3 \in [0.3, 0.4]$ (multiplier for the ***cos***-expression). However, this is a very limiting view, and overly tied to the particulars of how these three programs happen to be encoded. Considering the space *behaviorally* (right of Figure 4), a number of additional knobs can be imagined which might be turned in meaningful ways, such as:

1. numerical constants modifying the phase/frequency of the cosine expression,
2. considering some weighted average of x and y instead of one or the other,
3. multiplying the entire expression by a constant,
4. adjusting the relative weightings of the two arguments to +.

2.1.2. Syntax and Semantics

This kind of representation-building calls for a correspondence between syntactic and semantic variation. The properties of program spaces that make this difficult are over-representation and chaotic execution, which lead to *non-orthogonality*, *oversampling of distant behaviors*, and *undersampling of nearby behaviors*, all of which can directly impede effective program evolution.

Non-orthogonality is caused by over-representation. For example, based on the properties of commutativity and associativity, $a_1 + a_2 + \ldots + a_n$ may be expressed in exponentially many different ways, if + is treated as a non-commutative and non-associative binary operator. Similarly, operations such as addition of zero and multiplication by one have no effect, the successive addition of two constants is equivalent to the addition of their sum, etc. These effects are not quirks of real-valued expressions; similar redundancies appear in Boolean formulae (***x AND x*** ↔ ***x***), list

[6] First because this will increase the number of samples needed to effectively model the structure of knob-space, and second because this modeling will typically be quadratic with the number of knobs, at least for the BOA or hBOA [12, 13].

manipulation (***cdr(cons(x, L))*** ↔ ***L***), and conditionals (***if x then y else z*** ↔ ***if NOT x then z else y***).

Without the ability to exploit these identities, we are forced to work in a greatly expanded space which represents equivalent expression in many different ways, and will therefore be very far from orthogonality. Completely eliminating redundancy is infeasible, and typically at least NP-hard (in the domain of Boolean formulae it is reducible to the satisfiability problem, for instance), but one can go quite far with a heuristic approach.

Oversampling of distant behaviors is caused directly by chaotic execution, as well as a somewhat subtle effect of over-representation, which can lead to simpler programs being heavily oversampled. Simplicity is defined relative to a given program space in terms of minimal length, the number of symbols in the shortest program that produces the same behavior.

Undersampling of nearby behaviors is the flip side of the oversampling of distant behaviors. As we have seen, syntactically diverse programs can have the same behavior; this can be attributed to redundancy, as well as non-redundant programs that simply compute the same result by different means. For example, ***3*x*** can also be computed as ***x+x+x***; the first version uses less symbols, but neither contains any obvious "bloat" such as addition of zero or multiplication by one. Note however that the nearby behavior of ***3.1*x***, is syntactically close to the former, and relatively far from the latter. The converse is the case for the behavior of ***2*x+y***. In a sense, these two expressions can be said to exemplify differing organizational principles, or points of view, on the underlying function.

Differing organizational principles lead to different biases in sampling nearby behaviors. A superior organizational principle (one leading to higher-scoring syntactically nearby programs for a particular problem) might be considered a *metaptation* (adaptation at the second tier), in the terminology of King [21]. Since equivalent programs organized according to different principles will have identical scores, some methodology beyond selection for high scores must be employed to search for good organizational principles. Thus, the resolution of undersampling of nearby behaviors revolves around the management of *neutrality* in search, a complex topic beyond the scope of this paper.

These three properties of program spaces greatly affect the performance of evolutionary methods based solely on syntactic variation and recombination operators, such as local search or genetic programming. In fact, when quantified in terms of various fitness-distance correlation measures, they can be effective predictors of algorithm performance, although they are of course not the whole story [22]. A semantic search procedure will address these concerns in terms of the underlying behavioral effects of and interactions between a language's basic operators; the general scheme for doing so in MOSES is the topic of the next subsection.

2.2. Neighborhoods and Normal Forms

The procedure MOSES uses to construct a set of knobs for a given program (or family of structurally related programs) is based on three conceptual steps: *reduction to normal form*, *neighborhood enumeration*, and *neighborhood reduction*.

Reduction to normal form - in this step, redundancy is heuristically eliminated by reducing programs to a *normal form*. Typically, this will be via the iterative application of a series of local rewrite rules (e.g., $\forall x,\ x+0 \rightarrow x$), until the target program no longer changes. Note that the well-known conjunctive and disjunctive

normal forms for Boolean formulae are generally unsuitable for this purpose; they destroy the hierarchical structure of formulae, and dramatically limit the range of behaviors (in this case Boolean functions) that can be expressed compactly. Rather, *hierarchical normal forms* for programs are required.

Neighborhood enumeration - in this step, a set of possible atomic *perturbations* is generated for all programs under consideration (the overall perturbation set will be the union of these). The goal is to heuristically generate new programs that correspond to behaviorally nearby variations on the source program, in such a way that arbitrary sets of perturbations may be *composed* combinatorially to generate novel valid programs.

Neighborhood reduction - in this step, redundant perturbations are heuristically culled to reach a more orthogonal set. A straightforward way to do this is to exploit the reduction to normal form outlined above; if multiple knobs lead to the same normal forms program, only one of them is actually needed. Additionally, note that 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 twiddling some knob significantly decreases its size, it can be assumed to be a source of oversampling, and hence eliminated from consideration. 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 knobs that individually produce small changes).

At the end of this process, we will be left with a set of knobs defining a subspace of programs centered around a particular region in program space and heuristically centered around the corresponding region in behavior space as well. This is part of the *meta* aspect of MOSES, which seeks not to evaluate variations on existing programs itself, but to construct parameterized program subspaces (representations) containing meaningful variations, guided by background knowledge. These representations are used as search spaces within which an optimization algorithm can be applied.

2.3. Dynamics

As described above, the representation-building component of MOSES constructs a parameterized representation of a particular *region* of program space, centered around a single program of family of closely related programs. This is consonant with the line of thought developed above, that a representation constructed across an arbitrary region of program space (e.g., all programs containing less than ***n*** symbols), or spanning an arbitrary collection of unrelated programs, is unlikely to produce a meaningful parameterization (i.e., one leading to a compact problem decomposition).

A sample of programs within a region derived from representation-building together with the corresponding set of knobs will be referred to herein as a *deme*;[7] a set of demes (together spanning an arbitrary area within program space in a patchwork fashion) will be referred to as a *metapopulation*.[8] MOSES operates on a metapopulation, adaptively creating, removing, and allocating optimization effort to various demes. Deme management is the second fundamental *meta* aspect of MOSES, after (and above) representation-building; it essentially corresponds to the problem of effectively allocating computational resources to competing regions, and hence to competing programmatic organizational- representational schemes.

[7] A term borrowed from biology, referring to a somewhat isolated local population of a species.

[8] Another term borrowed from biology, referring to a group of somewhat separate populations (the demes) that nonetheless interact.

2.4. Algorithmic Sketch

The salient aspects of programs and program learning lead to requirements for competent program evolution that can be addressed via a representation-building process such as the one shown above, combined with effective deme management. The following sketch of MOSES presents a simple control flow that dynamically integrates these processes into an overall program evolution procedure:

1. Construct an initial set of knobs based on some prior (e.g., based on an empty program) and use it to generate an initial random sampling of programs. Add this deme to the metapopulation.

2. Select a deme from the metapopulation and 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.
   b) Considering the promising programs as collections of knob settings, generate new collections of knob settings by applying some (competent) optimization algorithm.
   c) Convert the new collections of knob settings into their corresponding programs, reduce the programs to normal form, evaluate their scores, and integrate them into the deme's sample, replacing less promising programs.

3. For each new program that meets the criterion for creating a new deme, if any:
   a) Construct a new set of knobs (via representation-building) to define a region 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.

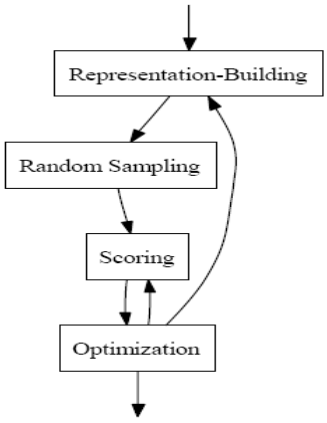


Figure 5: The top-level architectural components of MOSES, with directed edges indicating the flow of information and program control.

→###

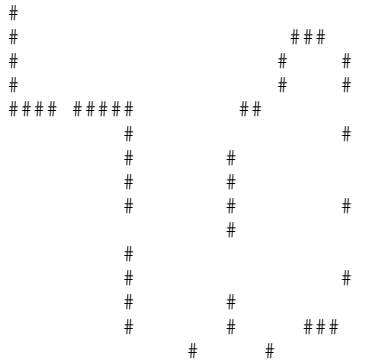


Figure 6: The top half of the Santa Fe trail. The ant's starting position and orientation is denoted by the →, and pieces of food by #s. The complete trail may be found in [10, p. 55].

The criterion for creating a new deme is *behavioral non-dominance* (programs which are not dominated by the exemplars of any existing demes are used as exemplars to create new demes), which can be defined in a domain-specific fashion. As a default, the scoring function may be used to induce dominance, in which case the set of exemplar programs for demes corresponds to the set of top-scoring programs.

2.5.Architecture

The preceding algorithmic sketch of MOSES leads to the top-level architecture depicted in Figure 5. Of the four top-level components, only the scoring function is problem-specific. The representation-building process is domain-specific, while the random sampling methodology and optimization algorithm are domain-general. There is of course the possibility of improving performance by incorporating domain and/or problem-specific bias into random sampling and optimization as well.

2.6. MOSES in Action

Let's go through all of the steps that are needed to apply MOSES to a small problem, the artificial ant on the Santa Fe trail [10], and describe the search process. The artificial ant domain is a two-dimensional grid landscape where each cell may or may not contain a piece of food. The artificial ant has a location (a cell) and orientation (facing up, down, left, or right), and navigates the landscape via a primitive sensor, which detects whether or not there is food in the cell that the ant is facing, and primitive actuators *move* (take a single step forward), *right* (rotate 90 degrees clockwise), and *left* (rotate 90 degrees counter-clockwise). The Santa Fe trail problem is a particular 32 x 32 toroidal grid with food scattered on it (Figure 6), and a fitness function counting the number of unique pieces of food the ant eats (by entering the cell containing the food) within 600 steps (movement and 90 degree rotations are considered single steps).

Programs are composed of the primitive actions taking no arguments, a conditional (*if-food-ahead*),[9] which takes two arguments and evaluates one or the other based on whether or not there is food ahead, and *progn*, which takes a variable number of

[9] This formulation is equivalent to using a general three-argument *if-then-else* statement with a predicate as the first argument, as there is only a single predicate (*food-ahead*) for the ant problem.

arguments and sequentially evaluates all of them from left to right. To score a program, it is evaluated continuously until 600 time steps have passed, or all of the food is eaten (whichever comes first). Thus for example, the program *if-food-ahead(m, r)* moves forward as long as there is food ahead of it, at which point it rotates clockwise until food is again spotted. It's can successfully navigate the first two turns of the Santa Fe trail, but cannot cross "gaps" in the trail, giving it a final score of 11.

The first step in applying MOSES is to decide what our reduction rules should look like. This program space has several clear sources of redundancy leading to over-representation that we can eliminate, leading to the following reduction rules:

1. Any sequence of rotations may be reduced to either a left rotation, a right rotation, or a reversal, for example:
   progn(left, left, left)
   reduces to
   right
2. Any *if-food-ahead* statement which is the child of an *if-food-ahead* statement may be eliminated, as one of its branches is clearly irrelevant, for example:
   if-food-ahead(m, if-food-ahead(l, r))
   reduces to
   if-food-ahead(m, r)
3. Any *progn* statement which is the child of a *progn* statement may be eliminated and replaced by its children, for example:
   progn(progn(left, move), move)
   reduces to
   progn(left, move, move)

The representation language for the ant problem is simple enough that these are the only three rules needed – in principle there could be many more. The first rule may be seen as a consequence of general domain-knowledge pertaining to rotation. The second and third rules are fully general simplification rules based on the semantics of *if-then-else* statements and associative functions (such as *progn*), respectively.

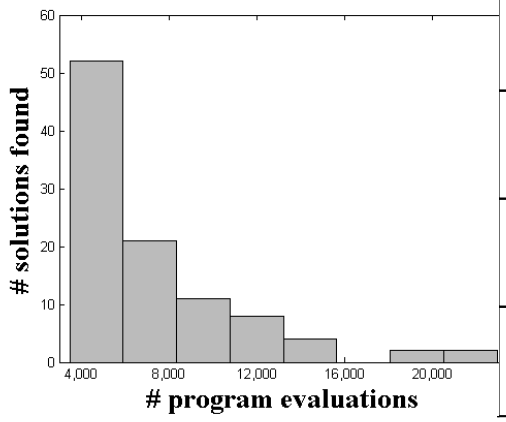
These rules allow us to naturally parameterize a knob space corresponding to a given program (note that the arguments to the *progn* and *if-food-ahead* functions will be recursively reduced and parameterized according to the same procedure). Rotations will correspond to knobs with four possibilities (*left, right, reversal, no rotation)*. Movement commands will correspond to knobs with two possibilities (*move, no movement)*. There is also the possibility of introducing a new command in between, before, or after, existing commands. Some convention (a "canonical form") for our space is needed to determine how the knobs for new commands will be introduced. A representation consists of a rotation knob, followed by a conditional knob, followed by a movement knob, followed by a rotation knob, etc.[10]

The structure of the space (how large and what shape) and default knob values will be determined by the "exemplar" program used to construct it. The default values are used to bias the initial sampling to focus around the exemplar: all of the *n* neighbors of the exemplar are first added to the sample, followed by a random selection of *n* programs at a distance of two from the exemplar, *n* programs at a distance of three, etc., until the entire sample is filled. Note that the hBOA can of course effectively recombine this sample to generate novel programs at any distance from the exemplar.

[10] That there be some fixed ordering on the knobs is important, so that two rotation knobs are not placed next to each other (as this would introduce redundancy). Based on some preliminary test, the precise ordering chosen (rotation, conditional, movement) does not appear to be critical.

The empty program *progn* (which can be used as the initial exemplar for MOSES), for example, leads to the following prototype subspace:

```
progn(
        rotate? [default no rotation],
        if-food-ahead(
                progn(
                        rotate? [default no rotation],
                        move? [default no movement]),
                progn(
                        rotate? [default no rotation],
                        move? [default no movement])),
        move? [default no movement])
```



Technique	Computational Effort
Genetic Programming [10]	450,000 evaluations
Evolutionary Programming [24]	136,000 evaluations
MOSES	23,000 evaluations

Figure 7: On the left, histogram of the number of global optima found after a given number of program evaluations for 100 runs of MOSES on the artificial ant problem (each run is counted once for the first global optimum reached). On the right, computational effort required to find an optimal solution for various techniques with p=.99 (for MOSES p=1, since an optimal solution was found in all runs).

There are six parameters here, three which are quaternary, and three which are binary. So the program *progn(left,if-food-ahead(move, left))* would be encoded in the space as [*left, no rotation, move, left, no movement, no movement*], with knobs ordered according to a pre-order left-to-right traversal of the program's parse tree (this is merely for exposition; the ordering of the parameters has no effect on MOSES). For an

exemplar program already containing an *if-food-ahead* statement, nested conditionals would be considered.

A space with six parameters in it is small enough that MOSES can reliably find the optimum (the program *progn(right, if-food-ahead(progn(),left), move)*), with a very small population. After no further improvements have been made in the search for a specified number of generations (calculated based on the size of the space based on a model derived from [23] that is general to the hBOA, and not at all tuned for the artificial ant problem), a new representation is constructed centered around this program.[11] Additional knobs are introduced "in between" all existing ones (e.g., an optional move in between the first rotation and the first conditional), and possible nested conditionals are considered (a nested conditional occurring in a sequence *after* some other action has been taken is not redundant). The resulting space has 39 knobs, still quite tractable for hBOA, which typically finds a global optimum within a few generations. If the optimum were not to be found, MOSES would construct a new (possibly larger or smaller) representation, centered around the best program that *was* found, and the process would repeat.

The artificial ant problem is well-studied, with published benchmark results available for genetic programming [10] as well as evolutionary programming based solely on mutation [24] (i.e., a form of population-based stochastic hill climbing). Furthermore, an extensive analysis of the search space has been carried out by Langdon and Poli [25], with the authors concluding:

- The problem is "deceptive at all levels", meaning that the partial solutions that must be recombined to solve the problem to optimality have lower average fitness than the partial solutions that lead to inferior local optima.
- The search space contains many symmetries (e.g., between left and right rotations),
- There is an unusually high density of global optima in the space (relative to other common test problems); even though current evolutionary methods can solve the problem, they are not significantly more effective (in terms of the number of program evaluations require) than random sample.
- "If real program spaces have the above characteristics (we expect them to do so but be still worse) then it is important to be able to demonstrate scalable techniques on such problem spaces".

A review of scalability results for MOSES across a range of problems is beyond the scope of this paper (see [26]), but results for the artificial ant problem may be given briefly to indicate the magnitude of improvement that may be experienced. Koza [10] reports on a set of 148 runs of genetic programming with a population size of 500 which had a 16% success rate after 51 generations when the runs were terminated (a total of 25,500 program evaluations per run). The minimal "computational effort" needed to achieve success with 99% probability was attained by processing through generation 14 was 450,000 (based on parallel independent runs). Chellapilla [24] reports 47 out of 50 successful runs with a minimal computational effort (again, for success with 99% probability) of 136,000 for his stochastic hill climbing method.

One hundred runs of MOSES were executed. Beyond the domain knowledge embodied in the reduction and knob construction procedure, the only parameter that needed to be set was the population scaling factor, which was set to 30 (MOSES

[11] MOSES reduces the exemplar program to normal form before constructing the representation; in this particular case however, no transformations are needed. Similarly, in general neighborhood reduction would be used to eliminate any extraneous knobs (based on domain-specific heuristics). For the ant domain however no such reductions are necessary.

automatically adjusts to generate a larger population as the size of the representation grows, with the base case determined by this factor). Based on these "factory" settings, MOSES found optimal solutions on every run out of 100 trials, within a maximum of 23,000 program evaluations (the computational effort figure corresponding to 100% success). The average number of program evaluation required was 6952, with 95% confidence intervals of ±856 evaluations.

Why does MOSES outperform other techniques? One factor to consider first is that the language programs are evolved in is slightly more expressive than that used for the other techniques; specifically, a *progn* is allowed to have no children (if all of its possible children are "turned off"), leading to the possibility of *if-food-ahead* statements which do nothing if food is present (or not present). Indeed, many of the smallest solutions found by MOSES exploit this feature. This can be tested by inserting a "do nothing" operation into the terminal set for genetic programming (for example). Indeed, this reduces the computational effort to 272,000; an interesting effect, but still over an order of magnitude short of the results obtained with MOSES (the success rate after 50 generations is still only 20%).

Another possibility is that the reductions in the search space via simplification of programs alone are responsible. However, the results past attempts at introducing program simplification into genetic programming systems [27, 28] have been mixed; although the system may be sped up (because programs are smaller), there have been no dramatic improvement in results noted. To be fair, these results have been primarily focused on the symbolic regression domain; I am not aware of any results for the artificial ant problem.

The final contributor to consider is the sampling mechanism (knowledge-driven knob-creation followed by probabilistic model-building). We can test to what extent model-building contributes to the bottom line by simply disabling it and assuming probabilistic independence between all knobs. The result here is of interest because model-building can be quite expensive ($O(n^2N)$ per generation, where n is the problem size and N is the population size[12]). In 50 independent runs of MOSES without model-building, a global optimum was still discovered in all runs. However, the variance in the number of evaluations required was much higher (in two cases over 100,000 evaluations were needed). The new average was 26,355 evaluations to reach an optimum (about 3.5 times more than required with model-building). The contribution of model-building to the performance of MOSES is expected to be even greater for more difficult problems.

Applying MOSES without model-building (i.e., a model assuming no interactions between variables) is a way to test the combination of representation-building with an approach resembling the probabilistic incremental program learning (PIPE) [29] algorithm, which learns programs based on a probabilistic model without any interactions. PIPE has no been shown to provide results competitive with genetic programming on a number of problems (regression, agent control, etc.).

It is additionally possible to look inside the models that the hBOA constructs (based on the empirical statistics of successful programs) to see what sorts of linkages between knobs are being learned.[13] For the 6-knob model given above for instance an analysis the linkages learned shows that the three most common pairwise dependencies uncovered, occurring in over 90% of the models across 100 runs, are between the rotation knobs. No other individual dependencies occurred in more than 32% of the

[12] The fact that reduction to normal tends to reduce the problem size is another synergy between it and the application of probabilistic model-building.

[13] There is in fact even more information available in the hBOA models concerning hierarchy and direction of dependence, but this is difficult to analyze.

models. This preliminary finding is quite significant given Landgon and Poli's findings on symmetry, and their observation [25] that "[t]hese symmetries lead to essentially the same solutions appearing to be the opposite of each other. E.g. either a pair of Right or pair of Left terminals at a particular location may be important."

In summary, all of the components of MOSES appear to mesh together to provide superior performance, although further experimentation and analysis across a range of problems is clearly needed.

2.7.Discussion

The overall MOSES design as described herein is unique. However, it is instructive at this point to compare its two primary facets (representation-building and deme management) to related work in evolutionary computation.

Rosca's *adaptive representation* architecture [30] is an approach to program evolution which also alternates between separate representation-building and optimization stages. It is based on Koza's genetic programming [10], and modifies the representation based on a *syntactic* analysis driven by the scoring function, as well as a modularity bias. The representation-building that takes place consists of introducing new compound operators, and hence modifying the implicit distance function in tree-space. This modification is uniform, in the sense that the new operators can be placed in any context, without regard for semantics.

In contrast to Rosca's work and other approaches to representation-building such as Koza's *automatically defined functions* [31], MOSES explicitly addresses on the underlying (semantic) structure of program space independently of the search for any kind of modularity or problem decomposition. This preliminary stage critically changes neighborhood structures (syntactic similarity) and other aggregate properties of programs.

Regarding deme management, the embedding of an evolutionary algorithm within a superordinate procedure maintaining a metapopulation is most commonly associated with "island model" architectures [8]. One of the motivations articulated for using island models has been to allow distinct islands to (usually implicitly) explore different regions of the search space, as MOSES does explicitly. MOSES can thus be seen as a very particular kind of island model architecture, where programs never migrate between islands (demes), and islands are created and destroyed dynamically as the search progresses.

In MOSES, optimization does not operate directly on program space, but rather on a subspace defined by the representation-building process. This subspace may be considered as being defined by a sort of template assigning values to some of the underlying dimensions (e.g., it restricts the size and shape of any resulting trees). The messy genetic algorithm [32], an early competent optimization algorithm, uses a similar mechanism - a common "competitive template" is used to evaluate candidate solutions to the optimization problem which are themselves underspecified. Search consequently centers on the template(s), much as search in MOSES centers on the programs used to create new demes (and thereby new representations). The issue of deme management can thus be seen as analogous to the issue of template selection in the messy genetic algorithm.

3. Summary and Conclusions

Competent evolutionary optimization algorithms are a pivotal development, allowing encoded problems with compact decompositions to be tractably solved according to normative principles. We are still faced with the problem of *representation-building* – casting a problem in terms of knobs that can be twiddled to solve it. Hopefully, the chosen encoding will allow for a compact problem decomposition. Program learning problems in particular rarely possess compact decompositions, due to particular features generally present in program spaces (and in the mapping between programs and behaviors). This often leads to intractable problem formulations, even if the mapping between behaviors and scores has an intrinsic separable or nearly decomposable structure. As a consequence, practitioners must often resort to manually carrying out the analogue of representation-building, on a problem-specific basis.

Working under the thesis that *the properties of programs and program spaces can be leveraged as inductive bias to reduce the burden of manual representation-building, leading to competent program evolution*, I have developed the MOSES system. Experimental results and theoretical analyses have been carried out to substantiate many of the claims made in this paper regarding the properties of program spaces and how to transform them via representation-building, which will be presented in [26], along with results and analyses of MOSES itself; order-of-magnitude reductions in the number of scoring function evaluations needed to find an optimal solution have already been achieved relative to genetic programming [10], on a number of benchmarks, in addition to the ant results presented herein. Future work will focus on advanced problem domains (higher-order functions, list manipulation, recursion, etc.), more adaptive representation-building for better scalability, exploiting additional inductive bias in the behavioral space, and, most importantly in the long-term, integration with other AI components.

Acknowledgements

Thanks to Ben Goertzel for many suggestions and discussions which have been instrumental in developing the ideas presented in this paper. Thanks to the anonymous reviewer for constructive suggestions that have hopefully made this a more cogent and comprehensible paper than it would have been otherwise.

References

[1] T. Deacon. *The Symbolic Species*. Norton, 1997.

[2] D. Marr. *Vision: a computational investigation into the human representation and processing of visual information*. W. H. Freeman, 1982.

[3] M. Looks and B. Goertzel. Mixing cognitive science concepts with computer science algorithms and data structures: An integrative approach to strong AI. In *AAAI Spring Symposium Series*, 2006.

[4] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[5] G. M. Edelman. *Neural Darwinism: The Theory of Neuronal Group Selection*. Basic Books, 1988.

[6] W. H. Calvin. The brain as a Darwin machine. *Nature*, 1987.

[7] W. H. Calvin and D. Bickerton. *Lingua ex Machina*. MIT Press, 2000.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[9] N. Cramer. A representation for the adaptive generation of simple sequential programs. In *International Conference on Genetic Algorithms and their Applications*, 1985.

[10] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
[11] M. Pelikan, D. E. Goldberg, and F. G. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 2002.
[12] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. Linkage problem, distribution estimation, and Bayesian networks. *Evolutionary Computation*, 2002.
[13] M. Pelikan and D. E. Goldberg. A hierarchy machine: Learning to optimize from nature and humans. *Complexity*, 2003.
[14] M. Pelikan and D. E. Goldberg. Hierarchical BOA solves Ising spin glasses and MAXSAT. Technical report, University of Illinois at Urbana-Champaign, 2003.
[15] F. Rothlaud, D. E. Goldberg, and A. Heinzl. Bad codings and the utility of well-designed genetic algorithms. Technical report, University of Illinois at Urbana-Champaign, 2000.
[16] R. A. Watson, G. S. Hornby, and J. B. Pollack. Modeling building-block interdependency. In Parallel Problem Solving from Nature, 1998.
[17] M. Hutter. Universal algorithmic intelligence: A mathematical top-down approach. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence*. Springer-Verlag, 2005.
[18] J. Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence*. Springer-Verlag, 2005.
[19] M. Looks, B. Goertzel, and C. Pennachin. Novamente: An integrative architecture for artificial general intelligence. In *AAAI Fall Symposium Series*, 2004.
[20] E. B. Baum. *What is Thought?* MIT Press, 2004.
[21] D.G. King. Metaptation: the product of selection at the second tier. *Evolutionary Theory*, 1985.
[22] 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.
[23] M. Pelikan and T. Lin. Parameter-less hierarchical BOA. In Genetic and Evolutionary Computation Conference, 2004.
[24] K. Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation,* 1997.
[25] W. B. Langdon and R. Poli. Why ants are hard. In Genetic Programming, 1998.
[26] M. Looks. *Competent Program Evolution*. PhD thesis, Washington University in St. Louis, 2006 (forthcoming).
[27] A. Ekárt. Shorter Fitness Preserving Genetic Programming. In Artificial Evolution, 2000.
[28] P. Wong and M. Zhang. Algebraic Simplification of GP Programs During Evolution. In Genetic and Evolutionary Computation Conference, 2006.
[29] R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 1997.
[30] J. Rosca. *Hierarchical Learning with Procedural Abstraction Mechanisms*. PhD thesis, University of Rochester, 1997.
[31] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.
[32] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 1989.