Report Number: WUCSE-2005-23

2005-04-06

Learning Computer Programs with the Bayesian Optimization Algorithm

Moshe Looks and R. P. Loui

The hierarchical Bayesian Optimization Algorithm (hBOA) [24, 25] learns bit-strings by constructing explicit centralized models of a population and using them to generate new instances. This thesis is concerned with extending hBOA to learning open-ended program trees. The new system, BOA programming (BOAP), improves on previous probabilistic model building GP systems (PMBGPs) in terms of the expressiveness and open-ended flexibility of the models learned, and hence control over the distribution of individuals generated. BOAP is studied empirically on a toy problem (learning linear functions) in various configurations, and further experimental results are presented for two real-world problems: prediction of... **Read complete abstract on page 2.**

Learning Computer Programs with the Bayesian Optimization Algorithm

Moshe Looks and R. P. Loui

Complete Abstract:

The hierarchical Bayesian Optimization Algorithm (hBOA) [24, 25] learns bit-strings by constructing explicit centralized models of a population and using them to generate new instances. This thesis is concerned with extending hBOA to learning open-ended program trees. The new system, BOA programming (BOAP), improves on previous probabilistic model building GP systems (PMBGPs) in terms of the expressiveness and open-ended flexibility of the models learned, and hence control over the distribution of individuals generated. BOAP is studied empirically on a toy problem (learning linear functions) in various configurations, and further experimental results are presented for two real-world problems: prediction of sunspot time series, and human gene function inference.

2005-23

Learning Computer Programs with the Bayesian Optimization Algorithm, Master's Thesis, May 2005

Authors: Looks, Moshe

April 6, 2005

Abstract: The hierarchical Bayesian Optimization Algorithm (hBOA) learns bitstrings by constructing explicit centralized models of a population and using them to generate new instances. This thesis is concerned with extending hBOA to learning open-ended program trees. The new system, BOA programming (BOAP), improves on previous probabilistic model building GP systems (PMBGPs) in terms of the expressiveness and open-ended flexibility of the models learned, and hence control over the distribution of individuals generated. BOAP is studied empirically on a toy problem (learning linear functions) in various configurations, and further experimental results are presented for two real-world problems: prediction of sunspot time series, and human gene function inference.

SEVER INSTITUTE OF TECHNOLOGY

MASTER OF SCIENCE DEGREE

THESIS ACCEPTANCE

(To be the first page of each copy of the thesis)

DATE: April 6, 2005

STUDENT'S NAME: Moshe Looks

This student's thesis, entitled Learning Computer Programs with the Bayesian Optimization Algorithm, has been examined by the undersigned committee of three faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Master of Science.

APPROVAL: _____ Chairman

_____

_____

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LEARNING COMPUTER PROGRAMS WITH THE BAYESIAN OPTIMIZATION ALGORITHM

by

Moshe Looks, B.Sc.

Prepared under the direction of Professor R. P. Loui

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

Master of Science

May, 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

LEARNING COMPUTER PROGRAMS WITH THE BAYESIAN OPTIMIZATION ALGORITHM

by Moshe Looks

ADVISOR: Professor R. P. Loui

May, 2005
Saint Louis, Missouri

The hierarchical Bayesian Optimization Algorithm (hBOA) [24, 25] learns bit-strings by constructing explicit centralized models of a population and using them to generate new instances. This thesis is concerned with extending hBOA to learning open-ended program trees. The new system, BOA programming (BOAP), improves on previous probabilistic model building GP systems (PMBGPs) in terms of the expressiveness and open-ended flexibility of the models learned, and hence control over the distribution of individuals generated. BOAP is studied empirically on a toy problem (learning linear functions) in various configurations, and further experimental results are presented for two real-world problems: prediction of sunspot time series, and human gene function inference.

Contents

List of Tables

List of Figures

Acknowledgments

- Thanks to Dr. Ben Goertzel for the general ideas and motivation behind the work described here, and for many entertaining discussions fleshing out the details.
- Thanks to my employer, Object Sciences Corp., for essential financial support. Further thanks to OSC in general, and Steve Luce in particular, for flexibility, generosity, and all-around niceness.
- Thanks to Prof. Loui for time, advice, additional entertaining discussions, and for keeping CS at Washington University fun *and* deep.
- Thanks to Prof. Smart and Prof. Zhang for taking time from their inordinately busy schedules to advise, speculate, and serve on my thesis committee.
- Thanks to Cassio Pennachin for being a great collaborator on much of this work, and to Murilo Saraiva de Queiroz for help on the gene function inference problem (described in chapter 4).
- Final thanks goes to family and friends for helping me stay sane over the last six months. I *really* appreciate it!

Moshe Looks

Washington University in Saint Louis
May 2005

Chapter 1

Background: GA, GP, and BOA

The Genetic Algorithm (GA), introduced by Holland [13], and Genetic Programming (GP), introduced by Koza [16], are classic optimization techniques that attempt to incrementally improve populations of candidate-solutions to maximize a predefined "fitness function", using operators inspired by crossover and mutation in biological evolution. In the GA the candidate-solutions are fixed-length strings, whereas GP uses variable- sized tree structures. After overviews of GA and GP, we pay a visit to *schema theory*, which attempts to explain why GA and GP work, and has motivated the design of the Bayesian Optimization Algorithm [24] (BOA) and hierarchical Bayesian Optimization Algorithm [25] (hBOA), which are the next stops on the journey. The initial sections may be skipped by readers with background in evolutionary computation.

1.1 The Genetic Algorithm

The GA originated as a way of modeling, on an abstract level, the basic mechanisms of Darwinian evolution with sexual reproduction (selection, crossover, and mutation). These are defined for the GA as follows:

- **Selection** is via a predefined fitness function which assigns real values to bit-strings in the space; the goal of the GA is to find a string which maximizes this value.
- **Crossover** is an operation that can be defined on pairs of bit-strings. Simple one-point crossover takes two strings (of the same length), picks a random location in the middle, and swaps the two halves, to create two new

strings. Formally, for two strings, $(a_1, a_2, ..., a_l)$ and $(b_1, b_2, ..., b_l)$, pick a random $i \in \{1, 2, \ldots, l-1\}$, and return $(a_1, ..., a_i, b_{i+1}, ..., b_l)$ $(b_1, ..., b_i, a_{i+1}, ..., a_l)$. For example, crossing 10010 and 00100 at position $i = 2$ produces the strings 10100 and 00010.

- **Mutation** is simply flipping some bits in a string. It is carried out with probability p, independently, on every bit in a string.

In fact, the GA has proven effective enough as an optimization algorithm to be applied to many performance-driven applications. The basic GA optimizes on the space of bit-strings of a specific predefined length, and operates as follows (adapted from [19]):

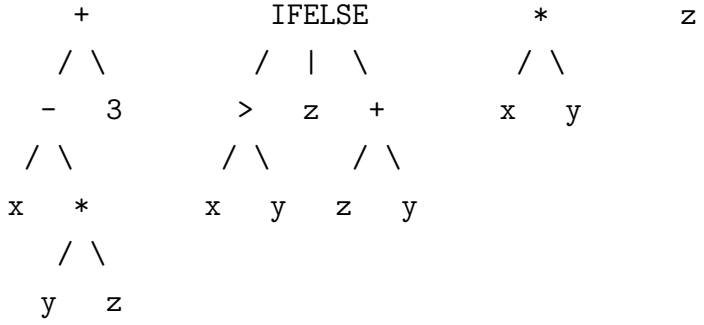
1. Initialize a population of n random bit-strings, all of a particular predefined length.
2. Compute the fitness of every instance in the population.
3. Until n instances have been created, select (with replacement) two instances from the population as an increasing function of fitness. With some predefined probability, cross them to create two new instances (otherwise keeping them intact). For odd n, discard a random instance.
4. Mutate all of the new instances.
5. Replace the old instances with the new ones.
6. Goto step 2.

To understand how the GA works, consider the case where crossover and mutation are non-existent (i.e., their probabilities are set to zero). In this case, the GA will eventually replace the entire population with copies of the fittest instance(s) in the initial population. The hope is that crossover and mutation will occasionally produce instances with improved fitness, allowing the search to progress. Somewhat surprisingly, they indeed do so for many problems of interest. This is taken up again after the introduction of *genetic programming* in the next subsection.

For a more detailed account of the genetic algorithm, see Mitchell's excellent introduction [19].

1.2 Genetic Programming

Genetic programming is similar to the GA in utilizing the three basic operations of selection, crossover, and mutation. However, the GP was designed from the beginning to be applied to problem-solving; in this case, automatically producing computer programs to serve a particular function. Instead of bit-strings, GP typically operates on an open-ended representation known as "parse trees" or S-expressions. An S-expression can be defined recursively based on a predefined set of terminals (e.g., x, y, z) and functions (e.g., $+, -, /, *, IFELSE, <$). Each function has a predefined arity (> 0). An S-expression is either a terminal, or an ordered list where the first item is a function, and the remaining items (equal in number to the function's arity) are S-expressions. Some sample S-expressions are:



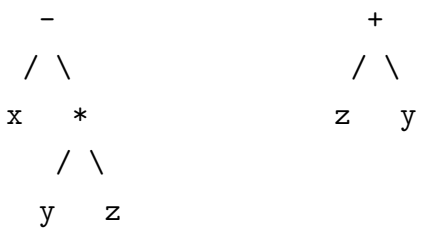
These correspond to the functions:

$$(x - (y * z)) + 3 \qquad (x > y)\ ?\ z\ :\ (z + y) \qquad x * y \qquad z \tag{1.1}$$

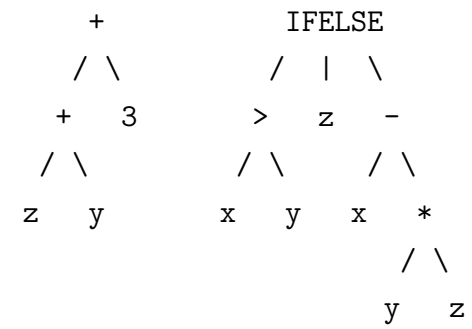
An important property of standard GP systems is *closure*. This means that any function can take as an argument the output of any other function or terminal. So in the second example from the left above, for instance, $IFELSE$ can be defined as taking three numbers, and returning the second if the first is non-zero, else the third. $>$ can be defined as taking two numbers, and returning 1 if the inequality holds, else 0.

Selection operates the same for GP as for the GA, with a fitness function defined on S-expressions rather than bit-strings. Crossover is somewhat different; since two trees may have totally different shapes, it must two-point rather than one-point. Random subtrees are chosen from each of the two trees (one each) and swapped to create two new trees. A subtree can be any non-root node. For example, if the

two leftmost trees above were being crossed, we might pick the following subtrees for crossover:



The new trees would then be:

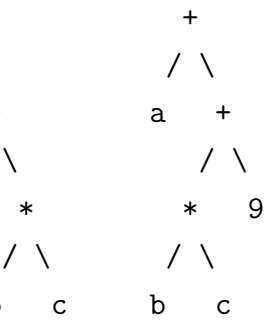


Mutation works similarly to the GA case. Typically, a random node is replaced with a new node with the same arity drawn from the set of terminals and functions. The basic GP algorithm is the same as that given for the GA. Mitchell's introduction to the GA [19], mentioned above, also contains a discussion of genetic programming. For the canonical account, see Koza's seminal work [16].

1.3 Schema Theory

A GA schema is a candidate solution (string) where some of the values may be ? (don't-care). The fitness of a schema is defined as the average fitness of all the strings that match it (by replacing the ?s with actual symbols). According to Holland's schema theory [13, 19], GA works by implicitly maintaining and recombining fit schema in parallel. Schema theory gives some approximations for the number of schema a given GA system can manipulate; see [19] for details. Schema are also known as "building blocks". Although there are some cases where the schema theory does not appear to be valid [4], by and large is accepted as an explanation of when and why the GA works [19].

Several similar theories have been proposed for GP, although the definition of a schema is not as straightforward. Whatever a GP schema is, the intuition is that GP works by recombining them in way comparable to the GA. Recently, it has been proposed by Rosca [27] that there are two fundamental kinds of schema in GP "top down" schema, defined by absolute tree position, and "bottom up" schema, defined by of relative position subtrees. To clarify, consider the trees below:



They both share the absolute-position schema $+(a, ?)$, and the relative-position schema $*(b, c)$ (i.e., $b * a$). There is significantly less consensus in the GP community with respect to the GA community on the applicability of schema theory. See for example [2], which questions the role of crossover in GP (and hence whether or not GP works by using crossover to manipulate fit schema).

1.4 The Bayesian Optimization Algorithm

The Bayesian Optimization Algorithm (BOA) is a qualitative improvement over the GA that moves beyond the metaphor of biological evolution, by maintaining an explicit centralized probabilistic model of the population of "good" candidate-solutions. Such algorithms are called probabilistic model building genetic algorithms (PMBGAs).

To understand how BOA works, let's first consider the case where the value at each position on a string makes an independent contribution to the string's overall fitness. With no other knowledge, the optimal way to generate a new string is to randomly assign a value to each position drawn from the distribution encoded by all of the strings we have evaluated so far, weighted by fitness. Most problems, unfortunately, do not possess this convenient feature; the fitness contribution of a particular value at a particular position depends, to a greater or lesser extent, on those in other positions. BOA attempts to dynamically discover these dependencies

between positions and utilize them in instance generation; instead of drawing from the entire distribution, we use conditional probabilities. By preserving important dependencies when new instances are generated, BOA can perform a more focused, effective search of the solution space.

Let's get more specific; BOA, as originally introduced [24], is a member of the family of "population-based search algorithms" [23], along with the GA, where instances are represented as bit strings. BOA significantly outperforms the genetic algorithm on a range of simple optimization problems by maintaining a centralized probabilistic model of the population it is evolving. When an initial population has been evaluated for fitness, BOA seeks to uncover dependencies between the variables that characterize "good" candidate solutions (e.g., the correct value for position 5 in the genome depends on which value is in place at position 7). In this way, it is hoped that BOA will explicitly discover and utilize probabilistic "building blocks", which are then used to generate new candidate solutions to populate the next generation. Different variations on BOA can be obtained by using different types of probabilistic models, but the basic algorithm remains the same (adapted from [24]):

1. Generate a random initial population P(0)
2. From the promising instances in P(t) learn a model M(t)
3. Generate a new set of instances O(t) from M(t)
4. Merge O(t) and P(t) according to some criteria, creating P(t+1)
5. Iterate steps (2) through (4) until termination criteria are satisfied

New instances are generated by sequentially assigning each variable. The value that each variable takes is chosen based on the values of the variables that they depend on; hence, there must be no dependency cycles (e.g., if we need to know the value of A to select the which value B will take, we must be able to compute the value of A without knowing the value of B). This implies that there is an ordering of the variables, $a_1 a_2 ...$ such that every a_i depends only on $a_1 ... a_{i-1}$.

Hierarchical BOA is BOA with two additions: a niche-friendly replacement scheme called restricted tournament replacement (RTR), and the use of decision graphs [26, 21] to model dependencies, introduced in [25]. Our implementation uses decision trees, based on the findings of [20] that they produce equivalent results.

Decision trees are used as a way of partitioning the population's distribution of a particular variable's values based on the values of other variables in the population.

For example, if our genome has two positions, and the decision tree for the second variable splits on the first variable, the population $(01, 11, 10, 00, 00)$ will be divided into $(01, 00, 00)$ and $(11, 10)$. So in choosing a value for the second variable, we would look at the value for the first variable (note that this must already be instantiated). If the first variable is 0, the value for the second variable will be drawn from $(1, 0, 0)$, so $P(0) = 2/3$, $P(1) = 1/3$, and if the first variable is 1, the distribution is $(1, 0)$, so $P(0) = 1/2$, $P(1) = 1/2$. In more complex genomes of course, decision trees can be more elaborate, with multiple splits based on different variables partitioning the distribution into many subsets.

Following this algorithm, when new instances are generated, good collections of variable assignments will be preserved, if the modeling is accurate. Thus, BOA can explore new areas of the search space in a much more directed and focused way than GA/GP, while retaining the positive traits of a population-based search algorithm (diversity of candidate solutions and non- local search).

In practice, constructing the optimal decision trees is intractable [26, 25]. Instead, a scoring metric is used to greedily add edges to initially empty trees, taking care to avoid adding edges which would produce dependency cycles.

Chapter 2

Generalizing BOA

This chapter is the meat of the thesis, describing the challenges needed to adapt BOA to learning computer programs, and how they have been overcome.

The first question addressed, with a relatively straightforward answer, is:

- *How can we model non-binary variables?*

Clearly we could encode anything we'd like to in sufficiently large binary strings, but this is usually not the best approach. The next series of questions tackled is more open-ended, and will not admit straightforward, direct, solutions:

- *What is a variable*? In fixed-length strings the answer is clear, but for trees we have the problem of absolute tree position vs. relative position (manifested in the two GP schema types described in chapter 1).
- *How to deal with alignment of different sized/shaped trees?*
- *How to deal with unified modeling of operators with different type signatures and ensure that type-correct trees are generated?*
- *What will drive the instances in the population to grow/increase in complexity (i.e., recombine bottom-up schemata)?*

2.1 Beyond Bits

Previous work by Ocenasek [20] includes an approach extending binary BOA to fixed-length strings with non-binary discrete and continuous variables (individual variables

prespecified to be either discrete or continuous). The principal obstacle that must be overcome is here is that the scoring metric and types of splits used must be able to handle pairwise combinations of discrete and continuous variables (i.e., decision tree nodes that split the distribution of a discrete variable based on the value of a continuous variable, discrete based on discrete, continuous based on continuous, and continuous based on discrete). There are a number of types of splits that can be considered. For example, we can split a tree using a discrete variable with n possible values into anywhere from 2 to n leaves. The Bayesian-Dirichlet metric described in [11] and adopted by Ocenasek is utilized, which generalizes nicely to characterize hybrid discrete-continuous splits. This is a start toward getting BOA to work on program trees; we can use continuous variables directly, and use discrete variables to encode operators and terminals.

However, there are still some fundamental issues that need to be dealt with. We must allow for discrete and continuous values in the same position. In order to split on such variables, the most natural solution is to introduce a predicate indicating whether the variable is discrete or continuous. After splitting on such a predicate, it is then possible to split again on the values themselves. For example, after splitting the distribution of Y into one leaf where X takes only discrete values, and one where X takes only continuous values, the former could then be split again into a leaf where X is 1, 3, or 5, and one where X is 0, 2, or 4, and the latter could be split into a leaf where $X < 3.2$ and one where $X > 3.2$.

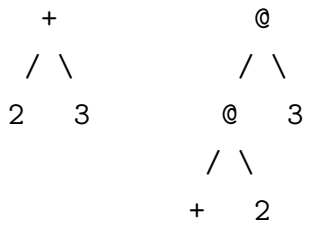
In addition to splitting on these variables, we must be able to instantiate them (i.e., maintain decision trees representing hybrid discrete-continuous variables). The natural solution now is to first split such a tree on whether its own value is discrete or continuous. And how do we decide which one it is? We can represent it using another decision tree! So for each hybrid variable X we have two trees; one which maintains a distribution over whether X is discrete or continuous (a type tree), and a second value tree which maintains X's actual distribution and is split based X's type.

2.2 Beyond Fixed-Length Strings

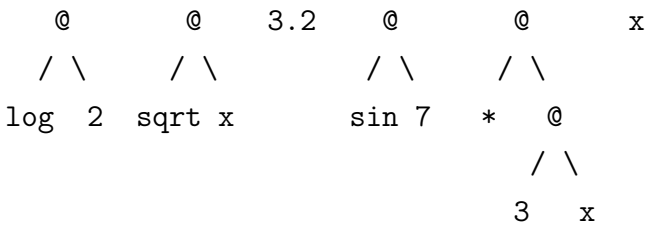
A serious issue that must be addressed is that BOA works on fixed-length strings, while GP typically uses variable-shaped trees. Thus, there is no consistent addressing scheme as there is for strings (e.g., if operators have different arities). We have considered two general solutions to this difficulty.

The first is a variation of gene expression programming (GEP) [8], where a linear, fixed-length genotype is used to represent a variable-sized tree phenotype. This was found to be acceptable for simple problems, but we consider this approach unscalable, due to the nature of the modeling process used by BOA. The difficulty is that the values for leaves that are in the same position on the tree may be in stored in different positions on the linear genome. Thus, BOA will not have any knowledge of their correspondence, and will have difficulty modeling them correctly. This difficulty is exacerbated as both problem complexity and tree size grow.

A second (partial) solution to this problem is to represent program trees in the curried form used by some functional programming languages [9, 36], where all function applications are single argument. So for example, + is viewed as an operator that takes a number as its argument, and returns an operator that takes a number as its argument, and returns a number. This binary application operator is denoted by @, and lives in all of internal nodes in our trees. See below for an example of this representation (right) and the standard GP tree form (left).



The obvious thing to do now is to create a separate variable for each unique address where variables are found. By address we mean the descending path taken from the root of the binary tree to reach a particular position, e.g., right-right-left-right, denoted by (0010). The root address is (). Consider the following population:



the distribution at the address () is $@, @, 3.2, @, @, x$, at (1) is $log, sqrt, sin, *$, at (0) is $2, x, 7, @$, at (01) is 3, and at (11) is x. The main problem with this approach is generating new instances that are guaranteed to be syntactically correct. For example, BOA could decide to put x at () and 2 at (0). The solution to this is for each decision

tree representing a variable's type other than the root address to be split on whether or not its parent is a leaf; if it is, then instantiation of the tree, as well as of its value tree, leads to a "null leaf", and no value is instantiated. Otherwise, a value is always instantiated, ensuring that all @ nodes, and no leaves, have their children instantiated. This approach was more effective than the first one, but still ineffective, and not entirely satisfying.

There is a fundamental distinction here between the GA case and the GP case; as described in chapter 1, the definition of a schema for fixed-length GA is fairly straightforward, whereas there is compelling evidence that there are *two* types of schema for GP; top-down schema corresponding to absolute tree position, and bottom-up schema, corresponding to relative position subtrees [27]. Since absolute tree position is sometimes relevant and sometimes not, one would like a system to be sensitive to this. The right way to do this is to allow the system to construct models that can vary the absolute position in which particular values end up (as in the gene expression programming solution), while keeping the semantics of different variables relatively intact (as in the absolute position example).

2.2.1 Zigzag Trees

As a solution a new representational scheme is introduced here, zigzag trees, which are a special kind of binary tree.[1] Let Z_N denote the set of zigzag tree with N leaves, using leaves drawn from the set L. This set can be defined inductively as follows. For $N = 1$, it is L. For $N > 1$, it the union of $@(l, t)$ and $@(t, l)$, where l is drawn from L, and t is drawn from Z_N. That is to say, zigzag trees are an intermediate form between left-deep and right-deep linear trees. So a zigzag tree of size N can be encoded as a list of N leaves, and a list of $N-1$ Boolean values (whether to zig or zag at a particular location). Here are for some sample zigzag trees and their encodings (L means zig left, R means zag right).

[1] After defining this kind of tree and coining the name, I discovered that they had already been defined, with the same name, for a completely different purpose (speeding up parallel database queries) [38].

```
      @                    @
     / \                  / \
    @   x                @   3
   / \                  / \
  +   @                @   42
     / \              / \
    @   x            if  @
   / \                  / \
  *   @                @   x
     / \              / \
    @   x            <   3
   / \
  +   @            [ < 3 x if 42 3,
     / \              R R L  R  R  ]
    @   x
   / \
  *   x

[ x * x + x * x + x,
  L R L R L R L R  ]
```

Now, what does representing programs as zigzag trees for BOA learning buy us? First of all, programs can be represented more concisely, since internal nodes are now implicit. Second, model position has been (partially) decoupled from absolute tree position.

This can now be modeled very naturally by BOA. For a population with a minimal instance size M (i.e., M leaves), and a maximal instance size of N, there will first be $N-M$ Boolean variables used to model instance length each dependent on the next. The semantics of these variables will be "instance has at least k leaves". Once one variable in the sequence has been set to false, all of the remainder automatically go to false (since they are dependent), which makes sense, since if a tree doesn't have $> k$ leaves, it clearly can't have $> k+1$, etc. leaves. There are also $N-1$ Boolean variables to determine where the tree zigs and zags (i.e., how the tree is constructed from the linear sequence). Finally, there are N leaf models, as described in the previous section (deciding if a value is a discrete symbol or a continuous value, then modeling the symbol/value content). These models are set up so that for a

particular position, the instance length variable must be set before the leaf value is set, so that we never set the values of non-existent leaves.

Note that, when simple conditions are met, zigzag trees are fully general, and can represent anything that can be represented with a binary tree. When no higher-order functions are used, in fact, they are *overgeneral*, in the sense that a simple right-leaning list of nodes can represent any function (see the next subsection for an explanation). For the case where higher-order functions are present, it can be proven that with the addition of a couple of simple rewrite rules (combinators) to the leaf set, any binary tree with N leaves will have an equivalent zigzag tree with at most $2N$ leaves; see chapter 5 for more details, and appendix A for the proof.

2.3 Sloppy Program Evaluation

One further difficulty that must be overcome in adapting BOA to modeling computer programs is types. BOA can very easily produce syntactically incorrect structures such as $@(@(@(+, 42), *), 2)$ (note that this is different from the syntactic requirement mentioned earlier of not inserting leaves in place of application nodes). Three general solutions exist to this difficulty; the first is to simply discard any such incorrect instances that are generated, the second is to introduce modeling on types, and the third is to use a more general evaluation scheme that can evaluate such incorrect expressions. The solution chosen here is the third, which seems to provide the most natural fit to the rest of the system.

Sloppy program evaluation works as follows (recall that all applications are binary). In applying A to B, where $@(A, B)$ is valid, it is returned. If it is not, but $@(B, A)$ is valid, that is returned instead. If neither is valid, the interpreter will "listify" A and B, meaning that they are merged into a single node containing a list of entities, with the left-most item coming first. Lists are treated as a single entity, with the left-most item being used for application. When one list is called on another, as many evaluations as possible are executed, moving from left to right.

Note that operators (e.g., $+, -, *, /$), might receive only a single number; in this case the number is held "in reserve" until another is found and the operator can be evaluated. A short example follows, utilizing the convention that unparenthesized applications are left-associative:

```
42 + (+ -) (3 4) 9 8 *
```

First 42 and + are inverted, while (+ −) and (3 4) are listified separately, to obtain:

```
+(42) [+, -] [3, 4] 9 8 *
```

The first two items must now be listified:

```
[+(42), +, -] [3, 4] 9 8 *
```

Now, when the two lists are merged, +(42) is given the argument 3, to produce 45, which can be applied to the next + along with the argument 4, producing $45+4 = 49$. This now becomes the first argument to the minus sign:

```
-(49) 9 8 *
```

The 9 is now fed to the −(49) to obtain:

```
40 8 *
[40, 8] *
320
```

Note that such a program might return multiple values or no values at all. In the former case we can adopt the convention of using the leftmost result. In the latter case, the instance is assigned zero fitness (this is extremely rare in practice).

2.4 Incremental Program Construction

Unfortunately, while the BOA variant described in the previous sections is effective at optimizing small subcomponents, and keeping components correctly intact, it does not, on its own, lead to the addition of new components. This is because there is no way to grow trees beyond the size of the largest tree in the population, and there is no way to move components to entirely new tree positions (although small changes can be accomplished in zigzag trees, by switching between zigs and zags).

To overcome this, a low-frequency two-point crossover operator (20% in all our experiments) has been added, which can cause trees to grow, and suffices to move subcomponents to new locations, where they can be optimized by BOA[2]. This overcomes the limitations of a pure probabilistic-variable-based approach, while maintaining its advantages, since instance generation is still by and large much more focused than it would be when using crossover and mutation exclusively. Also, we can start with a population of smaller trees, and have them naturally grow to fit the problem size.

[2]Two-point crossover mimics the behavior of GP crossover by swapping two random substrings of a zigzag tree (the leaves as well as the left/right zigzags).

2.5 Summary

The novel features of this research can be summarized in the form of brief answers to the questions with which we began this chapter:

- *How can we model non-binary variables?* Utilize Ocenasek's scheme [20] for modeling discrete and continuous variables, with additional type trees to allow discrete and continuous values to coexist in the same position.

- *What is a variable?* Separate tree structure from tree data and model them with different variables. This is accomplished by having curried operators live in the tree leaves and binary application nodes at all internal positions in the tree (ala functional programming), and introducing combinators (in the leaves), which can modify tree structure.

- *How to deal with alignment of different sized/shaped trees?* Simplify the modeling process by using zigzag trees, a subset of binary trees, which can be encoded as linear genomes and are provably general in that any binary tree can be converted into an equivalent zigzag tree using combinators (see A). The worst-case overhead, in term of tree size, is to double the number of leaves.

- *How to deal with unified modeling of operators with different type signatures and ensure that type-correct trees are generated?* Evaluate sloppily dynamically transforming syntactically incorrect trees into a nearby correct ones in the course of evaluation, with very little overhead.

- *What will drive the instances in the population to grow/increase in complexity (i.e., recombine bottom-up schemata)?* Use two techniques, corresponding to absolute (top-down) and relative-position (bottom-up) schema; BOA for the former, and zigzag tree structure combined with relative position crossover for the latter.

Chapter 3

Comparison with Prior Work

3.1 BOAP's relationship to BOA

Clearly, the prior work which BOAP owes the most to is BOA/hBOA [24, 25, 22], and Ocenasek's extension thereof [20]. The extent to which BOAP modifies the basic BOA/hBOA algorithm is been covered in the previous chapter.

3.2 BOAP's relationship to other PMBGPs

The field of probabilistic model building genetic programming (PMBGP) [30] is quite new, but there are already several approaches, which are compared and contrasted with BOAP, below.

3.2.1 Probabilistic incremental program evolution (PIPE)

An early PMGP was probabilistic incremental program evolution (PIPE) [28], later extended to hierarchical PIPE (hPIPE) [29]. PIPE uses standard GP function trees (S-expressions) as its representation. Instead of crossover, new trees are generated by drawing, in a depth-first fashion, terminals and leaves from the distribution at each (absolute) tree position, with an additional mutation operator. hPIPE allows subtrees to be stored in individual tree positions.

In comparison with BOAP, the primary limitations of PIPE and hPIPE are the fixed complexity of their models (no dependencies learned between positions), and the complete reliance on absolute position. Furthermore, using S-expressions means that arguments to functions with different arities are modeled together in a fairly

arbitrary way. For example, in the two trees $+(A, B)$ and $sqrt(C)$, C is modeled as overlapping with A rather than B (since they are both the first argument). Using curried form avoids this kind of arbitrary alignment.

3.2.2 Grammar-based approaches

Other recent work [31, 32, 5], uses more complex models, in the form of program generation grammars.

For example, program evolution with explicit learning (PEEL) [31], learns a grammar with ant colony optimization. In contrast to PIPE and EDP (see below), PEEL, is capably of learning variable-complexity models with an arbitrary number of production rules. However, the expressiveness of these rules is extremely limited; they can only "split" the distribution based on location in the tree, not based on tree content. For example, let's say that in a particular location some trees utilize $+$, while others utilize $*$. PEEL cannot express the notion that the numerical arguments are dependent on the choice of operator, and should be drawn from different distributions. Of course, it might be possible to add this kind of expressiveness in the future, along with a metric for deciding when such a split was worthwhile, making the resultant learning mechanism more BOA-like. As has been pointed out [31], PEEL has a desirable feature which it shares with BOAP; unlike in GP, there is no impetus for code bloat. On the contrary, more compact instances are more likely to be modeled and reproduced correctly.

PRODIGY [32] is a generalization of the PEEL approach which uses a stochastic context-free grammar (SCFG) and is agnostic with respect to how the grammar is learned. Bosman and de Jong [5] have reported on a similar SCFG-based approach.

3.2.3 Extensions of PIPE

Recently, systems described as performing estimation of distribution programming (EDP) [34, 35] have been developed which extend PIPE by taking note of dependencies (still between absolute positions in S-expressions). Unlike BOAP, these are not learned, but hardwired in a particular topology with fixed complexity (e.g., nodes are always dependent on their parents). This is closer to BOAP, but still much more primitive, and the limitations of PIPE mentioned above still apply.

Another generalization of PIPE is extended compact genetic programming (eCGP) [30], which is closer to BOAP in that the structure of the model can vary;

instead of generating from an independent distribution for each node, nodes can be iteratively merged based on an MDL metric (instantiation for a group of merged nodes is done according to the joint distribution). This is more expressive then PIPE; however, as with PEEL, dependency modeling is still not very fine grained; there is no way to restrict the dependency between a pair of variables to a particular context (as in BOAP), as they are either merged (completely dependent) or not (completely independent).

Chapter 4

Experiments

In all of the following experiments, BOAP is configured to generate a new population as described by Pelikan [24, 22], namely, by combining the top half from the previous generation with new instances generated from the model. As mentioned previously, the crossover operator is then applied to a random selection of 20% of the instances, with the exception of the single best instance in the population. Instances have been limited to 128 leaves, though this is rarely reached.

4.1 Symbolic Regression with Constants

To simply compare a number of configurations, a series of experiments have been run on fifty random linear functions $f(x) = ax + b$, where a and b are drawn uniformly from $[-5, 5]$, and x is in the interval $[-1, 1]$. As is typical in[16], each fitness evaluation consists of evaluation of absolute error against a fixed set of twenty random points. A total of 25,000 fitness evaluations are allowed per run, spread over 40 generations. The terminals are x and the ephemeral random constant, and the functions are $+, -, *, /$. Here, unlike in the following two sections, the point is not to demonstrate BOAP's superiority to competing methodologies, but rather to show the combined effectiveness of the various techniques presented (i.e., modeling with learning of dependencies, and zigzag trees), for a fairly arbitrary and not carefully tuned set of parameters. "Empty Models" refers to applying BOAP without learning any dependencies between variables, This means that the values for variables are drawn from the entire distribution at that location (as in PIPE; see chapter 3). "No Modeling" is 90% crossover (with population size adjusted accordingly), and no model-building or instantiation (i.e., GP on zigzag trees with sloppy evaluation).

Table 4.1: Symbolic Regression with Constants

	Success Rate	**Average Error**
BOAP	0.62	0.471±0.2655
Empty Models	0.14	1.05±0.338
No Modeling	0.00	2.33±0.654

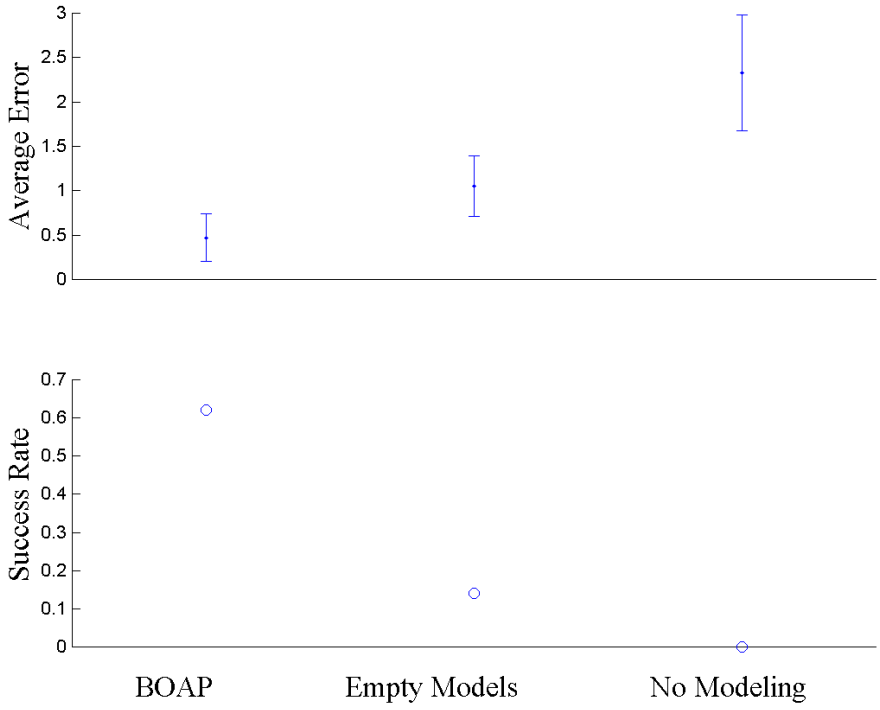


Figure 4.1: Symbolic Regression with Constants

Average error is the sum of absolute errors over the data points with 95% confidence intervals.

As can be seen from the table and figures, BOAP is fairly effective at learning this class of functions, and for zigzag trees at least, learning constants via modeling is more effective than via composition. Success is defined as in [16], as being within 0.01 of the target value for all twenty points. Note that for simple problems like this, BOAP is generally inferior to GP. It is only on problems difficult enough to warrant a large number of fitness evaluations (and hence a larger population size for BOAP) where GP is outdistanced. We shall explore two such problems below.

4.2 Sunspot Time Series Prediction

A well-studied time series prediction problem is the yearly sunspot data for 1700-1979. It is a useful candidate for BOAP testing because published benchmarks are available for PEEL [31] and GP [16]. In order to allow for meaningful comparison to these results, fitness was calculated using the error measure:

$$\frac{1}{\sigma^2} \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} (x_i - \hat{x}_i)^2 \tag{4.1}$$

where n is the number of data points, the x_i are the values, and $\sigma^2 = 0.41056$ is the variance of the entire dataset. In [15, 31], 600,000 fitness evaluations are used per run. Here, we use a total population size of 10,000, and a crossover rate of 0.2 (meaning 6,000 fitness evaluations per generation, since 40% of the population will be copied without modification). BOAP runs for 98 generations and hence utilizes 598,000 fitness evaluations (6,000 per generation, and 10,000 for initialization. As in [31], 15 independent runs were carried out. The only terminal is x, and the operators are $+, -, *, /, sin, cos, log, exp$.

Note that better results can be achieved on this problem with other methods (e.g., back-propagation neural networks [33]). We are concerned here with how BOAP compares to PEEL and GP under roughly identical circumstances. It is important to note in this regard that there is some ambiguity as to which fitness cases are included in which time periods. In particular, there is a peak at the beginning of the second test set which is ignored in the test scheme used by PEEL (and adopted here) which significantly degrades the accuracy of many of these models (learned by GP, PEEL, and BOAP). Also, PEEL and GP were configured somewhat differently; only 10 runs of GP were carried out, and four were excluded from the data given here due to poor performance. Furthermore, PEEL was given the previous six data points with which to make a prediction, whereas GP was given at least eight. In these cases of conflict, we have consistently chosen to follow the configuration taken by PEEL, to ensure that this comparison is a meaningful one.

The "best" instance given is the best on the training data. Two generations later (generation 100) on this run, BOAP discovered an instance with slightly better training performance that reduced the error score to 0.125 on (1921-1955), and only 0.117 on (1956-1979), outperforming both PEEL and GP ('x' in the figure).

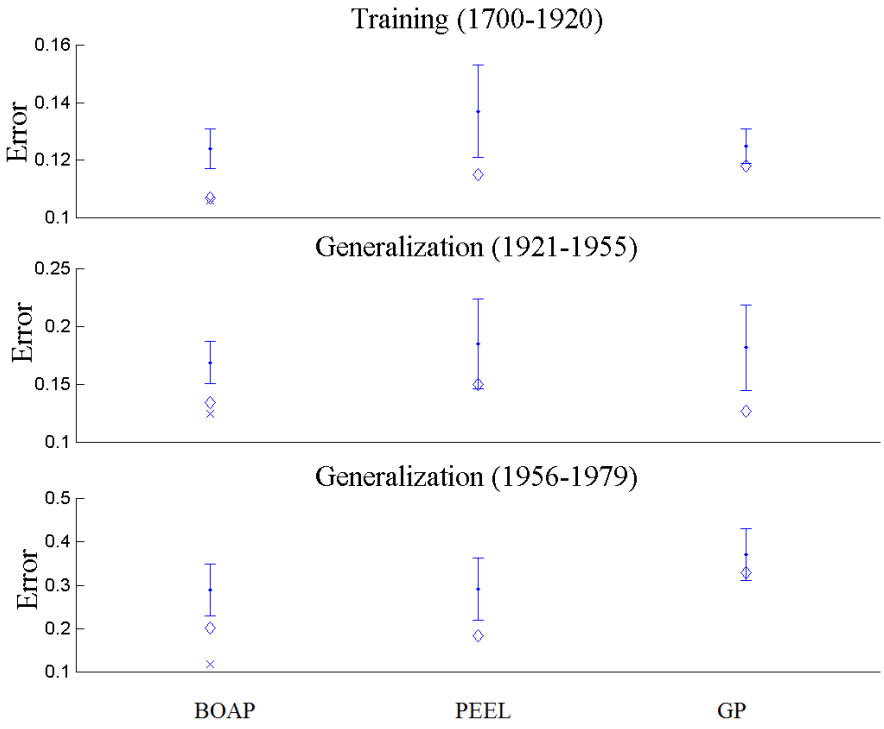


Figure 4.2: Sunspot Time Series Prediction

Table 4.2: Sunspot Time Series Prediction

	Training (1700-1920)	**Generalization** (1921-1955)	**Generalization** (1956-1979)
BOAP	0.124±0.007	0.169±0.018	0.289±0.06
BOAP best	0.107	0.134	0.202
PEEL	0.137±0.016	0.185±0.039	0.291±0.071
PEEL best	0.115	0.150	0.184
GP	0.125±0.006	0.182±0.037	0.37±0.06
GP best	0.118	0.127	0.329

4.3 Gene Function Inference

This set of experiments turns to a new and challenging supervised categorization problem: the automated placement of human genes in functional categories based on gene expression data. Only a brief background overview is given here (a paper focusing exclusively on this problem is in preparation, in collaboration with Ben Goertzel and Cassio Pennachin, who are coauthors the background material below).

In order to cope with the vast amounts of new bioinformatics data made available by new biological technologies, researchers have developed a variety of tools, including controlled vocabularies for discussing biological phenomena, and collaborative knowledge bases constructed using these vocabularies. One such effort is the multi-organism Gene Ontology (GO) project, which presents collections of terms divided into three ontologies: cellular component, biological process, and molecular function [6]. Each of the ontologies is a directed acyclic graph (DAG), in which terms are linked to other terms via specialization and part-whole relationships. Proteins are associated to GO terms according to knowledge of their biological roles.

The GO is powerful but far from complete, a fact that presents a number of interesting computational problems. While the online GO databases are reasonably extensive, not all genes discussed in the literature have been assigned a position in the GO hierarchy. Also, there are a large number of genes that have not yet even been discussed in the research literature (let alone placed in the GO). Gene expression microarrays are one among a number of recent technologies that allow us to acquire quantitative data on the whole-genome level. This data can be used to make educated guesses regarding the correct GO classification for genes that have not yet been categorized by humans. If a little-explored gene belongs in a certain GO

Table 4.3: Gene Function Inference

Gene Ontology Category	BOAP In-Sample	BOAP Out-Of-Sample
Catalytic Activity (GO:0003824)	0.729±0.009	0.464±0.025
Intracellular (GO:0005622)	0.766±0.01	0.526±0.027
Ribosome (GO:0005840)	0.907±0.005	0.814±0.022
Cell Growth/Maintenance (GO:0008151)	0.751±0.016	0.5084±0.043
Metabolism (GO:0008152)	0.773±0.007	0.549±0.023

category, then one may be able to tell this based on its expression levels obtained in various cells and experimental conditions.

Allocco et al [1] have explored the relationship between gene co-expression and co-regulation, using microarray and transcription factor binding data. Azuaje and Bodenreider [3] showed a reasonable correlation between gene co-expression and two different measures of Gene Ontology similarity. The closest correlate of the current work is that of Hvidsten et al [14] and Lagreid et al. [17], which uses a supervised learning algorithm to place genes in GO categories based on their expression values. Their accuracy figures are roughly comparable to ours, but their algorithm requires gene expression time-series data, which is much less common than the static data used here.

After preprocessing to normalize the data and limit the expression levels used to those that might plausibly be use to infer gene ontology (84 categories), 100 positive and 100 negative examples were selected for each category. That is, we end up with five supervised categorization data sets with 200 data points (with 84 category values per data point). Details on the preprocessing are omitted here, but will be included in the in-preparation paper. GP and BOAP were both tested on this data, and allowed 280,000 fitness evaluations each. BOAP was configured as in the previous experiments, but with a population size of 10,000. Continuous values were utilized. The operator set was $+$, $-$, $/$, $*$, $<$, and probabilistic OR ($OR(a,b) = a + b - a * b$) and NOT ($NOT(a) = 1 - a$). GP was configured to use the same operators, and tested with population sizes of 1000 and 5000 (note that a population size of 5000 for the GP is roughly equivalent to a population size of 10000 for BOAP).

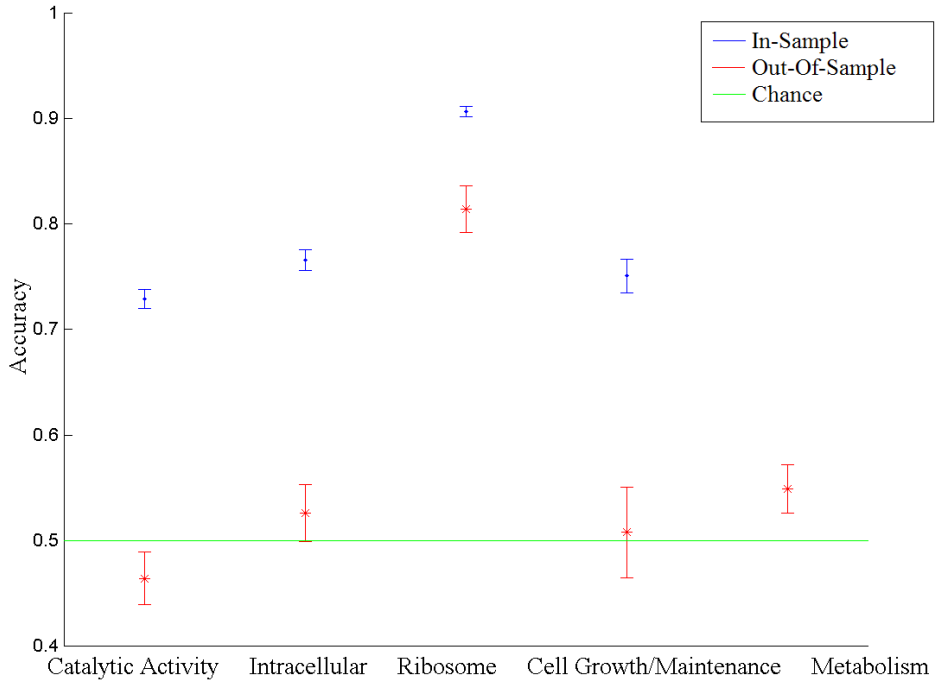


Figure 4.3: Gene Function Inference

Three-fold cross-validation was used, with five independent runs for each configuration, giving a total of fifteen runs per data set. The results were mixed; since this is a new data set, there were no guarantees that any meaningful classification rules could be discovered. However, BOAP found such rules conclusively on one set (Ribosome), and suggestively on two of the others (Intracellular, and Metabolism). GP failed to achieve results above chance on any of this data (none of the training accuracy levels were over 51%), as did a preliminary investigation utilizing support vector machines. One final thing to note is the pleasing simplicity of the rules found by BOA. For example, an effective rule discovered by BOAP for the Ribosome data set is:

```
x58 / ((x55 / !(x2 OR (x28 + x79))) * x52)
```

Overall, the average size of the rules found by BOA here is 17 symbols.

4.4 A Note on Runtime Comparisons

Conventional wisdom in the GP community is that runtimes are typically dominated by time spent in computing the fitness function (i.e., evaluating trees in the population) [16]. Thus, algorithms are benchmarked, as above, by allocating the same number of fitness evaluations to each algorithm. However, when comparing systems with radically different instance generation mechanisms as above, this is challenged on two fronts. The first is that the modeling overhead of BOAP might offset any percieved gains. The second is that all fitness evaluations are not created equal: GP may exhibits code bloat, for example, which can produce much slower evaluations for semantically equivalent trees. If, based on the former, one were to penalize BOAP for modeling, it would appear to be at a disavantage for the problems studied here. In order to clarify the latter, more conclusive tests would be needed to specifically study the instance size dynamics of BOAP vis a vis GP.

Unfortunately, runtime and profiling results are not included in [15, 31], so a finer-grained comparison is not possible at this time. Emperically, BOAP's C++ implementation spends about half of its time on modeling and half on computing the fitness function. Since BOAP is gear towards more demanding problems (typically with more expensive fitness functions), this should be considered as an upper bound for the percentage of time BOAP will spend on modeling.

For at least one such class of demanding problems outside the scope of this thesis, controling a simulated robot, preliminary results have shown that fitness evaluation indeed dominates modeling costs. The tasks attempted in this domain so far have been "fetch" and the game of tag.

Chapter 5

Conclusion

5.1 Summary

A novel extension of the hBOA algorithm, BOAP, has been developed for learning computer programs. Effective solutions to representational problems have been described, and the algorithm has been compared, theoretically and experimentally, to GP and to PMBGPs. Below is a description of some further directions this research could move in, including ongoing efforts to learn trees utilizing more powerful programmatic constructs.

5.2 Further Directions

5.2.1 Partial Evaluation, Combinators, and Zigzag Trees

While not a concern for the kind of work described in this paper, which uses the standard set of GP operators and terminals, goals for the development of BOAP include more ambitious utilization of functional techniques such as those described in [36], for example, implicit recursion. We are also experimenting with combinators, which are a kind of higher order function that always produce a new higher order function when applied. Some combinators are defined as follows:

$$Ix => x \quad Kxy => x \quad Bxyz => x(yz) \tag{5.1}$$

$$Wxy => xyy \qquad\qquad Sxyz => xz(yz) \tag{5.2}$$

Note that x, y, etc... can be arbitrarily complex parenthesized expressions. Also, all combinators are curried; again, this means that S, for instance, is a function which takes one argument (x), and produces a function which takes one argument (y), and produces a function which takes one argument (z), and produces the result of x applied to z, applied to the result of y applied to z. More details on combinatory logic can be found in [7, 9]. Combinators allow programs to be expressed quite compactly. For example, a zigzag tree evolved by BOAP with combinators and sloppiness to solve the symbolic regression problem $f(x) = x + x^2 + x^3 + x^4$, is the extremely terse $x(*(+x)S)Wx$. The reader is encouraged to try evaluating this tree as an exercise (remember that combinators can take anything as an argument, and that evaluation is left-associative by default). An annotated evaluation trace can be found in appendix B.

Furthermore, a synergy of using combinators with zigzag trees is that, as mentioned in chapter 2, zigzag trees now become a complete representation. That is, it can be proven fairly simply by induction that any binary tree with N leaves can be encoded as a zigzag tree utilizing a couple of simple combinators with at most $2N$ leaves (see appendix A). As a trivial example, the non-zigzag tree $f\ x\ (g\ y)$ is equivalent to the zigzag tree $B\ (f\ x)\ g\ y$. For non-higher order functions this is not important, as expressions can be written linearly without any ambiguity. For higher-order functions however, being able to encode any possible tree is an important feature. This seems to be a fairly novel direction; one previous which applies combinators in an evolutionary computing context is [10], in the context of automated theorem proving.

5.2.2 More Powerful Modeling Techniques

Following [24, 25, 20], BOAP models a population using greedily constructed decision trees, one per variable. One possible direction for future research, somewhat orthogonal to the present work, is replacing this with a more powerful, less greedy learning scheme. But there is a trade-off here: decision trees are not that powerful but learning them is fast. Replacing them with something cleverer will increase the time required in the modeling phase of the algorithm, a cost to be balanced against the benefit of having instances that are on average fitter. As a guess, it seems that replacement of the decision tree learning algorithm will valuable, but only for highly expensive fitness functions.

More ambitiously, the entire concept of what constitutes a variable to be modeled may be reconsidered; it need not be in unique correspondence with the set of values in a particular tree position. An elaboration of this idea is presented below.

5.2.3 Beyond Optimization Algorithms

In Metamagical Themas [12], Douglas Hofstadter develops the metaphor of "twiddling knobs" for the kind of exploration that takes place when manipulating a fixed representation such as an array of bits or numbers. To move beyond optimization algorithms, one needs to focus on methods for creating, removing, and refining knobs, what Hofstadter calls the "crux of creativity". While Hofstadter's view of knob manipulation for well-defined search spaces (and hence optimization and much work in classic AI) as trivial seems an extreme position, the meta-problem of how to pose such problems effectively is certainly is different in kind, and requires a different toolbox.

Part of this distinction is captured by the dichotomy between the straightforward definition of a schema for GA, and the nuanced difficulties involved in defining similar abstractions for GP (see chapter 1). But what is really essential is not the distinction between fixed and open-ended representations, but the one between algorithms that rely on static assumptions regarding problem structure, and those that can *dynamically identify and exploit problem structure*, on multiple levels. Different levels of structure might be, for example, properties of all instances of the traveling salesman problem (most general, "class level" structure), properties of all instances of the TSP with Euclidean edge weights (intermediate), or properties of a particular instance of the TSP (most specific, "instance level" structure). See [37] for an illustration of an algorithm that exploits structure on multiple levels, albeit in a narrow domain (the TSP). GP and BOAP, by the nature of their representations, can handle (in principle) a much greater range of such phenomena. One could say, for example:

$$Hillclimbing < GA < GP < BOAP < ? \tag{5.3}$$

A future system, rather than modeling raw tree variables as BOAP and all current competing approaches do, might dynamically construct a higher-level "invariant representation", tailored to the problem domain and distribution at hand. This creates a feedback loop between perception (representation-building) and cognition (the learning algorithm) as new representations (ways of seeing the problem) lead to new learning, which in turn leads to the construction of new representations, etc...

Appendix A

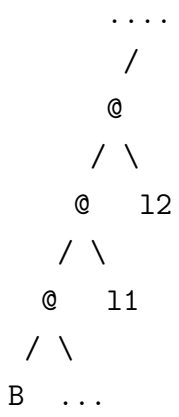
Proof of the Generality of Zigzag Trees

Here is a brief constructive proof of the generality of zigzag trees with the addition of a couple of simple combinators. The algorithm generates zigzag trees with at most twice as many leaves as the input full binary tree. To the best of my knowledge, this is a novel proof; there appears to be no prior work on the utilization of zigzag trees in conjunction with combinators.

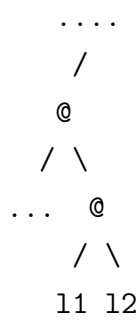
The proof is by induction on N, the number of leaves. For $N = 1$, the tree is already zigzaggy. For any tree T with $N > 1$ leaves ($|T| = N$), let's consider some subtree $@(l_1, l_2)$, where l_1 and l_2 are leaves. For any full binary tree at least one such subtree must exist, otherwise the tree would have infinite depth (i.e., such a subtree exists at the maximal depth of the tree). Consider the tree T' obtained by replacing $@(l_1, l_2)$ with a leaf node l_3. Since $|T'| = N - 1$, there exists an equivalent zigzag tree $Z(T')$ satisfying with $|Z(T')| = k$ and $k \leq 2(N - 1)$.

Let's represent $Z(T')$ as $[x_1, x_2, ..., x_k]$ and $[z_1, z_2, ..., z_{k-1}]$, where the x_i are the leaves, and the z_i are the zigzags ($Left$ or $Right$). Let's say that $l_3 = x_j$, so $1 \leq j \leq k$.

If $j = 1$, we can simply let $Z(T)$ be $[l_1, l_2, x_2, ..., x_k]$ and $[Right, z_1, z_2, ..., z_{k-1}]$. The number of leaves is $k+1$, and since $k \leq 2(N-1)$, $k+1 \leq 2(N-1)+1 < 2N$, we're done. Otherwise, $j > 1$. If $z_{j-1} = Right$, let $Z(T)$ be $[x_1, ..., x_{j-1}, B, l_1, l_2, x_{j+1}, ..., x_k]$ and $[z_1, ..., z_{j-2}, Left, Right, z_j, ..., z_{k-1}]$. This tree will look something like:



Since $Bxyz => x(yz)$, when B is evaluated this becomes:



Thus, evaluation proceeds exactly like for $Z(T')$, only with $@(l_1, l_2)$ in the place of l_3. That is, $Z(T)$ and T are equivalent. The number of leaves is $k + 2$, and since $k \leq 2(N - 1)$, the inequality $k + 2 \leq 2N$ holds, and we're done.

The final possibility to consider now is where $j > 1$ and $z_{j-1} = Left$. This is symmetrical to the case where $z_{j-1} = Right$. Instead of a B combinator however, we will define $Pxyz => (yz)x$, and use it instead. This will place $@(l_1, l_2)$ in the correct position (on the left instead of on the right). Since the size of the tree is the same as the previous case, we're done.

In all cases then, for any tree T, there exists some equivalent zigzag tree $Z(T)$ with at most twice as many leaves. Q.E.D.

Appendix B

Evaluation of $x(*(+x)S)Wx$

Here is what the evaluation of $x(*(+x)S)Wx$, which was discovered by BOAP as a short representation of $f(x) = x+x^2+x^3+x^4$, looks like. The evaluation order is lazy, and the output is from an the interpreter that BOAP uses for evaluation, running in verbose mode. Here is the expression, fully parenthesized.

```
(((x ((* (+ x)) S)) W) x)
```

The + sign binds x as its first argument, and is listified with $*$.

```
(((x ((* +(x)) S)) W) x)
(((x ([*, +(x)] S)) W) x)
```

The combinators now execute in a cascade, to produce a significantly larger expression. Note some combinators produce other combinators, etc., eventually bottoming out in non-combinators.

```
(((x S([*, +(x)])) W) x)
((S([*, +(x)],x) W) x)
((([*, +(x)] W) (x W)) x)
((W([*, +(x)]) (x W)) x)
((([*, +(x)] (x W)) (x W)) x)
((([*, +(x)] W(x)) (x W)) x)
((((x [*, +(x)]) [*, +(x)]) (x W)) x)
((([*(x), +(x)] [*, +(x)]) (x W)) x)
(([*(x), +(x), *, +(x)] (x W)) x)
(([*(x), +(x), *, +(x)] W(x)) x)
(((x [*(x), +(x), *, +(x)]) [*(x), +(x), *, +(x)]) x)
```

Evaluation will now proceed straightforwardly according to the rules of sloppy evaluation to produce a single list.

```
(([*(+(x, *(x, x))), +(x)] [*(x), +(x), *, +(x)]) x)
([*(+(x, *(x, x))), +(x), *(x), +(x), *, +(x)] x)
([*(+(x, *(x, x))), +(x), *(x), +(x), *, +(x)] x)
[*(+(x, *(x, +(x, *(+(x, *(x, x)), x))))), +(x)]
```

To extract a value from this we take the first item, omitting the leading $*$ (since it only has a single argument) to obtain:

```
+(x, *(x, +(x, *(+(x, *(x, x)), x))))
```

Or equivalently:

```
x + (x * (x + ((x + (x * x)) * x))) = x + x*x + x*x*x + x*x*x*x
```

which is equivalent to $x + x^2 + x^3 + x^4$. Q.E.D.

References

[1] D. Allocco, I. Kohane, and A. Butte. Quantifying the relationship between co-expression, co-regulation and gene function. *BMC Bioinformatics*, 5(18), 2004.

[2] P. J. Angeline. Subtree crossover: Building block engine or macromutation? In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Palo Alto, CA, 1997. Morgan Kaufmann.

[3] F. Azuaje and O. Bodenreider. Incorporating ontology-driven similarity knowledge into functional genomics: An exploratory study. In *Proceedings of the IEEE Fourth Symposium on Bioinformatics and Bioengineering (BIBE-2004)*, pages 317–324, Tiachung, Taiwan, 2004. IEEE Computer Society Press.

[4] E. B. Baum, D. Boneh, and C. Garrett. Where genetic algorithms excel. *Evolutionary Computation*, 9:93–124, 2001.

[5] P.A.N. Bosman and E.D. de Jong. Learning probabilistic tree grammars for genetic programming. In *Parallel Problem Solving from Nature (PPSN VIII)*, pages 192–201, Birmingham, UK, 2004. Springer-Verlag.

[6] The Gene Ontology Consortium. Gene ontology: Tools for the unification of biology. *Nature Genet.*, 25:25–29, 2000.

[7] H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, Holland, 1958.

[8] C. Ferreira. Gene expression programming: a new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[9] A. J. Field and P. G. Harrison. *Functional Progamming.* Addison-Wesley, Boston, MA, 1998.

[10] M. Fuchs. Evolving combinators. In *Proceedings of the 14th International Conference on Automated Deduction*, pages 416–430, Berlin, Germany, 1997. Springer-Verlag.

[11] D. Heckerman, D. Geiger, and D. Chickering. Learning bayesian neworks: The combination of knowledge and satistical data. *Machine Learning*, 20:196–243, 1994.

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

[13] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[14] T. Hvidsten, A. Laegreid, and J. Komorowski. Learning rules-based models of biological process from gene expression time profiles using gene ontology. *Bioinformatics*, 19:1116–1123, 2003.

[15] H. Jaske. Prediction of sunspots by gp. In *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, pages 79–88, Vassa, Finland, 1996. University of Vaasa Press.

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

[17] A. Lagreid, T. Hvidsten, H. Midelfart, J. Komorowski, and A. Sandvid. Predicting gene ontology biological process from temporal gene expression patterns. *Genome Research*, 13:965–979, 2003.

[18] M. Looks, B. Goertzel, and C. Pennachin. Learning computer programs with the bayesian optimization algorithm. In *Genetic and Evolutionary Computation Conference (GECCO2005) (forthcoming)*, Washington, DC, 2005. Springer-Verlag.

[19] M. Mitchell. *An Introduction to the Genetic Algorithm*. MIT Press, Boston, MA, 1996.

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

[21] J. J. Oliver. Decision graphs - an extension of decision trees. In *Proceedings of the Fourth International Workshop on Artificial Intelligence and Statistics*, pages 343–350, Fort Lauderdale, FL, 1993. Springer-Verlag.

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

[23] M. Pelikan, D. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic model. Technical Report IlliGAL Report No. 1999018, University of Illinois at at Urbana-Champaign., Illinois Genetic Algorithms Laboratory, 1999.

[24] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. Boa: The bayesian optimization algorithm. Technical Report IlliGAL Report No. 1999003, University of Illinois at at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, 1999.

[25] M. Pelikan, D. E. Goldberg, and K. Sastry. Bayesian optimization algorithm, decision graphs, and bayesian networks. Technical Report IlliGAL Report No. 2000020, University of Illinois at at Urbana-Champaign., Illinois Genetic Algorithms Laboratory, 2000.

[26] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[27] J. Rosca. Genetic programming acquires solutions by combining top-down and bottom-up refinement. In *Foundations of Generic Programming*, Orlando, FL, 1999. Morgan Kaufmann.

[28] R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[29] R. P. Salustowicz and J. Schmidhuber. H-pipe: Facilitating hierarchical program evolution through skip nodes. Technical Report IDSIA-8-98, IDSIA, 1998.

[30] K. Sastry and D. E. Goldberg. *Probabilistic Model Building and Competent Genetic Programming*, pages 205–220. Kluwer Academic Publishers, 2003.

[31] Y. Shan, R. I. McKay, H. A. Abbass, and D. Essam. Program evolution with explicit learning: a new framework for program automatic synthesis. Technical Report CS04/03, Univ. of New South Wales, School of Computer Science, Univ. College, 2003.

[32] Y. Shan, R. I. McKay, R. Baxter, H. Abbass, D. Essam, and Nguyen H.X. Grammar model-based program evolution. In *Proceedings of the Congress on Evolutionary Computation*, Portland, OR, 2004. IEEE.

[33] A. S. Weigend, B. A. Huberman, and D. E. Rumelhart. *Predicting sunspots and exchange rates with connectionist networks*, pages 395–432. Addison-Wesley, 1992.

[34] K. Yanai and H. Iba. Estimation of distribution programming based on bayesian network. In *Proceedings of Congress on Evolutionary Computation (CEC-2003)*, pages 1618–1625, Canberra, Australia, 2003. IEEE.

[35] K. Yanai and H. Iba. Program evolution by integrating edp and gp. In *Genetic and Evolutionary Computation Conference (GECCO2004)*, pages 774–785, Seattle, WA, 2004. Springer-Verlag.

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

[37] W. Zhang and M. Looks. A novel local search algorithm for the traveling salesman problem that exploits backbones. In *19th International Joint Conference on Artificial Intelligence (IJCAI2005) (forthcoming)*, Edinburgh, Scotland, 2005. Morgan Kaufmann.

[38] M. Ziane, M. Za'it, and P. Borla-Salamet. Parallel query processing with zigzag trees. *VLDB Journal*, 2:277–301, 1993.

Vita

Moshe Looks
6652 Washington Ave. Apt. C1
St. Louis, MO 63130
phone 202-641-2157
email *moshelooks@yahoo.com*

Work Experience

2003-present Software Design and Research on the Novamente AI Engine, Object Sciences Corp.
2002-2003 Software Design, Novamente LLC
2000-2001 Programming, Wmap (Wireless Mobile Advanced Push)
2000 Programming, GoldNames Ltd.
2000 Web Development, Darche Noam Institutions
1999-2000 Undergraduate Assistant in Physical Chemistry for Professor Haim Levanon
1999-2000 Web Development and Translation, Catalytic Systems Technologies

Education

2002-2003, 2004-2005 Washington University in St. Louis
Master's Degree in Computer Science (expected). GPA: 4.0. Thesis: Learning Computer Programs with the Bayesian Optimization Algorithm.
1998-2002 The Hebrew University of Jerusalem
Bachelor's Degree in Computer Science, Magna Cum Laude. Dean's List, 2000-2002.
1998-2000 Midrash Shmuel Institute of Advanced Torah Studies (Part-time).

Software Design Skills

Object oriented, generic, and functional paradigms. Extensive experience in C++, Java, C, Awk, Perl, Matlab, PHP and JavaScript. Some experience with Scheme, Prolog, and others. SQL.

Publications

Weixiong Zhang and Moshe Looks. A Novel Local Search Algorithm for the Traveling Salesman Problem that Exploits Backbones *19th Intern. Joint Conf. on Artificial Intelligence (IJCAI-05).*
Moshe Looks, Ben Goertzel and Cassio Pennachin. Learning Computer Programs with the Bayesian Optimization Algorithm, *Genetic and Evolutionary Computation Conf. (GECCO 2005).*
Moshe Looks, Ben Goertzel and Cassio Pennachin. Novamente: An Integrative Architecture for General Intelligence, *The 2004 AAAI Fall Symp. on Achieving Human-Level Intelligence.*
Weixiong Zhang, Ananda Rangan and Moshe Looks. Backbone Guided Local Search for Maximum Satisfiability, *The 18th Intern. Joint Conf. on AI (IJCAI-03).*

May 2005