WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COMPETENT PROGRAM EVOLUTION

by

Moshe Looks, M.S., B.Sc.

Prepared under the direction of Professor R. P. Loui

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

December 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY

SEVER INSTITUTE

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

COMPETENT PROGRAM EVOLUTION

by Moshe Looks

ADVISOR: Professor R. P. Loui

December 2006

Saint Louis, Missouri

Heuristic optimization methods are adaptive when they sample problem solutions based on knowledge of the search space gathered from past sampling. Recently, *competent* evolutionary optimization methods have been developed that adapt via probabilistic modeling of the search space. However, their effectiveness requires the existence of a compact problem decomposition in terms of prespecified solution parameters.

How can we use these techniques to effectively and reliably solve program learning problems, given that program spaces will rarely have compact decompositions? One method is to manually build a problem-specific representation that is more tractable than the general space. But can this process be automated? ***My thesis is 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.***

The central contributions of this dissertation are a synthesis of the requirements for competent program evolution, and the design of a procedure, *meta-optimizing semantic evolutionary search* (MOSES), that meets these requirements. In support of my thesis, experimental results are provided to analyze and verify the effectiveness of MOSES, demonstrating scalability and real-world applicability.

Contents

List of Tables

List of Figures

Preface

> "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, The Symbolic Species* [17]

The work described in this dissertation began most directly nearly four years ago, in collaboration with Dr. Ben Goertzel and Cassio Pennachin [61, 60]. The decision to study evolutionary learning was arrived at top-down, from a system-level view of cognition where dynamics of random sampling, selection, and recombination of possible solutions play key roles in problem solving. Rather than modeling human thought, however, our goals have been to consider cognitive processes' essential high-level characteristics and goals (Marr's level of *computational theory* [64]) and to develop corresponding algorithms, possibly designed and implemented along radically different lines.[1]

The particular goals of "cognitive evolutionary learning" include the on-line synthesis of relatively small nested, sequenced structures for tasks such as rock throwing and sentence generation [12, 13]. This leads us to consider *programmatic representations* for evolutionary learning. It should be clear that the aim of program evolution in this scheme is not to emulate the human programmer, but merely one particular mechanism underlying the human ability to achieve complex goals in complex environments, including (as a very particular special case) the goal of writing a program to carry out some computation.

In attempting to synthesize the requirements for *competent* program evolution – solving hard problems quickly, accurately, and reliably [28] – I have drawn on insights from a number of areas, especially:

- **Evolutionary optimization** – the theoretical and empirical understanding of optimization problem difficulty has advanced to the point where evolutionary approaches to optimization – genetic algorithms [40, 27] and estimation-of-distribution algorithms [71] – have been developed which may be considered *competent* in the sense outlined above. In particular, the hierarchical Bayesian optimization algorithm [82] stands out as a powerful approach to solving hard optimization problems via adaptive hierarchical decomposition.

[1] [59] elaborates on this approach.

- **Current approaches to program evolution** – genetic programming [16, 48], probabilistic incremental program evolution [98], and automatic design of algorithms through evolution [73], have had the most impact on the ideas presented herein.
- **Studies of problem spaces** – considering the distribution of solutions and solution quality in various problem spaces can often indicate what makes particular problem instances and classes difficult. This has led to some useful theory along with many experimental studies – the most thorough work here to-date for program spaces has been Langdon and Poli's [56].
- **Genotype-phenotype mappings** – a basic feature of programs is that they are executed to generate some behavior (a phenotype), and that many programs (genotypes) can map to the same or nearly the same behavior. The properties of such many-to-one mappings form a basis for the neutral theory of molecular evolution [45], and can have important implications for the design of evolutionary algorithms [19, 75]. For instance, different programs mapping to the same behavior might have different organizational principles, leading to a "second order" differential in evolvability (cf. [46]). A related observation is that programs with distinct behaviors may be equally effective at solving some problem, but achieve that effectiveness in different ways.
- **Algorithmic information theory, algorithmic probability, and learning theory** – in order to effectively learn programs (when possible), one needs a notion of when it is not possible to do so (learning theory) [7]. Together with an understanding of some theoretical properties of program distributions [14] and idealized inductive inference [110], novel inductive biases may be formulated specifically for learning programs. Baum's discussion of the role inductive bias plays in human cognition and its relation to compact programs [5] is particularly topical.
- **Dynamics of representation-building** – in considering ways to incorporate effective inductive bias into program evolution, the notion of a *representation-building process* emerged as the most promising candidate. The most effective representation of an event or object is contextual (cf. [64]), and representations must be allowed to shift in an ongoing interplay between perception and cognition (cf. [39]). Hawkins' discourse on the centrality to invariant representations [36], and Yudkowsky's notion of a "codic cortex" analogous to the primate visual cortex [125] were both inspirational and helped lay the conceptual foundations for my work.

Beyond the areas listed above and the sources cited herein, I am deeply indebted on a personal level to many friends and mentors.

I can honestly say that without the formative guidance and assistance of Dr. Ben Goertzel and Dr. Ron Loui, this dissertation would not exist in any recognizable form.

Dr. Bill Smart has never dismissed any of my wacky ideas that have had merit. He has provided valuable feedback and support beginning with my course project for his Machine Learning class in Spring 2003, and continuing through service on my M.S. and D.Sc. committees.

Dr. Robert Pless very kindly overcame his sensible skepticism of my initial half-baked ideas and agreed to serve on my D.Sc. committee, offering valuable and pragmatic advice.

Dr. Martin Pelikan has provided significant technical suggestions and encouragement since I first met him three years ago; I am grateful to know him and to have him on my committee.

Special thanks to Dr. David E. Goldberg for giving of his extremely valuable and scarce time to serve on my committee, a debt I cannot repay. Without his depth of experience and advice (conveyed both through one-on-one discussion and through his books and articles), my views on the theory and practice of evolutionary computation would be greatly impoverished.

Dr. Guy Genin has been extremely friendly and supportive while offering a valuable outsider's perspective on my D.Sc. committee.

Dr. Weixiong Zhang has a deep understanding of heuristic search and combinatorial optimization; the lessons I learned spending a year as his research assistant continue to influence my thinking.

Cassio Pennachin (of Vetta Labs) continues to be a valued collaborator, providing knowledge of the real-world application of evolutionary learning and, recently, essential machine time for experiments. Additional thanks to Lúcio de Souza Coelho (also of Vetta Labs) for help on datasets for the supervised categorization experiments described Section 6.3.

Thanks to my supervisors over the past three years, Steven Luce (at Science Applications International Corporation – SAIC) and Dr. John Lockwood (at Washington University in St. Louis), who have provided valuable feedback and grounding, along with crucial financial support from SAIC.

Final acknowledgments and words of gratitude where words are inadequate: to my parents, sisters, grandparents, and Hannah.

Moshe Looks

Washington University in Saint Louis
December 2006

Chapter 1

On Problems and Problem Solving

> "The primary task of the biologist is to discover the set of forms that are likely to appear, for only then is it worth asking which of them will be selected." *P. T. Saunders, interpreting Alan Turing* [103]

There are two primary levels of analysis when thinking about problems and problem-solving procedures:

- The *pre-representational* – How should problems be described as the formal inputs to problem-solving procedures?
- The *post-representational* – How effective will different problem-solving procedures be, given a particular problem formalization?

Conflating these levels is dangerous. A comparison between problem-solving algorithms (post-representational level) is meaningful only to the extent that the formal description (pre-representational level) remains fixed.[1] Along these lines, another way to view the distinction is that the pre-representational level considers the effort that must be expended by a practitioner (or superordinate procedure) in order to apply a problem-solving procedure, whereas the post-representational level considers the effort that must be expended within the procedure itself.

These two levels broadly correspond to a distinction between "knob creation" (the discovery of novel values worth parameterizing) and "knob twiddling" (adjusting the values of existing parameters), eloquently explicated by Hofstadter [38]. A similar paradigm has been articulated in evolutionary biology – King [46] distinguishes between evolved patterns that directly increase fitness, *adaptations*, as opposed to *metaptations*, organizational changes that increase fitness indirectly by constraining and directing variation along particular axes.

[1] The problem description as considered here contains *all* input that must be given to the algorithm, including settings for any internal parameters.

1.1 Competent Optimization

General optimization is the class of problems with formal representations specifying a solution space (a set of knobs to twiddle, as outlined above) and a scoring function on solutions. "Solving the problem" corresponds to the procedure returning a solution with a sufficiently high score. Taking the above into consideration, there are two ways of advancing the state of the art: pre-representational improvements that ease the burden of the descriptive process, and post-representational improvements that allow formally stated problems to be solved with higher scores, or with less computational effort. An example of the former would be automatically adjusting the value of a parameter that would otherwise need to be set manually (i.e., given as part of the formal problem statement to the algorithm). An example of the latter would be modifying a search algorithm to reduce the risk of convergence to local optima.

In the general case where the scoring function is unrestricted, techniques such as linear programming that rely on it having particular properties are inapplicable. Assuming there are insufficient resources to exhaustively search the space, a heuristic approach must be employed, such as random sampling, hillclimbing, or simulated annealing. These procedures tell us how to generate new solutions to score, possibly based on the scores of past solutions that have been tested.

The only way an approach to optimization can outperform random search or enumeration is by assuming some regularities in the scoring function.[2] Thus, the question to ask is: *What regularities occur post-representationally across the range of problems we are interested in solving that distinguish them from arbitrary problems?* Assuming they can be characterized, the next step must be to design principled algorithms to robustly exploit these regularities. A fundamental source of such regularities derives from attempting to *decompose* a problem – break it down into smaller subproblems that may be solved independently. Classically, many real-world systems are observed to possess *near decomposability* [109].

How successfully an optimization algorithm can apply such an approach post-representationally depends on how clever an encoding has been chosen pre-representationally. Ideally, in searching an n-dimensional parameter space, there will be complete separability (the strongest kind of decomposability) for each parameter. If this can in fact be assumed, it is quite straightforward to apply a normative probabilistic approach – to generate random points in the search space according to a uniform distribution, score them, update the distribution to tend toward higher-scoring points, and iteratively resample new points.

[2] And, of course, thereby *underperforming* random search averaged over problems where these regularities are not present – cf. [117, 67].

Complete separability thus amounts to probabilistic independence, and a separate probability distribution may be maintained for each parameter.[3]

This method is essentially the simplest member of a family of approaches to optimization known as estimation-of-distribution algorithms (EDAs), *population-based incremental learning* [3]. For an overview of this area of research, see Pelikan et al. [85]. If the correct decomposition is computed pre-representationally, one can incorporate it into the sample-score-adjust cycle outlined above, obtaining the factorized distribution algorithm [70], another EDA. These approaches leverage their independence assumptions to perform empirical credit assignment (allocating probability density among competing assignments to parameters or sets of parameters), according to Bayesian principles. Over time, knowledge of the overall solution space is accumulated and exploited in the generation of new solutions, making EDAs *adaptive optimization algorithms*, as conceived by Holland [40].

Such normative approaches of course are applicable beyond the realm of optimization as well – for any problem where the correct representational structure (which parameters to pay attention to and their interactions) is known, it is often possible to deploy simple probabilistic-statistical methods to achieve superior performance. Elkan, for example, describes the winning entry in a data mining contest based on naive Bayesian learning, modified (manually by the author) to incorporate a few key dependencies between variables and to discard irrelevant variables [21]. In this case, the pre-representational problem has been solved directly and completely by the human encoding the task.

A more general approach to adaptive optimization, given a solution space defined by a set of parameters, is to attempt to learn an accurate problem decomposition dynamically. Recently, a number of such *competent* adaptive optimization algorithms have indeed been developed, and their ongoing proliferation dubbed the *competence revolution* [28]. These approaches are *evolutionary* inasmuch as they operate via incremental processes of selection and recombination to iteratively improve an initial sampling of solutions (generated according to some prior probability distribution).

Two of the most advanced competent adaptive optimization algorithms are the Bayesian optimization algorithm (BOA) [84], and hierarchical BOA (hBOA) [82]. The BOA is an extension of the population-based incremental learning approach described above that, like the factorized distribution algorithm, does not assume the independence of all parameters. Instead of a fixed model, however, a problem decomposition represented as a Bayesian

[3]There is still the question of selecting an appropriate prior (nearly any will do in the limit) – this turns out to be somewhat problematic in practice when dealing with continuous variables, though not insurmountable [9].

Figure 1.1: The structure of OneMax, a paradigmatic *separable* optimization problem.

network with nodes (probabilistic variables) corresponding to parameters in the space is dynamically learned (e.g., via greedy learning with a minimum-description-length metric).[4] In the hBOA, the Bayesian network representation is augmented with local structure, allowing for more expressive (hierarchical) problem decompositions, and *niching* is incorporated into the selection process, which promotes the preservation of diverse solutions.[5] In problem decomposition via probabilistic modeling (as in the BOA and hBOA), one represents interactions between parameters in the representation as probabilistic dependencies between the corresponding variables in the model.

The effectiveness of the BOA and hBOA on problems with known decompositions that they can represent has been demonstrated [84, 82]. A further question to ask is if they are effective, empirically, on real-world problems with unknown decompositions (which may or may not be effectively representable by the algorithms). On problems they have been tested on to date, the answer is affirmative; robust, high-quality results have been obtained for Ising spin glasses and MaxSAT [81], as well as a real-world telecommunication problem, even with a "bad" encoding [97].

In summary, encodings that eliminate interactions between variables lead to easy problems (Figure 1.1), which are tractably solvable via iterative random sampling, as well as classic heuristic optimization procedures such as local search and genetic algorithms. Encodings where interactions can be restricted to bounded sub-problems, possibly with a hierarchical structure (Figure 1.2), are tractably solvable when an accurate model of this structure can be manually encoded or learned by a competent optimization algorithm. It is also important to recognize the possibility of intractable encodings, where all variables critically depend on all others, and no compact decomposition exists (Figure 1.3).

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* [64]

It should be apparent that there is a fine line between representation-building and robust adaptive optimization (and hence between the pre- and post-representational levels),

[4] The appeal of Bayesian networks and related knowledge representation schemes is their marriage of statistical inference and problem decomposition principles.

[5] This is important because in practical implementations, some sample points must be discarded as the search progresses.

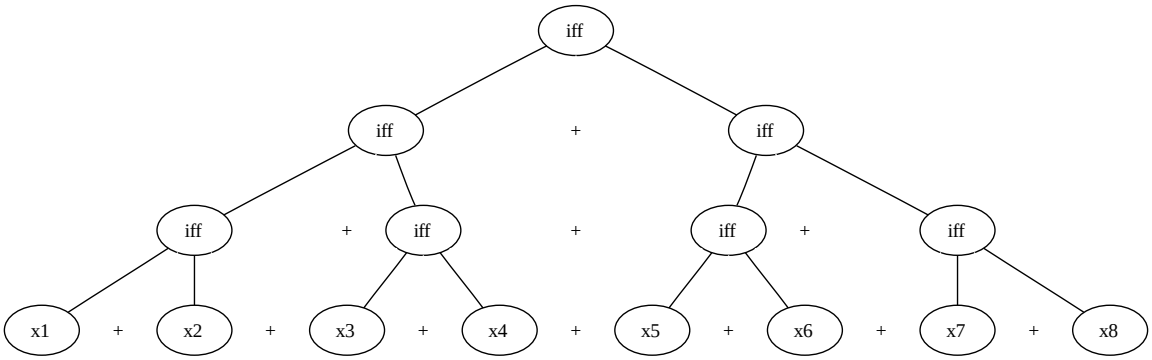

Figure 1.2: The structure of hierarchical if-and-only-if [118], a paradigmatic *nearly decomposable* optimization problem.

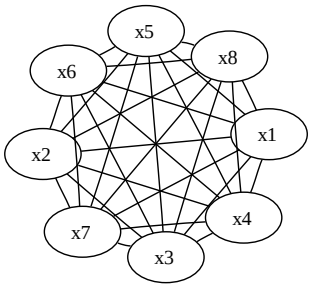

Figure 1.3: The structure of an intractable optimization problem, such as a uniform random scoring function, where changing the assignment of any variable always results in a chaotic change in the overall score.

inasmuch as competent methods such as the BOA and hBOA can adjust their problem decompositions to overcome difficulties with the representation.[6]

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[7] and reasonably correct decomposition in the space (of decompositions, not solutions) being searched.

Difficulty arises when no such decomposition exists, 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, a "universal" approach to general problem-solving is indeed possible [43, 104].

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. Note that this requirement is fairly universal across a diversity of approaches; even "unsupervised" learning algorithms have some prespecified formal measure, implicit or explicit, which they attempt to maximize. This is of course to be expected – otherwise we would have no means of preferring one result over another (or one learning algorithm over another). We may have some vague, informal goal in mind, but any fully realized procedure must necessarily embody a formal target. The primary focus of my dissertation is the set of issues surrounding the representation of solutions, rather than how solutions are scored. How solutions are scored will be addressed as well, but less ambitiously; a full treatment of would require a much broader scope.

[6] Principal component analysis may be seen as another generic means of overcoming representational difficulty.

[7] 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 also in accordance with the general notion of learning corresponding to compression [7, 5].

1.3 Why is Program Learning Different?

Recall the description of an optimization problem given in Section 1.1: a solution space S is specified, together with some scoring function on solutions, where "solving the problem" corresponds to discovering a solution in S with a sufficiently high score. Let's define program learning as follows: given a program space P, a behavior space B, an execution function $exec : P \rightarrow B$, and a scoring function on *behaviors*, "solving the problem" corresponds to discovering a program p in P whose corresponding behavior, $exec(p)$, 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[8] regarding the program and behavior spaces, and the execution and scoring functions (the additional inductive bias mentioned above in Section 1.2):

- **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 that are themselves members of P or some related program space. This provides a natural family of distance measures on programs, in terms of 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 factors conspires to *scramble* scoring functions –

[8] As a technical note, *exec* is assumed to be computable and B is assumed to be finite. As we will only ever run our programs on finite hardware for finite amounts of time, this is in no way a limiting assumption. Throughout this dissertation we will encounter issues of computational complexity which are quite relevant for competent program evolution. Issues of computability are not relevant, however, even when dealing with "Turing complete" program classes. This assumption is certainly important, but is uninteresting.

even if the mapping from behaviors to scores is separable or nearly decomposable, the complex[9] program space and chaotic execution function will often quickly lead to intractability as problem size grows. This fundamental claim will be substantiated throughout this dissertation.

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 [5] 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[10] – 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 generally assigns greater probability mass to smaller programs.
- **Simplicity preference** – given two programs mapping to the same behavior, we generally prefer the smaller program. This may be seen as a secondary scoring function.
- **Behavioral decomposability** – the mapping between behaviors and scores is separable or nearly decomposable. This is akin to Haynes' contention that building blocks for program evolution are most appropriately defined phenotypically (i.e, on the behavioral level) [37]. 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.[11] This partial order will never contradict the total ordering of scalar scores.
- **White box execution** – the mechanism of program execution is known *a priori*, and remains constant across many problems.

[9] Here "complex" means open-ended, over-representing, and hierarchical.

[10] Otherwise, humans could not write programs significantly more compact than lookup tables.

[11] For example, in supervised classification one rule dominates another if it correctly classifies all of the items that the second rule classifies correctly, as well as some which the second rule gets wrong.

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, is the central preoccupation of this dissertation.

1.4 Thesis

In summary:

- When thinking about problems and problem solving it is important to explicate the often implicit distinction between the pre- and post-representational levels.
- Competent evolutionary optimization algorithms are a pivotal development, allowing encoded problems with compact decompositions to be tractably solved according to normative principles (post-representationally).
- 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.

My thesis is 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.

A framework with mechanisms for exploiting these properties, *meta-optimizing semantic evolutionary search* (MOSES), will be presented in the next chapter.

Chapter 2

Mechanics of Competent Program Evolution

In this chapter I present *meta-optimizing semantic evolutionary search* (MOSES), a framework for competent program evolution. Based on the viewpoint developed in the previous chapter, MOSES is designed around the central and unprecedented capability of competent optimization algorithms, 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).

The process of representation-building given a particular program or family of programs is covered in Section 2.1, followed by a description of how representation-building meshes dynamically with an ongoing optimization process in Section 2.2.

2.1 Statics

The basic goal of MOSES is to exploit the regularities in program spaces outlined in Section 1.3, 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.

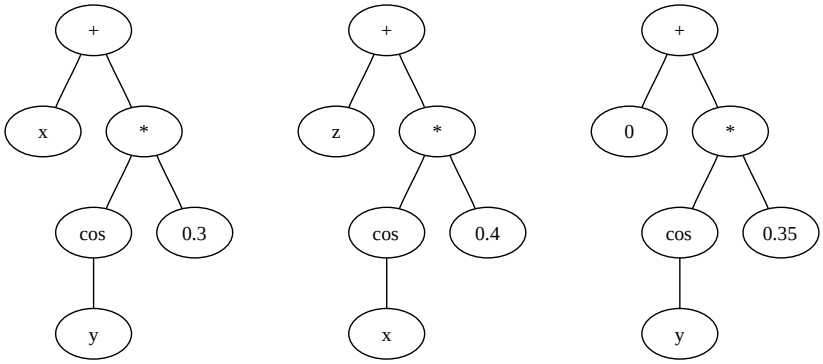

Figure 2.1: Three simple program trees encoding real-valued expressions, with identical structures and node-by-node semantics.

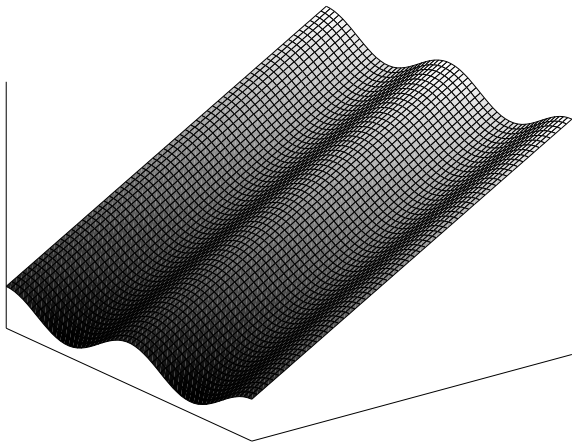

Figure 2.2: The corresponding behavioral pattern for the programs in Figure 2.1; the horizontal axes correspond to variation of arguments (i.e., x, y, and/or z), with the vertical axis showing the corresponding variation of output.

2.1.1 Variations on a Theme

Let's start with an easy example. What knobs (meaningful parameters to vary) exist for the family of programs in Figure 2.1?[1] 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 in Figure 2.2. 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*. We are searching through the space of combinations of knob settings (not a single change at a time, but a set of changes). The effects of knobs should be as different from each other as possible, and any knob whose effects are equivalent to another knob or combination of knobs is undesirable.[2] 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* (Figure 2.2), a number of additional knobs can be imagined which might be turned in meaningful ways, such as:

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

[1] Throughout this dissertation, programs will be depicted for convenience as rooted trees with ordered children, with atomic functions in all internal nodes taking their children as arguments. This is equivalent to a parse tree or symbolic expression. To express sequential execution, the *progn* function will be used, which takes a variable number of arguments, evaluates them from left to right, and returns the result of the rightmost evaluation. Note that this is for expository purposes only - MOSES does not require or expect any particular underlying program representation, as we shall see.

[2] 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 [84, 82].

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, $a_1 + a_2 + ... + a_n$ 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 ($x\ AND\ x \leftrightarrow x$), list manipulation ($cdr\ (cons\ x\ L) \leftrightarrow L$), and conditionals ($if\ x\ then\ y\ else\ z \leftrightarrow if\ NOT\ x\ then\ z\ else\ y$).

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, as will be shown in the experimental results in later chapters.

Oversampling of distant behaviors (that is, oversampling of programs that are behaviorally distant) is caused directly by chaotic execution, as well as an 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.[3]

Undersampling of nearby behaviors (that is, undersampling of programs that are behaviorally close) 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" [54, 56] 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.

[3]Since *exec* is assumed to be computable and B to be finite (see footnote to Section 1.3), minimal length, while expensive to calculate exactly, is in fact a computable quantity in this context, in contrast to the corresponding quantity studied in algorithmic information theory [14].

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, in the terminology of King [46] introduce at the beginning of Chapter 1. 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 to be addressed later.

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 [114], although they are of course not the whole story (cf. [47]). 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.

2.1.3 Neighborhoods and Normal Forms

The procedure MOSES uses to construct a set of knobs, or representation, 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 [119, 41].

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; 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. 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 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 point in program space and heuristically centered around the corresponding point 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.

2.2 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 (the *exemplar*) or family of closely related programs. This is consonant with the line of thought developed in Chapter 1, 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 representation will be referred to herein as a *deme*;[4] a set of demes (together spanning an arbitrary area within program space in a patchwork fashion) will be referred to as a *metapopulation*.[5] 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.

2.2.1 Algorithmic Sketch

The salient aspects of programs and program learning described in Section 1.3 lead to requirements for competent program evolution that can be addressed via a representation-building process such as the one shown above in Section 2.1, combined with effective deme

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

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

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 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 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, 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 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.

The hierarchical Bayesian optimization algorithm (hBOA) [82] is used for step 2(b).

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.

The exemplar used to generate a representation also defines default values for all knobs; these are the knob settings for the exemplar itself. The default values are used to bias the initial sampling for a new deme 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

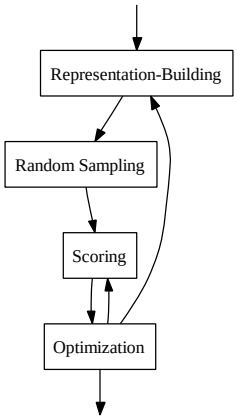

Figure 2.3: The top-level architectural components of MOSES, with directed edges indicating the flow of information and program control.

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.

Many details of this procedure need to be filled in, and will be in the next three chapters. Chapters 3 and 4 delve into program spaces and their transformation via representation-building in the domain of Boolean formulae, and provides empirical verification for many of the claims and arguments made thus far. Chapter 5 then addresses some fundamental issues relating to problem difficulty, arguing for a bipartite model unique to program evolution, distinguishing between program-level and representation-level difficulty.

2.2.2 Architecture

The preceding algorithmic sketch of MOSES leads to the top-level architecture depicted in Figure 2.3. 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-independent. There is of course the possibility of improving performance by incorporating domain and/or problem-specific bias into random sampling and optimization as well.

2.3 Example: Artificial Ant

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 [48]. I will then describe the search process, and present comparative experimental results. After an in-depth study of program spaces and problem difficulty in the next three chapters, which will incidentally fill in the details of MOSES's operation, results from more complex domains and problems will be presented.

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). It 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, and a scoring function counting the number of 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). The beginning of the trail is as follows with food denoted by #s, and the ant located in the upper left, facing right:

```
###
  #
  #
  #
  #
  #### #####
           #
           #
```

Programs are composed of the primitive actions taking no arguments, a conditional (*if-food-ahead*),[6] 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 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 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

[6] 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.

2. Any *if-food-ahead* statement that 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 that 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. Representations are be organized with a *progn* knob in the root. The leftmost child is a rotation knob, followed to the right by a conditional knob, followed by a movement knob, followed by a rotation knob, etc., as below.[7]

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 empty program *progn* (used as the initial exemplar for MOSES), for example, leads to the representation shown in Figure 2.4.

There are six parameters here, three quaternary[8] and three binary. So the program *progn(left, if-food-ahead(move, left))* would be encoded in the space as [*left, no rotation,*

[7]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.

[8]The original hBOA is designed to operate solely on binary parameters; I have use Ocenasek's generalized approach [72] here, and elsewhere in the dissertation where non-Boolean parameters are called for.

```
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])
```

Figure 2.4: The initial representation built by MOSES for the artificial ant domain.

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 [89] 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.[9] 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 [48] as well as evolutionary programming based solely on mutation [15] (i.e., a form of population-based stochastic hillclimbing). Furthermore, an extensive analysis of the search space has been carried out by Langdon and Poli [55], with the authors concluding:

[9] 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.

1. 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.

2. The search space contains many symmetries (e.g., between left and right rotations),

3. 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 required) than random sampling.

4. "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".

The scalability of MOSES will be analyzed in later chapters. For now, let's look at results for the artificial ant problem, which indicate the magnitude of improvement that may be experienced. Koza [48] 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 minimum "computational effort" needed to achieve success with 99% probability, which was attained by processing through generation 14, was 450,000 (based on parallel independent runs). Chellapilla [15] reports 47 out of 50 successful runs with a minimum computational effort (again, for success with 99% probability) of 136,000 for his stochastic hillclimbing method based on evolutionary programming.

Langdon and Poli [53] study the performance genetic programming with problem-specific modifications to the scoring function, requiring the ant to traverse the path sequentially, adding a time penalization term and other terms. These are also considered in combination with variations on the genetic programming crossover operation. Out of twelve sets of runs (generally consisting of fifty independent trails each) with various configurations, the highest success rate achieved was 0.38%; the lowest computational effort figure was 120,000 program evaluations. When a size restriction was added, the minimum computational effort achieved over all configurations decreased to 104,000.

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 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, 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 (see Figure 2.5).

Table 2.1: The computational effort required to find an optimal solution to the artificial ant on the Santa Fe trail problem for various techniques with $p = .99$ (for MOSES $p = 1$, since an optimal solution was found in all runs).

Technique	Computational Effort
Genetic programming [48]	450,000 evaluations
Evolutionary programming [15]	136,000 evaluations
MOSES	23,000 evaluations

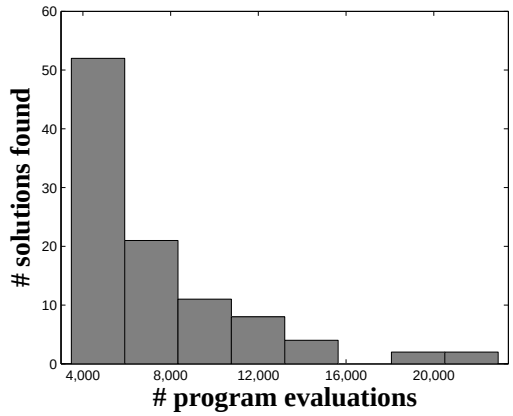

Figure 2.5: A histogram of the number of program evaluations required to find an optimal solution to the artificial ant problem for 100 runs of MOSES. Each run is counted once for the first global optimum reached.

Why does MOSES outperform other techniques? One factor to consider first is that the language that programs are evolved in is slightly more expressive; 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 that do nothing if food is present (or not present). Indeed, many of the smallest solutions found by MOSES exploit this feature. This effect can be controlled 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 far short of the results obtained with MOSES (the success rate after 51 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 of past attempts at introducing program simplification into genetic programming systems [42, 20, 120] have been mixed; although the system may be sped up (because programs are smaller), there have been no dramatic improvement noted in the quality of results. 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).[10] 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 26,356 evaluations to reach an optimum. This is about 3.5 times more than the average number of evaluations required with model-building.

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 stochastic hillclimbing and the probabilistic incremental program evolution (PIPE) [98] algorithm, which learns programs based on a probabilistic model without any interactions (see Appendix A). PIPE has 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.[11] For the 6-knob model given above for instance, analysis the linkages

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

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

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 models. This preliminary finding is quite significant given Landgon and Poli's findings on symmetry, and their observation [55] 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.

2.4 Discussion

The overall MOSES design as described in this chapter 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 [96] is an approach to program evolution that also alternates between separate representation-building and optimization stages. It is based on Koza's genetic programming [48], 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 program-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* [49], MOSES explicitly addresses the underlying (semantic) structure of program space independently of the search for any kind of modularity or problem decomposition. This critically changes neighborhood structures (syntactic similarity) and other aggregate properties of programs, as will be shown experimentally in the following chapters.

Regarding deme management, the embedding of an evolutionary algorithm within a superordinate procedure maintaining a metapopulation is most commonly associated with "island model" architectures [27, 50]. 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 subspaces defined by the representation-building process. These subspaces may be considered as being defined by templates assigning values to some of the underlying dimensions (e.g., they restrict the size and shape of any resulting trees). The messy genetic algorithm [30], an early competent optimization algorithm, uses a similar mechanism - a common "competitive template" is used to evaluate candidate solutions to the optimization problem which may be underspecified. Search consequently centers on the template(s), much as search in MOSES centers around the exemplar programs used to create new demes (with new representations). The issue of deme management can thus be seen as analogous to the issue of template selection in the messy genetic algorithm.

A review of approaches to program evolution in relation to MOSES on the system rather than the feature level may be found in Appendix A.

Chapter 3

Understanding Program Spaces

> "In the discrete world of computing, there is no meaningful metric in which 'small' changes and 'small' effects go hand in hand, and there never will be."
> *Edsger Dijkstra* [18]

This chapter is an in-depth study of the regularities inherent in program spaces, much of it empirical. Interestingly however, it can be characterized to a large extent with tools from algorithmic information theory [14]. Boolean function and formula spaces are introduced as a useful domain for illustration and demonstration.

An n-ary Boolean *function* is a mapping from $\{true, false\}^n$ to $\{true, false\}$. An n-ary Boolean *formula* is a symbolic expression (i.e., a parse tree) with leaves drawn from the set of variables $\{x_1, x_2, ..., x_n\}$, and internal nodes drawn from some operator set (the formula's *basis*) consisting of Boolean functions, such that the number of children of every internal node is equal to the arity of its corresponding operator. Every formula computes a function:

1. Formulae which are a single leaf x_i compute the function mapping tuples $(v_1, v_2, ..., v_n) \in \{true, false\}^n$ to v_i.

2. Formulae which are rooted in an internal node, an m-ary operator σ, compute the function mapping tuples $(v_1, v_2, ..., v_n) \in \{true, false\}^n$ to $\sigma(r_1, r_2, ..., r_m)$, where every r_i (with $i \in 1, ..., m$) is the result computed by the node's ith subexpression's function for the input $(v_1, v_2, ..., v_n)$.

We will consider program spaces of Boolean formulae, with the execution function mapping to Boolean functions (the behavior space). The problems we will be interested in solving will concern the discovery of n-ary formulae computing particular prespecified functions, with the score given by the Hamming distance between all or a subset of a function's (2^n different) outputs and those of the target (lower being better, of course). All things being equal, a smaller formula (one containing less symbols) will be preferred. The

operator set will consist of conjunction (AND), disjunction (OR), and negation (NOT), which can be used to compute any given Boolean function (cf. [119]).

Why study this particular program space? As described in Chapter 1, program learning is most basically defined by the decomposition of the scoring function into execution (translation from program space to behavior space) and evaluation (in behavior space). The behavior space for Boolean formulae and the scoring function (bit-strings and Hamming distance, respectively), have a uniformly scaled, separable structure. This mirrors the OneMax problem mentioned in Chapter 1, whose bit-string scoring function is simply the sum of the input bits (i.e., Hamming distance from the string of all ones). Furthermore the operator set is small and simple, and the variable set is uniform and tunable via a single parameter (the arity) that controls the size of the space. It is thus possible to focus on the essential aspects of program evolution. And as we shall see, there is a wide range of easy-to-hard problems that may be considered.

A basic question to ask at the outset is if the program/behavior space of Boolean formula learning (or any particular programmatic domain) in fact aligns with the the fundamental properties of program learning I have posited, namely: open-endedness, over-representation, compositional hierarchy, chaotic execution, a simplicity prior, a simplicity preference, behavioral decomposability, and white box execution (see Section 1.3). The first three of these follow immediately from definitions of functions, formulae, and the mapping of formulae to functions. The last three follow immediately from the mapping of formulae to functions and the family of scoring functions under consideration. Given no additional information, a simplicity prior over such formulae is pragmatically and philosophically justifiable [7, 5], as well as in accordance with human problem-solving judgments [23].

Theory and experimentation will both be brought to bear on the study of chaotic execution, along with the effects of non-orthogonality, oversampling of distant behaviors, and undersampling of nearby behaviors (see Section 2.1.2). In this context is is important to consider both global and local properties of the space. In this chapter we will pursue a global-local-global theme, considering:

- *Globally:* How are behaviors distributed in program space?
- *Locally:* How do small syntactic perturbations in the structure of a program alter its behavior?
- *Globally:* How do the syntactic distances between formulae relate to the semantic distances between their corresponding behaviors?

The answers to these questions vary based on how programs are sampled and the composition of the space (e.g., the arity of our formulae), as we shall see. For Boolean formulae, we shall sample according to a uniform random procedure where the number of

literals (variables or their negations) in the desired formula is prespecified. First, a binary tree shape with a particular number of leaf nodes (corresponding to the desired number of literals) is uniformly randomly chosen. Then all internal nodes are uniformly randomly labeled with *AND* or *OR*, and all leaves with literals. Technically negation involves replacing a leaf with an internal node labeled *NOT* and a single child-leaf labeled with a variable, according to the definitions given above, but it is convenient to consider negated variables to be single leaf nodes.[1]

3.1 The Distribution of Behaviors

Previous studies [48, 56, 33, 96, 114], have found that Boolean formula spaces, when sampled syntactically according to various schemes, compute a highly skewed distribution of functions ("simpler" functions are overrepresented). The interested reader is directed to Langdon and Poli's work in particular for further results and discussion, encompassing non-Boolean domains and programs-in-general as well [56]. Furthermore, this skewed distribution cannot be avoided in general by sampling larger or smaller programs – convergence results can be obtained across program spaces satisfying fairly general requirements, indicating that the distribution of behaviors as a function of program size remains essentially fixed past a certain size [56].

This stands in contrast to related work in combinatorial optimization; in traveling salesman [8] and maximum satisfiability [128] many problem instances of interest exhibit strong local regularities (leading to the effectiveness of stochastic local search algorithms) as well as global *big valley* structure (which may be exploited by simple metaheuristics [128, 127]).

These results can be qualitatively understood as corollaries of: (1) the incompressibility of minimal programs [14]; and (2) that an output x's algorithmic probability may be effectively approximated by 2^{-l}, where l measures (in bits) the length of the minimal program computing x [110, 111]. This value of x may be considered its *intrinsic complexity* in the given program space, corresponding to Kolmogorov complexity for Turing-complete program spaces.

Boolean formulae in particular follow this general relationship (i.e., a geometric distribution) as illustrated in Figure 3.1. This shows the distribution of formulae with one hundred literals (when sampled as described above), grouped by minimal program length (the number of literals in the shortest corresponding formula). Because computing the minimal program length for functions of high arity is prohibitively expensive, data are limited

[1] This sampling procedure is equivalent to the one employed by Holman [41] and later by Langdon [52]. As these authors have done, my implementation is based on logarithms of the Catalan numbers (the recurrence relation used to calculate the number of binary trees of a given size) to avoid overflow.

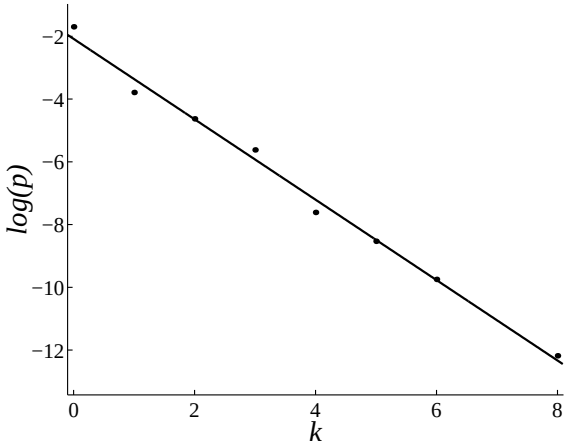

Figure 3.1: Minimal formula length measured in literals, k, vs. log-of-density, $log(p)$, in the space of ternary Boolean formulae with one hundred literals. Density (p) is the proportion of formulae sampled with a given minimal length. The solid line is a regression fit to the data, $log(p) = -2.083 - 1.281k$.

to ternary functions. A sampling of a million programs was used. All ternary functions except for parity are represented; *AND/OR/NOT* formulae computing 3-parity are too rare to regularly appear in a random sampling of a million formulae (their minimal formulae have ten literals). Note that tautology and contradiction are displayed as containing zero literals.

The graph in Figure 3.1 fits the theory well because for formula with one hundred literals the *limiting distribution* for ternary Boolean formulae has effectively been reached, and because one million samples is sufficient to compute reasonably accurate probabilities for all but the rarest of behaviors (i.e., the parity functions). What are the consequences for smaller sample sizes and samples of smaller programs? Consider how many unique behaviors we can expect in a sample of m programs of a particular size (k) in a particular space (for Boolean formulae this will grow with n, the arity). In particular, we will consider number of unique behaviors as a fraction of the total number of programs sampled (henceforth the *behavioral diversity* of the sample).

The graph on the left in Figure 3.2 shows that as the sample size m increases, behavioral diversity decreases, which is to be expected given the heavy skew towards programs with short minimal lengths outlined above. An observation based on this result is that simply increasing our (syntactic) sample size is often an inefficient method for achieving behavioral diversity (as shown). The graph on the right in Figure 3.2 shows the same qualitative effect as a function of program sample length k. However, program space grows so quickly that in practice this can only really be observed for very small k. For $k = 10$ for

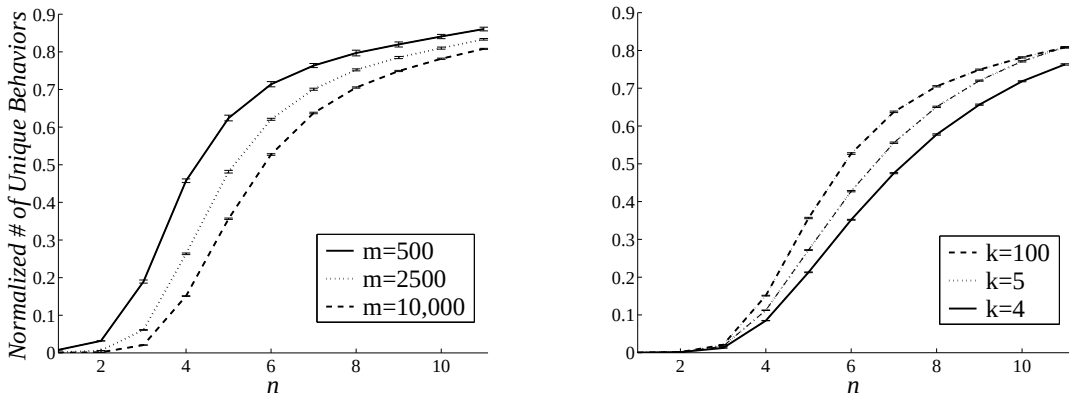

Figure 3.2: The proportion of unique behaviors (Boolean functions) in a given sample of programs (Boolean formulae), as a function of the arity of the function space, n. This can be seen to vary based on the sample size m (left), and formula size measured in literals, k (right). Formula size for the left graph is one hundred literals, and sample size for the right graph is ten thousand. Error bars are 95% confidence intervals based on thirty independent trials.

instance, the behavioral diversity for a random sample is nearly identical to the distribution for $k = 100$ (and in fact to that for $k = 2000$ as well).

A final feature to consider in these graphs is the ubiquitous S-shaped curve as n increases. The left side of the transition ($n \leq 2$) can be understood as the regime where behavioral diversity is effectively bounded by the number of unique behaviors in the space (2^{2^n}). The right side of the transition (around $n \geq 8$), correspondingly, is the regime where the number of unique behaviors in the space dominates distributional skew and sample size, leading to high behavioral diversity. A future area of interest would be to consider the ubiquity of these phase transitions in program spaces and their relationship to problem difficulty and the underlying combinatorics of the space, as has been successfully achieved for optimization problems such as Boolean satisfiability [68], and the traveling salesman [126, 24].

3.2 Semantic vs. Syntactic Distance

To characterize the syntactic neighborhood structure of program space in relation to semantic (behavioral space) neighborhood structure, we will calculate the semantic distance distribution of a (random) program's syntactic neighbors.

A neighborhood structure for these computations 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,

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 (this is symmetrical). 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). The number of neighbors may be computed as a function of the number of literals k and the arity n. It will be the sum of the number of possible replacement operations, $k \cdot (2n - 1)$, the number of possible removals of a single operator and literal child, k, and the number of possible insertions of a single operator and literal-child, $(2k - 1) \cdot 2 \cdot 2n$. The total neighborhood size is thus

$$k \cdot (2n - 1) + k + (2k - 1) \cdot 2 \cdot 2n = 10nk - 4n. \tag{3.1}$$

This particular edit distance and neighborhood structure corresponds to syntactic processing based on contiguous subtrees; e.g., to obtain $AND(x_1, x_2)$ from $OR(x_1, x_2)$ requires two edits, each with a cost of two, for a total distance of four. The corresponding mathematical model of stochastic subtree swapping is the general schema theory [92, 56] developed for genetic programming. Edit distance as a tool for studying program evolution was first proposed by O'Reilly [78]. Typically, it is used when considering dynamics such as diversity among programs and the effects of selection and recombination over time [78, 32].

Figure 3.3 shows the distribution of behaviors and *unique* behaviors among the neighbors of random formulae, as a proportion of the total neighborhood size. Only neighbors which are behaviorally different from the source formula (a minority) are shown. Results for formulae with arity five are on the left and with arity ten on the right. The data are based on one thousand random formulae with one hundred literals for each arity, generated as described above.

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 fixed size sample as the arity increase). 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.).

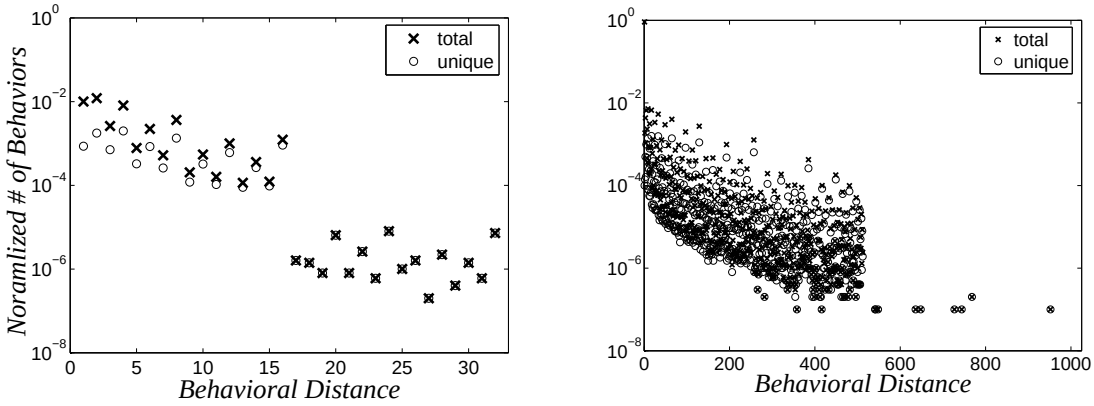

Figure 3.3: Distribution of behaviors and unique behaviors as a proportion of total neighborhood size for random formulae with arities five (left) and ten (right). Note the logarithmic scale on the y axis. The wall at 2^{n-1} (16 on the left, 512 on the right) is due to symmetries in the space – see footnote.

When considering the distribution of the remaining four or nine percent of neighbors, of particular interest is the long tail of the distribution up until the halfway point (2^{n-1}),[2] and the fact that the total density of neighboring formulae with a behavioral distance of one or two is around an order of magnitude greater than the density of unique behaviors. That is, 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 over-representation! This is yet another consequence of the skewed distributions present in program spaces.

A final basic question that needs answering is how the distribution of behaviors and neighborhood properties translates to the global relationship between semantic and syntactic distance. The may be undertaken using the edit distance metric defined above. Experimentally, what we would like to compute is thus the set of pairwise distances (syntactic and semantic) for a set of random formulae.

To actually compute the distance between two arbitrary (non-neighbor) formulae, dynamic programming may be used to find an alignment between the formulae (represented as labeled trees) that minimizes a corresponding cost function. For the case of two binary trees, this may be computed in $O(k^4)$, for two formulae with k literals each – the recurrence relation is $f(i,j) = 2f(i,j/2) + 2f(i/2,j)$, since the four possible pairwise alignments between the two trees' roots' child-subtrees must be considered, which gives $f(k,k) = k^4$.

[2]The density of syntactic neighbors past this distance declines so sharply because of the symmetries of the space – all literals have pairwise behavioral distances of 2^{n-1}, excepting variables and their negations. Furthermore, binary conjunctions and disjunctions will be at most a distance of 2^{n-1} from the behaviorally closer of their two arguments.

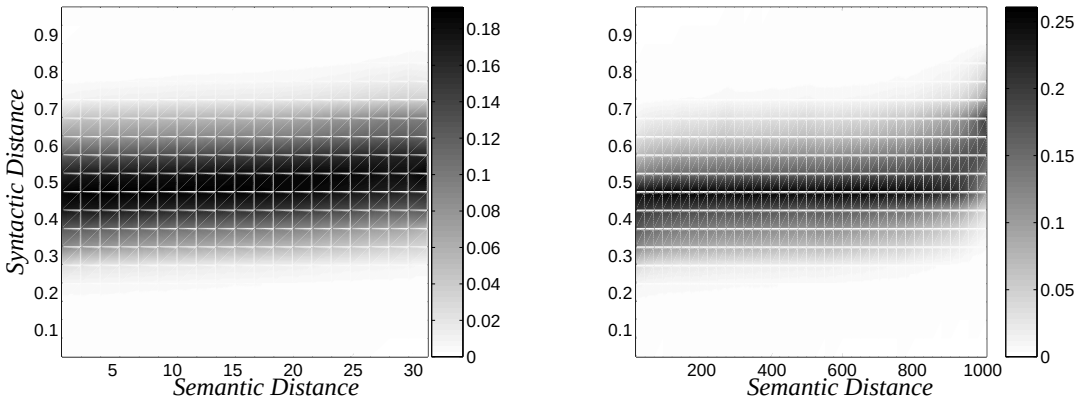

Figure 3.4: Pairwise distribution for edit distance in program space (syntactic) vs. Hamming distance in behavior space (semantic) for random formulae with arities five (left) and ten (right).

Due to this high computational complexity, data are restricted to random formulae (generated again as described above) with only thirty literals, still a fair approximation of the limiting distribution (see above). Five thousand *syntactically and semantically unique* formulae were generated each for arities five and ten, and all pairwise distances computed. The requirement of uniqueness allows us to specifically study the relationship between syntactic and semantic distance when both are strictly positive. The resultant distributions are shown in Figure 3.4. Syntactic distance is normalized based on the data 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 for arity five, and a 20×40 mesh for arity ten. Interpolation is used to generate smooth figures.

Except for very high semantic distances at arity ten, little variation is shown in the distribution based on semantic distance. In other words, 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. This result is complementary to the previous study of the semantic distribution for syntactically close programs.

We have now seen how behaviors are distributed in program spaces, the properties of the local neighborhood structure linking syntactically close programs together, and the global relationship between syntactic and semantic distance in program spaces. It is possible to state affirmatively that execution in Boolean formulae spaces is chaotic, and that effects of non-orthogonality, oversampling of distant behaviors, and undersampling of nearby behaviors are all felt. It is expected based on the results of Langdon and Poli [56] and algorithmic information theory that most program spaces of interest will have these

properties. Now let's move beyond the study of fixed landscapes and distributions, and return to representation-building. Once it has been described in more detail, the effects of representation-building on program and behavior space properties will be analyzed.

Chapter 4

Transforming Program Spaces

This chapter describes representation-building mechanisms in depth, and examines their effects in transforming program spaces. The experimental results of these transformations are instrumental to verifying my thesis that representation-building can dramatically change the properties of program spaces (detailed in the previous chapter) to facilitate competent program evolution. As in the previous chapter, Boolean function and formula spaces are used to demonstrate.

4.1 Hierarchical Normal Forms

In accordance with the vision outlined in Chapter 2, the first step in applying MOSES to learning in any given domain is to design an appropriate hierarchical (i.e., structure-preserving) normal form. In this section I will posit some general guidelines for this process, and fill in the details in the context of Boolean formulae. The effects of representing programs in a hierarchical normal form (HNF) are then analyzed.

The primary goal of reduction to normal form for MOSES is to eliminate redundancy (e.g., a subformula *AND(x,x)* would be replaced with x) and explicate programs' essential structure (e.g., a subformula *AND(AND(x, y), AND(z, q))* would be replaced with *AND(x, y, z, q)*). This kind of reduction can be expected to transform program space in a number of ways. Consider:

- *Globally:* The syntactic distances between formulae will ideally become more closely correlated with the semantic distances between their corresponding behaviors.
- *Locally:* The set of perturbations in the structure of a program considered "small" will change, ideally becoming more similar to the set of small semantic perturbations.
- *Globally:* If we only consider programs in normal form, the distribution of behaviors in program space will change, ideally becoming more uniform.

One fairly general procedure for reaching an effective normal form is to break the transformation process down into two stages. First, knowledge of operator associativity, commutativity, idempotence, etc., is applied locally to reduce the program representation – associativity allows nested compositions to be flattened, commutativity allows ordered sets of arguments to be labeled as unordered, and idempotence allows elements and possibly expressions to be simply removed. Second, heuristic search is performed to identify correspondences between program elements leading to reductions (e.g., search along all paths from the root to leaves of a tree, or over all pairs of siblings). If there is utility in doing so, the two stages may be iterated until a stable result is reached.

This is similar to the heuristic approach taken in computer algebra systems (cf. [22]). For representations such as Boolean formulae and real-valued expressions, the resultant normal-form programs will have alternating levels of linear and non-linear operators, as in a scheme recently adopted for program evolution as an aid to producing human-interpretable results [65]. This increased interpretability is an added bonus from the perspective of representation-building.

In the domain of Boolean formulae, Holman has developed an "elegant normal form" (ENF) [41] along these lines which is both computationally efficient to derive and heuristically effective. Holman reports for instance on experiments involving randomly generated Boolean formulae with between one hundred and five hundred literals, where 99% of the formulae required fewer than 10,000 atomic operations to reduce to ENF, and retained fewer than 2% of their original literals. Formulae in ENF use the basis $\{AND, OR, NOT\}$, as in the previous chapter. To define ENF, it is convenient to 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 4.1. 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, as defined by Holman, if all of the following hold:

1. Negation appears only in literals.
2. Levels of conjunction and disjunction alternate.

[1] For clarity, these definitions are slightly different from those presented in [41].

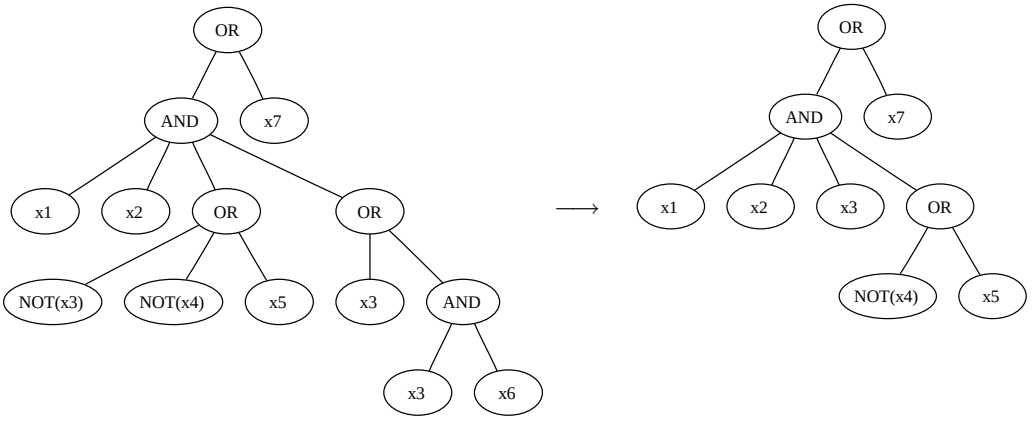

Figure 4.1: A redundant Boolean formula (left) and its equivalent in hierarchical normal form (right).

3. No conjunction or disjunction has both a literal and its negation, or multiple copies of the same literal, as children.

4. No branch set contains a literal and its negation.

5. The intersection of all of the children of any disjunction's guard sets is empty.

6. The intersection of any conjunction's guard set and dominant set is empty.

Thus, the formula in Figure 4.1 on the left is not in ENF, because the intersection of the *OR* node in the lower right's children's guard sets is non-empty – it contains x_3 (condition 5). Holman [41] presents an algorithm reducing any formula to ENF which executes in $O(n \cdot min(n, k))$, where n is the arity of the space, and k is the number of literals in the formula. Essentially, this procedure consists of a set of eight reduction rules which are executed iteratively over the entire formula until no further reductions are possible. I have extended Holman's ENF and the corresponding reduction procedure to obey the following additional constraints:

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

8. No node's guard set is a subset of any of 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 $S_1 \cup S_2$.

The rationale behind the the first of these additions of is merely symmetry – there appears to be no reason to reduce redundancy in conjunctions-of-disjunctions (item 5)

and not in disjunctions-of-conjunctions (item 7). The latter two additions were chosen based on experimentation with small hand-crafted formulae. Empirically, their addition can reduce random formulae further than ENF alone about 7% of the time, by an average of around three literals, for formulae with one hundred literals.[2] From a computational complexity standpoint, the reductions needed to implement items 8 and 9 (searching all pairs of siblings for matching literals and their negations, then searching for subsets) add an additional multiplicative term which is quadratic in the maximum arity of any conjunction or disjunction. Empirically however, the impact on runtimes relative to the reduction to ENF is negligible.[3]

4.2 Semantic vs. Syntactic Distance Revisited

Now that we have defined a hierarchical normal form and reduction procedure for Boolean formulae, we can experimentally study its effectiveness at eliminating redundancy and aligning syntactic and semantic variation.

First, consider the last experiment run in the previous chapter, on the global relationship between semantic and syntactic distance, based the pairwise distances between five thousand random formulae. This experiment is now repeated on the same formulae, reduced to HNF. 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). The results are shown in Figure 4.2.

In contrast to the case of unreduced formulae (Figure 3.4), the results for both arities show a marked correlation between semantic and syntactic distance for small semantic distances (signified by the diagonal sloping of the distribution towards the origin). Syntactic distances are now being computed primarily 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.

Also considered in the previous section was the behavioral distribution of the syntactic neighbors of random formulae. Certainly the syntactic neighbors of formulae in HNF might be expected to be somewhat closer to their source than for random formulae, but we can do better. The neighborhood enumeration and neighborhood reduction steps of MOSES

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

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

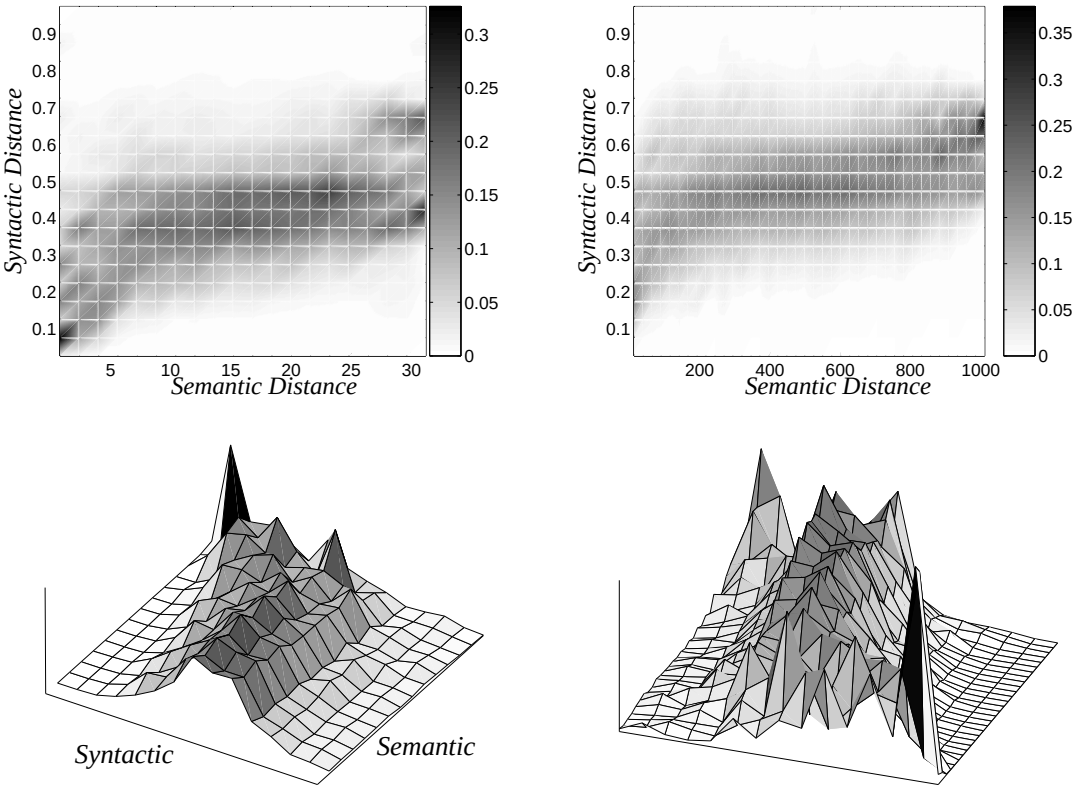

Figure 4.2: Pairwise distribution for edit distance in program space (syntactic) vs. Hamming distance in behavior space (semantic) for random formulae with arities five and ten, after reduction to hierarchical normal form. The lower surface plots are details of the figures above them, with the origin located in the far corner.

(see Section 2.1.3) may be used to create a dynamically sized neighborhood that will hopefully consist of programs that are not only closer to their source, but also behaviorally diversified (based on the neighborhood reduction step).

To accomplish this, we must first define a set of atomic perturbations. Because the transformation to HNF typically reduces program size, we can consider a larger atomic set (the neighborhood size will be bounded by the product of the atomic set size and the program size). Here we will exploit domain knowledge to create knobs leading to greater orthogonality. Consider a knob corresponding to insertion of a literal; inserting the literal's negation as a sibling will always lead to a contradiction or a tautology. Similarly, inserting a copy of a literal as its own sibling is completely redundant. It is thus more effective to create, for every junctor (*AND* or *OR*) in a formula, at most one knob for every literal in the variable set, with three possible settings (*present*, *negated*, and *absent*).

Beyond inserting, negating, or removing a literal as a child of an existing junctor, we can consider introducing a new junctor (with children) in between any existing junctor and literal in the formula, as a child of an existing junctor, or above the root. Here again domain knowledge (of the evaluation of Boolean formulae) may be applied. Introducing a junctor of the same type as a child of an existing junctor (e.g., and *AND* node directly under an *AND* node) is entirely redundant; only opposing junctors need be considered. For each of these existing junctors and the new junctor above the root, a new (opposing) child junctor with literal children may also be considered (according to the same scheme). We can also consider flipping an existing junctor between *AND* and *OR*.

Given a formula in HNF with j junctors and k literals of arity n, how many knobs are possible?

1. Every junctor may have n knobs corresponding to literal children ($j \cdot n$ knobs).
2. Every junctor may have a new opposing junctor inserted underneath it, which may have n literal children ($j \cdot n$ knobs).
3. Every junctor may be flipped (j knobs).
4. An opposing junctor may be inserted above the root, which may have n knobs corresponding to literal children (n knobs).
5. The new junctor inserted above the root may have an opposing junctor inserted underneath it, which may have n literal children (n knobs).
6. The new junctor inserted above the root may be flipped (1 knob).
7. Every literal may have an opposing junctor inserted above it, which may have additional $n - 1$ knobs corresponding to literal children ($(n - 1) \cdot k$ knobs).

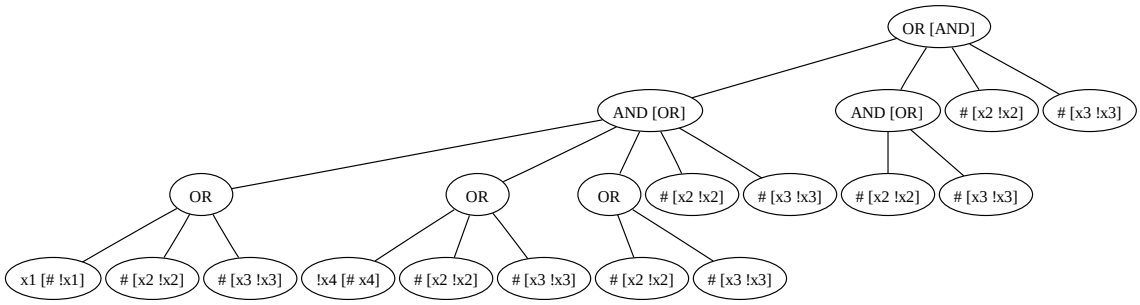

Figure 4.3: The representation built by MOSES for the formula $AND(x_1\ not(x_4))$.

This gives a total of $n \cdot (2j + k + 2) + j - k + 1$ possible knobs. The important thing to note is that this quantity grows with the product of the size of the formula and the arity of the space (as does the size of the syntactic neighborhoods studied in the previous chapter). To heuristically eliminate redundant knobs, each possible knob setting is experimentally set (with all other knobs kept in their default positions), and the resultant program reduced to normal form. If the number of symbols in the reduced program is less than in the original program, then the knob is eliminated as redundant. This is because any large reduction in program length can be expressed as a set of smaller length reductions. Of course, this is a heuristic; a transformation that had a redundant effect on its own might lead to a unique program if applied in conjunction with other transformations. Over-representation might be decreased still further by retaining only one of a set of knobs transforming the exemplar program to the same program, but at the cost of decreased expressiveness.

As an example, the formula $AND(x_1, not(x_4))$ in quaternary formula space will have 24 possible knobs; this number is reduced to 17 after heuristic knob reduction. The reduced representation is shown in Figure 4.3. Knobs are depicted with their default values leftmost, followed by a bracketed list containing other possible settings. A '#' denotes absence from the formula as a possible knob setting, and a '!' prefix indicates negation.

Assume for the moment 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) and programs grow larger. 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 all of the transformations described above involving the set of n literals, and 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

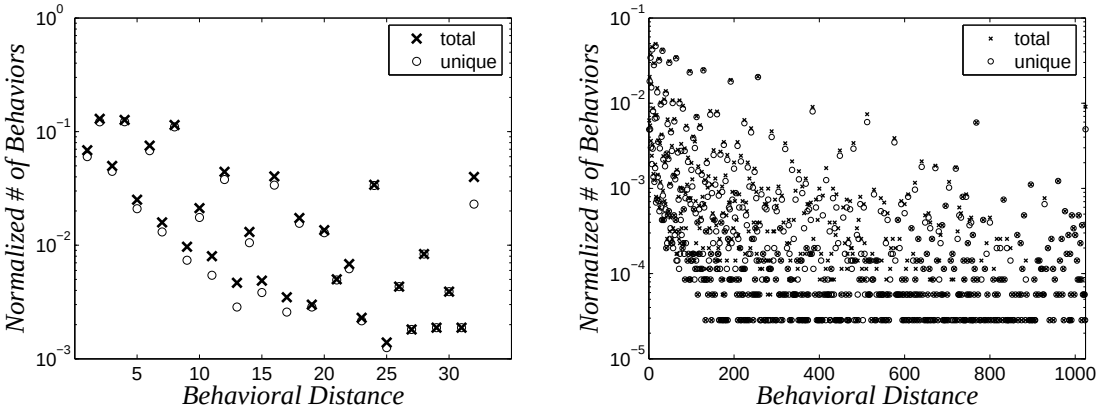

Figure 4.4: Distribution of behaviors and unique behaviors as a proportion of total neighborhood size for random formulae with arities five (left) and ten (right), reduced to hierarchical normal form. Note the logarithmic scale on the y axis.

in HNF 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 will not be affected. The motivation behind sampling from this particular 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).

Now that the neighborhood structure used by MOSES for program evolution has been described, we can study it in comparison to the syntactically constructed neighborhood structure from the last chapter. Figure 4.4 (comparable to Figure 3.3 in the previous chapter) shows the distribution of behaviors and unique behaviors among the neighbors of random formulae, as a proportion of the total neighborhood size (not including neighbors which are behaviorally identical to the source formula). Results for formulae with arity five are on the left and with arity ten on the right. The data are based on one thousand random formulae with one hundred literals (then reduced to HNF) for each arity, generated as in the last chapter.

A number of behavioral differences between the syntactic and representation-building neighborhood structures are worth noting. One is that syntactic variations of large random formulae typically have no behavioral effect (96% of the time for arity five, and 91% for arity ten). The opposite is true however for the representation-building approach, with only 10% of neighbors exhibiting behavioral neutrality (for both arities tested). 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.

4.3 Discussion

The conclusion I draw from these results is that hierarchical normal forms can lead to more effective program evolution. One might discount this by arguing that large random formulae are not a good model for the programs on which evolutionary search typically operates. However, Langdon and Poli [54, 56] have argued and demonstrated that a tendency of evolved programs to exhibit progressively greater redundancy over time (i.e., "bloat") is a generic consequence of search through program-space. It will manifest itself in any incremental search procedure (genetic programming, local search, etc.) not carefully designed to prevent it. The situation for evolved programs might thus be even more dire than for random ones.

These results are a nice illustration of a general point made by Baum and Durdanovic in the context of evolving collections of agents in a simple artificial economy [6]. So-called "strong weak methods" [2] such as evolutionary computation attempt to modify their search behavior via empirical credit assignment. As Baum and Durdanovic have pointed out however, many credit assignment schemes implicitly assign credit to elements that do not advance the overall effectiveness of the systems (in terms of explicitly stated problem-solving goals). This is akin to an economic "tragedy of the commons". Such systems cannot scale, because they become inundated by non-productive elements (which are combinatorially easier to discover than elements that actually contribute to solving hard problems). Reduction to normal form should thus succeed because it facilitates empirical credit assignment to those elements of program with semantic meaning (i.e., that might actually help solve a given problem).

Chapter 5

On Problem Difficulty

The previous two chapters have developed an understanding of program spaces and representation-building in program learning problems. We can now proceed towards a deeper understanding problem difficulty for program evolution. This will allow us to flesh out the details of how MOSES will search within and across regions of program space, and how to decide when new demes should be created.

5.1 Program-Level Difficulty

The first question to consider is search within a particular deme, where a competent adaptive optimization algorithm is applied over a fixed-length representation (a set of knobs). A well-developed theory of problem difficulty for evolutionary optimization that applies in this context (the "design approach" [28]) revolves around a model of the required sample size that must be maintained over time (N, the population size) and the expected number of resampling cycles (G, the number of generations) required until convergence. Assuming the sample size has be correctly calculated, one expects the convergence point to be a global optimum, and can thus bound the expected number of scoring function evaluations required by $N \cdot G$.

The computation of N is derived from considerations of problem subsolution structure – recall the discussion in Chapter 1 of interaction and separability between variables in an encoding. The requirements for correct optimization and integration of partial solutions, or *building blocks*, may be decomposed into:[1]

1. **Sufficient initial diversity** [31] – N must be large enough to for the initial sample to contain all possible subsolutions with high probability. This minimum sample size depends on the size of largest subsolution we want (k). Note that this is assuming

[1]Only very brief summaries of the relevant models and rationale are given; see the references for full descriptions and derivations.

a recombinative approach such as the BOA or hBOA where the dominant source of novelty as search progresses is recombination of existing subsolutions (rather than the mutative creation of new subsolutions). Generally speaking, the minimal N required here grows exponentially with k, and is dominated by the lower bound imposed on N by correct decision-making.

2. **Correct decision-making** [29, 35] – At every selective step, N must be large enough for the superior subsolutions to be (statistically) distinguishable. This requirement leads to a minimal N which is $O(2^k\sqrt{n})$, where n is the number of problem variables (assumed here to be binary). The dependence on n is due to "collateral noise", which is essentially variance which arises because we can only directly compute the scores of entire solutions (assignments to all n variables), not subsolutions.

3. **Maintaining diversity** [66, 113] – as search progresses, the noise introduced though finite sampling may cause important subsolutions present at the beginning of the search to be lost (this is also referred to as *drift*). How large a population (N) must be maintained to avoid this depends on the strength of the selection pressure and, asymptotically, on the relative scaling of the subproblems (i.e., how they are weighted in the scoring function). In the easy case of uniform scaling all subproblems may be optimized in parallel, and N is $O(\sqrt{n})$. For exponential scaling, the subproblems will be optimized sequentially from most to least important, and the bound becomes $O(n)$.

4. **Learning the correct problem decomposition** [80, 83] – the bounds given above all assume that the correct problem decomposition is used, otherwise the required population size may clearly grow exponentially with the problem size for hard problems. For the BOA, the required population size to learn a correct problem decomposition may be computed by considering the sample size needed to distinguish actual dependencies from spurious ones introduced by noise and selection. This is $O(2^k \cdot n^{1.05})$, as shown in [80].[2]

The $O(2^k \cdot n^{1.05})$ population sizing requirement for learning the correct problem decomposition dominates here. We clearly know n, but what about k? The easiest thing to do is treat k as a constant across a given program evolution problem, the approach currently taken in MOSES. However, it is still unknown. One approach would be to estimate k based on observing the critical population size required over repeated runs on small-scale program optimization problems with varying population size (e.g., via bisection). This would allow curve-fitting to $c \cdot n^{1.05}$ or $c_1 \cdot n^{1.05} + c_2$, which could be use to compute the population size for large problems. Another approach would be to use the population sizing technique

[2] A more recent derivation [124] gives $O(2^k n \cdot log(n))$.

of the parameterless hBOA [89]. This involves maintaining a sequence of exponentially increasingly larger populations; the final population size might be useful in formulating a guess for the further hBOA runs (i.e., when creating any new "offspring" demes). The approach currently taken in MOSES is the simplest possible; a user-defined parameter c is taken, and populations are sized according to $c \cdot n^{1.05}$.

What about G, the number of generations needed? We typically have no way of knowing, at the program level, when an optimum within some given region has been reached. If we are using the BOA, we can wait until the population converges on a single solution, hopefully the optimum. The hBOA, however, uses *niching* to maintain diverse promising solutions simultaneously, and may never converge. Fortunately, the methodology of the parameterless hBOA [89] can offer guidance regarding when a given population has "played itself out"; namely after a number of generations equal to the number of bits in the solution encoding. This recommendation is given based on practical experience, although it agrees well with convergence theory for the BOA (without niching), which predicts that this will be between $O(\sqrt{n})$ and $O(n)$ (see [83, 80]); one would expect each semi-separate niche in the hBOA to converge at least this quickly.

An additional guide that may be used is to consider the number of generations without improvement; according to the same model, the convergence time given no improvements (i.e., the *takeover time*) is expected to be $O(\sqrt{n})$. For the hBOA however, the window size for restricted tournament replacement (RTR) may be additionally taken into account to provide a tighter bound. RTR is used in the hBOA to integrate new instances into the population. For every new instance generated, a random sampling of w existing instances is taken, and compared against the new instance to find the one most similar to it (based on Hamming distance). The new solution replaces this instance in the population if its score is higher (if the score are identical, I use a coin flip to decide). The population (of size N) is thus implicitly subdivided into w (overlapping) niches that may converge in parallel. Accordingly, the hBOA search is terminated if no improvements have been observed (across the entire population) in the last $\lceil \sqrt{N/w} \rceil$ generations.

Problem difficult on the program level is thus a function of n and k, which are determined by the interplay of program size, program evaluation semantics, and representation-building.

5.2 Deme-Level Difficulty

> "Redundancy is necessarily neutral, but neutrality is not necessarily redundant."
> *Marc Toussaint* [115]

Demes with different representations exist in MOSES based on the hypothesis that problem decomposition will be easier within properly defined subspaces of programs than

across the entire space. But the use of demes raises additional issues. From an algorithmic perspective, there are questions of exactly how to decide when to create a new deme, and how to choose between demes. From a problem-solving perspective, there is the question of which problem classes are solvable by program evolution with demes based on representation-building. We will need to understand what makes problems easy or difficult on the deme level. Some definitions are called for.

A program that has the highest score of all programs in the region centered around it (for some given representation-building mechanism) may be called a *deme-local optimum*. Compare this to the notion of a local optimum, a program with the highest score of all programs in its neighborhood. Neighborhood size is typically a linear function of program size, whereas the size of a region is typically exponential; hence there may be no deme-local optima in a search space containing many local optima (although conversely, every deme-local optimum is also a local optimum).

A program y is *strictly deme-reachable* from some program x if there exists a sequence of programs $p_1, p_2, ..., p_n$ such that: (1) p_1 is the unique deme-local optimum in the region centered around x; (2) y is contained within the region centered around p_n; and (3) for $1 < i \leq n$, p_i is the unique deme-local optimum in the region centered around p_{i-1}. The *strict deme-distance* from x to y is the length of the shortest such sequence; if y is contained within the region centered around x, the strict deme-distance from x to y is zero.

A problem may be considered *strictly deme-separable* if any of its global optima are strictly deme-reachable from the initial (empty) program. Note that there must be a unique strict deme-distance n for all such global optima, if any exist. In this case, MOSES can hopefully proceed directly to a global optimum, although it may require the solution of a sequence of $n+1$ difficult global optimization problems (i.e., by the hBOA) along the way.[3]

Unfortunately, assuming strict deme-separability is unwarranted; a basic objection is the prevalence of *neutrality*[4] or near-neutrality in many program spaces [56, 75]. This is in consonance with the over-representation discussed and studied in previous chapters. Reduction to normal form will eliminate (most) redundancy, but this is only one form of neutrality. Thus, we will not have a unique chain of strictly dominant solutions leading to a global optimum, but rather a large ramifying network, with possibly very few branches leading to global optima. In other words, after running sufficient generations of the hBOA within a fixed region, we will probably be faced with a large number of programs sharing the highest score in the deme, or nearly so.

[3]Throughout the discussion in this section, I assume that whatever optimization algorithm is applied to search within regions is adequately configured and provided with sufficient computational resources to effectively and reliably discover optima.

[4]Neutrality most simply refers to search spaces where a large number of solutions have the same score (extreme neutrality makes gradient-based search impossible). A stronger sort of neutrality occurs when large sets to distinct solutions have the same behavior.

The definition of strict deme-reachability may be relaxed by dropping the uniqueness requirement on deme-local optima, leading to a general notion of (non-strict) deme-reachability, and an analogous general measure of deme-distance. A problem may be considered *deme-separable* if any of its global optima are deme-reachable from the initial program. Note that there may be many such deme-reachable global optima scattered at different deme-distances from the initial program.

If we were to create demes corresponding to the regions centered around all optima uncovered in search (assuming we could find them all), and recursively create new demes for all optima discovered in these regions, etc., the solution of (non-strictly) deme-separable problems could certainly be guaranteed. Unfortunately, this is computationally intractable for most problems of interest. A simple knowledge-free solution is to take some kind of random walk over the network of regions defined by the optima discovered by the underlying optimization algorithm (a so-called "neutral walk. [44, 4]). However, this ignores useful biases that can potentially winnow down the search space (based on the assumptions set out in Chapter 1). Also, it is unclear whether problems will even be non-strictly deme-separable!

To preferentially distinguish between potential exemplars for new demes, and between competing demes in general, I posit the following heuristics:

1. If one program *behaviorally dominates* another, the dominated program need not be considered.

2. If multiple programs are *behaviorally equivalent*, only the shortest programs (e.g., measured by the number of discrete symbols) need be considered.

Recall that behavioral dominance and equivalence are based on behavioral decomposability (described in chapter 1), and are domain dependent. For Boolean formula learning for instance, formulae that compute the same function are behaviorally equivalent. One formula dominates another if it has a higher score *and* computes a function that matches the target for all input combinations that the subordinate formula's function matches. So for example, if the target function's truth table is 0101, then a formula computing the truth table 1101 dominates a formula computing 1111, but not one computing 0010, even though the score for 0010, 1, is lower than the score for 1111, 2. Note that is does not imply that a formula computing 0010 dominates one computing 1101 (since the ordering is only *partial*).

The partial ordering defined by behavioral dominance is a generalization of the notion of Pareto-optimality in multi-objective optimization (given multiple scoring functions, a solution is Pareto-optimal if no other solution both scores as highly as it on all of the functions *and* scores higher than it on at least one of them). This is because any scoring function that can be decomposed into multiple objectives implies a partial ordering, whereas a partial ordering based on behavioral dominance may be defined without explicitly decomposing the

scoring function. Krawiec [51] discusses further the notion of behavioral dominance ("hypothesis outranking" in his terminology). He also argues the need for the incomparability of hypotheses, and gives some experimental results in an evolutionary learning context (based on genetic programming).

The current approach to deme management in MOSES is based on running the hBOA on a deme without interruption until its termination (see Section 5.1). All programs in the final population are considered in competition with each other and with extant demes for insertion into the metapopulation (as exemplars for the formation of new demes), according to the heuristics given above. To select a new deme for applying optimization, the one whose exemplar program has the highest score is chosen. In the case of a tie, which may be quite likely in domains such as Boolean function learning with a discrete number of possible scores, one of the tied programs is chosen by roulette selection weighted geometrically on program size (number of symbols) with a factor of 0.5. For example, if there are four tied programs with sizes of 10, 11, 13, and 13, their respective probabilities of selection will be 0.57 $(8/(8+4+1+1))$, 0.29 $(4/(8+4+1+1))$, 0.07 $(1/(8+4+1+1))$, and 0.07.

A previous approach which was discarded used a finer-grained parallelism, with selected demes run for only a single generation at a time. However, this was prohibitively expensive in terms of memory usage, as many (potentially very large) populations had to be maintained simultaneously. The current approach may be implemented lazily so that the initial population of a deme is not created until the deme is chosen for hBOA optimization. Since the hBOA is run on a deme without interruption until its completion, only one complete population needs to be maintained in memory at a time. Conceptually however, demes are still processed according to a "parallel terraced scan" [95].

Greater stochasticity on the deme level does not appear to be necessary for many program search problems, but might increase robustness on pathological problems. Specifically, consider a hypothetical problem which is "deceptive" on the deme level with respect to the heuristics given above (e.g., finding the global optimum requires sometimes ignoring the most promising and shortest solutions found within regions). In this case, a bias that allowed less promising directions to sometimes be selected could be partially effective. Note however that such a "globally deceptive" problem would appear to be very difficult for any search mechanism without some initial bias towards the correct region of program space (e.g., overall programmatic organization) – evaluating the score of a region of program space is much more expensive than evaluating the score of a single program.

Whether or not a given problem will be deme-separable depends on the particular programmatic structures and representation-building primitives that are chosen. In a trivial sense, any problem can be made deme-separable by iteratively expanding the region size to include more and more of the overall program space. In fact, this could be the basis

for an iterative deepening approach based on MOSES which would be quite robust, but potentially extremely expensive.

5.3 Discussion

In this chapter I have proposed a novel bipartite conception of problem difficulty for program learning. Existing (unipartite) models are typically based on generalizations of Holland's schema theory [40], where schema correspond to various syntactically defined fragments of program trees, *quantified over the entire program space* (see Poli and Langdon [56] for a review of these efforts). Advances are currently being made according to two approaches in particular.

The first of these two approaches is adapting the facetwise design approach to problem difficulty outlined in Section 5.1 to directly model programs, where subsolutions (schema) consist of contiguous tree fragments that may appear anywhere in a program. Population sizing relations have been derived for initial diversity [102] and decision making [101]. The second approach is based on a mathematically complex general exact schema theorem [92, 56], equivalent to a Markov chain model, which encompasses many syntactic recombination operations on strings and trees as special cases (including standard crossover for genetic algorithms and genetic programming).

Both approaches have already begun to deliver useful insights into the operation of syntactically guided program evolution procedures, and will most likely continue to do so as they are further developed. However, they are both formulated to quantify schema over the entire program space, and are both syntactic. 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.[5] 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 program semantics). An interesting topic for further study would be to attempt to quantify this pessimism, and adjust the model(s) accordingly.

[5]Technically x and y are more closely linked in the first schema (being less likely to be disrupted by genetic programming's crossover operation, for instance), while x and z are more closely linked in the second. But they are the same in the sense that swapping between them will never change a program's score.

Chapter 6

Evaluation

This chapter analyzes the results of applying MOSES to problems in three domains: Boolean formulae, the JOIN expression mechanism, and supervised classification. Comparative results are given for all problems (see Table 6.1).

The domain of Boolean formulae was chosen based on its fundamental nature (see Chapter 3), potential for difficulty, and the existence of complex, principled normalization rules (Holman's elegant normal form [41] and the extensions presented in Chapter 4). Furthermore, problems such as multiplexer and even-parity learning may be easily tuned to be more or less difficult by adjusting their arity, to study scalability.

The ORDER expression mechanism [79] was designed to study subsolution acquisition and recombination. Because the expression mechanism is very simple, particular aspects of problem difficulty may be studied in isolation. It is very easy to test scalability on these problems as well.

The next question is whether or not the gains realized on synthetic program evolution problems will transfer to real-world problems of interest, as they have in the bit-string domain for the BOA and hBOA [81, 97]. To begin to answer this, I have applied MOSES to challenging problems in computational biology that have recently been studied using

Table 6.1: The domains that MOSES is tested on in this chapter, and other methods applied to these domains which it is compared against.

Domain	Other Methods Applied
Boolean formulae	genetic programming [48], evolutionary programming (i.e., stochastic hillclimbing) [15]
JOIN expression mechanism	genetic programming and extended compact genetic programming [100], probabilistic incremental program evolution [76]
Supervised classification	genetic programming and support vector machines [25, 26]

supervised categorization: (1) predicting chronic fatigue syndrome (CFS) based on single nucleotide polymorphisms (SNPs, the smallest possible variations in the genome) [26]; (2) distinguishing between types of lymphoma based on gene expression data [108, 112, 25]; and (3) between young and old human brains [62, 25], also based on gene expression data.

These data are gathered with microarray profiling technology, a means of measuring the expression levels across thousands of genes (and of detecting SNPs); a gene's expression level is a rough indicator of its activity level (i.e., translation into protein). Gene expression profiling thus generates, for some particular cell type in a patient, a (noisy) high-dimensional numerical vector profiling activity in the particular genes chosen to be sampled.

The CFS dataset is of interest because the only technique to yield statistically significant results to date is simple enumerative search, outperforming support vector machines (SVMs) [26]. The lymphoma and aging brain datasets are of interest because while SVMs yield superior performance, the classifiers themselves are often opaque. This has led researchers to use methods such as genetic programming (GP) which, while delivering poorer results by the numbers, can give valuable insights into exactly how the classification is being performed. Furthermore, the powerful program simplification and representation-building mechanisms developed for Boolean formula learning may be leveraged here as well.

An important question to consider is how to compare learning results across potentially very different methods in an informative way. A common methodology is to consider the quality of the results achieved as a function of the number of calls to the scoring function. This has two problems. Firstly, there may be sources of heavy computational workload not captured by the metric (e.g. MOSES's model-building and representation-building, neither of which depend on calling the scoring function). Secondly, not all calls to the scoring function are of equal cost, particularly when dealing with program evolution. Typically, larger programs execute more slowly than smaller ones. This is compounded by the phenomenon of bloat in GP and other methods, which causes programs to rapidly grow in size without any corresponding improvement in performance.

MOSES does not suffer from bloat, as we shall see. However, the cost of model-building rises about quadratically with the size of the representation, which typically grows linearly with the size of the exemplar programs being used. The real cost of any given run thus depends not only on the number of evaluations, but also, as a first approximation, on the size of the best solution found by MOSES (see Sections 6.4 and 6.5).

Beyond these difficulties, the central quantity reported for program evolution experiments, *computational effort*, is problematic (cf. [63]). Computational effort, computed over some set of independent runs, is the minimal number of evaluations needed to find a global optimum with some specified probability (typically 0.99). This minimum is over all possible combinations of number of generations (up to the maximum number of generations carried out in a run) and number of independent runs (see Koza [48] for more details). An issue

Table 6.2: All of the tunable parameters for MOSES and their setting across all problems which MOSES has been tested on.

Description	Default	Exceptions
Population size (N) is $c \cdot n^{1.05}$ (derived in [80]), where n is the number of bits in the encoding. This quantifies the degree of interaction among problem variables (i.e., the higher the degree, the higher c must be).	$c = 5$	For the artificial ant $c = 30$, based on preliminary experiments and the analysis in [55] indicating a very high degree of interaction.
The window size (w) for restricted tournament replacement in the hBOA, which implements niching.	$w = min(0.05N, n)$, recommended in [86]	None.
The tournament size (selection pressure) used for model-building in the hBOA.	2, recommended in [80]	None.
The complexity penalty for model-building in the hBOA (to avoid overfitting).	$log_2(N)$, derived in [80]	None.
A deme is terminated after t_1 generations of hBOA, or t_2 generations with no improvement in the best score.	$t_1 = n$, $t_2 = \lceil \sqrt{N/w} \rceil$	None.
The maximal number of scoring function evaluations allowed per run (if a global optimum is found, the run is terminated immediately).	∞	$1,000,000$ evaluations for 5-parity, $8,000,000$ for 6-parity, and $100,000$ for supervised classification.

with relying on computational effort to distinguish between the performance of algorithms is that it is not usually computationally feasible to compute statistical significance; each set of independent runs used to calculate the measure contributes a single point, and typically only one set of runs is used.

Furthermore, it has been observed empirically that under some circumstances the computational effort measure attributes better performance to techniques with higher variance but lower mean scores [63], a misleading judgment to say the least. An additional problem is that the measure of course cannot be used at all in cases where the global optimum is not known. When a technique solves a problem reliably with low variance, the computational effort figure will typically be about the *maximum* number of evaluations taken in any of the runs.[1] With these caveats, I do present computational effort figures when appropriate, in order to facilitate comparison with other methods for which it may be the only data available. However, they are not presented exclusively, and, when possible, are supplemented with mean-based measures for which statistics may be calculated.

The goal of these experiments is not to show that MOSES will always beat other search methods. MOSES is a general procedure; when a hand-tuned problem specific approach may be brought to bear, it will nearly always trump general methods. Rather, the goal is to demonstrate scalability and robustness. Therefore, an attempt has been made to keep parameter settings as constant as possible across problems, rather than tune them to squeeze out the best numbers possible. All parameter settings for MOSES are listed in Table 6.2. The fact that MOSES can achieve strong results with very little parameter-tuning is quite significant given that commercial machine-learning systems often resort to performing multiple runs in parallel of methods such as GP and SVMs with different random configurations, in order to achieve reliable performance.[2]

6.1 Boolean Formulae

Experimental results are presented in this section for three sets of Boolean function learning problems: even-parity, multiplexer, and hierarchically composed parity-multiplexer functions.

6.1.1 Even-Parity Functions

Parity formulae with the basis $\{AND, OR, NOT\}$ are very difficult for evolutionary systems to learn. This appears to be due to two main properties. Firstly, even-parity formulae are extremely rare in formula-space (their minimal formula size grows quadratically with the

[1] To be precise, it will be the length of the run that is slower than all but 1% of all other runs, or the length of the longest run if under one hundred runs are performed.

[2] Two examples of such systems are Discipulus, from RML Technologies (*http://www.aimlearning.com/*) and ArrayGenius, from by Biomind LLC (*http://www.biomind.com/*).

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 very well-studied from both a theoretical and an experimental perspective [119, 56, 48, 96].

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 [49]), or simply using a different basis (containing the equality or the exclusive-or function, for instance). This eliminates the first source of difficulty, as the minimal formula length will now grow linearly with the arity. A method introduced by Poli and Page, sub-machine code genetic programming with smooth uniform crossover [93], 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, what is of theoretical 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 analyzed herein for the original formulation ([93] summarizes the results for transformed problem variants as well).

Fifty independent runs of MOSES were executed for parities 3, 4, and 5. Ten independent runs were executed for 6-parity (due to the high computational cost). For 3-parity and 4-parity, an optimal solution was found in all runs. For 5-parity, runs were terminated after one million fitness evaluations and for 6-parity after eight million fitness evaluations. To study the effects of probabilistic model-building on the performance of MOSES (as opposed to representation-building and deme management), equivalent runs were carried out with a fixed univariate model (i.e., assuming no interactions between variables). This configuration will henceforth be referred to as "univariate MOSES" (uMOSES).

Computational effort and overall success rates are shown in comparison to other methods in Tables 6.3 and 6.4, respectively. Note that a non-zero success rate for 6-parity might be achieved for evolutionary programming or genetic programming with larger populations and longer runs (evolutionary programming used a population of 500 and 250 generations [15], while genetic programming used a population of 16,000 and 51 generations [49]).

MOSES appears to achieve the best performance of all techniques listed. However, its scaling still appears to be at least exponential. An analysis of the problem reveals this to be unsurprising, and in fact unavoidable without dramatically changing the space and search biases. Recall that the cost of a model-building approach will grow exponentially

Table 6.3: The computational effort required to find optimal solutions to even-parity problems for various techniques with $p = .99$.

Technique	Computational Effort			
	3-parity	4-parity	5-parity	6-parity
Univariate MOSES	6,151	73,977	2,402,523	342,280,092
Evolutionary programming [15]	28,500	181,500	2,100,000	no solutions
Genetic programming [49]	96,000	384,000	6,528,000	no solutions
MOSES	5,112	72,384	1,581,212	100,490,013

Table 6.4: The success rate for even-parity problems for various techniques.

Technique	Success Rate			
	3-parity	4-parity	5-parity	6-parity
Univariate MOSES	100%	100%	86%	10%
Evolutionary programming [15]	100%	84%	22%	0%
Genetic programming [49]	100%	100%	46%	0%
MOSES	100%	100%	96%	30%

with the degree of interaction between variables; because of the symmetries in parity (see above), this will be at least equal to the arity.

Let's derive a very rough theoretical estimate of ideal scaling on parity as a function of the arity n, in terms of the number of evaluations required. Consider that the size of a minimal perfect solution will grow with $O(n^2)$. The size of the largest representations built by MOSES will thus be $O(n^3)$ (the product of the arity and the program size, as derived in Chapter 4). The number of generations may be lower-bounded by $O(\sqrt{n^3}) = O(n^{1.5})$, giving $O(n^{4.5}2^k)$ evaluations, where k is the degree of interaction. The depth of a parity formula must be at least $log_2(n)$, otherwise its overall size will grow exponentially (a proof may be found in [119]). The degree of interaction depends on this, because the effects of a literal may be altered by nodes on the path to the root and their literal children (see Chapters 3 and 4). The aforementioned symmetries in parity necessitate multiplying this by n (for the kinds of representations built by MOSES), to obtain a total asymptotic degree of interaction of $n \cdot log(n)$. The number of evaluations required as a function of the arity will thus be expected to grow with about $n^{4.5}2^{n \cdot log_2(n)} = n^{n+4.5}$. This is plotted alongside the computational effort figures for MOSES is Figure 6.1.1, indicating rough agreement. A more rigorous future model combined with more accurate experimental results (more runs, particularly for 6-parity) could prove instructive.

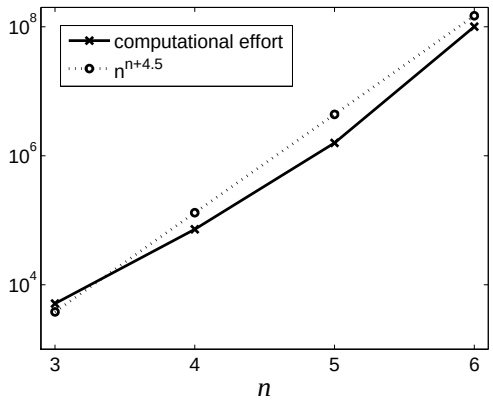

Figure 6.1: Scaling of MOSES on even-parity problems in terms of computational effort as a function of arity (n).

6.1.2 Multiplexer Functions

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. [48]). Restricting the basis to the familiar $\{AND,\ OR, NOT\}$ makes them somewhat more difficult.

To obtain comparative results for GP in this set of experiments, the lil-gp system was used (version 1.1., *http://garage.cse.msu.edu/software/lil-gp/*), with a population size of 4000. Following [49], tournament selection is used with a tournament size of 7, and the crossover rate is 90% (with the remainer of the population carried over without modification).[3] Computational effort and overall success rates based on 50 independent runs for each configuration are shown in Tables 6.5 and 6.6, respectively.

A significant improvement may be observed for both MOSES and uMOSES over GP. For the 11-multiplexer, the average number of evaluations to achieve a solution is $132,215$ evaluations with 95% confidence intervals of $\pm 13,698$, for MOSES. For uMOSES this is $153,258 \pm 15,403$, indicating that more experiments would be needed to confirm the hypothesis that model-building improved performance. The next problem to be studied will allow us to confidently make such statements, and also analyze the models built.

[3] A number of results for GP learning multiplexers are available in the literature ([48, 49], and others) with varying population sizes and selection methods. These results are qualitatively similar to those presented herein.

Table 6.5: The computational effort required to find optimal solutions to multiplexer problems for various techniques with $p = .99$.

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

Table 6.6: The success rate for multiplexer problems for various techniques.

Technique	Success Rate	
	6-multiplexer	11-multiplexer
Univariate MOSES (without *if*)	100%	100%
Genetic programming (with *if*)	100%	68%
Genetic programming (without *if*)	100%	24%
MOSES (without *if*)	100%	100%

6.1.3 Hierarchically Composed Parity-Multiplexer Functions

As a final Boolean formula problem, let's consider hierarchically composing parity and multiplexer functions (first studied in [11]). The outputs of a set of non-overlapping parity functions of arity k_1 will be used as the inputs to multiplexer with k_2 address inputs (instead of bits, the $k_2 + 2^{k_2}$ arguments will all be parity functions). The overall arity will thus be $k_1 \cdot (k_2 + 2^{k_2})$. Because this grows very quickly, only the case of $k_1 = 2$, $k_2 = 1$ will be considered here. As above, the basis $\{AND,\ OR, NOT\}$ will be used.

This problem is of interest because one would expect to be able to clearly analyze and demonstrate the benefits of model-building. Variables corresponding to the pairs of inputs combined to form parity might be expected to have the strongest interactions. It is possible to test this hypothesis in MOSES; in the representations constructed for Boolean formulae there are specific probabilistic model variables corresponding to the removal, insertion, and negation of the Boolean variables in the problem domain ($\{x_1, x_2, ..., x_n\}$) at various positions in the formula. We can aggregate data by source and destination for these variables over the course of multiple runs of MOSES to construct an n by n dependency matrix, with cell (i, j) containing the relative strength of the dependency from x_i to x_j.

150 runs of MOSES were aggregated to derive the dependency matrix shown in Figure 6.2. Frequencies are normalized to have a maximum of 1. The address is based on the first two inputs (i.e., $XOR(x_1, x_2)$), are the two values in the addressing space are

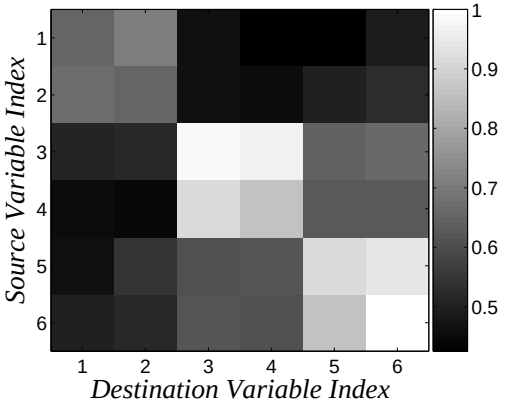

Figure 6.2: Pairwise interactions between formula variables for the hierarchically composed 2-parity-3-multiplexer problem as discovered by MOSES.

Table 6.7: The computational effort and average number of evaluations (with 95% confidence interval) required to find optimal solutions to the hierarchically composed 2-parity-3-multiplexer problem.

Technique	Computational Effort	Avg. # of Evaluations
Univariate MOSES	374,781	127,264±12,609
MOSES	218,093	105,184±6,399

$XOR(x_3, x_4)$ and $XOR(x_5, x_6)$. These results lead to some interesting observations. The hypothesis that the tightest dependencies would occur within parity functions has been confirmed. This can be seen as the three light-colored boxes along the upper-left to lower-right diagonal. Although there are dependencies uncovered between all of the variables, intraparity linkages are around twice as strong as the weakest dependencies. Additionally, there is a rough symmetry in the strengths of dependences. A more sophisticated analysis might be able to uncover clear directional dependencies occurring in particular contexts, as well as looking for hierarchical dependencies.

Based on this analysis, it is clear how representation-building is allowing the hBOA to learn the structure of the problem. The data in Table 6.7, based on 150 runs per method, demonstrate that this translates into a statistically significant improvement for MOSES in comparison to uMOSES; a 17% decrease in the average number of evaluations required to find an optimal solution leading to a 42% decrease in computational effort. In the future it would be interesting to apply MOSES and uMOSES to larger parity-multiplexer problems to study comparative scalability.

6.2 The JOIN Expression Mechanism

The program space for the JOIN expression mechanism [79] consists of a single binary function, *JOIN*, and an even-numbered 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 as for formulae. Specifically, the spaces are n and 2^n bits, respectively. 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 this parse, the ith bit of the output is 1. If $\bar{X}_i$ appears first, or neither terminal appears in the program, the ith bit is 0. For example, the $n = 4$ JOIN program

$$JOIN(JOIN(\bar{X}_4, X_2), JOIN(JOIN(X_3, X_3), X_4)) \tag{6.1}$$

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 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 quite effectively solve the bit-string equivalents of the problems that we will consider in this section. Instead we will "play along", and not exploit our knowledge of the expression mechanism. Thus, no reduction rules will be used in the JOIN domain.

Like *progn* and conditional operators in the artificial ant domain studied in Chapter 2 and 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 in Boolean formulae, a random sampling procedure is used. 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 knobs corresponding to insertions at a particular program location, a random JOIN tree with $k+1$ leaves is generated. One of these leaves is uniformly 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. Each of the new leaves will be a binary knob in the representation. In the experiments reported here, $k = [log_2(n)] - 1$ is used. For a tree with l leaves, this will lead to $2k(l-1)$ knobs, since all trees are binary. Additionally, removal of existing leaves may be considered (l more knobs), for a total of $2([log_2(n)] - 1)(l-1) + l = (2[log_2(n)] - 1)(l-1) + 1$ binary knobs. For the program shown on the left of Figure 6.3 with $n = 4$ for instance, the representation will

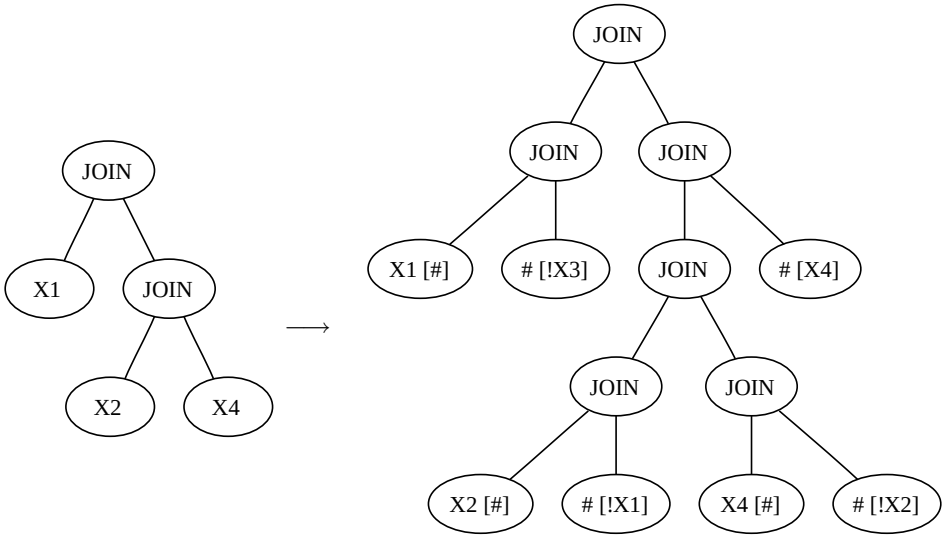

Figure 6.3: An exemplar program from the JOIN domain (left) and a corresponding representation built by MOSES (right).

contain seven knobs. One such possibility is shown on the right; nodes containing knobs have the default value listed first, followed by the other possible values in brackets; '#' means no node present, and a '!' prefix means complement.

6.2.1 ORDER

ORDER is the simplest scoring function for the JOIN domain; it simply sums the number of bits in the output (i.e., a OneMax). There are thus no interactions between variables. It is the an easy problem in the sense of not requiring any decomposition in order to be solved quickly and reliably. Previous studies of this problem [100, 76] have observed reasonable scaling performance for GP, extended compact genetic programming (ECGP), and probabilistic incremental program evolution (PIPE). MOSES and uMOSES were both executed for 50 independent runs with $n = 4, 8, 16, 32, 40,$ with a 100% success rate. The average number of evaluations required to find an optimal solution is shown in Figure 6.4.

For this problem, a univariate model seems to give slightly better performance as n grows. This effect appears to be 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. Based on theory this is not surprising; a simpler model will typically achieve better performance when a complex one is unnecessary. Furthermore, the best results reported in the literature [76] are for PIPE, a univariate algorithm. The results for MOSES and uMOSES are comparable to those presented in [100] for GP and ECGP, which both appear

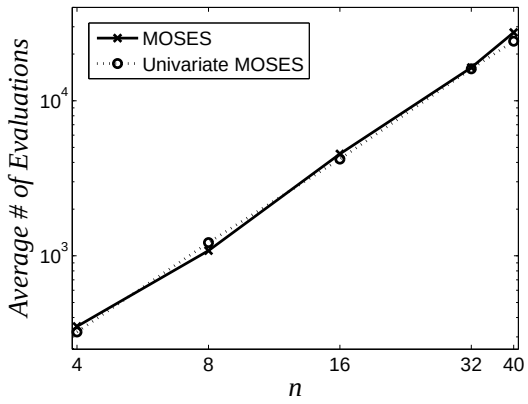

Figure 6.4: Scalability of MOSES and univariate MOSES or the ORDER problem. Note log-log scale.

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.

6.2.2 TRAP

TRAP computes a bit-string deceptive traps function on the output bits for the JOIN domain. This function is defined to be completely separable into subproblems of size k; within each group however, a trap function leads local optimizers away from the global optimum. That is, not only must all within-group subsolutions of order k be examined in order to solve the problem reliably, but local searchers will be consistently led *away* from optima, and naive recombination operators will consistently disrupt optimal subsolutions once discovered. For traps of order three and signal difference of 0.25 for example (as in [100] and as will be studied here), we can define the scoring function $f^3_{deceptive}$ as follows:

$$f^3_{deceptive}(x) = \sum_{i=1}^{n/3} trap(x_{3i-2} + x_{3i-1} + x_{3i}), \qquad trap(u) = \begin{cases} 0.75 & \text{if } u = 0 \\ 0.375 & \text{if } u = 1 \\ 0 & \text{if } u = 2 \\ 1 & \text{otherwise.} \end{cases} \tag{6.2}$$

Here, x_i is the ith bit of the string. Note that n should be a multiple of three.

Based on the theory of difficulty summarized in Chapter 5, TRAP should be harder to learn than ORDER because of the higher order of dependency (three versus one). In [100], no successful runs were reported for GP for $n > 24$, while the univariate PIPE appears to scale even more poorly [76]. MOSES and uMOSES were both executed for 50 independent

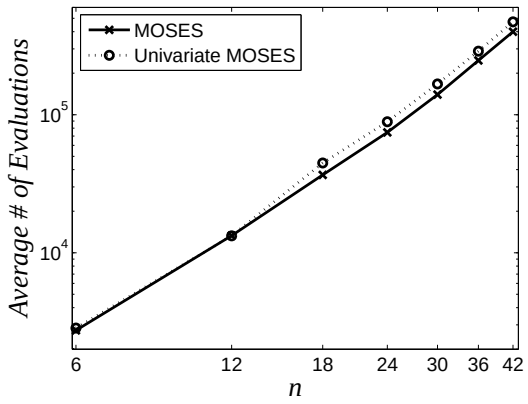

Figure 6.5: Scalability of MOSES and univariate MOSES or the TRAP problem. Note log-log scale.

runs with $n = 6, 12, 18, 24, 30, 36, 42,$ with a 100% success rate. The average number of evaluations required to find an optimal solutions is shown in Figure 6.5.

For this problem, a univariate model seems to perform *worse* as n grows, as expected. This effect is small, but statistically significant with 95% confidence for $n = 36$ ($288,653 \pm 22,314$ vs. $247,497 \pm 17,439$) and for $n = 42$ ($471,155 \pm 37,237$ vs. $399,335 \pm 31,963$). Furthermore, the variance is less for MOSES, as indicated by the narrower confidence intervals. The performance of MOSES appears to be roughly comparable to and perhaps slightly better than that of the ECGP on this problem [100], where the average number of evaluations is between 300,000 and 400,000 for $n = 36$, and between 400,000 and 500,000 for $n = 42$.[4]

What is somewhat surprising at first is why uMOSES, while not as strong as MOSES, performs so much better on this problem than other methods without dependency learning (i.e., GP and PIPE). A bit of theory from the bit-string case can help us here again. On the bit-string deceptive trap problem of order k, the BOA is expected to scale with $O(2^k n^{1.5})$, the simple genetic algorithm with $O(2^n)$, and stochastic hillclimbing with $O(n^k log(n))$ – Chapter 3 of [80] provides a more in-depth analysis and an experimental comparison. Although in the limit as n grows the BOA outperforms stochastic hillclimbing, neither scales nearly as poorly as the simple GA on this problem class.

[4]The figures for GP and ECGP from [100] for ORDER and TRAP are for the number of evaluations required to converge to a population with a score no more than one less than the optimum (i.e., may be missing a single building block). Hence, comparative performance for MOSES may be somewhat better than indicated.

MOSES has dynamics which are a mixture of optimization guided by probabilistic-modeling (the hBOA) and stochastic population-based hillclimbing (the iterative resampling centered around existing good solutions when new demes are created).[5] For various problems, one or the other may dominate to an extent, as it appears stochastic population-based hillclimbing does here. To test this, MOSES and uMOSES should be applied to larger TRAP problems with an increased the trap size (e.g., 5 instead of 3), and decreased signal difference (e.g., 0.1 instead of 0.25). This would be expected to increase the performance gap between MOSES and uMOSES.

6.3 Supervised Classification Problems

In this section, MOSES is applied to learn Boolean formulae as classifiers for a few difficult binary supervised learning problems in computational biology.[6] These are different from the experiments run in previous sections in that the evaluation methodology will not depend primarily on discovering high-scoring solutions, but on test set performance. Before applying MOSES to supervised classification, a number of general machine-learning issues that are relevant to any supervised classification methods must be addressed. One is how to deal with noisy continuous data, another is how to deal with a huge number of problem variables (potentially thousands), and third is how to avoid overfitting.

The goal here is to focus on MOSES rather than on sophisticated data processing; these issues are hence resolved uniformly in the simplest possible manner. Continuous expression levels are discretized into binary values based on the median of the data. For high-dimensional datasets, only the fifty most differentiating features are used. Additional parsimony pressure is provided by penalizing classifier programs half a point for each symbol they contain (where a point corresponds to classifying a single case correctly), and restricting the number of distinct features utilized by a single classifier to be under five.[7] This favors smaller classifiers for smaller datasets roughly in accordance with algorithmic probability and learning theory; the larger a training set, the larger the classifier programs that may be supported without overfitting.

A further configurational issue that must be addressed is that MOSES does not return a single solution but a set of good solutions. In the future this may be exploited further to obtain greater insight into the problem being studied. For now, only the solutions

[5] Representation-building is expected to improve performance synergistically with both optimization heuristics.

[6] I gratefully acknowledge Dr. Ben Goertzel, Lúcio de Souza Coelho, and Cassio Pennachin for assistance and helpful suggestions in preparing the experiments and analysis presented in this section.

[7] The *ad hoc* way that parsimony pressure was implemented in this study is a deficiency which should be remedied in future studies by a rigorous minimum-description-length-based derivation and/or empirical studies.

Table 6.8: Supervised classification test set accuracy for MOSES, genetic programming, support vector machines, and max-prior (the frequency of the most common classification). * indicates a result achieved over the *entire* dataset, rather than test accuracy.

Technique	CFS	Lymphoma	Aging Brain
Max-prior	57.4%	75.3%	52.9%
SVM	66.2%*	97.5%	95.0%
Genetic programming	67.3%	77.9%	70.0%
MOSES	67.9%	94.6%	95.3%

with the highest score are considered; the results presented herein are uniform averages over these.

The results presented in the following subsections are summarized in Table 6.8.

6.3.1 Diagnosing Chronic Fatigue Syndrome

Significant genetic variation in humans is due to single nucleotide polymorphisms (SNPs), the smallest possible variations in the genome. The existence of SNPs or combinations of SNPs that can act as accurate predictors for some phenotypic trait (such as a disease) indicates at least a partial genetic basis. Chronic fatigue syndrome (CFS) is currently a largely exclusionary diagnosis; it is posited when other possibilities have been eliminated. There is still controversy concerning the status of CFS as a medical condition, and its causes remain unknown. The preliminary evidence of a genetic basis presented in Goertzel et al. [26], implied by the discovery of statistically significant predictors based on combinations of SNPs, is thus of great interest.

The Goertzel et al. study involved significant preprocessing of a dataset consisting of 43 CFS patients together with 58 controls, with a feature space containing 28 SNPs that might plausibly be related to CFS. For each patient, these were marked as not present, present in heterozygosis (a single copy), or present in homozygosis (two copies). Based on disappointing performance of traditional machine-learning approaches, the authors took the drastic step of dispensing with the test-training set distinction, and carrying out enumerative search for classifiers.

"Pattern strength classifiers" were used, which are simply sets of SNPs together with a threshold. To evaluate a pattern strength classifier on a given patient, values for each SNP in the set are summed, with 0 being assigned for not present, 1 for heterozygosis, and 2 for homozygosis. If the sum is above the threshold, the patient is classified as CFS. The enumerative search considered all combinations of up to five SNPs, with thresholds computed to give maximum accuracy. Accuracies of up to 76.3% were attained with this

protocol. Based on a permutation test over 115 shuffled versions of the data, this result was determined to be significant with $p < 0.01$.

Since I did not have the computational resources available to perform a permutation test based on hundreds of sets of MOSES runs, a more typical 10-fold cross-validation protocol was used, with 10 independent runs of MOSES per fold. Each run was carried out for 100,000 evaluations. The preprocessed data of [26] were used, with each of the 26 SNPs generating two binary features for MOSES ("in heterozygosis" and "in homozygosis"), leading to a search space with arity 56, greater than any of the problems studied above.

The best result attained on this dataset based on 1000 runs with SVMs in randomized configurations as reported in [26] was 68% accuracy over the *entire* dataset, with an average accuracy of 66.2% (in comparison, the overall average accuracy of the classifiers learned by MOSES was 77.3%). Genetic programming, when operated in a standard configuration for supervised classification (arithmetic, trigonometric, and logical operators, and ephemeral random constants), produced essentially chance results on the test set.

MOSES on this dataset attained an average test accuracy or 67.9%. When GP was run in a comparable configuration (Boolean formulae with the basis {*AND, OR,NOT*}, a half-point penalty for each symbol, etc.), the average accuracy was 67.3%, based on 100 indpendent runs. The 95% confidence upper bound for the GP test set performance was 67.8%, and insuffucient independent runs of MOSES were executed to compute confidence intervals. Thus, it cannot be said conclusively that MOSES outperforms GP for formula learning on this dataset.

On the other hand, the classifiers learned by MOSES were generally more complex than those discovered by genetic programming, leading to significantly better training set performance (78.3% for MOSES vs. 72.7% for GP). Given the small size of the dataset relative to the dimensionality of the space (only 101 samples vs. 56 binary features), MOSES might be expected to achieve demonstrably superior performance via a protocol such as leave-one-out cross-validation.

An additional contribution of the analysis in [26] is an ordering of the nine genes in the study (there are multiple SNPs per gene) based on their frequency of occurrence in pattern intensity classifiers outperforming the maximum prior. This leads to recommendations for future analysis. When a similar analysis is performed on the set of top-performing classifiers found by MOSES, this shrinks to a set of only seven genes; 5HTT, COMT, CRHR1, CRHR2, NR3C1, NRC1, and TPH2 (see [26] for a lexicon). This list is in agreement with the suggestions proposed in [26] for additional research (based on which genes appear most promising for future study).

6.3.2 Diagnosing Lymphoma Type

Lymphoma is a broad term for a number of particular kinds of cancer occurring in lymphocytes (a type of white blood cell). Various subtypes require different treatments, but are difficult to distinguish clinically. Discovering good classifiers between lymphoma types based on microarray expression levels is of interest for yielding tests, as well as understanding which genes in fact have roles in lymphoma and how they may interact. I report here on work with the dataset of [108] containing 58 cases of diffuse large B-cell lymphoma, and 19 cases of follicular lymphoma (77 total). This dataset has a dimensionality of 6817. Results are reported in the literature for SVMs, K-nearest neighbor, and two kinds of neural network [112, 25], and for GP [25]. All of these results are based on optimizing algorithm parameters over the training data - see the respective references for details.

MOSES was run on this dataset with a total of 50 Boolean features corresponding to the most-differentiating genes (discretized based on their medians, as described above). Ten independent runs with 10-fold cross-validation were used, with a total of 100,000 evaluations per run. Average test accuracy was 94.6%, slightly worse that for SVMs (97.5%), and significantly better than K-nearest neighbor (87%), backpropagation neural networks (89.6%), probabilistic neural networks (67.7%), and genetic programming (77.9%). Unlike with SVMs, we can very easily look inside and across the classifiers learned by MOSES to determine which genes are used in classification, how, and how often. On this dataset, the best classifiers found by MOSES, taken in aggregate, used 33 out of the 50 possible genes.

6.3.3 Classifying Aging Brains

The final computational biology dataset to be examined concerns a microarray dataset with around 11,000 genes gathered to study the effects of aging on the human brain [62]. A subset of the data consisting of 11 "old" samples (over 72 years old) and 9 "young" samples (under 42 years old) was used for supervised classification with GP and SVMs in [25]. The configuration and findings here are essentially the same as for the lymphoma experiments described above; MOSES significantly outperforms GP (95.3% vs. 70.0%), and achieves parity with SVMs (95.0%). Fully 46 out of the 50 possible genes were used by MOSES in constructing high-quality classifiers, indicating that this is a fairly easy dataset.

6.4 Dynamics of Program Growth

Throughout this chapter, I have primarily focused on the improvements in quality of results, success rate, etc., made possible by MOSES. Of secondary interest is how large a program

MOSES will typically discover relative to its intrinsic complexity (minimal length). Section 6.3 introduced a very strong parsimony pressure for supervised classification which essentially limited program size empirically based on the size of the training set.

How does program size vary over the course of evolution with MOSES otherwise? Based on reduction to normal form, preferential allocation of computational effort to demes based on smaller exemplars, and lexical parsimony pressure (breaking ties in tournaments based on program size), program growth is not expected without a corresponding improvement in solution quality. Exactly how this will play out in practice will be determined by the reduction rules used, the size of the representations built, and the deme-level attractors.

This can be examined by considering how the sizes of the programs that are used as exemplars to construct demes vary against the number of evaluations. This is shown for ten search trajectories each for 5-parity, 11-multiplexer, ORDER, and TRAP in Figure 6.6. These trajectories demonstrate that indeed excessive program growth does not occur on these problems (nor in any of those I have run MOSES on).

The trajectories can also give additional insight into search dynamics. The 5-parity problem (upper left) typically requires many demes to be searched, indicative of significant neutrality, as expected. On the 11-multiplexer problem, in contrast, most search can take place within demes (i.e., deme-local optima are rarely encountered). The $n = 40$ ORDER problem (lower left) is an even more extreme example of this. The $n = 42$ TRAP problem shows an interesting sawtooth pattern in program size; sudden jumps indicate a corresponding improvement in performance, while parsimony pressure effectively drives down program size when no improvement in score is taking place.

6.5 Computational Complexity and Profiling

A final question that must be answered is how the number of evaluations performed by MOSES corresponds to actual computational complexity and runtimes. To summarize, the computational complexity of MOSES within a particular deme may be bounded by $O(N \cdot n^3)$, where N is the population size, and n is the size of the representation; n^2 for modeling[8] and n for the number of generations. The number of scoring function evaluations here will be $O(N \cdot n)$. We can let $N = O(2^k \cdot n^{1.05})$, where k is the highest degree of subproblem interaction, and $n = O(a \cdot l)$, where a is the arity of the problem's function and terminal set, and l is the size of the exemplar program.[9] Letting $c(l)$ be the problem-dependent cost of scoring a program of size l, we get:

[8] As a technical note, this assumes that the depth of the decision trees (local structures) learned by the hBOA may be bounded by a constant. For hierarchical problems an additional factor of $log(n)$ or n may need to be introduced here, and the ensuing calculations modified accordingly. In practice however, a constant bound on the decision tree depth seems reasonable.

[9] This is the relation which perhaps can and should be improved most critically in future design developments – see the next chapter.

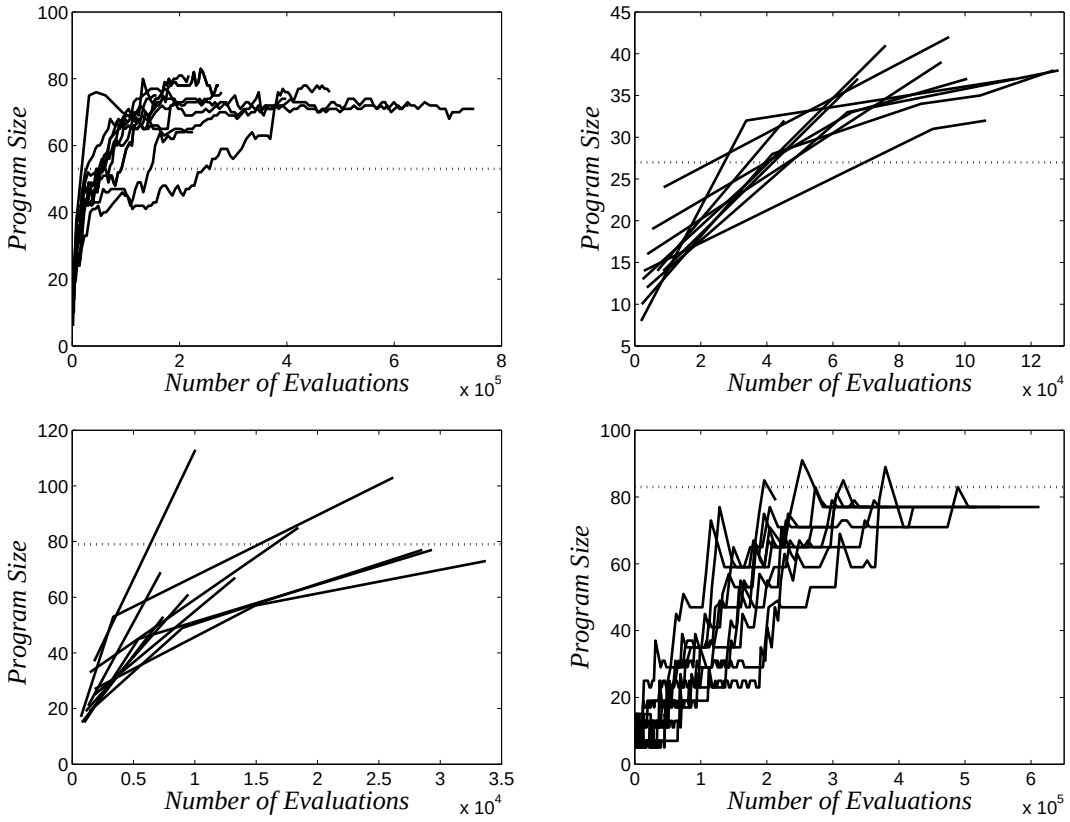

Figure 6.6: Exemplar program size vs. number of evaluations for 5-parity (top left), 11-multiplexer (top right), $n = 40$ ORDER (bottom left), and $n = 42$ TRAP (bottom right). The dotted lines are the minimal optimal solution sizes (approximated for parity and multiplexer).

Table 6.9: For various problems, the percentage of total MOSES execution time spent on the hBOA, on representation-building, and on program evaluation. The last row shows computational complexity (per generation), for comparison. N is the population size, l is the program size, a is the arity,and c is the number of test cases.

Problem	hBOA	Representation-Building	Program Evaluation
5-Parity	28%	43%	29%
11-multiplexer	5%	5%	89%
CFS classification	80%	10%	11%
Complexity	$O(N \cdot l^2 \cdot a^2)$	$O(N \cdot l \cdot a)$	$O(N \cdot l \cdot c)$

$$\begin{aligned} O(N(nc(l) + n^3)) = \\ O(2^k n^{2.05} \cdot (c(l) + n^2) = \\ O(2^k (a \cdot l)^{2.05} (c(l) + (a \cdot l)^2)). \end{aligned} \tag{6.3}$$

A conjecture for future investigation is that when representation-building is properly configured for a domain at hand, the above equation will hold as a bound on the computational complexity of the *entire* MOSES procedure, taking n as the size of the target program to be learned. This involves assuming no deception on the deme level, as described in Chapter 5. In this case, the complexity of the final deme optimization problem should dominate the cost of any single optimization problem encountered previously. For problems that may be solved via a few demes such as the artificial ant or the multiplexer, this argument is sufficient. But what about problems such as parity that require significant neutral drift and consequently many successive demes to solve, such as parity? In this case, I would further conjecture that the optimal configuration of MOSES is with asymptotically smaller populations than the above equation would suggest for individual demes, leaving the relation itself to hold in aggregate. Clearly, there is much work here still to be done.

The implementation of MOSES used for all experiments in this dissertation has been written in high-level C++, compiled with g++ version 3.4.4. Runs have been executed on machines with between two and four gigabytes of RAM, which is more than adequate; the code is typically CPU-bound rather than memory-bound. Profiling was carried out for the 5-parity, 11-multiplexer, and CFS classification problems; results are shown in Table 6.9. In comparison, simpler program evolution systems such as GP typically spend nearly all cycles on program evaluation (cf. [48]).

The high percentage of time spent on representation-building was somewhat surprising; in retrospect, this appears to be due to suboptimal implementation decisions rather than any intrinsic computational bottleneck (in practice *or* asymptotically). I believe that a

superior implementation could render the cycle counts for representation-building (at least on Boolean formulae) insignificant relative to the hBOA and program evaluation.

Which computational task is most intensive varies quite understandably by problem in accordance with the computational complexities given and problem parameters. For 11-multiplexer $c = 2^{11} = 2048$, and so program evaluation is dominant. For CFS classification the arity is relatively high (56), and so the cost of hBOA, which is quadratic in arity (a), dominates. 5-parity has relatively low arity, and c and l are of roughly the same magnitude; there is no clear dominating factor here in comparing the costs of hBOA and program evaluation.

Actual runtimes of MOSES were measured on a 2.2 GHz AMD Opteron machine with 4GB RAM, running GNU/Linux. The longest were the 6-parity runs, typically clocking in at around twelve hours. Runs for 5-parity, 11-multiplexer, and supervised classification typically ran for between one and three hours. Runs for other problems took less than one hour (per run).

Chapter 7

Towards a Theory of Competent Program Evolution

Empirical and theoretical results have demonstrated that competent optimization algorithms such as the hBOA can learn a wide range of nearly decomposable functions. They are robust to deception, noise, exponential scaling of subproblems, etc., and scale polynomially with problem size. To a great extent, their success has followed from the clear articulation of exactly what problem classes they are trying to solve [28]. The corresponding effort to describe an interesting class of "nearly decomposable" program evolution problems is just beginning.

I have attempted to advance this effort by first describing the salient features of programs in particular that make them difficult to evolve in a direct fashion, as well as those that might be exploited as inductive bias (Chapter 1). Chapter 2 then outlines a general search mechanism that exploits this inductive bias. Chapters 3, 4 and 5 show, concretely, how program spaces may be studied to uncover the aforementioned features and transformed to exploit the aforementioned biases via representation-building. Finally, Chapter 6 presents and analyzes the results of applying this approach to solve problems.

7.1 Propositions

A representation-building and probabilistic-modeling approach to program evolution has been presented. On this basis I believe that progress should be possible towards a theory of competent program evolution by proceeding according to a corresponding set of propositions. That is to say, the propositions are organized isomorphically to the design of MOSES, and in their totality constitute a proof of (or if unproven, a line of argument for) the effectiveness of a MOSES-like approach to program evolution.[1]

[1] Ben Goertzel has greatly contributed to the ideas in this section through discussion.

7.1.1 Representation-Building Improves Distance Properties

I have demonstrated empirically in Chapter 4 that representation-building improves the correlation between syntactic and semantic distance (for small syntactic distances). To move from this observation to a theorem, one approach would be to describe a class of program evaluation functions where representation-building provably improved the correlation between syntactic and semantic distance. One might even start with individual evaluation functions. The next step would be to investigate a class of *scoring* functions and optimization algorithms where increased correlation improved performance (on average). This seems like a fairly straightforward property of the hBOA and many other evolutionary optimization algorithms.

7.1.2 Improved Properties are Manifest Primarily for Smaller Distances

Assume a quantified relationship between syntactic and semantic distance under representation building, and that it follows the pattern observed for Boolean formulae (strong correlation only for small syntactic distances). It should then be possible to quantify the expected utility of processing based on demes, and even compute the optimal deme size (i.e., how big representations should be). This will depend on the representation-level properties of problem difficulty described in Chapter 5.

7.1.3 Programs Contain Complex Cross-Modular Dependencies

If the dependency structure of a program space is closely aligned with the internal tree-like structure of programs (with no deception), then an approach based on the recombination of program subtrees such as genetic programming would be expected to scale well. If there were *no* dependencies, then a univariate model such as probabilistic incremental program evolution would be appropriate. Based on results presented in this dissertation and elsewhere, neither of these is generally correct. If significant cross-modular dependencies can be demonstrated on the program level, then an adaptive approach based on probabilistic modeling should achieve superior performance. The catch is that the structure of the dependencies must be compact enough that the probabilistic modeler can learn it with a reasonable sample size.

7.1.4 Representation-Building Reduces Dependencies

By eliminating redundancy and matching up similar programmatic substructures in the encoding, representation-building is expected to reduce the level of interaction (dependencies) needed to accurately model program space. Of course, representation-building is not generally capable of transforming program learning tasks into problems with separable encodings; a competent optimization approach (such as the hBOA) is thus called for.

Taken together, these propositions imply that competent program evolution may be attained by exploiting domain knowledge to build representations over bounded regions of program space (demes), and applying a competent optimization algorithm to learn within the representations.

7.2 Future Development Plans

As noted in the Preface, the intent of this work it not to lead to a program evolution system that directly matches the capabilities of human programmers, but rather to develop one of the essential underlying mechanisms that must be synthesized to achieve this goal. Nevertheless, this will certainly require a system that encompasses more of what human programmers do than MOSES is capable of at present. The list below articulates a number of desiderata that will furthermore greatly expand the utility of MOSES for direct real-world problem-solving. None of these extensions represent radical departures from the MOSES paradigm detailed in this dissertation; they are evolutionary, not revolutionary.

7.2.1 Real-Valued Functions

MOSES may by extended to learn real-valued functions. A hierarchical normal form for arithmetic and trigonometric functions analogous to that for Boolean formulae is possible, consisting of alternating levels of linear and non-linear operations. The corresponding representation-building operations are also quite similar. Some aspects of such a system have been implemented and successfully deployed in the domain of circuit modeling [65]. Based on the strong results achieved for supervised categorization when augmented with a simplicity bias, MOSES can be expected to perform well in this area.

7.2.2 A Bias for Hierarchical Code Reuse

The representation-building processes of MOSES may be extended to incorporate a bias towards hierarchically structured code reuse (one kind of modularity). This would manifest itself as additional knobs whose twiddling would enable reuse (possibly with slight modification) of code from one part of a program somewhere else. This is the basic goal of Koza's automatically defined functions [49]. How exactly to accomplish this most effectively in a MOSES context is not yet clear; promising directions that build on existing work include a directed acyclic graph representation [91], combinators [61, 58], and the creation of local functions [74, 123].

7.2.3 Advanced Programmatic Constructs

MOSES may be extended to learn list-manipulation functions, in conjunction with the addition of a rudimentary type system, e.g.:

1. primitive types (e.g., *Boolean*, *integer*, *real*),
2. *list-of-*T types, where T is some primitive type,
3. functions from $(T_1, T_2, ..., T_N)$ to O, where O and all T_i $(1 \leq i \leq N)$ are either primitive types, lists of some primitive type, or functions of a lower or equal order.

Applying such a typing scheme to program evolution is reminiscent of strongly typed genetic programming [69], and functional genetic programming [123]. As with the bias towards code reuse, a bias towards uniform list processing may be incorporated into representation-building. For example, an exemplar program containing a few special cases processing elements of a list might lead to knobs that, when enabled, caused the entire list to be processed uniformly. This is another kind of modularity bias.

This type system has serious limitations; for instance, lists of functions are not supported, nor are polymorphic functions.[2] The rationale for these restrictions is to reduce combinatorial explosion and simplify the representation-building process and corresponding reduction rules. At the moment, there does not appear to be any "killer app" for program evolution that would necessitate a richer type system.

The learning of list-manipulation functions in a typed language leads quite naturally to the development of iterative functional programming constructs such as *map* (apply a unary function to all items in a list, producing a new list) and *fold* (apply a binary function successively to all items in a list, "folding" it into a result consisting of a single item). Based on these and other linear (and sublinear) list manipulation methods, polynomial-time programs may be evolved via appropriate nesting of linear-time operators.

7.2.4 Enhancing the Underlying Optimization Algorithm

A wider range of underlying optimization algorithms and extensions should be explored. There have been a number of promising approaches to extending the hBOA to continuous and categorical variables [72, 1, 87]. Numerous efficiency enhancements have also been proposed, such as sporadic model-building [90], integration of local search [57], and parallelization of model-building [88]. I believe that all of these may be effectively integrated into MOSES to provide superior performance. It is also worth exploring the impact of using a simpler (and possibly faster) yet still competent optimization algorithm instead of the hBOA, such as the extended compact genetic algorithm [34].

[2] It may be convenient however to implement built-in functions over a wide space of possible types as if they were polymorphic. For example, addition of reals and addition of integers might correspond to the same underlying operation even if, to the system, they appeared as two separate functions.

7.2.5 Adaptive Representation-Building

As demonstrated in Chapter 6, the current MOSES system can effectively scale to handle program evolution over spaces with large arities (up to around fifty), provided the programs to be learned are small. It can also scale to learn large programs (over one hundred nodes), provided the arity of the program space is low (around six). This is because the number of parameters in the corresponding search space will generally grow with the *product* of the arity and the program size. Model-building will then be quadratic in this product.

In order to effectively learn large programs in high-dimensional spaces, this scaling behavior must be improved. Beyond the generic efficiency enhancements to the hBOA outlined above, there are two basic ways to accomplish this.

One way is to exploit knowledge of program spaces to heuristically reduce the space of models considered by the hBOA. For example, a locality bias might restrict the search for dependencies among variables to pairs of variables within a constant program-tree-traversal distance of each other, which would reduce the computational complexity of model-building to linear time. However, care would have to be taken that potentially useful dependency relations were not excluded. This could be tested through an extended study of the models built by the (unrestricted) hBOA across a range of program-learning problems.

Another way to scale better is to adapt the representation-building process as search progresses by limiting which knobs are constructed. MOSES already heuristically reduces the size of representations by exploiting domain knowledge (for Boolean formulae) to eliminate potential knobs that will probably not produce meaningful program transformations. Additional heuristic knowledge may be obtained late in search (or across a set of similar problems) regarding which types of knobs have facilitated improved programs in the past. A model based on the multi-armed bandit problem would be a reasonable starting point, with each type of knob corresponding to an arm, and a pull corresponding to introducing a knob into the representation.

> "We can only see a short distance ahead, but we can see plenty there that needs to be done." *Alan Turing*, final sentence of the essay *Computing Machinery and Intelligence* [116]

Appendix A

Related Approaches to Program Evolution

In Section 2.4 I reviewed a few evolutionary learning systems with similar features to those found in MOSES: representation-modification, demes, and biased/partially specified solution templates. In this appendix I compare MOSES to previous program evolution systems based on probabilistic modeling.

Broadly speaking, current approaches to program evolution based on probabilistic modeling may be divided into three categories: 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.[1] MOSES does not fall into any of these categories, as we shall see.

The archetypal prototype tree approach to program evolution is probabilistic incremental program evolution (PIPE) [98]. PIPE is based on a rooted schema model of programs trees. All trees in the population are aligned with a probabilistic model that has 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").[2]

To generate new solutions, PIPE independently samples from the distribution of functions and terminals at each node in the prototype tree. So for example 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 may thus be considered univariate (no interactions between variables), making PIPE a program evolution analogue of the population-based incremental learning approach [3] mentioned in Chapter 1.

[1] For a more detailed survey and analysis of approaches in the first two categories, see [107].

[2] All functions must thus have bounded arities (typically unary or binary), and the branching factor of the prototype tree equals the largest arity. A function with arity less than the maximum takes arguments from left to right and ignores any remaining ones.

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

Based on this outlook, two primary deficiencies 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, absolute addressing may be seen to be problematic by considering that $f(x)$ and $0.99 \cdot f(x)$, while nearly identical behaviorally, may share no rooted-tree schemata at all. Regarding the latter, program evolution problems may contain significant across-node dependencies, as demonstrated in Chapters 2 and 3. Note however that these two problems are linked; the good models don't exist for bad representations (see Chapter 1).

Several direct extensions of PIPE have been developed:

- Hierarchical PIPE [99] introduces hierarchical instructions and skip nodes. Hierarchical instructions refer to a fixed structuring of the function set which biases sampling towards programs with a particular overall organization (e.g., a linear combination of nonlinear elements). Skip nodes allow nodes to contain instructions for expressing only one of their child subprograms, ignoring the remainder.

- Estimation-of-distribution programming (EDP) [121] augments PIPE by considering a Bayesian network to represent dependency relationships between nodes. However, rather than learning a network as in the BOA, a fixed topology of dependency relationships is assumed (as in the factorized distribution algorithm [70]). Specifically, nodes are dependent on their parent and left sibling (if they exist). In [122], this approach is hybridized with genetic programming.

- Extended compact genetic programming (ECGP) [100] similarly extends PIPE to a non-univariate model. Rather than a static model as used in EDP however, the ECGP dynamically learns a marginal product model after each generation. This is essentially a partition of the variables (i.e., prototype tree nodes), with full intra-partition dependence and full inter-partition independence.

All of these extensions can be seen as addressing 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 based

on prior knowledge. Hybridizing genetic programming 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 of course directly addresses the assumption of independence between nodes.

A number of estimation-of-distribution algorithms for program evolution have been developed based on grammar models:

- Program evolution with explicit learning (PEEL) [105] models the distribution of good programs with a stochastic context-free grammar (SCFG) extended to allow rules to be tagged with depth and location restrictions. This model is learned locally by incrementally splitting existing rules into sub-rules.

- Grammar model-based program evolution (GMPE) [106] is also based on an SCFG model, here learned with a minimum message length metric.

- Grammar-transformation-based estimation-of-distribution algorithm for genetic programming (EDA-GP) [10] is yet another SCFG model, learned with a minimum description length metric (note that [106] and [10] were effectively introduced simultaneously).

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 the 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 clearly be introduced.

To date, two approaches to program evolution have been developed that are based on learning linear encodings that get mapped to programs:

- BOA programming (BOAP) [61, 58] is based on a probabilistic model (dynamically learned Bayesian networks with local structure) with two sets of variables; one set models the symbol content of programs (as in PIPE), while the other models program structure (i.e. tree shape). The two sets are explicitly coupled to the extent that the number of symbols generated must match the program structure. While schemata are no longer rooted in absolute positions, neither are they completely relative, as the structure-description language is constrained (see [58] for details and a description of other unique features of BOAP, most notably a generalized "sloppy" evaluation procedure designed to increase evolvability and facilitated list-based processing).

- Bayesian automatic programming (BAP) [94] is a unique approach which is essentially prototype-based; it evolves fixed-length integer vectors which are optimized by the

BOA (dynamically learned Bayesian network model). The vectors are mapped to programs for evaluation, as in grammatical evolution [77]; integers are treated as indexing the rules in a context-free grammar defining the programming language.

There are clearly program learning problem classes that necessitate more powerful modeling procedures (at any rate, we can invent them), while problem classes with simpler dependencies will be more effectively solved by techniques employing simpler models. This point holds regardless of which of the three basic approaches described above is taken. How simple or complex modeling needs to be for a program-learning tasks of interest is a question that is still far from being answered.

A more fundamental question which I believe must be answered first that of how programs and program subcomponents should be represented. It may be that one of the three approaches 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 certainly be proven based from a standpoint of computational complexity, learning theory, and algorithmic information theory. I would go further however and argue that in all but the simplest languages, the improvement to be had by leveraging knowledge and dynamically adjusting the representation *within* a program space will be dramatically greater than the improvement from shifting *across* general program spaces.

The probabilistic model constructed in MOSES is based on evolving sets of plausible transformations on an exemplar program. This was chosen in contrast to the three approaches given above primarily because it more effectively facilitates leveraging background knowledge and performing specialized representation-building (in determining which transformations are plausible). It can be seen as a generalization of the prototype tree approach; while modifying single symbols in a prototype tree may be transformations considered by MOSES, not all such transformations will be plausible (based on background knowledge), and many other kinds of transformations may be considered as well. Some of these may be viewed as tunneling through program-space and the overall set need not be fixed or tied to a particular distance metric.

In summary, the new approach to program evolution I have proposed is best characterized by the emphasis placed on exploiting background knowledge and building specialized representations, and not by the particular program spaces and optimization algorithms that have been used.

References

[1] C. W. Ahn, R. S. Ramakrishna, and D. E. Goldberg. Real-coded Bayesian optimization algorithm: Bringing the strength of BOA into the continuous world. In *Genetic and Evolutionary Computation Conference*, 2004.

[2] P. J. Angeline. Genetic programming and emergent intelligence. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*. MIT Press, 1994.

[3] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie Mellon University, 1994.

[4] L. Barnett. Netcrawling - optimal evolutionary search with neutral networks. In *Congress on Evolutionary Computation*, 2001.

[5] E. B. Baum. *What is Thought?* MIT Press, 2004.

[6] E. B. Baum and I. Durdanovic. Evolution of cooperative problem solving in an artificial economy. *Neural Computation*, 2000.

[7] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, 1987.

[8] K. D. Boese. *Models for Iterative Global Optimization*. PhD thesis, University of California, Los Angeles, 1996.

[9] P. A. N. Bosman and J. Grahl. Matching inductive search bias and problem structure in continuous estimation-of-distribution algorithms. Technical report, Manheim Business School, 2005.

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

[11] M. V. Butz. *Rule-based Evolutionary Online Learning Systems: Learning Bounds, Classification, and Prediction*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

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

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

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

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

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

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

[18] E. W. Dijkstra. On the cruelty of really teaching computing science (EWD1036). Circulated privately, 1988.

[19] M. Ebner, M. Shackleton, and R. Shipman. How neutral networks influence evolvability. *Complexity*, 2001.

[20] A. Ekárt. Shorter fitness preserving genetic programming. In *Artificial Evolution*, 2000.

[21] C. Elkan. Magical thinking in data mining: Lessons from CoIL challenge 2000. In *Knowledge Discovery and Data Mining*, 2001.

[22] R. J. Fateman. *Essays in Algebraic Simplification*. PhD thesis, MIT, 1972.

[23] J. Feldman. Minimization of Boolean complexity in human concept learning. *Nature*, 2000.

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

[25] B. Goertzel, L. Coelho, C. Pennachin, I. Goertzel, M. Queiroz, F. Prosdocimi, and F. Lobo. Learning comprehensible classification rules from gene expression data using genetic programming and biological ontologies. In *Computational Intelligence Methods for Bioinformatics and Biostatistics*, 2006.

[26] B. Goertzel, C. Pennachin, L. Coelho, B. Gurbaxani, E. B. Maloney, and J. F. Jones. Combinations of single nucleotide polymorphisms in neuroendocrine effector and receptor genes predict chronic fatigue syndrome. *Pharmacogenomics*, 2006.

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

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

[29] D. E. Goldberg, K. Deb, and J. H. Clark. Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, 1992.

[30] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 1989.

[31] D. E. Goldberg, K. Sastry, and T. Latoza. On the supply of building blocks. In *Genetic and Evolutionary Computation Conference*, 2001.

[32] S. Gustafson. *An Analysis of Diversity in Genetic Programming*. PhD thesis, University of Nottingham, 2004.

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

[34] G. Harik. Linkage learning via probabilistic modeling in the ECGA. Technical report, University of Illinois at Urbana-Champaign, 1999.

[35] G. Harik, E. Cantú-Paz, D. E. Goldberg, and B L. Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 1999.

[36] J. Hawkins and S. Blakeslee. *On Intelligence*. Times Books, 2004.

[37] T. Haynes. Phenotypical building blocks for genetic programming. In *International Conference on Genetic Algorithms*, 1997.

[38] D. R. Hofstadter. *Metamagical Themas: Questing for the Essence of Mind and Pattern*. Basic Books, 1985.

[39] D. R. Hofstadter. *Fluid Concepts and Creative Analogies*. Basic Books, 1995.

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

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

[42] D. Hooper and N. S. Flann. Improving the accuracy and robustness of genetic programming through expression simplification. In *Genetic Programming*, 1996.

[43] M. Hutter. Universal algorithmic intelligence: A mathematical top-down approach. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence*. Springer-Verlag, 2005.

[44] M. A. Huynen. Exploring phenotype space through neutral evolution. *Journal of Molecular Evolution*, 1996.

[45] M. Kimura. Evolutionary rate at the molecular level. *Nature*, 1968.

[46] D.G. King. Metaptation: the product of selection at the second tier. *Evolutionary Theory*, 1985.

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

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

[49] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[50] J. R. Koza and D. Andre. Parallel genetic programming on a network of transputers. Technical report, Stanford University, 1995.

[51] K. Krawiec. Pairwise comparison of hypotheses in evolutionary learning. In *International Conference on Machine Learning*, 2001.

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

[53] W. B. Langdon and R. Poli. Better trained ants for genetic programming. Technical report, University of Birmingham, 1998.

[54] W. B. Langdon and R. Poli. Fitness causes bloat. In *Online World Conference on Soft Computing in Engineering Design and Manufacturing*, 1998.

[55] W. B. Langdon and R. Poli. Why ants are hard. 1998.

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

[57] C. F. Lima, M. Pelikan, K. Sastry, M. Butz, D. E. Goldberg, and F. Lobo. Substructural neighborhoods for local search in the Bayesian optimization algorithm. Technical report, University of Missouri-St. Louis, 2006.

[58] M. Looks. Learning computer programs with the Bayesian optimization algorithm. Technical report, Washington University in St. Louis, 2005.

[59] 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.

[60] M. Looks, B. Goertzel, and C. Pennachin. Novamente: An integrative architecture for artificial general intelligence. In *AAAI Fall Symposium Series*, 2004.

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

[62] T. Lu, Y. Pan, S. Y. Kao, C. Li, I. Kohane, J. Chan, and B. A. Yankner. Gene regulation and DNA damage in the aging human brain. *Nature*, 2004.

[63] S. Luke and L. Panait. Is the perfect the enemy of the good? In *Genetic and Evolutionary Computation Conference*, 2002.

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

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

[66] B. L. Miller and D. E. Goldberg. Genetic algorithms, selection schemes and the varying effects of noise. *Evolutionary Computation*, 1996.

[67] T. M. Mitchell. The need for biases in learning generalizations. Technical report, Rutgers University, 1980.

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

[69] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 1995.

[70] H. Mühlenbein and T. Mahnig. Convergence theory and applications of the factorized distribution algorithm. *Journal of Computing and Information Technology*, 1998.

[71] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. binary parameters. In *Parallel Problem Solving from Nature*, 1996.

[72] J. Ocenasek. *Parallel Estimation of Distribution Algorithms.* PhD thesis, Brno University of Technology, 2002.

[73] J. R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 1995.

[74] J. R. Olsson. How to invent functions. In *European Conference on Genetic Programming*, 1999.

[75] J. R. Olsson and B Wilcox. Self-improvement for the ADATE automatic programming system. In *WSES International Conference on Evolutionary Computation*, 2002.

[76] R. Ondas, M. Pelikan, and K. Sastry. Scalability of genetic programming and probabilistic incremental program evolution. In *Genetic and Evolutionary Computation Conference*, 2005.

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

[78] U. M. O'Reilly. Using a distance metric on genetic programs to understand genetic operators. In *IEEE International Conference Conference on Systems, Man, and Cybernetics*, 1997.

[79] U. M. O'Reilly and D. E. Goldberg. How fitness structure affects subsolution acquisition in genetic programming. In *Genetic Programming*, 1998.

[80] M. Pelikan. *Bayesian Optimization Algorithm: From Single Level to Hierarchy*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[81] M. Pelikan and D. E. Goldberg. Hierarchical BOA solves Ising spin glasses and MAXSAT. Technical report, University of Illinois at Urbana-Champaign, 2003.

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

[83] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. Bayesian optimization algorithm, population sizing, and time to convergence. In *Genetic and Evolutionary Computation Conference*, 2000.

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

[85] M. Pelikan, D. E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. Technical report, University of Illinois at Urbana-Champaign, 1999.

[86] M. Pelikan, D. E. Goldberg, J. Ocenasek, and S. Trebst. Robust and scalable black-box optimization, hierarchy, and ising spin glasses. Technical report, University of Illinois at Urbana-Champaign, 2003.

[87] M. Pelikan, D. E. Goldberg, and S. Tsutsui. Combining the strengths of the Bayesian optimization algorithm and adaptive evolution strategies. Technical report, University of Illinois at Urbana-Champaign, 2001.

[88] M. Pelikan and J. D. Laury. Order or not: Does parallelization of model building in hBOA affect its scalability? Technical report, University of Missouri-St. Louis, 2006.

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

[90] M. Pelikan, K. Sastry, and D. E. Goldberg. Sporadic model building for efficiency enhancement of hierarchical BOA. In *Genetic and Evolutionary Computation Conference*, 2006.

[91] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In *International Conference on Genetic Algorithms*, 1997.

[92] R. Poli and N. McPhee. General schema theory for genetic programming with subtree-swapping crossover. *Evolutionary Computation*, 2003.

[93] 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.

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

[95] J. Rehling and D. Hofstadter. The parallel terraced scan: an optimization for an agent-oriented architecture. In *IEEE International Conference on Intelligent Processing Systems*, 1997.

[96] J. Rosca. *Hierarchical Learning with Procedural Abstraction Mechanisms*. PhD thesis, University of Rochester, 1997.

[97] 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.

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

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

[100] 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 Publishers, 2003.

[101] K. Sastry, U. M. O'Reilly, and D. E. Goldberg. Population sizing for genetic programming based upon decision making. In U. M. O'Reilly, T. Yu, R. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice*. Kluwer Academic Publishers, 2004.

[102] K. Sastry, U. M. O'Reilly, D. E. Goldberg, and D. Hill. Building block supply in genetic programming. In R. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practice*. Kluwer Academic Publishers, 2003.

[103] P. T. Saunders. Introduction. In P. T. Saunders, editor, *The Collected Works of Alan Turing: Morphogenesis*. North-Holland, 1992.

[104] J. Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence*. Springer-Verlag, 2005.

[105] 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.

[106] 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.

[107] Y. Shan, R. I. McKay, D. Essam, and H. A. Abbass. A survey of probabilistic model building genetic programming. In M. Pelikan, K. Sastry, and E. Cantú-Paz, editors, *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*. Springer, 2006.

[108] M. A. Shipp, K. N. Ross, P. Tamayo, A. P. Weng, J. L. Kutok, and R. C. T. Aguiar *et al.* Diffuse large B-cell lymphoma outcome prediction by gene-expression profiling and supervised machine learning. *Nature Medicine*, 2002.

[109] H. A. Simon. *The Sciences of the Artificial*. MIT Press, 1996.

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

[111] R. Solomonoff. Complexity-based induction systems: Comparisons and convergence theorems. *IEEE Transactions on Information Theory*, 1978.

[112] A. Statnikov, C. F. Aliferis, I. Tsamardinos D. Hardin, and S. Levy. A comprehensive evaluation of multicategory classification methods for microarray gene expression cancer diagnosis. *Bioinformatics*, 2004.

[113] D. Thierens, D. E. Goldberg, and A. B. Pereira. Domino convergence, drift and the temporal-salience structure of problems. In *IEEE International Conference on Evolutionary Computation*, 1998.

[114] 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.

[115] Marc Toussaint. Self-adaptive exploration in evolutionary search. Technical report, Institut für Neuroinformatik, 2001.

[116] A. Turing. Computing machinery and intelligence. *Mind*, 1950.

[117] S. Watanabe. *Knowing and Guessing - a formal and quantitative study.* John Wiley and Sons, 1969.

[118] R. A. Watson, G. S. Hornby, and J. B. Pollack. Modeling building-block interdependency. In *Parallel Problem Solving from Nature*, 1998.

[119] I. Wegener. *The Complexity of Boolean Functions.* John Wiley and Sons, 1987.

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

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

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

[123] T. Yu. *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming.* PhD thesis, University College London, 1999.

[124] T. L. Yu, K. Sastry, D. E. Goldberg, and M. Pelikan. Population sizing for entropy-based model building in genetic algorithms. Technical report, University of Illinois at Urbana-Champaign, 2006.

[125] E. Yudkowsky. Levels of organization in general intelligence. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence.* Springer-Verlag, 2005.

[126] W. Zhang and R. E. Korf. A study of complexity transitions on the asymmetric traveling salesman problem. *Artificial Intelligence*, 1996.

[127] W. Zhang and M. Looks. A novel local search algorithm for the traveling salesman problem that exploits backbones. In *International Joint Conference on Artificial Intelligence*, 2005.

[128] W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *International Joint Conference on Artificial Intelligence*, 2003.