Semantically-Driven Search Techniques for Learning Boolean Program Trees

by
Nicholas Charles Miller

Bachelor of Science
Computer Engineering
Georgia Institute of Technology
2006

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May 2013

We the undersigned committee
hereby approve the attached thesis

Semantically-Driven Search Techniques for Learning Boolean Program Trees

by
Nicholas Charles Miller

Philip K. Chan, Ph.D.
Associate Professor
Computer Science
Principal Advisor

Marius C. Silaghi, Ph.D.
Assistant Professor
Computer Science

Georgios C. Anagnostopoulos, Ph.D.
Associate Professor
Electrical and Computer Engineering

William D. Shoaff, Ph.D.
Associate Professor and Department Chair
Computer Science

Abstract

Title: Semantically-Driven Search Techniques for Learning Boolean Program Trees
Author: Nicholas Charles Miller
Principal Advisor: Philip K. Chan, Ph.D.

Genetic programming has been around for over 20 years, yet most implementations are still based on sub-tree crossover and node mutation, in which structural changes are made that manipulate the syntax of programs. However, it is not clear why manipulating program syntax should have any desirable effect on program behavior (or semantics). One sub-field of genetic programming which has gained recent interest is semantic genetic programming, in which programs are evolved by manipulating program semantics instead of program syntax. A semantic GP (SGP) implementation exists that operates on program semantics through composition of sub-programs, but has the drawback that the evolved programs are large and complex. This paper will propose two new algorithms, SGP+ and SDPS, that aim to search the semantic space of programs in a more effective manner than the existing SGP algorithm. Experimental results on "deceptive" Boolean problems show that programs created by the SGP+ and SDPS algorithms are 3.8 and 32.5 times smaller than SGP respectively, while still maintaining accuracy as good as, or better than, SGP. Additionally, a 17.6% improvement in program accuracy was observed for several high-arity Boolean problems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Genetic programming (GP) has been around for well over 20 years, and sub-tree crossover remains the most widely used recombination operator. This type of genetic programming, popularized by John Koza [4], represents a program as a tree, and crossover works by swapping sub-trees. This is an operation on the structure (or syntax) of the program.

The reason for swapping sub-trees is not entirely clear or justified. Why should swapping one part of a random program with another part of a different random program create a better offspring program? There are no guarantees that swapping program syntax will have a desirable effect on program semantics. In short, there is no rigorous schema theory on the building blocks in genetic programming, such as there is with genetic algorithms [10]. In reality, the relationship between program syntax and program behavior, or semantics, is a complex one. Even minor changes to program syntax can have drastic changes to program semantics. This relationship is also sometimes referred to as the genotype-phenotype mapping.

Most of the syntax-modifying GP algorithms are of the generate-and-test variety. In other words, they focus on randomly generating a new program by modifying syntax of existing programs and then test how well the behavior matches the desired behavior. Stated another way, there exists some desired program behavior in a *semantic space*, and traditional GP operates on programs in a *syntax space* with the hope that the generated program will have semantics that are close to the desired semantics in the semantic space. Because the mapping between syntax space and semantic space is often very complex, this generate-and-test approach may not work well for all types of problems.

There has been an increased interest in semantically-driven genetic programming in recent years as an alternative to syntax-based GP representations[13]. The goal is to perform a more

direct search in semantic space, as opposed to an indirect search via the complicated syntax-semantic mapping. The goal for this paper is the same, but will not be strictly limited to genetic programming, but rather to semantically-aware search techniques in general. Furthermore, the focus will be on solving "deceptive" problems, which are problems with a particularly complicated syntax-semantic mapping, as these are the types of problems in which traditional GP falls short.

Boolean problems will be the focus of this paper, though the ideas proposed are extensible to other domains (e.g. regression). Potential applications of Boolean function learning are optimization of FPGA combinatorial logic, electronic design automation (EDA), discrete classification problems, or any problem that can be expressed in truth table form.

1.1 Problem Statement

Given a set of n input-output pairs (or instances) $T = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_n}, y_n)\}$, the problem is to find a model, or hypothesis, $h : \mathbf{X} \to Y$ that interpolates all known input-output pairs:

$$\forall(\boldsymbol{x_i}, y_i) \in T, h(\boldsymbol{x_i}) = y_i \tag{1.1}$$

This is essentially the problem of supervised machine learning. Each input $\boldsymbol{x_i} \in T$ is a vector of attributes and each output y_i is a single value, sometimes referred to as the class for discrete domains. The focus of this paper is on the Boolean problem domain, so the input space (domain) is $\boldsymbol{X} = \{0, 1\}^n$ and the output space (codomain) is $Y = \{0, 1\}$. The restriction to the Boolean domain is done primarily for simplicity of implementation and analysis, but the ideas apply equally well to other domains.

The space of hypothesis functions, H, is the space of all possible Boolean program trees. An example of a Boolean program tree is shown in Figure 1.1.

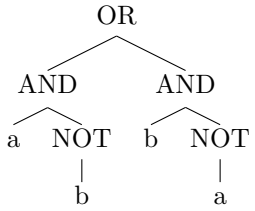

Figure 1.1: Example of a program tree in H

This tree is an example of the 2-input odd-parity (a.k.a. 2-input XOR) function, which returns 1 if there are an odd number of 1s in the input vector $\boldsymbol{x_i}$. To preserve space, trees

will sometimes be represented by their corresponding S-expression, for example (OR (AND a (NOT b)) (AND b (NOT a))).

As is the case with most supervised learning, the goal is to find a model that is both simple, comprehensive in prediction of instances in T, and generalizes well to *unseen* instances that are not in T.

1.2 Overall Approach

The focus will be on constructing models by directly utilizing program tree semantics, or behavior. The semantics of a program tree h' can be expressed as a vector $\boldsymbol{Y}'$ corresponding to the output of the tree for each $\boldsymbol{x_i} \in T$. For example, if h' were the program tree (OR x_1 x_3), then the semantics of h' would be the output of the tree for each input case in T. This can be visualized in truth table form as in Table 1.1.

Table 1.1: Example of program semantics for hypothesis function $h' =$ (OR x_1 x_3)

x_1	x_2	x_3	$y' = h'(\boldsymbol{x_i})$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Here, each row corresponds to an instance in T and the final column represents the semantics of h' (e.g. 01011111). Bitstrings will be used as a notational convention for semantics throughout this paper.

The notion of semantic space will be used throughout. This is the multidimensional hyperspace of semantic vectors Y'. The number of dimensions is equal to the number of input-output pairs in T. The semantics of a hypothesis program h' can be represented as a single point in this space, as can the target semantics Y from T. The problem of model construction then becomes a search for a suitable hypothesis program h' in this semantic space. Also note that a single point can correspond to many different possible hypotheses. That is, there may be more than one program tree that can produce the semantics represented by a point. Using the example from Table 1.1, a program with equivalent semantics would be (NOT (AND (NOT x_1) (NOT x_2))) and would be represented by the same point in semantic space.

Two distinct approaches will be taken to search for a suitable h'. The first is based on the

semantic *composition* of programs. In other words, new programs are created by composing two sub-programs with an associated Boolean operator. An example of this type of composition is shown in Figure 1.2. The semantics are shown above each node to convey that it is the *semantics*

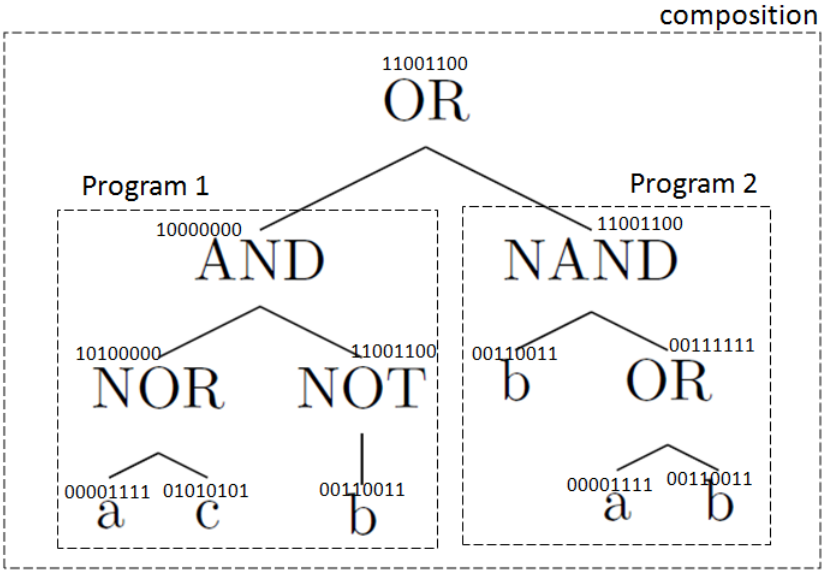

Figure 1.2: An example of program tree composition

that are being composed. Also note that each of the inputs have associated semantics as well. These correspond to the input columns in the truth table.

The second type of approach taken will be based on the *decomposition* of program semantics. This approach starts with the desired output (a.k.a. the target semantics) and recursively decomposes the target into distinct sub-targets, creating a kind of divide-and-conquer approach. An example of this type of decomposition is provided in Figure 1.3.

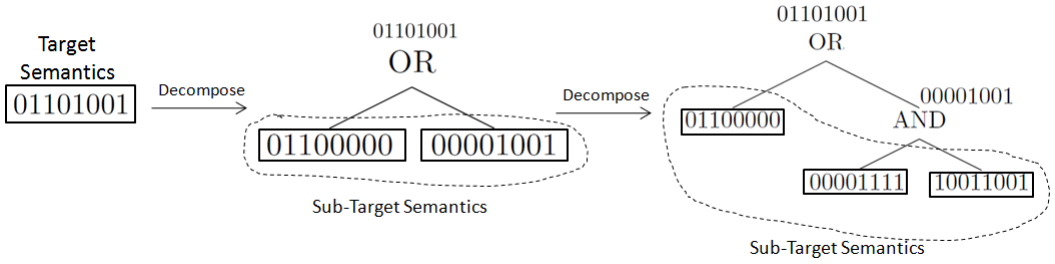

Figure 1.3: An example of semantic program tree decomposition

Composition of programs is a "bottom-up" type of approach. Starting from some base set of trivial programs (i.e. leaves), progressively larger and more complicated programs are built via composition. When the program exhibits the desired semantics the algorithm terminates with the final compositional operator acting as the program root. In contrast, the decomposition of semantics is a "top-down" type of approach, where the program tree is built one node at a time, starting from the root and ending at the leaves. When the sub-target semantics match the semantics of an input variable, the sub-tree terminates in a leaf.

The compositional approach will be implemented as a genetic programming algorithm. It is similar in nature to a randomized beam search for programs in semantic space, where the beam width is the size of the population and the randomization comes from the stochastic nature of the selection, crossover, and mutation operators. The decompositional approach will be implemented as a hill-climbing greedy search, where semantic sub-target designations are made using heuristics. In contrast to genetic programming where multiple programs are considered, the greedy search will only operate on a single program.

Many of the Boolean problems discussed are *deceptive* in nature. In the context of GP, deceptive means that the search may be deceived if there is not a clear path in the search space from a promising individual to the individual that solves the problem. Stated informally, the fitness landscape is rocky and the path to the goal is unclear. These types of problems are prone to reduced population diversity, as locally optimum solutions begin crowding the population. The Boolean parity problems are deceptive in nature, because minor changes to program structure can result in drastic changes in program fitness, which impedes the search from moving in a potentially promising direction. Solving these types of problems will be the primary focus of this paper.

1.3 Overview of Contributions

- An improved semantic GP algorithm (SGP+) is proposed that searches semantic space directly. It is an improved version of the existing SGP algorithm, and is more selective in choosing parents for recombination so that offspring programs will be closer to the target program in semantic space.

- A greedy search-based algorithm is proposed, called Semantic Decomposition for Program Search (SDPS), that navigates semantic space by creating semantic sub-problems. The key idea is to minimize the number of branches by using heuristics to choose sub-problems that minimize the semantic distance to the program tree leaves (i.e. inputs).

- Experimental results show a significant reduction in final program size compared to the

existing SGP algorithm. For the deceptive Boolean parity problems, the program output by SGP+ is 3.8 times smaller than SGP, on average. For SDPS, the average program size is 32.5 times smaller than SGP.

- Experimental results show SGP+ and SDPS exhibit a significant 17.6% improvement in classification accuracy for high-arity deceptive Boolean problems.
- The proposed algorithms exhibit better generalization to unseen instances than the existing SGP algorithm on 4 out of 5 tested UCI classification problems.

1.4 Overview of Chapters

This paper is divided into five main chapters, including this one. Chapter 2 provides a brief overview of past research in the field of semantic search. This includes the sub-fields of semantic novelty, semantic genetic programming, and semantic modularity and decomposition. Chapter 3 introduces two algorithms - SGP+ and SDPS - that are designed to be improvements on existing semantic algorithms. Chapter 4 presents the results of experiments on synthetic and real-world classification problems and provides some analysis and interpretation of results. Finally, Chapter 5 summarizes results, draws conclusions, and discusses potential areas of improvement and future research.

Chapter 2

Related Work

This chapter will provide a brief history of related research in the field of semantically-inspired program search techniques. The first few sections discuss existing techniques and models for solving Boolean problems. The remainder of the chapter is dedicated to prior research in the field of semantic search. The final section includes a detailed description of the SGP algorithm, from which the SGP+ algorithm proposed in Chapter 3 will be based upon.

2.1 Boolean Simplification Techniques

It is important to note that there exist many different Boolean simplification algorithms and techniques. Among the most well-known are heuristic techniques such as Karnaugh maps and the Quine-McClusky algorithm. Also, Espresso is a popular software package for performing logic minimization. These algorithms have good heuristics, and can often produce near-optimal Boolean simplification for small- to moderately-sized Boolean functions. However, one limitation of these techniques is that they only work in the Boolean domain. In other words, they cannot be extended to other domains (e.g. regression). These existing methods are useful as a baseline for comparison for the effectiveness of the algorithms proposed in this paper.

2.2 Classification Models

The model chosen is a program tree, where each of the internal nodes represents a Boolean function and the leaves are the program inputs. This is a fairly non-restrictive model, as the tree can grow in any direction and to any depth. Contrast this with the model created by the ID3 algorithm (for example), and the potential utility of this model becomes clear, as depicted

in Figure 2.1. In this case, the decision tree model is restricted to be a disjunction of Horn

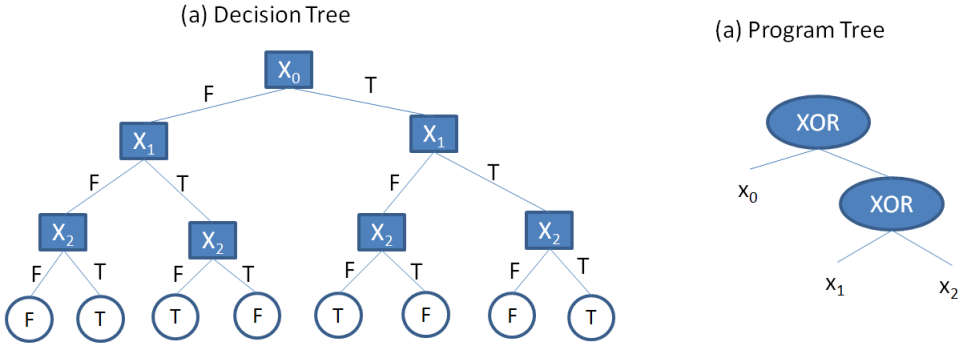

Figure 2.1: (a) Decision tree model created by ID3 for the 3-input odd parity problem. (b) Program tree model for the same problem.

clauses (i.e. If-Then rules for each path), whereas the program tree model is potentially more flexible/less restricted, which can result in a more compact and efficient representation.

2.3 Syntactic Search vs. Semantic Search

There are several key distinctions between program search in syntactic space and semantic space. First, syntactic search explores the space of program *representations* whereas semantic search explores the space of program *behaviors*. Second, a single point in the syntactic space represents a syntactically unique program, and corresponds with a *single* point in the semantic space (i.e. a program only has one behavior). In semantic space, a single point represents a semantically unique program, and has *multiple* corresponding points in syntactic space (i.e. behavior can be represented by multiple different syntactically-unique programs). Finally, and perhaps most importantly, the syntactic space is *infinite* whereas the semantic space is *finite* for discrete domains (i.e. there are only so many unique program behaviors). This means that semantic search should be easier for discrete problems because of the smaller space.

In syntactic space search, programs are manipulated by changing the syntax of programs. For example, changing the operator at an intermediate node in the tree, or performing sub-tree crossover between different trees. The key point is that only the syntax and structure of the program are changed directly, and any effect these have of the semantics of the tree will be indirect. This usually results in a generate-and-test methodology, where program syntax is manipulated and program semantics are checked for accuracy. In contrast, semantic space search focuses on manipulating programs by changing the behavior of trees directly. This is more difficult than syntax-based manipulations because in general it is not known how changing

the semantics of a tree will affect the syntax of the tree. There are multiple ways to address this difficulty, such as utilizing a small library of known programs or combining programs so as to multiplex their outputs. Both of these techniques will be used in Chapter 3.

2.4 Semantic Novelty

The idea of *semantic novelty* was introduced in [9]. This paper is focused on searching for novel genetic programs in an effort to promote diversity and reduce overfitting in deceptive problem domains. Rather than guiding the evolution using an objective fitness function, they guide *solely* based on program novelty (i.e. how different the behavior is from other programs in the population). They motivate this idea by pointing out that natural evolution continually produces novel forms and is more open-ended in that it does not necessarily have a final objective. In this sense, it is a divergent search. They propose a *novelty metric* to replace the fitness function which will reward programs with more novelty. The metric is computed based on the behavior (or phenotype) of an archive of programs, and programs are added so as to penalize future programs exhibiting similar behavior. The calculation consists of measuring the distance of a program from other programs in *semantic space* (i.e. the sparseness of a program in the space of unique behaviors). This "novelty score" is calculated as the average distance to the k nearest neighbors (from the archive) in semantic space.

$$\rho(x) = \frac{1}{k}\sum_{i=0}^{k} dist(x, \mu_i) \tag{2.1}$$

The higher the value, the more sparse the region is, and the more likely the program is novel. Programs from a particular generation are added to the archive with uniform probability, which they claim works better than adding only high-novelty programs because it allows further exploration of a local subspace. The authors also claim that although novelty search is general enough to be applied to any domain, it works best when the fitness landscape is deceptive and the domain restricts the number of possible program behaviors. If there are only so many possible program behaviors, then the novelty search may stumble upon the target function. Their results indicate that the novelty search had a higher success rate than standard GP and exhibited smaller program sizes (i.e. less bloat) than the fitness-based or random search approaches. The reduction in program bloat is hypothesized to be due to the fact that bloat would be maladaptive to novel programs. In other words, bloat can be thought of as a "guard" against change, which is used to preserve high-fitness individuals in fitness-oriented evolution. But this "guard" is not present in novelty search because it is rewarded solely *for* change.

In [2] it was pointed out that sometimes the search for novelty alone is not enough. This

paper compares traditional objective-based fitness functions with novelty-based (i.e. no objective) fitness functions. Additionally, they show how novelty search can be used in traditional objective-based search to sustain diversity. The takeaway from this paper is that using novelty as the sole criterion for selection results in high behavioral diversity, but not necessarily improved overall fitness. By combining novelty with objective fitness, better solutions are found.

In [3], different methods for promoting semantic diversity in genetic programming are investigated. By introducing a new crossover operator that promotes unique program behaviors in a population, a significant improvement in success rate was observed over a range of different problem types. The first part of the paper measures structural, behavioral, and fitness diversity using a standard tree-based GP. Results showed that for most problems, structural diversity is preserved and actually increasing at each generation. Behavioral diversity on the other hand always decreased with each generation. This makes intuitive sense because later generations will typically have members with many introns (i.e. "dead code") that produce structurally diverse programs, but which exhibit identical program behavior. To improve behavioral diversity, a two-step process is used - (1) establish sufficient diversity in the initial population, and (2) maintain diversity in each generation with a new diversity-promoting crossover operator. For the second step, crossover is modified to repeat until a unique individual has been created, with some maximum number of attempts.

The search for novelty continues in [12], where deceptive problems are solved with a multi-objective evolutionary algorithm designed to promote both behavioral diversity within the population as well as behavioral novelty overall. Results showed that utilizing behavioral diversity and behavioral novelty improves the evolution of neural networks and is similar in performance to the well-known NEAT algorithm.

2.5 Semantic Genetic Programming

In [10], an investigation is made into the nature of semantic building blocks in genetic programming. This paper analyzes the effects of the crossover operator in genetic programming on the semantics of the resulting program. They find that the majority of crossovers operations produce programs that are *semantically identical* to the parent programs, resulting in no movement towards the final objective. Their experiments are performed in the context of Boolean problems, but should generalize to other domains (e.g. symbolic regression). The authors begin by stating that subtree crossover involves two distinct components - the subtree to be replaced as well as the context, which is the rest of tree not including the subtree. This decomposes the problem of describing the semantic impact of crossover to describing the semantics of subtrees, the semantics of contexts, and their interactions after crossover is performed. The semantics of

subtrees are fairly straightforward. Given all possible inputs, determine the resulting outputs. This completely encompasses the semantic behavior of a subtree. For example, the tree (AND x y) could be semantically described by the string 0001. This semantic string can be thought of as the final vertical column in a truth table. The semantics of contexts are a little more complicated, as they must take into account the parent operator of the subtree removed, the parent semantics obtained by removing the subtree, and the subtree semantics of any other arguments to the parent operator (see Figure 2.2). An example of a tree with fully specified

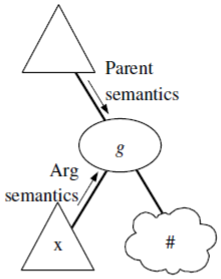

Figure 3: Illustration of the interaction of the different components in computing the context semantics. We have a tree with some subtree removed (the insertion point, indicated by the # in the lower right). g is the parent node of the insertion point, and x represents the other argument of g (i.e., the sibling subtree of the insertion point). Note that x is not necessarily a leaf but can represent an arbitrarily complex node. The semantics of this context is a function of the specific operator g, the semantics of the context obtained by removing the subtree rooted at g, and the subtree semantics of x.

Figure 2.2: Example of components needed for calculating tree context (from [10])

sub-tree semantics and context semantics is provided in Figure 2.3. As a simpler example of fixed semantic context, consider the tree (AND # false), where # is a removed subtree. In this case, the value output by the tree will always be false, regardless of the sub-tree at #. This means that any crossover that occurs within the sub-tree rooted at # will have no effect. The authors analyze crossover as it is applied to several different even-parity problems. During evolution, they track three primary metrics

- The proportion of fixed contexts. Out of all possible contexts in all population members, how many are fixed.
- Construction likelihood. The probability of constructing the target program via subtree crossover.
- Proportion of compatible contexts. Out of all possible contexts in all population members, how many are compatible with the target program. A context is compatible if all

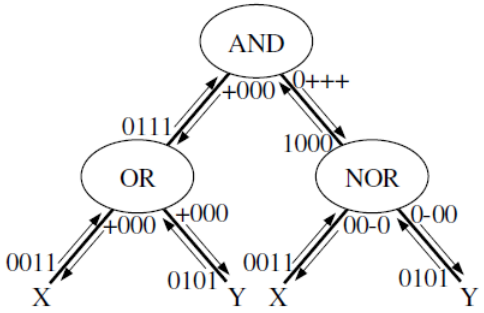

Figure 2: A sample syntax tree showing both subtree and context semantics. The arrows pointing upward (on the left of the edges) are the semantics of the subtree below them, e.g., the semantics of `(or x y)` is `0111`. The arrows pointing downward (on the right on the edges) are the semantics of the context obtained by removing the subtree below the arrow, e.g., the semantics of `(and # (nor x y))` is `+000`.

Figure 2.3: Example of fully-specified tree semantics and context (from [10])

semantically fixed positions match the corresponding position in the target program.

Their results indicate that for easy problems, the percentage of compatible contexts is nearly 100%. However, for harder problems, the compatible contexts level off indicating they are stuck in local optima and that it is unlikely that crossover will break out of it. Additionally, they found a strong correlation between the percentage of compatible contexts with the probability of constructing the target. This indicates that having the correct context is critical to the success of the search, and most likely more important than the choice of the subtree itself. Finally, they report that the percentage of fixed contexts generally doesn't change much over time, and that it is always over 60%. This means that crossover with randomly chosen subtrees will result in no semantic change 60% of the time, which is an alarming result. The idea of program context will be revisited in Chapter 3.

Various methods have been proposed for utilizing program semantics in genetic programming to counteract the inefficiencies discovered in [10]. For example, in [14], a semantically-aware crossover operator is created in order to approximate the Gaussian Q-Function, for which no closed form currently exists. To begin, the authors introduce Sampling Semantics (SS) for determining the semantics of a sub-tree. SS is a set of values obtained by evaluating the sub-tree on inputs corresponding to a sequence of points from training data. Next, they define Sampling Semantics Distance (SSD) between two sub-trees. For two SSs U and V (obtained using identical input sequences), the SSD is calculated as the mean absolute difference between corresponding values in U and V. Finally, these are tied together to form the Most Semantic

Similarity based Crossover (MSSC). In essence, the idea is to perform crossover between two parents such that small semantic changes are made. In MSSC, this is done by randomly selecting N crossover point (i.e. sub-tree) pairs from the parents and calculating SSD of each pair. The pair of parents that produces the smallest SSD value is chosen for crossover. In other words, the parents with the most semantically similar (but not equivalent) sub-trees are chosen, and then crossover proceeds as normal. Their results demonstrated that they were able to evolve approximations that were better than the best known human-made approximation in 3 out of 50 runs. With standard crossover, a better approximation was never found.

In [7], an investigation is made into the semantic similarities and differences between a random sampling of programs. In particular, they analyze the variety of tasks solved (semantic diversity), the complexity of tasks solved, as well as the modularity of programs that solve a particular task. The concept of a program is formalized and implemented in a simple programming language (Push) for simplicity of analysis, but the results should generalize well to programs in general. To begin, they define a program as a finite set of instructions that, when executed, will change the contents of memory. Before execution, memory contains the program inputs, and after execution, the program outputs. Additionally, intermediate memory contents during execution are also considered. To generalize to all programs (not just a particular program), they use the concept of memory *state*, which is a finite set of memory contents (i.e. multiple "copies" of memory, each with different contents). Before execution, state s_0 represents the set of all possible inputs, which is the starting point for all programs. During execution, new states are reached by executing an instruction, which changes the memory contents for each of the possible inputs. A program trace is a path through these memory states, starting from s_0 and ending in state s, which is the set of outputs generated by the program for all possible inputs. State s is also defined as the task solved by a program whose trace ends in s. Additionally, multiple programs may end in state s, meaning that they solve the same task. Furthermore, programs may reach the same intermediate memory states both between programs, and within a single program. Figure 2.4 provides a visualization of this process. Rather than try to solve benchmark problems or analyze a suite of existing programs, they randomly generate programs and observe the different tasks solved, or the program semantics. They define three measures of programs:

1. Semantic diversity - Number of unique tasks solved by a population of programs
2. Internal semantic diversity - Number of unique memory states reached during execution of a single program
3. Task complexity - The minimum number of execution steps to reach a task s amongst all programs

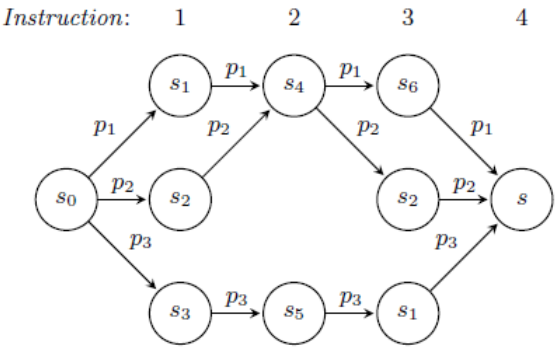

Figure 1: Graphical representation of three distinct traces of three programs p_1, p_2, p_3, each of length 4, that solve task s. Vertical columns of states correspond to consecutive instructions executed by programs. Note that a program can revisit the same state (p_2 visits state s_2 twice), and different programs can visit the same state on different stages of their execution (both p_1 and p_3 visit s_1).

Figure 2.4: Example of program traces (from [7])

The author uses these metrics in three different experiments to analyze program complexity, program diversity profiles, and program modularity. For the first experiment on program diversity and complexity, their results indicate that for programs limited to a small number of instructions, semantic diversity levels off. Likewise, if a high number of instructions is allowed, the semantic diversity curve is steeper. These results seem consistent with intuition, as longer programs are allowed to explore larger portions of the semantic space. Finally, they found that nearly all tasks had complexity less than 20 (i.e. required no more than 20 instructions) and that the majority of tasks had complexity of 10. These results indicate that the length of the program has an ideal number of instructions needed to solve the problem, where less will be unable to solve the task and more will be unnecessary. The second experiment investigates diversity profiles, or how the semantic diversity of all programs changes during execution. Before doing that, they formalize the idea of modularity. For a set of programs that solve a task, if all of the tasks pass through a memory state s', then the problem is said to modular, in that it could be split into two independent subtasks, one from s_0 to s', and another from s' to s. This can be visualized as the "waistline" in a program trace diagram. The second experiment consists of calculating relative diversity amongst all programs solving the task at each execution step. The relative diversity metric takes into account the number of unique states reached by all programs. A low relative diversity indicates that a "waistline" exists in

the task trace diagram. The relative diversities are then clustered to create similar diversity groups, which are then plotted. It can be seen that the there are clearly program types that exhibit dips in diversity, which indicates that they are modular in nature. The third experiment seeks to overcome the shortcomings of the second experiment - namely that it is naive to assume that semantically similar programs will reach the same memory state at the same instruction step. They introduce a new metric to calculate the centrality of a state s', which will be larger if two conditions exist: the state s' is near the middle of the task trace diagram, and many tasks pass through s' at some point during their execution. They then identify the state s' that has maximal centrality, as this will be the most likely place to decompose a problem into subtasks. They plot task complexity versus maximal centrality and observe a monotonous tendency for centrality to increase with complexity. What this means is that as tasks become more complex, they also become more modular. This is the main result of the paper. The paper concludes by mentioning that this is an investigation into the nature of programs solving a particular task, but in general, we are trying to solve the task via genetic programming, so more work is needed to incrementally find modularity as the program evolves.

In [5], a semantic-based search technique which involves transforming program space into semantic space is discussed. Typically, program space is very large, containing many syntactically unique variations. However, many of these programs are semantically the same, in the sense that they produce the same outputs given a particular set of inputs. This paper is concerned with identifying all unique program semantics and placing them in the context of a semantic space, which can then be searched using GP. This search is more efficient primarily because the search space is smaller, and the fitness landscape is smoother than its program space counterpart. The main achievement of this paper shows that semantic embeddings of small programs (i.e. semantic spaces where each point represents the semantics of a depth-limited tree-based program) can be exploited in a compositional manner to build larger compound programs, resulting in more effective search in the larger space of programs. Figure 2.5 provides some intuition about the meaning of semantic embedding. Here, the semantic space is an abstract space where each point in the space represents a set of equivalent programs (syntactically unique programs with identical semantics). The point itself is represented as vector of multiple program outputs corresponding to program inputs. Additionally, a neighborhood is associated with each point, with the goal that neighbors have similar semantics. A similarity measure, referred to as locality, exists based on the euclidean distance between program semantics (not to be confused with the distance in the abstract semantic space). There exists a bijective mapping between semantic equivalence classes and points in the semantic space. It is important to note that semantic space is independent of any specific problem or fitness function. Rather, it is representation of all programs of a specified length, organized by semantic

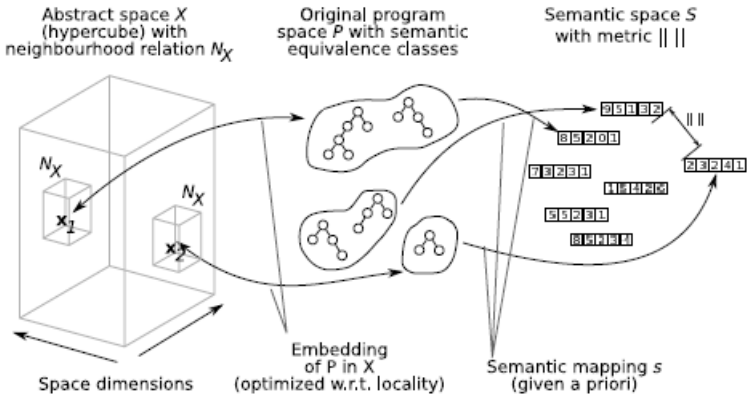

Figure 1: Conceptual sketch of semantic embedding.

Figure 2.5: Semantic embedding (from [5])

locality. Obviously, this semantic space will be much smaller, but additionally, it is desired that the fitness landscape be smoother so the search is easier. The paper demonstrates a link between locality and fitness landscape based on the triangle inequality. In words, as two points in semantic space get closer, so does the difference in their fitness values with respect to some target point. This means that small changes in semantic space do not drastically change the fitness, resulting in a smoother landscape across all programs. Furthermore, this smooth landscape applies for all possible target points in the semantic space, resulting in smoother fitness landscapes for all possible problems that can be represented in the space. This is in contrast to the syntactic space, where "nearby" programs can have wildly different fitness, resulting in a rocky fitness landscape. Therefore, it is important to create a semantic space with high overall locality. This semantic space organization can be done effectively with heuristic greedy algorithms. To determine a near-optimal organization of the semantic space, a simple greedy algorithm is applied:

1. Randomly distribute program semantics in the semantic space.
2. For each point, calculate the change in overall locality (i.e. locality over all points) as a result of swapping with all points within its neighborhood. Choose the neighbor swap that maximizes the increase in overall locality.
3. Repeat the step above until overall locality converges.

An illustrative example of the output of this algorithm is shown in Figure 2.6. Note also that

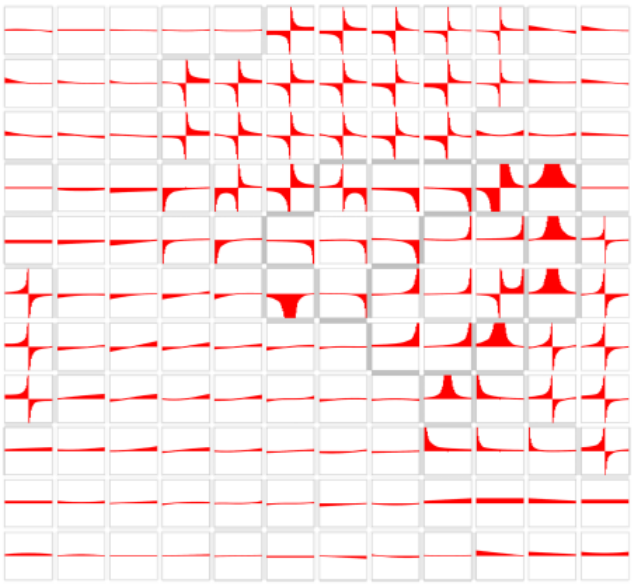

Figure 2: An exemplary optimized two-dimensional toroidal embedding of all 132 unique semantics obtained from 1024 programs of depth 3 composed of instructions $+, -, *, /, v, 1$. Each tile plots the function calculated by the corresponding representative program (its semantics) for $v \in [-1, 1]$. The darker the shading between the neighboring tiles, the greater the semantic distance. The locality of this embedding is higher than that of random embedding, which could be obtained by reshuffling the tiles randomly (see text for more details).

Figure 2.6: Greedy organization of abstract semantic space (from [5])

the choice of semantic space size and dimensionality is mostly arbitrary. The authors use a maximum dimensional size of n and number of dimensions d such that n^d is at least as big as the number of semantically unique programs. Ultimately, this semantic space is used to guide GP. To this end, they define a population member as a point within the abstract semantic space, crossover as a random point geometrically between two parents, and mutation that takes two flavors: either a single step in a particular dimension or a completely random point in the space. Fitness is calculated as the semantic distance from a point to some target.

2.6 Semantic Modularity and Decomposition

This section will present research related to semantic modularity and decomposition. In other words, using the semantics of known programs to improve semantic program search.

In [8], module identification and module exploitation in GP is considered. The paper introduces *monotonicity* as a means of assessing how useful a particular subgoal is in finding good modules. A good module is a subprogram whose output is highly correlated with the fitness of the entire program, independent of the subprograms context. In general, their approach is to generate many random programs (GP trees), define some constant tree decomposition that is used for all programs (decompose into subprogram and context), and then observe the monotonicity of all possible subgoals over the population of subprograms. To begin, they formalize program decomposition as an invertible function that decomposes a program x into a subprogram, or part, p and context c, i.e. $d(x) = (p, c)$. Then they define the part quality function that assesses the similarity between a vector of outputs of a part and a "subgoal", which is vector of expected outputs. Next they define monotonicity using Spearman's rank correlation coefficient. In short, this measures how much the quality of the parts are correlated with the fitness of the program. Next, they define the optimal part quality as the part quality function (tied with a subgoal) that maximizes the monotonicity. Finally, a problem is called α-modular if the monotonicity of the optimal part quality function is greater than or equal to α. The first question the paper wishes to answer is: "Given a problem, are there any differences between subgoals in terms of monotonicity?" In other words, they wish to observe the distribution of monotonicity over all subgoals for a given problem (e.g. XOR-3). To answer this, they generate 100,000 random Boolean-valued GP trees of depth 17. All possible subgoals are enumerated, where a subgoal is a vector of outputs corresponding to fitness cases (2^8 subgoals for 3 inputs). Monotonicity (correlation between part quality and fitness) is evaluated for each subgoal and then plotted. It is observed that for the XOR-3 problem, the distribution of monotonicity is highly skewed, with a very few number of subgoals having good monotonicity. This is expected, as the XOR-3 problem is intuitively unmodular. In contrast, the distribution for the OR-AND

problem is more evenly distributed, indicating that is may be more modular or more decomposable. This is depicted in Figure 2.7, where sub-goals on the x-axis are sorted by increasing monotonicity. Another question the paper wishes to answer is: "Is it possible to reliably es-

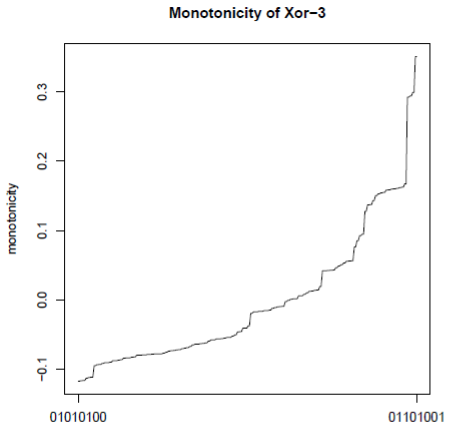

Figure 1: Monotonicity of all subgoals for Xor-3 problem estimated from 100,000 random individuals.

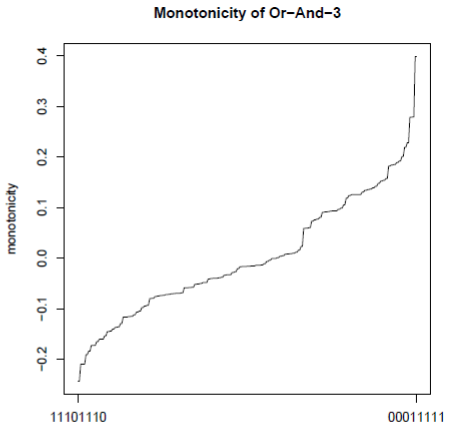

Figure 2: Monotonicity of all subgoals for Or-And problem estimated from 100,000 random individuals.

Figure 2.7: Monotonicity for XOR3 and AND-OR functions (from [8])

timate monotonicity from an evolving population?" They run a standard fitness-based GP algorithm and at the end, they calculate monotonicity of each subgoal (averaged over 100 runs) and plot against the unbiased monotonicity distribution. They find high correlation between the two curves, indicating that monotonicity can be estimated in a fitness-biased sample of programs. Furthermore, the OR-AND function (which is expected to exhibit more modularity) has a higher correlation to the unbiased curve, possibly indicating that more modular programs have monotonicity that is easier to estimate. The paper concludes by saying that monotonicity can be used to effectively identify modularity, and can be exploited during population evolution where there is fitness-bias. This is a useful finding, because it means this method could be applied to identify modules and break them off as a separate sub-problem, reducing the search to two smaller problems - finding a program that produces the subgoal, and finding the context that, in combination with the subgoal solution, will produce the desired final program output.

2.7 Geometric Semantic GP

Geometric semantic GP is a sub-type of semantic GP that focuses on producing offspring that hold some geometric relationship with their parents in semantic space. This is desirable because

it allows the search to explore the space in a predictable and manageable way.

The idea of *semantic mediality* was discussed in [6] where the goal is to find an approximately medial crossover. In other words, given two parents the goal is to produce offspring that are near the geometric midpoint of the parents in semantic space. Because this midpoint may not exist in discrete domains, an approximate nearest point is found. The search for the point nearest the midpoint is exhaustive (considers all possible programs) which is infeasible, so the algorithm operates only on sub-programs (e.g. sub-trees) of reasonable size. The idea is that by performing approximate medial crossover of sub-programs, the overall program will also evolve to be approximately medial. Formalities are first introduced, such as the definition of a geometric offspring, which is $||o, p1|| + ||o, p2|| = ||p1, p2||$, where o is the offspring of parents p_1 and p_2. In Euclidean space, this can be interpreted as all points o which lie on the line segment between p_1 and p_2. Next, it is proven that the expected fitness (assuming uniform distribution of o between p_1 and p_2) of o is equal to the average fitness of the parents. Furthermore, this expected fitness is minimized when the offspring lies on the geometric midpoint of the parents, $||o, p1|| = ||o, p2||$. It is also noted that the fitness of the offspring can be no worse than the worst of its parents. Since there is no guarantee that an ideal midpoint exists, it must be approximated. There are two factors to consider: how "geometric" is the offspring, and how close to equidistant is the offspring from the parents. In the ideal case, the offspring is perfectly geometric (i.e. lies on line segment between parents) and is equidistant from both parents. To approximate these two factors, the deviation from the ideal values is calculated as the *geometric divergence* and the *equidistance divergence*:

$$d_G(o, p1, p2) = ||o, p1|| + ||o, p2|| - ||p1, p2|| \tag{2.2}$$

$$d_E(o, p1, p2) = \text{abs}(||o, p1|| - ||o, p2||) \tag{2.3}$$

The primary result of the paper is that medial crossover can be effectively approximated by operating only on subprograms. The primary drawback of the method is the exhaustive search of all possible sub-programs of a particular length.

2.8 The SGP Algorithm

Geometric semantic GP algorithms typically focus on creating approximately geometric offspring using a generate-and-test methodology, but in [11], provably geometric crossover and mutation operators are proposed which allow direct search in the semantic space. Rather than

using traditional syntax-based crossover and mutation, this paper uses new operators that operate on the semantics of parent programs. This is accomplished through the use of randomized function composition of parent programs. Surprisingly, this technique is general enough to be applied to most any GP problem domain (Boolean, regression, symbolic, etc). There have been several attempts at performing evolutionary search in the semantic space, but the majority of these algorithms rely on indirect methods, such as performing multiple crossovers until a semantically better child emerges. Though this may eventually work, it results in a lot of wasted effort. The authors therefore propose using existing known functions and composing them in such a way as to directly manipulate the program outputs. The composition scheme is the key contribution of the paper, and it is proven that the compositions used are equivalent to geometric crossover/mutation in the semantic space. As an example, the Boolean recombination operator is T3 = (T1 AND TR) OR (NOT TR AND T2), where T1 and T2 are the parents, T3 is the offspring, and TR is a random Boolean function with known semantics. This operator is equivalent to a 2-input multiplexer, where T1 and T2 are the inputs and TR is the select bit. This is illustrated in Figure 2.8. For each input case in the training data, either T1's output

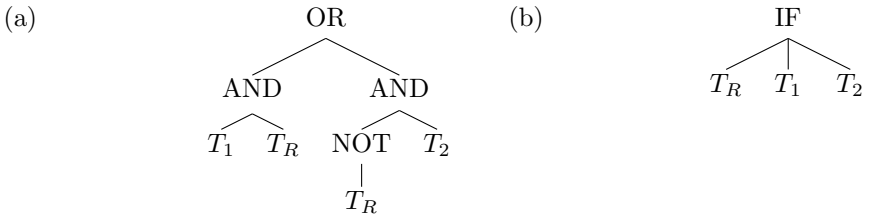

Figure 2.8: The semantic geometric crossover of parents T_1 and T_2 (from [11]). Note that (a) and (b) are equivalent representations.

is chosen or T2's output is chosen, based on the value of TR for that input case. Because each offspring program output comes from T1 or T2, this equates to a geometric crossover with respect to Hamming distance. Viewed another way, T3 is the result of applying a mask-based crossover on the semantics of T1 and T2. An example of the truth table associated with this operation is provided in Table 2.1. In this example, T_1 is the program (AND x_1 (NOT x_2)), T_2 is the program (NAND x_2 x_3) and T_R is the program x_1. The paper offers similar recombination operators for regression and symbolic problem domains. Similarly, a mutation operator is defined which will change T3's program output for a particular input case (i.e. change one bit in the final truth table column). Since each recombination results in a program containing both parents, the size of the program tree grows exponentially. The authors handle this problem by performing incremental simplification at each generation, without changing

Table 2.1: Example of semantic geometric crossover of parents T_1 and T_2 ($T_R = x_1$)

x_1	x_2	x_3	T_1	T_2	T_R	T_3
0	0	0	0	1	0	1
0	0	1	0	1	0	1
0	1	0	0	1	0	1
0	1	1	0	0	0	0
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	0	1	1	0
1	1	1	0	0	1	0

the program semantics. Standard GP, semantic stochastic hill climber (SSHC), and semantic genetic programming (SGP) are compared experimentally on Boolean problems, polynomial regression problems, and classification problems. SSHC is simply applying the semantic mutation operator discussed in the paper in a hill-climbing fashion. There is no test set, so only training set accuracy is compared. For the Boolean problems, both SSHC and SGP were near 100% accuracy on the training set, while standard GP was typically much less. Program size of SGP and SSHC is typically 2-3 times bigger than standard GP. Similar accuracy results are seen for the regression and classification domains. The authors state that the semantic operators may have heavy biases in the offspring distributions that hinder performance.

Because this algorithm will be used as the basis of research presented in Chapter 3, more algorithmic details will be provided based on this authors interpretation of the algorithm described in [11]. The genetic programming algorithm is provided in Algorithm 2.8.1.

Typical parameters for the algorithm are provided in Table 2.2. Note that the crossover

Table 2.2: Typical parameters for Algorithm 2.8.1

Parameter	**Value**
Population Size	200
Maximum Generations	50
Mutation Rate	0.1
Crossover Rate	1.0
Function Set	{AND, OR, NAND, NOR}

rate is 1.0, meaning that crossover is always performed.

The structure of the algorithm follows the standard GP template and uses elitism, crossover, and mutation. On line 2, the population is initialized randomly. This initial population provides the "primordial soup" from which to compose future programs. The semantic diversity of this initial population is a key element to the convergence rate. These initial programs are typically

Algorithm 2.8.1 Semantic Genetic Boolean Programming algorithm

Input: Train - A set of input-output pairs $(\boldsymbol{x_i}, y_i)$
Input: popSize - The size of the population
Input: maxGens - The maximum number of generations to evolve
Input: mutRate - Mutation rate, range [0.0, 1.0]
Input: funcSet - The set of functions to use as internal nodes in initial population
Output: A program tree that interpolates all input-output pairs in Train

```
1: function SEMANTIC-GP(Train, popSize, maxGens, mutRate, funcSet)
2:     Initialize P with popSize randomly generated program trees        ▷ Current Population
3:     perfectFitness ← size(Train)
4:     gen ← 0
5:     while gen <maxGens and best.fitness < perfectFitness do
6:         gen ← gen +1
7:         nextP ← {}
8:         ∀p ∈ P, evaluate fitness w.r.t. Train
9:         best ← argmax_{p∈P} {p.fitness}
10:        Add best to nextP                                              ▷ Elitism
11:        for i = 1 to size(P) do                                        ▷ Crossover
12:            p_1 ← TOURNSEL(P)
13:            p_2 ← TOURNSEL(P)
14:            p_r ← RANDOMMINTERMPROGRAM( )
15:            child ← (OR (AND p_1 p_r) (AND p_2 (NOT p_r)))
16:            child.semantics ← EVALUATESEMANTICS(child,Train)
17:            Add child to nextP
18:        for i = 1 to [mutRate ∗ size(P)] do                            ▷ Mutation
19:            r ← Randomly chosen program from nextP
20:            p_r ← RANDOMMINTERMPROGRAM( )
21:            if RANDINT(0,1) = 1 then
22:                mutant ← (OR r p_r)
23:            else
24:                mutant ← (AND r (NOT p_r))
25:            mutant.semantics ← EVALUATESEMANTICS(mutant,Train)
26:            nextP[r] ← mutant
27:        P ← nextP
28:    return best
29: end function
```

tree-based programs, but they can be any other type of program representation (e.g. linear program, stack-based program, etc).

On line 8, the maximizing fitness function is evaluated on all population members. For the Boolean domain, this will simply be the number of matching bits in the programs output and the target semantics.

Crossover is then performed by first selecting two parents using tournament selection (tournament size 2). Next a random minterm program is generated that will act as the T_R program (a.k.a. crossover mask). This program is a subset of minterms of inputs, and can be constructed in a mechanical fashion. A few examples of minterm programs constructed from inputs $\{x_1, x_2, x_3\}$ are provided in Table 2.3.

Table 2.3: Examples of generated minterm programs

Minterm	Program Representation	Semantics
$x_1\overline{x_2}$	(AND x_1 (NOT x_2)	00001100
$x_1x_2x_3$	(AND x_1 (AND x_2 x_3))	00000001
x_3	x_3	01010101
$\overline{x_2x_3}$	(AND (NOT x_2) (NOT x_3))	10001000

Once the minterm is generated, the geometric crossover is performed by composing both parents, and the offspring is added to the next generation.

On line 18, mutation is performed, which is analogous to randomly toggling one of the bits in the program semantics. Again, a minterm program is generated for the mutation operation, but this time all inputs are present in the minterm so as to set only a single bit in the semantic vector. The corresponding bit in the program to mutate is then randomly chosen to be set or cleared.

The algorithm terminates when either some maximum number of generations have elapsed, or a perfect fitness individual is found.

To provide some intuition about the nature of the algorithm, an example of a few generations is provided. In generation 0 (before the main loop), the population is initialized with random programs, as depicted in Figure 2.9.

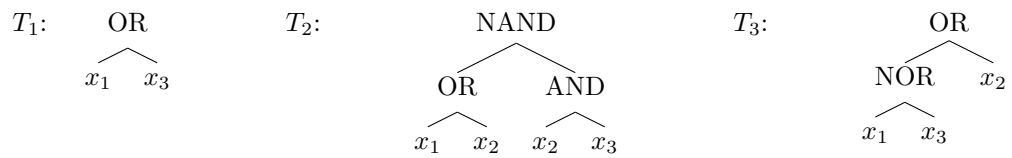

Figure 2.9: Generation 0: The "primordial soup" from which to evolve and compose program trees

The internal nodes of these randomly generated programs are chosen from the funcSet input of the algorithm, which is the set of functions allowed. In this example, only {AND, OR, NAND, NOR} are allowed.

At generation 1, crossover and mutation has been performed on the initial programs from generation 0. This is depicted in Figure 2.10.

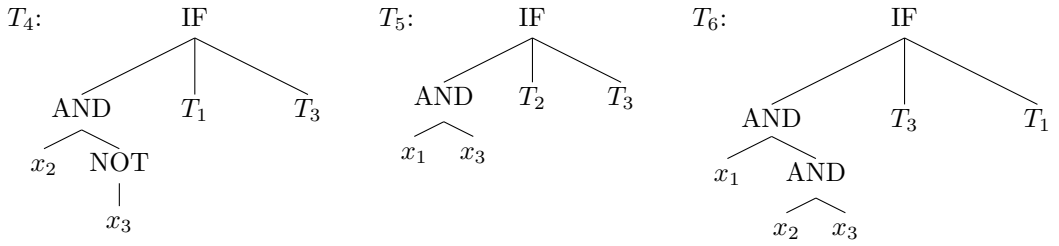

Figure 2.10: Generation 1: Composition of programs from generation 0

Note that all trees in this generation are IF trees because of the regular way in which programs are composed. The first input to the IF operator is the condition, the second input is the "true" part, and the last input is the "false" part.

At generation 2 (Figure 2.11), crossover and mutation has been performed on the programs from generation 1.

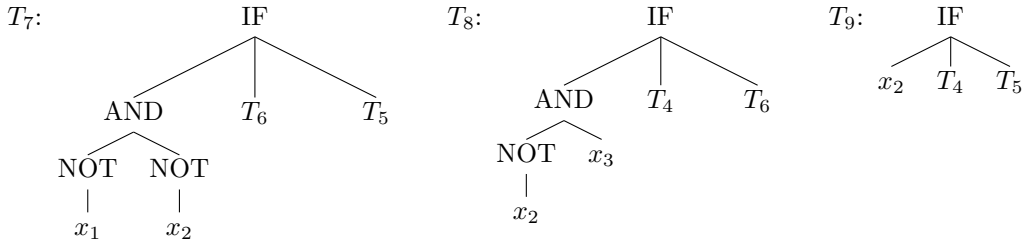

Figure 2.11: Generation 2: Composition of programs from generation 1

Similar to generation 1, all trees are of the IF variety. Also note that programs in this generation contain programs from both generation 1 *and* generation 0. In general, programs in generation n will contain programs from all previous $(n-1)$ generations. Because of this, the growth in the size of programs will be exponential in the number of generations.

2.8.1 Limitations of SGP

As of this writing (and to the best of this author's knowledge), the SGP algorithm is the only GP algorithm that searches the semantic space *directly*. This is a powerful property that allows it to solve many deceptive problems which traditional GP algorithms struggle with. However, the cost of this property is fairly large - namely, the program size grows exponentially with the number of generations. This is the biggest limitation of the algorithm and the area that offers the most room for improvement.

Another limitation of SGP is the randomized structure of the program. Inserting random sub-programs into the overall program is counter-intuitive, and makes the program complex and difficult for human readers to interpret. Furthermore, the motivation for using randomized sub-programs is lacking. The primary purpose it serves is to force the crossover and mutation to be geometric in nature (which allows for direct search in semantic space).

Another limitation of SGP is that crossover and mutation create offspring that are unnecessarily randomized. For example, the crossover operator will create offspring that are semantically intermediate with respect to the parents, with the distance to each parent being randomized. However, the motivation for doing this is unclear - why not just choose a semantically intermediate offspring that is closest to the target semantics? It's possible the choice was made in an effort to promote population diversity. However, it also delays the convergence to a solution, which is important in SGP considering that the program size grows exponentially with each generation.

These primary limitations will be addressed by the proposed algorithms in Chapter 3.

Chapter 3

Approach

This chapter will propose two algorithms for learning Boolean program trees. The first is a revised version of the Semantic-GP algorithm described in [11] with improved crossover and mutation operators. This algorithm is based on the *composition* of program semantics to produce the desired output semantics. Due to weaknesses identified in the first algorithm, a second algorithm is introduced that grows a Boolean program tree by utilizing greedy search, and is based on the *decomposition* of program semantics.

3.1 Improved Semantic-GP

The Semantic-GP algorithm described in [11] and detailed in Algorithm 2.8.1 is one of the few genetic programming algorithms which searches directly in the semantic space of programs. This is particularly useful for *deceptive* Boolean problems, where the fitness landscape can lead a traditional generate-and-test evolutionary search into local optima instead of the global optimum. However, one of the primary weakness of the algorithm is that the evolved tree size is too large. We wish to harness the power of semantic search while overcoming the weakness of tree size using an improved Semantic-GP+ algorithm, or SGP+.

3.1.1 Motivation for SGP+

To address the weakness of tree size in the SGP algorithm, we must consider what makes the tree grow. The tree grows in depth for every crossover and mutation operation that occurs. This is dangerous, as it means the tree will grow in size exponentially with each generation. Therefore, we wish to reduce the number of crossover and mutation operators by converging to a solution more quickly. The general strategy will be to choose parents whose crossover is

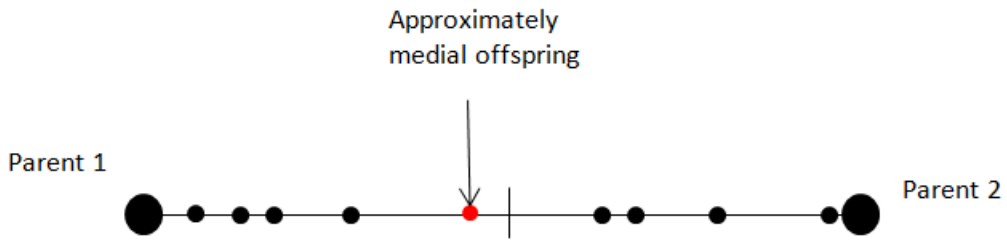

Figure 3.1: Mediality dictated by choice of p_r. Smaller dots represent potential offspring points in semantic space of parents 1 and 2, depending on p_r. The middle line represents the point of a perfectly medial offspring.

more likely to produce offspring closer to the target semantics, at the expense of computational time per generation. This may slow down the evolution, but should produce smaller trees if the crossover and mutation operators are indeed choosing better parents.

3.1.1.1 Random Program Archive

As mentioned in section 2.7, because the crossover operation is *geometric* we know that the expected fitness of the offspring is optimal when it is equidistant from each parent [6]:

$$||o, p_1|| = ||o, p_2|| \tag{3.1}$$

This means that it may be worthwhile to consider constraining the crossover to produce medial, or equidistant, offspring. However, as mentioned in [6], this is technically infeasible due to the complexity of the genotype-phenotype mapping. In other words, the mapping between program syntax and semantics is too complex to be able to directly synthesize a semantically medial offspring. Also note that this complex mapping is what makes search techniques like GP necessary. The genotype-to-phenotype mapping is typically one-to-one and easily found by simply evaluating the program output, but the phenotype-to-genotype mapping is often one-to-many, and there is no direct way to learn the mapping unless a database of known programs happens to contain a program with the desired phenotypic behavior.

Given that it is infeasible to produce a perfectly medial offspring, we can relax the constraint to get an approximately medial offspring. In the SGP algorithm, recall that p_r is an input to the crossover operator that acts as a crossover mask and dictates the distance between the offspring and each parent. This is illustrated in Euclidean semantic space in Figure 3.1. The intermediate dots represent the semantics of offspring for various choices of p_r. Note that there is a single best approximate medial crossover, and that this p_r should be preferred over the

others for crossover because it will result in optimal expected fitness. In the extreme cases, the offspring will be identical to one of the parents, which can occur if the semantics of p_r are $0 \ldots 0$ or $1 \ldots 1$.

If an approximate medial offspring is desired, then it is important to have a diverse selection of p_r crossover masks to choose from. In the best case, all possible masks are available, in which case the optimal medial offspring can be produced. In SGP, the p_r programs are small programs generated using a subset of minterms of inputs. For example, if program inputs are labeled $\{x_1 \ldots x_n\}$, then p_r could be a program that generates the function $(x_1\overline{x_3})$. This program can be constructed mechanically as (AND x_1 (NOT x_3)) and fed into the crossover operator. For $n = 3$, the semantic behavior of such a program would be 00001010. However, using programs that are subsets of minterms will only allow a finite number of p_r behaviors. In an effort to increase the number of p_r behaviors, we will add an archiving step to the main generational loop. A random program archive (or RPA) will be maintained that will initially contain many minterm programs. At each generation, a randomly chosen subset of the population will be added to the archive, which will increase the number of p_r choices available for the crossover operator. This p_r archive is similar to the archive described in [9] in that the archive contains a history of programs observed throughout *all* generations. For two given parents, the RPA will be queried for a particular ideal crossover mask, and the program p_r with minimum distance $||q, p_r||$ with respect to query q will be chosen.

There are two main side-effects to using a random program archive. The first is that the archive will increase linearly in size with each generation, resulting in longer p_r query times for each crossover. The second side-effect is that the p_r programs inserted into the overall program tree can be much larger (i.e. may not just be a small minterm program). However, this side-effect is not a concern in practice, as the p_r tree will not need to be re-evaluated each time it is encountered. When a program tree is evaluated, the output of sub-trees is memoized, so p_r will only ever be evaluated once, which means that the size of the p_r sub-tree is not of primary concern.

3.1.1.2 Choosing Parents for Crossover

In the SGP algorithm, the selection of parents for crossover is done using normal GP selection methods such as roulette wheel selection, tournament selection, or rank selection. These selection methods do not assume a geometric crossover, which means that there may be more efficient selection methods that take advantage of the geometricity of offspring. In Euclidean space, the offspring semantics are represented as a point on the line segment connecting the two parents. With this knowledge, it seems advisable to select parents which straddle the target in semantic space. An example of this straddling is shown in Figure 3.2. In this example, parents

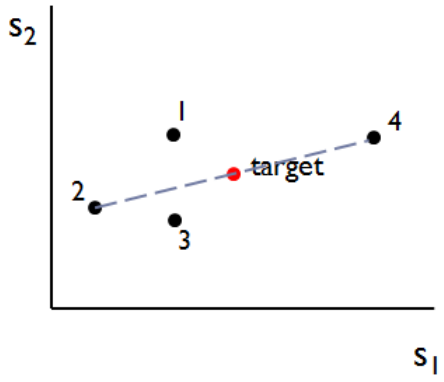

Figure 3.2: Example of choosing parents which straddle the target in semantic space. In this case parents 2 and 4 would be chosen over parents 1 and 3, despite being further away from the target.

2 and 4 straddle the target the best, so their offspring may have a better chance of landing near the target. This selection is in contrast to fitness-proportionate selection, which would choose parents 1 and 3.

The degree to which two parents straddle the target will be referred to as *divergence from geometricity* (as defined in [6]), and can be calculated using the triangle inequality:

$$d_G(t, p_1, p_2) = ||t, p_1|| + ||t, p_2|| - ||p_1, p_2|| \tag{3.2}$$

If the target semantics lie on the line segment between two parents, then d_G will be 0. In this case, the parents perfectly straddle the target, and the crossover operation will be more likely to produce offspring closer to the target. Parents should be chosen such that d_G is minimized. In practice, it is infeasible to calculate d_G for all pairs of parents, so a small pool of parents will be chosen using tournament selection. Each pair of parents in the pool will be considered, and the pair that minimizes d_G will be chosen.

In the previous section, it was observed that a medial geometric crossover will have optimal *expected* fitness. However, given two parents and a known target, the medial point may not be optimal. This is illustrated in Figure 3.3. The optimal choice of offspring semantics (i.e. the one that minimizes the distance to the target) occurs at the intersection of the line segment between the parents and the corresponding perpendicular line that passes through the target semantics. Therefore, the RPA will be queried for this point instead of the medial point.

One of the primary differences between SGP and SGP+ is the location of the crossover offspring in semantic space. Figure 3.4 illustrates this difference. Both algorithms utilize

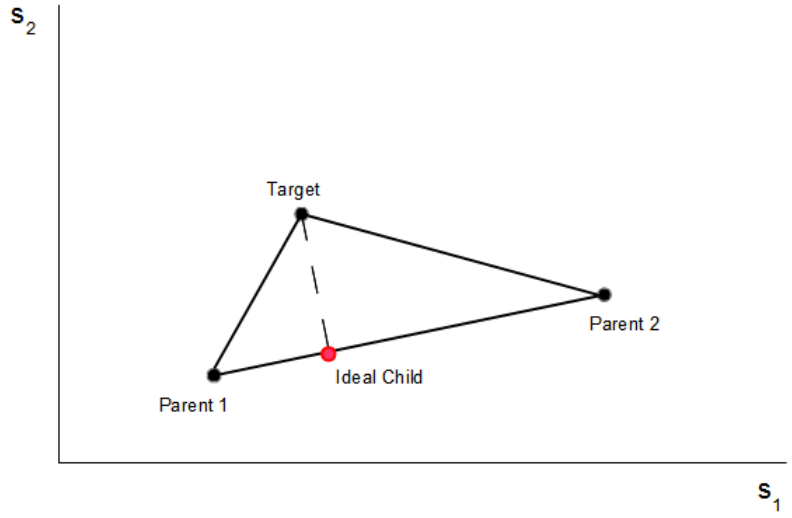

Figure 3.3: Example of an ideal geometric crossover in 2D Euclidean semantic space

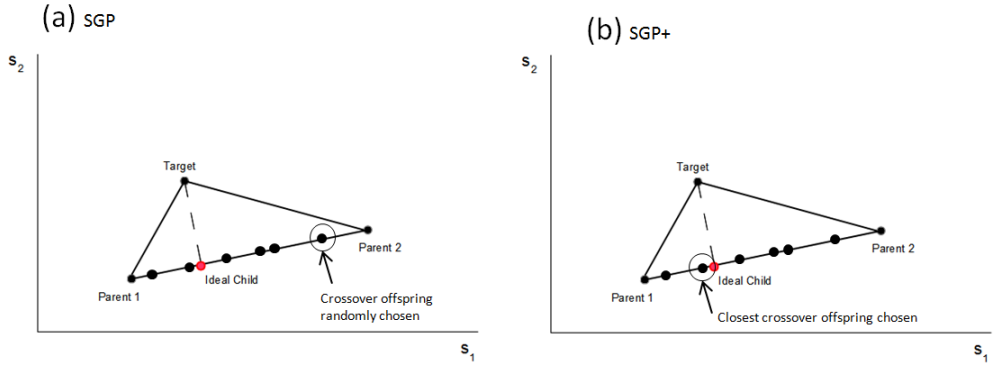

Figure 3.4: Each black dot represents a potential offspring program. (a) SGP crossover is geometric (on the line segment between parents), but the position along the line is randomly chosen. (b) SGP+ chooses the offspring that is closest to the target, based on the current contents of the RPA.

geometric crossover, but the offspring produced by SGP+ will be closer to the target because the RPA will be queried for the program that will produce offspring with minimal distance to the ideal child/offspring.

3.1.1.3 Greedy Mutation

The mutation operator in the SGP algorithm chooses a random program from the current population and mutates a single semantic bit randomly (either sets or clears the bit). However, given that we know what the target semantics are, it seems more efficient to make the chosen bit match the corresponding bit in the target. In other words, there is no clear motivation for randomly assigning the bit if we know what the correct assignment should be. Therefore, the mutation operator in SGP+ will identify the first semantic bit difference between the program to mutate and the target and set that bit to match. This is equivalent to taking a step in a single dimension in semantic space towards the target. Figure 3.5 illustrates the conceptual difference between mutation in SGP and SGP+ in 2-D semantic Euclidean space.

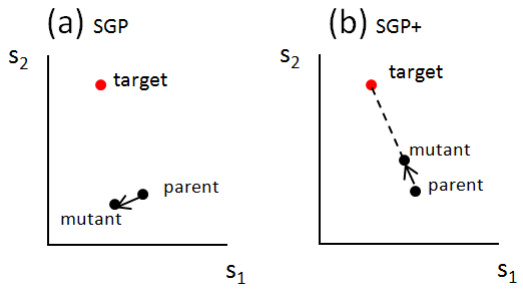

Figure 3.5: (a) Mutation in SGP takes a single step in a random direction. (b) SGP+ mutation takes a step in the direction of the target semantics.

Given that we can mutate individual bits to match corresponding bits in the target, the question arises: Why don't we just keep mutating bits until we've matched the target? This is generally a bad idea because the target semantic vector can be prohibitively large. Recall that the program tree grows in depth for each mutation performed. If there are k input cases, then the semantic vector is k bits long, and the resulting program tree would have depth $O(k)$. If k is large (say 10,000), then not only is the tree overly complex, but the evaluation of the tree will take a prohibitively long time. In general, mutating a single bit at a time will make tiny incremental steps towards the target, but crossover can make large jumps, resulting in faster convergence and smaller tree sizes. Therefore, crossover should be the preferred operator with mutation playing a lesser role (controlled by the mutation rate).

3.1.2 SGP+ Algorithm

The structure of the SGP+ algorithm is nearly identical to Algorithm 2.8.1. There are three primary differences:

- A new Semantic-Crossover operator
- A new Semantic-Mutation operator
- The addition of a random program archive

The revised algorithm is detailed in Algorithm 3.1.1.

As discussed previously, the random program archive is initialized with random minterm subsets (line 3). At the end of each generation, programs from the current population are archived to improve the diversity of the RPA (lines 19 to 21). The number of programs archived is controlled by the RPA rate, which is typically set to 0.2. Note that the archived programs are not necessarily fit programs. If programs were archived based on fitness, then highly fit or similar programs could begin to dominate the archive, which would be counter to the goal of the RPA, which is to have a diverse collection of unique programs for the purposes of a crossover mask.

The crossover operator has been modified to utilize the RPA. The details of the crossover function are provided in Figure 3.6.

First, the target semantics are extracted from the training set. Next, a small pool of parents is selected using fitness-based tournament selection (lines 3 to 5). The tournament is a size-2 tournament with $p = 0.8$, meaning that the more-fit individual wins the tournament 80% of the time. On line 6, the geometricity of each pair of parents in the parent pool is calculated (d_G from equation 3.2), and the pair that minimizes this value is chosen. Because the focus is on Boolean problems, the distance metric for geometricity is Hamming distance.

Lines 7 through 19 find an ideal crossover mask, as depicted in Figure 3.3. This is done by looping through each bit in the target semantic vector and checking whether any parent matches the target bit. If both parents match the target, then it doesn't matter which parent is chosen to be represented in the mask, so a value of 'X', or "don't care", is assigned to the ideal mask. Similarly, a value of 'X' is assigned if neither parent matches the target. If only a single parent matches the target, then that parent is chosen for representation in the crossover mask. After looping through all semantic bits, an ideal crossover mask is obtained. If this mask were to exist in the RPA, then the offspring of the chosen parents will be optimally close to the target. Line 20 queries the RPA for the program that most closely matches the ideal crossover mask. For the Hamming distance calculation, the bits assigned as 'X' in the ideal mask will not add to the distance. This is a linear search through the entire RPA. For large training

Algorithm 3.1.1 Improved Semantic Genetic Boolean Programming algorithm

Input: Train - A set of input-output pairs $(\boldsymbol{x_i}, y_i)$
Input: popSize - The size of the population
Input: maxGens - The maximum number of generations to evolve
Input: mutRate - Mutation rate, range [0.0, 1.0]
Input: rpaRate - Rate at which to add programs to the Random Program Archive, range [0.0, 1.0]
Input: funcSet - The set of functions to use as internal nodes in initial population
Output: A program tree that interpolates all input-output pairs in Train

```
1:  function SEMANTIC-GP(Train, popSize, maxGens, mutRate, rpaRate, funcSet)
2:      Initialize P with popSize randomly generated program trees      ▷ Current Population
3:      Initialize RPA with random minterm programs                     ▷ Random Program Archive
4:      perfectFitness ← size(Train)
5:      gen ← 0
6:      while gen < maxGens and best.fitness < perfectFitness do
7:          gen ← gen +1
8:          nextP ← {}
9:          ∀p ∈ P, evaluate fitness w.r.t. Train
10:         best ← argmax_{p∈P} {p.fitness}
11:         Add best to nextP                                           ▷ Elitism
12:         for i = 1 to size(P) do                                     ▷ Crossover
13:             child ← SEMANTIC-CROSSOVER(Train, P, RPA)
14:             Add child to nextP
15:         for i = 1 to [mutRate ∗ size(P)] do                         ▷ Mutation
16:             r ← Randomly chosen program from nextP
17:             mutant ← SEMANTIC-MUTATION(Train, r, RPA)
18:             nextP[r] ← mutant
19:         for i = 1 to [rpaRate ∗ size(P)] do                         ▷ Archiving
20:             r ← Randomly chosen program from P
21:             Add r to RPA
22:         P ← nextP
23:     return best
24: end function
```

```
Input: Train - A set of input-output pairs (x_i, y_i)
Input: P - The current population from which parents are chosen
Input: RPA - The Random Program Archive, from which crossover masks (p_r) are chosen
Output: A child program tree created from the composition of parents
function SEMANTIC-CROSSOVER(Train, P, RPA)
    Target ← {OUTPUT(t) ∀t ∈ Train}
    ParentPool ← {}
    for i = 1 to 5 do
        Add TOURNSEL(P) to ParentPool
        argmin_{p1,p2∈ParentPool} {GEOMETRICITY(Target, p1, p2)}
    IdealCrossMask ← {}
    for i = 1 to size(Target) do
        if Target[i] = p1.semantics[i] then
            if Target[i] = p2.semantics[i] then
                ▷ Both parents match target, so don't care which output is used
                IdealMask[i] ← X
            else
                IdealMask[i] ← 0                                  ▷ Choose output from p1
        else if Target[i] = p2.semantics[i] then
            IdealMask[i] ← 1                                      ▷ Choose output from p2
        else
            ▷ Neither parent matches target, so don't care which output is used
            IdealMask[i] ← X
    p_r ← argmin_{p∈RPA} {HAMMDIST(IdealMask, p.semantics)}
    child ← (OR (AND p1 p_r) (AND p2 (NOT p_r)))
    child.semantics ←
        (OR (AND p1.semantics p_r.semantics) (AND p2.semantics (NOT p_r.semantics)))
    return child
end function

Input: Target - The target semantics (vector of bits)
Input: p1 - Semantics of first parent (vector of bits)
Input: p2 - Semantics of second parent (vector of bits)
Output: A number representing the divergence from geometricity of p1 and p2
function GEOMETRICITY(Target, p1, p2)
    a ← HAMMDIST(p1.semantics, Target)
    b ← HAMMDIST(p2.semantics, Target)
    c ← HAMMDIST(p1.semantics, p1.semantics)
    return (a + b − c)
end function
```

Figure 3.6: SGP+ crossover function for Algorithm 3.1.1

sets, this could be improved by using locality-sensitive hashing (LSH), which would return an approximate nearest match.

Line 21 is the construction of the offspring program tree, which is equivalent to a 2-input multiplexer of parent semantics. Finally, the semantics of the offspring are calculated by applying the crossover mask to the semantics of both parents. Note that the offspring tree is not actually evaluated, because the output of each parent sub-tree is already known for each input case in the training set (this is the definition of parent semantics). This greatly speeds up the crossover operation and the overall speed of evolution because the programs do not need to be fully evaluated.

The mutation operator has been modified to take a single-dimensional step in the direction of the target. The details of the mutation function are provided in Figure 3.7.

Input: Train - A set of input-output pairs $(\boldsymbol{x_i}, y_i)$
Input: r - The program to mutate
Input: RPA - The Random Program Archive to which a minterm program will be added
Output: A program whose semantics are one bit different from r

```
1:  function SEMANTIC-MUTATION(Train, r, RPA)
2:      Target ← {OUTPUT(t) ∀t ∈ Train}
3:      i ← Index of first difference between r.semantics and Target
4:      if Target[i] = 1 then
5:          set ← True
6:      else
7:          set ← False
8:      IdealMask ← 0...0
9:      IdealMask[i] ← 1
10:     p_r ← GENMINTERM(IdealMask)
11:     Add p_r to RPA
12:     if set then
13:         mutant ← (OR r p_r)
14:         mutant.semantics ← (OR r.semantics p_r.semantics)
15:     else
16:         mutant ← (AND r (NOT p_r))
17:         mutant.semantics ← (AND r.semantics (NOT p_r.semantics))
18:     return mutant
19: end function
```

Figure 3.7: SGP+ mutation function for Algorithm 3.1.1

First the target semantics are extracted. Next, the target semantic vector is scanned (Line 3) to find the first semantic difference with the program to mutate. This is simply the index of the first 1 bit in the vector (XOR r Target). Next it is determined if the bit in the program to

mutate should be set or cleared to match the target bit.

An ideal mask is then constructed with a single bit set in the index of the first difference. This mask corresponds to a minterm, and a small program is generated that will have semantics that match the mask. Although the details are omitted, the construction of such a minterm program can be done mechanically. The bit that is set corresponds to a single input case in the truth table. For instance, if the mask were 00100000, this could corresponded with the input case $\{0, 1, 0\}$. The minterm for this bit is $(\overline{x_1} x_2 \overline{x_3})$, and the corresponding program is (AND (NOT x_1) (AND x_2 (NOT x_3))). The mutant is then constructed based on whether the bit is to be set or cleared. If being set, then the program can be bitwise-OR'd with p_r. Otherwise, the bit is to be cleared, and the program should be bitwise-AND'd with the negation of p_r. Finally, the semantics of the mutant are obtained in a similar fashion. As in the crossover operator, the mutant tree does not need to be fully evaluated because the original program semantics are known and will never change.

3.1.3 Complexity Analysis

The time complexity of fitness evaluation is $O(|T|)$, where $|T|$ is the size of the training set. This is because programs are constructed compositionally so that sub-trees do not need to be recomputed. All that needs to be done for fitness evaluation is to multiplex the $|T|$ bits from each parent and to calculate the Hamming distance to the target vector.

The time complexity for the entire evolutionary process is $O(G*|T|*|R|*M)$, where G is the number of generations, M is the size of the population and $|R|$ is the size of the RPA. This can be seen by observing that each of the $(G*M)$ programs created during the evolutionary process must perform $O(|T|*|R|)$ work to search the RPA during crossover for a nearest crossover mask, which involves computing Hamming distance to each program in the RPA.

The time complexity for tree evaluation (after the evolutionary process has completed) is $O(|T| * (|P| + |R|))$, where $|P|$ is the size of the Parent Program Archive (PPA). Although not previously discussed, the PPA is used for practical implementations for storing parent programs that are actually used in the crossover and mutation operations. If a program is never used in one of these operations, it can be discarded. Therefore, to evaluate the tree output, each of the programs from the parent and random program archives must be evaluated, because they occur in some part of the overall program tree. Each of these sub-tree evaluations take $O(|T|)$ time. In practice, the tree is evaluated depth-first starting from the root and results from each sub-tree are memoized. This means that each of the $O(|P| + |R|)$ sub-trees will only be evaluated once.

The space complexity of each program is $O(|T|)$ because each program (except for seed programs at tree leaves) only needs to store the semantics for each of the $|T|$ input cases, as

well as $O(1)$ metadata, such as pointers to parent trees. The space complexity of the seed programs is $O(k + |T|)$, where k is the upper limit on the size of the initial program trees.

The space complexity of the RPA is $O(G * M * |T|)$. This can be seen because αM programs of size $O(|T|)$ are added to the archive at each generation. The space complexity of the PPA is also $O(G * M * |T|)$ by similar reasoning.

The complexity analysis of SGP and SGP+ are summarized and compared in Table 3.1.

Table 3.1: Summary of SGP+ Complexity Analysis

Alg	Fitness	Evolution	Evaluation	Prog Size	RPA Size	PPA Size
SGP	$O(\|T\|)$	$O(G * \|T\| * M)$	$O(\|T\| * \|P\|)$	$O(\|T\|)$	-	$O(G * M * \|T\|)$
SGP+	$O(\|T\|)$	$O(G * \|T\| * M)$	$O(\|T\| * (\|P\| + \|R\|))$	$O(\|T\|)$	$O(G * M * \|T\|)$	$O(G * M * \|T\|)$

The only complexity differences between SGP and SGP+ are the tree evaluation time complexity and the RPA size complexity. Note that the RPA does not exist in SGP, as random minterm programs are generated on-the-fly. It would appear that despite the intentions, the evolutionary time and tree evaluation time have not improved. However, as will be shown in Chapter 4, the actual running time of SGP+ is better than SGP because the G term is lower. In other words, convergence to a solution occurs more quickly with SGP+, which improves most of the complexity measures from Table 3.1.

3.2 Semantic Decomposition for Program Search

This section discusses a new kind of semantically-driven algorithm for program search that is not biologically-inspired or based on any kind of evolutionary algorithm, called Semantic Decomposition for Program Search, or SDPS. Instead, it is based on traditional search techniques. First the motivation for this algorithm is established followed by some background information, the algorithm details, and finally the complexity analysis.

3.2.1 Motivation for SDPS

Given that the goal of SGP+ was to produce a smaller tree, it is not clear whether that goal is fully realized. On the one hand, the time to convergence is improved over SGP by making larger and more directed steps toward the target. But on the other hand, we still have an overly large tree that is extremely complex. The fact that the tree includes random programs is problematic because it is contrary to common sense. If a software engineer were tasked with designing a program to specifications, there will most likely not be any "random" design decisions made. Each sub-program would have a justifiable purpose for achieving the specification.

Therefore, we wish to create program trees that are more deterministic and are as simple as possible. To this end, a greedy algorithm called Semantic Decomposition for Program Search (SDPS) will be proposed. There are several key distinctions between the SGP+ and SDPS algorithms:

1. SGP+ is an evolutionary algorithm, while SDPS is a traditional greedy search algorithm.
2. SGP+ searches the semantic space by evolving multiple models, whereas SDPS only operates on a single model.
3. SGP+ builds programs from the bottom up by composing programs. SDPS builds programs from the top down by decomposing semantics.
4. SGP+ includes random programs and non-determinism. SDPS is completely deterministic.

The top-down, decompositional nature is probably the most important aspect of SDPS (this was depicted in Figure 1.3). In words, the algorithm starts with the target output (i.e. "what we want") and decomposes the target into multiple sub-targets, each of which will be independent sub-problems. The decomposition terminates when the sub-target semantics match the semantics of an input variable (i.e. "what we've got"), which will be the tree leaf. In other words, the tree is grown from output to inputs.

The algorithm is designed to produce short trees by using heuristics to determine how close the current semantic sub-target is to the tree leaf (i.e. one of the input variables). By creating shorter trees, the goal is to improve tree simplicity and generalization to unseen inputs. This should improve the accuracy of the tree on the training and test sets. Furthermore, it will be easier for a human to interpret and reason about than the trees output from SGP or SGP+.

One positive aspect of the SGP and SGP+ algorithms was the ability to solve deceptive Boolean problems such as the parity problem, where a change in a single input bit can result in a change in the output. This was possible because programs were searching directly in the semantic space, so any complexities resulting from the input-output mapping were side-stepped. Will SDPS be able to achieve this as well? In short, yes, because SDPS will utilize the semantic space to determine how far away a sub-target is from one of the input variables and for generating new sub-targets. Because of this semantic space utilization, the complexities from the input-output mapping are side-stepped. However, because of the greedy (i.e. non-backtracking) nature of the algorithm, it is possible that sub-optimal trees (overly complex/large) are generated for certain deceptive problems. A comparison of accuracy and tree size on deceptive Boolean problems will be explored further in Chapter 4.

3.2.2 Background

A few key terms and ideas should first be explained before diving into the SDPS algorithm.

To begin, the *semantic context* of a node n is defined similarly as in [10]. It is the overall program tree with the sub-tree rooted at node n removed. With n removed, the exact semantics of the overall tree will not be exactly known, but certain semantic bits may be precisely known due to other parts of the overall tree. Consider the tree depicted in Figure 3.8. Here, node n is

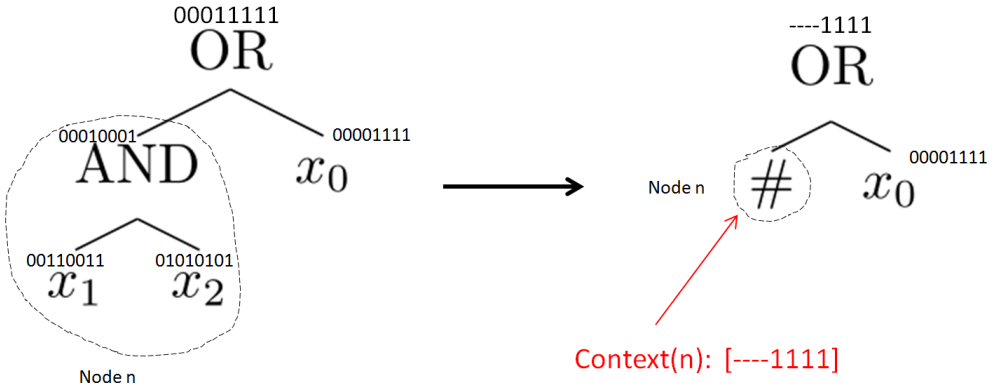

Figure 3.8: Example of determining semantic context associated with node n by removing it from the overall tree. Semantics are displayed above each node. Also note that removal of n affects the semantics of all ancestors of n (in this case, just the root node).

removed from the tree and replaced by the # symbol. Once removed, the output of the tree is no longer known. However, some of the outputs *are* known, because they will not be affected by the output of the sub-tree at node n. For instance, the last four bits of the overall program will be 1s, regardless of the sub-tree at #. These are referred to as *fixed* bits. For the unknown (non-fixed) context bits (i.e. the sub-tree at n matters), a '-' is used to indicate that the bit can be a 0 or a 1. As a notational convention, context will be written inside of square brackets. In this example, the context associated with node n is [- - - -1111], which means this sub-tree has the potential to change any of the first four semantic bits, but is unable to change any of the last four. The semantic context can be determined in a similar way for any node in the program tree.

The *input semantics* are the semantics associated with a particular input. This corresponds to an input variable column in the truth table. Also referred to as *leaf semantics* in some cases.

Sub-targets are semantic vectors, similar to target semantics previously discussed, that represent the desired sub-tree behavior at that location in the overall tree. It is a unique sub-problem that needs to be solved.

The *ancestors* of a node are the set of parents on the ancestral path from the node to the

root (i.e. parent, grand-parent, great-grand-parent, great-great-grand-parent, etc).

3.2.3 SDPS Algorithm

The overall structure of SDPS follows that of a recursive tree induction algorithm, such as the ID3 decision tree learning algorithm. The top-level pseudocode is provided in Algorithm 3.2.1. The inputs and outputs to SDPS are identical to that of SGP+: given a set of input-output

Algorithm 3.2.1 SDPS algorithm

Input: Train - A set of input-output pairs $(\boldsymbol{x_i}, y_i)$
Input: funcSet - The set of invertible functions to use as internal nodes
Output: A program tree that interpolates all input-output pairs in Train

```
1:  function SDPS(Train, funcSet)
2:      targetQ ← {}
3:      target ← {y_i | (x_i, y_i) ∈ Train}
4:      root ← NODE(target)                                  ▷ Initialize program tree
5:      Add root to targetQ          ▷ Initialize target queue with desired program output
6:      while targetQ not Empty do
7:          t ← SELECTTARGETNODE(targetQ)                    ▷ Choose next sub-target
8:          Remove t from targetQ
9:          f, subtargs ← DECOMPOSENODE(t, funcSet)
10:         t.func ← f                          ▷ Associate an f ∈ funcSet with this node
11:         Create branches from node t to subtargs                      ▷ Grow the tree
12:         UPDATEANCESTRALSEMANTICS(t)      ▷ Update semantics on ancestral path of t
13:         UPDATECONTEXT(root)                          ▷ Update context of entire tree
14:         for s ∈ subtargs do
15:             if s is not a leaf then
16:                 Add s to targetQ
17:     return root
18: end function
```

pairs and a set of functions to use as internal nodes, return a program tree that interpolates them. To begin, the desired tree output (or target) is determined by extracting the outputs from the training set (i.e. the final column in the truth table). On line 4, the program tree is initialized with this target and it is also added to the target queue. The main loop consists of decomposing nodes one at a time and adding any new sub-targets the to target queue. Once the target queue is empty (all targets reached), the program tree is returned.

A detailed discussion of each of the algorithm sub-procedures will follow:

- Choosing a node to decompose (SELECTTARGETNODE)
- Node Decomposition (DECOMPOSENODE)
  - Determining target semantics (GETTARGETSEMANTICS)
  - Choosing sub-targets (CHOOSESUBTARGETS)
  - Validating sub-targets (VALIDSUBTARGETS)
- Tree Update (UPDATEANCESTRALSEMANTICS and UPDATECONTEXT)

3.2.3.1 Choosing a Node to Decompose

The algorithm is biased to select the most difficult sub-target first because the harder sub-targets might require more node decompositions. These decompositions have an effect on the target semantics and context of other nodes in the tree, so it is desirable to get these difficult sub-targets out of the way first. Alternatively, the easiest sub-target could be chosen. This might be useful if fully learned sub-trees are allowed to be reused to achieve other sub-targets. In that case, it may make sense to solve the easier sub-targets first so they can be reused as soon as possible. Ultimately, the choice of easiest/hardest sub-target is not very significant because the learned sub-trees will be approximately the same, except for a few bits which may be more or less constrained by the current context.

The target queue is a priority queue of sub-targets. At each iteration of the main while loop, the highest priority sub-target is selected and extracted from the queue (lines 7 - 8). The selection of a sub-target is based on the Hamming distance from the sub-target semantics to the closest leaf semantics (a.k.a. input variable semantics). Consider the example in Figure 3.9. For each of the candidate nodes to decompose (on the frontier of the tree), the semantic Hamming distance between the sub-target and each input variable is computed. The input variable with minimum Hamming distance is associated with each of the candidates.

$$\min_{i \in \mathrm{I}} \{\textsc{HammDist}(i,\, s)\} \tag{3.3}$$

In formula 3.3, i represents the semantics for a particular input variable (i.e. the column in the truth table associated with that input) and s is the semantics of a particular candidate node to decompose. This is a crude heuristic that captures the expected number of decompositions before a leaf is reached on that path. In other words, how difficult the sub-target is to achieve. This is not a perfect heuristic, but is good enough for most cases. As an example of the crudeness of this heuristic, consider semantics between sub-target $s = 10101010$ and leaf $x_2 = 01010101$.

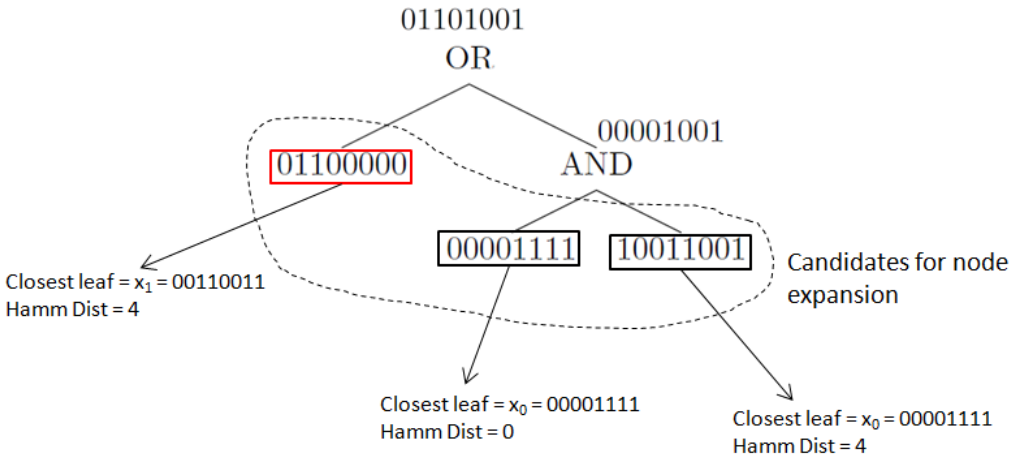

Figure 3.9: The most "difficult" sub-target (based on Hamming distance) is chosen for decomposition (in this case, 01100000). Note that 10011001 could have also been chosen, as it is equally as difficult.

Here, the Hamming distance is 8, the maximum possible, yet there is only one more node decomposition needed to reach the leaf, namely (NOT s).

The pseudocode for the SELECTTARGETNODE function is provided in Figure 3.10.

3.2.3.2 Node Decomposition

Node decomposition is the process of breaking down a target into two or more unique semantic sub-targets such that, when combined via a function, generates the original target semantics. The problem is to find the function and the sub-targets that will minimize the overall size of the tree. Additionally, we want the sub-targets to be *easier* to solve than the original target. In general, the ideal sub-targets are not known, so heuristics will be used. The choice of function and sub-targets is at the heart of the algorithm and will dictate the shape, size, and complexity of the final program. The overall goal of the algorithm is to reduce the total number of node decompositions to produce a smaller tree.

To choose the decomposition that will result in a smaller overall tree, heuristics will be used. We will prefer the decomposition that results in sub-targets which are *closest to the inputs in semantic space*. More precisely, we will choose sub-targets such that the Hamming distance to the semantics of the nearest inputs are minimized. This is done in an effort to minimize the depth of each branch of the tree. By minimizing the depth of every branch in the tree, we will reduce the overall size of the tree.

```
Input: targetQ - Queue of candidate nodes for decomposition
Output: difficultNode - Node selected for decomposition (the most-difficult sub-target)
 1: function SelectTargetNode(targetQ)
 2:     maxDist ← 0
 3:     difficultNode ← {}
 4:     for s ∈ targetQ do
 5:         dist ← min_{i∈I} {HammDist(i.semantics, s.semantics)}
 6:         if dist > maxDist then
 7:             maxDist ← dist
 8:             difficultNode ← s
 9:     return difficultNode
10: end function
```

Figure 3.10: SelectTargetNode function from Algorithm 3.2.1. Selects the most "difficult" target for decomposition.

Node decomposition occurs on line 9 of Algorithm 3.2.1. The function DecomposeNode returns two objects - a function from funcSet (to associate with the newly-decomposed node) and a set of new sub-targets. The pseudocode for DecomposeNode is provided in Figure 3.11.

DecomposeNode exhaustively checks all functions in funcSet and all possible combinations of inputs. For example, if there are 5 input variables $\{x_1, \ldots, x_5\}$ and f is chosen to be the 3 input OR function, then there are $\binom{5}{3}$ possible input combinations to f. For each of these (f,c) combinations (line 8), candidate sub-targets are created. In essence, this is testing potential decompositions of node t to determine the optimal decomposition, which will be returned by the function. Once all (f,c) pairs have been considered, the optimal function and sub-targets are returned.

Node decomposition consists of three primary sub-procedures - determining the target semantics (incorporates context), choosing the sub-target semantics, and validating the chosen sub-targets.

Determining target semantics. Because t is the node to be decomposed, it does not yet have an associated function or any children. However, it does have associated semantics that represent the target semantics that are to be achieved by the sub-tree rooted at t as well as an associated context, which encodes information about other parts of the overall program tree. The GetTargetSemantics function returns the target semantics associated with t, which is a modification of the semantics of t that incorporates the context. If context were *not* considered, the value returned by this function would simply be the semantics of t. However,

```
Input: t - Node to decompose. Has associated context and target semantics.
Input: funcSet - The set of invertible functions to use as internal nodes
Output: bestF - A function from funcSet to replace node t with
Output: bestSubtargs - A set of new sub-target nodes
 1: function DecomposeNode(t, funcSet)
 2:     target ← GetTargetSemantics(t)        ▷ Determine target based on t's context
 3:     minDistance ← 0
 4:     for f ∈ funcSet do
 5:         k ← arity of f
 6:         inputCombinations ←
 7:             Set of all k-combinations of input variables {x_1, ..., x_n}
 8:         for c ∈ inputCombinations do
 9:             d, subtargs ← ChooseSubtargets(target,f,c)
10:             if ValidSubtargets(subtargs) and d < minDistance then
11:                 bestF ← f
12:                 bestSubtargs ← subtargs
13:                 minDistance ← d
14:     return bestF, bestSubtargs
15: end function
```

Figure 3.11: DecomposeNode function from Algorithm 3.2.1. Exhaustively tries all possible node decompositions and returns the best one.

Table 3.2: Relationship between semantic bits, context bits, and target bits in GetTarget-Semantics

Semantic Bit	Context Bit	Target Bit
0	0	*
0	1	*
0	-	0
1	0	*
1	1	*
1	-	1

because certain bits are fixed by the context, the target semantics of t may be relaxed, because they cannot be modified by t's sub-tree. Fixed context bits of 0 or 1 are desired, as this will allow for more possible choices of sub-targets. In this way, t's sub-tree will be easier to learn. For any fixed bits in the context, the corresponding bit in the semantics of t will become a "don't care", indicated by the character '*'.

Recall that the context is represented as a vector of 0s, 1s, and dashes (-). If a context bit is a 0 or 1 then it is fixed, and the corresponding target bit becomes a '*'. If the context bit is a '-', then the target bit must still be learned by the sub-tree and will remain unchanged. Table 3.2 shows the mapping between semantic bit, context bit, and target bit.

The psuedocode for GetTargetSemantics is provided in Figure 3.12.

```
Input: t - Node to decompose. Has associated context and target semantics.
Output: target - Target semantics of t (incorporates context)
1: function GetTargetSemantics(t)
2:     target ← t.semantics
3:     L ← len(target)
4:     for i=1..L do
5:         if t.context[i] ≠ '-' then
6:             target[i] ← '*'
7:     return target
8: end function
```

Figure 3.12: GetTargetSemantics function from DecomposeNode. If the context is fixed, then set the corresponding target bit to a '*', or "don't care".

Choosing sub-targets. Given a particular function f, combination of inputs c, and target semantics (from GetTargetSemantics), the ChooseSubtargets function will return a set of new sub-target nodes such that the Hamming distance between the new sub-target's

semantics and the input semantics is minimized. By minimizing the Hamming distance, we are biasing towards sub-targets which are *closest to the leaves in semantic space*. This is done to minimize the depth and size of the overall tree.

Sub-targets are constructed one semantic bit at a time, based on the corresponding input bits, the target bit, and the output of f(c). For each bit, if the target bit and the output bit match, then the input bits are copied over to the corresponding sub-target bits. Similarly, if the target bit is a '*', then we are free to choose any sub-target bits, so we will prefer the input bits, which will be copied directly to the corresponding sub-target bits. If a mismatch between output and target occurs, then we must decide how to set the sub-target bits so that f(subtarget bits) = target bit (or equivalently, choose sub-target bits from f^{-1}(target bit)).

As an example, suppose f is the OR function and c is the input pair (x_1, x_3). The step-by-step process of setting the first four sub-target bits is illustrated in Figures 3.13 through 3.16. For each figure, the candidate node decomposition is shown on the left with semantics above each node.

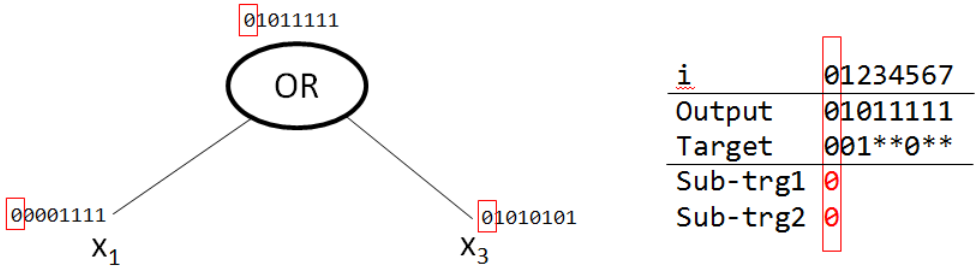

Figure 3.13: Setting sub-target bit 0. Since the output bit and the target bit match, then we can set the sub-target bits to match the input bits.

This process continues until all sub-target bits have been set. In this case, they would be set to 00001011 and 00110001, respectively. As a measure of quality for this node decomposition, the total Hamming distance between sub-targets and corresponding inputs is calculated and returned by the function. In this case it would be HammDist(00001111, 00001011) + HammDist(01010101, 00110001) = 4. This value will be used by the DecomposeNode function for tracking the best of the candidate node decompositions.

Finally, if a sub-target perfectly matches the input semantics, then a flag will be set for the sub-target to indicate that it is a leaf, and that no further decomposition is necessary.

The pseudocode for ChooseSubtargets is provided in Figure 3.17.

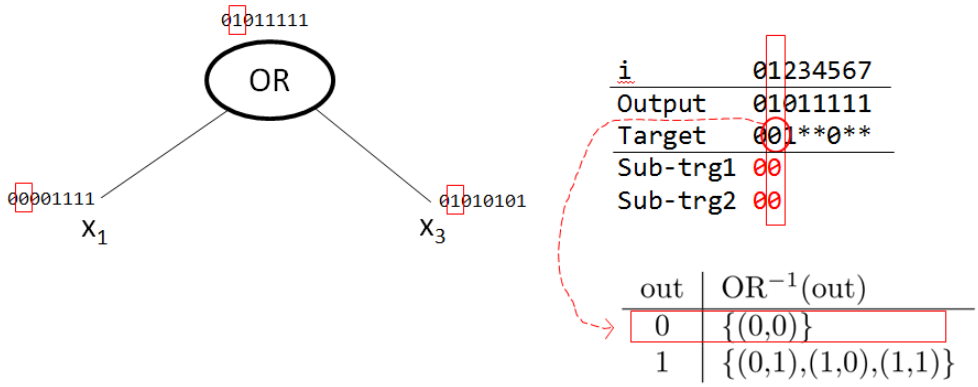

Figure 3.14: Setting sub-target bit 1. The output bit doesn't match the target bit, so we must decide how to set the sub-target bits. An inverse function lookup is performed to find the set of possible sub-targets we can choose from so that f(Sub-trg1,Sub-trg2)=Target. There is only one choice, so we must set the sub-targets to (0,0).

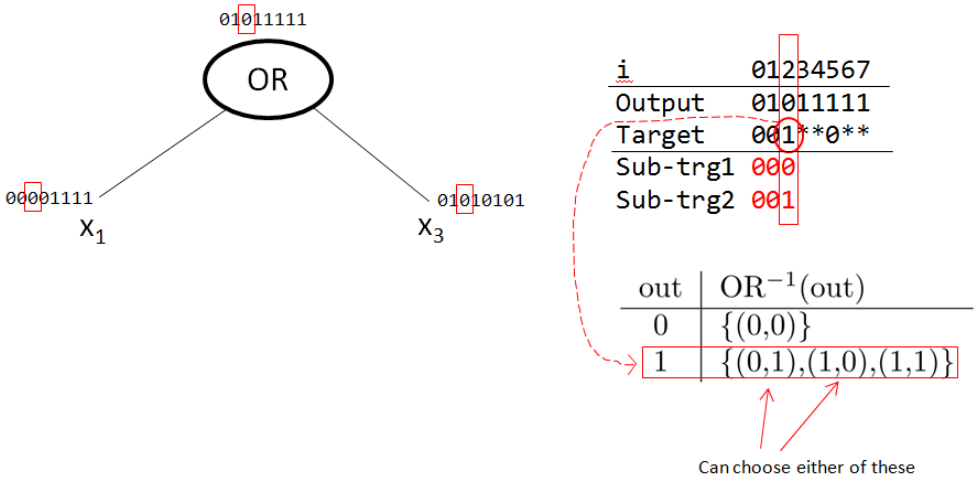

Figure 3.15: Setting sub-target bit 2. The output bit doesn't match the target bit, so an inverse function lookup is performed. We will prefer the sub-target bits that have minimal Hamming distance to the input bits. Note that there are two optimal choices of sub-target bits in this case, each of which have Hamming distance 1 to the input bits.

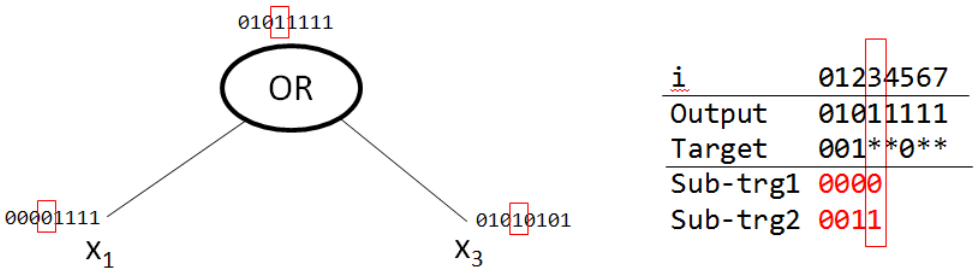

Figure 3.16: Setting sub-target bit 3. Since the target bit is '*', we are free to choose any sub-target bits, so we will choose the input bits.

Input: target - Target semantics for this node decomposition
Input: f - The function associated with this node decomposition
Input: c - The set of inputs associated with this node decomposition
Output: subtargs - Sub-target nodes that minimize distance to inputs in c
Output: d - Total distance between all sub-targets and corresponding inputs

```
1:  function ChooseSubtargets(target, f, c)
2:      d ← 0
3:      subtargs ← {Node({})}                          ▷ Initialize empty sub-target nodes
4:      L ← len(target)
5:      k ← arity of f
6:      for i=1..L do
7:          inputBits ← {x_j.semantics[i] | x_j ∈ c}
8:          if target[i] = '*' then                    ▷ target[i] is a "don't care"
9:              subtargetBits ← inputBits
10:         else
11:             B ← f^{-1}(target[i])                  ▷ Rows in truth table matching target bit
12:             subtargetBits ← argmin_{b∈B} {HammDist(b, inputBits)}
13:             d ← d + HammDist(subtargetBits, inputBits)
14:         for j=1..k do
15:             subtargs[j].semantics[i] ← subtargetBits[j]
16:     for j=1..k do
17:         if subtargs[j].semantics = c[j].semantics then
18:             subtargs[j].isLeaf ← True
19:     return d, subtargs
20: end function
```

Figure 3.17: ChooseSubtargets function from DecomposeNode. Chooses sub-targets that are closest to the inputs in semantic space.

Validating sub-targets. Once sub-targets have been chosen, then they must be validated to ensure that they don't match a previous target. If targets are allowed to repeat, then the tree decomposition will infinitely recurse. Therefore, we must enforce that all targets on a particular ancestral path are unique. If no repeated targets exist, then the function will return True. Otherwise, a repeated target exists and the function will return False.

The pseudocode for ValidateSubtargets is provided in Figure 3.18.

```
Input: subtargs - Sub-target nodes to verify
Output: True, if no repeated targets. False, otherwise.
1: function ValidateSubtargets(target, f, c)
2:     ancestralOutputs ← {a.semantics | a ∈ Ancestors(t)}
3:     for s ∈ subtargs do
4:         if s.semantics ∈ ancestralOutputs then
5:             return False                    ▷ Sub-targets exists in the ancestral path
6:     return True
7: end function
```

Figure 3.18: ValidateSubtargets function from DecomposeNode.

3.2.3.3 Tree Update

The tree update process includes creating branches to the newly-created sub-targets followed by updating the semantics and context of the entire tree.

The semantics must first be updated on the ancestral path of the node that was just decomposed. This is necessary because of constraint relaxation on the sub-target. When the node was decomposed, some of the target bits became "don't cares" because of fixed bits in the context. Therefore, these bits may not match the original target from before decomposition. These changes can also affect all of the nodes on the ancestral path, up to the root. These changes also necessitate an update of context of the *entire* tree, because semantic changes in one part of the tree can affect context of entirely different parts of the tree. The potential for context to change is important, as it will guide how relaxed or constrained each sub-target is when it is decomposed.

As an example of why the tree update is necessary, consider the example depicted in Figure 3.19. In this example, if the right node context is not updated, then it might produce incorrect results at the time of decomposition due to having a context that does not reflect the current state of the tree.

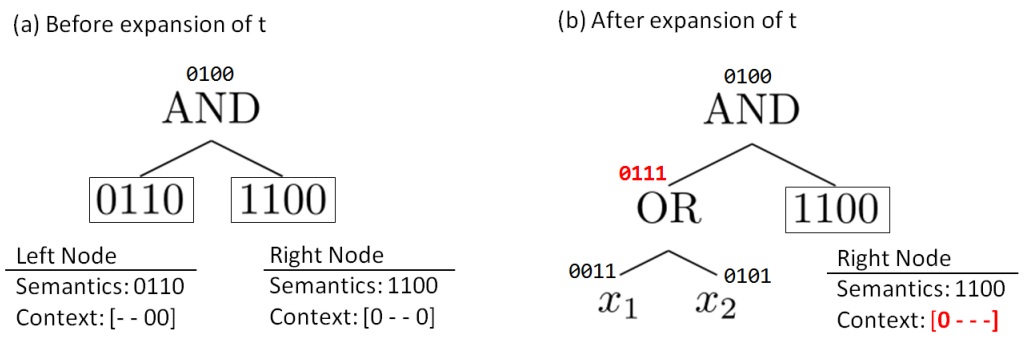

Figure 3.19: (a) Prior to decomposition, the semantics of the left node was 0110 and the context of the right node was [0 - - 0]. (b) After decomposition of the left node, the semantics changed to 0111 because of context relaxation. This change in semantics also changed the context of the right node on the *other side of the tree*. In this case, that side of the tree becomes *more* constrained because there are fewer fixed bits in the context. If the context was not updated, this additional constraint on the right node would be lost, possibly producing incorrect results at the output of the overall tree.

Lines 10 to 13 in Algorithm 3.2.1 update the program tree based on the function and sub-targets returned from the DecomposeNode function. First, the function is associated with the node and branches are created from the node to each of the sub-targets.

Next, UpdateAncestralSemantics updates the semantics of all nodes on the ancestral path of t, including t. All that needs to be done to update the semantics of a node is to re-evaluate the sub-tree rooted at that node, which can be done efficiently by updating nodes from leaf-to-root.

The change in semantics necessitates that the context be re-evaluated for the entire tree starting at the root. The tree is processed in a breadth-first fashion and the context of a node is evaluated as described in section 3.2.2.

3.2.4 Complexity Analysis

The time complexity of the DecomposeNode function is $O(|F| * \binom{n}{k} * |T| * k)$, where $|F|$ is the number of functions in funcSet, n is the number of input variables, k is the arity of $f \in$ funcSet with maximum arity, and $|T|$ is the size of the training data. The reason is that each of the $O(|F| * \binom{n}{k})$ pairs of function and input combinations are considered as potential target decompositions. Then for each of these pairs, $O(|T| * k)$ work is done to determine the new sub-targets.

Recall in line 4 of the DecomposeNode function that repeated sub-targets are disallowed to avoid recursive sub-targets. This means there are a maximum of $2^{|T|}$ possible sub-targets in

the tree. Therefore, the time complexity of the SDPS algorithm is $O(2^{|T|} * |F| * \binom{n}{k} * |T| * k)$. In reality, the number of node decompositions will be much less than $2^{|T|}$.

The space complexity of the program tree will be $O(2^{|T|})$, as this is the maximum number of sub-target nodes possible. As with the worst-case running time, the tree size will nearly always be much less than this.

The size of funcSet plays an important role in the execution time of the algorithm as well as the overall tree size. Although increasing $|F|$ will increase the worst-case running time of the DecomposeNode function, having more choices at each node decomposition generally reduces the total number of node decompositions needed, which reduces the overall running time. This will be observed in Chapter 4.

Chapter 4

Evaluation and Results

4.1 Evaluation Criteria

There will be four main criteria for evaluation of algorithms:

1. Program size (i.e. number of internal nodes in tree)
2. Accuracy on training and test sets
3. Training time
4. Tree evaluation time

For the SGP and SGP+ algorithms, the program size will be the combined size of the RPA and the PPA as described in the complexity analysis, as this represents the number of unique internal nodes in the program tree. For the SDPS algorithm, the number of internal nodes will be counted.

Accuracy on the training and test set data will be defined as the percentage of correctly classified instances:

$$Acc(h', T) = \frac{|\{h'(\boldsymbol{x_i}) = y_i \mid (\boldsymbol{x_i}, y_i) \in T\}|}{|T|} \tag{4.1}$$

where h' is the model returned by the algorithm and T can be either the training set or the test set.

Training time is the time it takes the algorithm to train and create the final model. The tree evaluation time is the time it takes for the model to evaluate the entire training set after the model has been created. All times will be in units of seconds as measured by the CPU time on a PC with a 3.0 GHz Intel Core 2 Duo CPU and 4 GB of RAM.

4.2 Experimental Procedures

Experiments will be performed to compare the relative effectiveness of the following three algorithms:

1. The SGP algorithm originally described in Section 2.7
2. The SGP+ algorithm proposed in Section 3.1
3. The SDPS algorithm proposed in Section 3.2

Note that the original SGP algorithm described in [11] included an extra simplification step at each generation to reduce the size of the program without altering the semantics. This step was omitted for ease of implementation in both the SGP and SGP+ algorithms. Removal of this step will not affect the comparison of algorithms because the simplification does not alter the semantics, and therefore the unsimplified and simplified models will be equivalent with respect to semantics. The program sizes and evaluation time will be inflated, but will be comparable between SGP and SGP+.

For all three algorithms, an identical function set will be used, namely {AND, OR, NAND, NOR}. This will allow for similar models to be built by all algorithms so that they may be fairly compared.

The algorithms will be compared using the evaluation criteria previously described against both synthetic Boolean datasets and real-world classification datasets from the UCI Machine Learning Repository. For the synthetic Boolean datasets, only a training set will be used due to the limited number of instances for each of the problems. For the UCI data sets, the data will be split into separate training and test sets, with the model being built against the training set alone. This is done to quantify how well a model generalizes to unseen instances in the test set.

Results reported will be averaged over 10 runs to eliminate elements of randomness in the GP-based algorithms as well as for the choice of instances in the training and test sets. A two-sample T-test will be used to determine the statistical significance of observed differences. For each run of the UCI data sets, 75% of the instances will be randomly chosen for the training set with the remaining 25% used for the test set.

4.2.1 Algorithm Parameters

The parameters for the genetic-programming based algorithms (SGP, SGP+) are described in Table 4.1. The selection of parents is done by tournament selection with tournament size 2 and probability 0.8, meaning that the more “fit” individual will be chosen 80% of the time. The

Table 4.1: Parameters for GP-based algorithms (SGP, SGP+)

Parameter	Value
Elitism?	Yes
Function Set	{AND, OR, NAND, NOR}
Population Size	200
Crossover Rate	1.0
Mutation Rate	0.1
Maximum Generations	50
Tournament Size	2
Tournament Probability	0.8
Initialization Method	Random

number of generations was limited to 50 to limit the depth of the tree for SGP/SGP+, as it will increase by one each generation.

The initial population (generation 0) will be randomly-created linear programs. A linear program is a sequence of instructions, similar to machine code, which includes a destination register, an op code, and two source registers. The initialization of these programs will be done based on the recommendations in [1]. Specifically, the programs will contain a random number of instructions between 10 and 30, with half of the instructions involving a constant (0 or 1) in one of the source registers. Each of the n inputs will be placed in registers 1 through n, and there will be an additional n registers for general usage. After execution of a linear program, it is assumed that the output resides in register 1. The decision to use linear programs in generation 0 was made for ease of implementation. It is important to note that this decision will only affect the *syntax* of the program. In theory, any kind of program syntax could be used without affecting the results (e.g. tree-based, grammar-based, stack-based, etc.).

There are some additional parameters that are specific to the SGP+ algorithm. These are described in Table 4.2. The values chosen are based on sensitivity analysis presented in Section 4.3.

Table 4.2: Parameters specific to the SGP+ algorithm

Parameter	Value
Initial RPA Size	(1.4 * Population Size)
RPA Rate	0.45
Parent Pool Size	7

There are no "magic numbers" for the SDPS algorithm. However, the choice of library functions used (funcSet in Algorithm 3.2.1) will play an important role. For most experiments,

this set will be limited to the same set used in the GP-based algorithms, namely {AND, OR, NAND, NOR}.

4.3 Synthetic Boolean Problems

This section will present results on several synthetic Boolean problems. Many of these are "deceptive" problems, meaning that the search may be deceived if there does not exist a clear path in the search space from a promising individual to the individual that solves the problem. Deceptive problems will help highlight the difference between traditional syntax-based GP algorithms and semantic-based algorithms. It is hard to qualify exactly what makes a problem deceptive, but an attempt has been made to distinguish between the deceptive and non-deceptive problems. Non-deceptive problems will also be considered, in order to observe how well the algorithms deal with "easy" problems.

4.3.1 Data Set Description

Table 4.3 lists the synthetic Boolean data sets used. For all problems, the number of inputs

Table 4.3: Synthetic Boolean Data Set Description

Name	Deceptive?	# Inputs	# Instances
PARITY5	Yes	5	32
PARITY6	Yes	6	64
PARITY7	Yes	7	128
PARITY8	Yes	8	256
MUX6	Yes	6	64
OR5	No	5	32
OR6	No	6	64
OR7	No	7	128
OR8	No	8	256
COMP6	No	6	64
COMP8	No	8	256
RANDOM5	-	5	32
RANDOM6	-	6	64
RANDOM7	-	7	128
RANDOM8	-	8	256

is limited to a maximum of 8 due to inherent limitations in the ability of the SGP and SGP+ algorithms to handle a large number of inputs.

The PARITY* problems will be computing odd parity, meaning that if there are an odd number of 1s in the input, the output will be a 1. The MUX6 problem will be computing

the multiplexer function with 4 input bits and two select bits. The OR* problems are simply computing the OR function with multiple inputs. The COMP* problems will be computing the comparator function, where half of the inputs are treated as input A, the other half treated as input B, and the output will be 0 if A $\leq$ B. Finally, the RANDOM* problems are completely randomized functions, which may or may not be deceptive. For instance, RANDOM5 will randomly choose one of the possible 2^{32} 5-input functions.

4.3.2 Results

There are four primary experiments performed on the synthetic Boolean problems, including parameter sensitivity experiments and an overall comparison of all algorithms.

4.3.2.1 SGP+ Parameter Sensitivity

The first experiment will be a parameter sensitivity test on the PARITY5 problem for the parameters that are specific to the SGP+ algorithm - namely the initial RPA size, the RPA rate, and the parent pool size. All values reported are averaged over 10 runs. The accuracy metric is omitted because it was 100% regardless of specific parameter value. The results for the initial RPA size sensitivity are provided in Table 4.4. The actual size of the initial RPA is equal to (Init RPA * Population Size), where the population size is fixed at 200. Generally speaking, the SGP+ algorithm is not very sensitive to the size of initial RPA, although the general trend seems to be that larger RPA sizes result in slightly smaller programs and shorter evaluation times. The learning time is largely unaffected because the RPA initialization occurs outside of the main evolutionary loop. The ideal value appears to be 1.4, so this will be the default RPA size used in subsequent experiments. With a population size of 200, this means that the RPA will be populated with 280 random minterm programs.

The results for the RPA rate sensitivity are provided in Table 4.5. This is the rate at which programs are added to the RPA each generation. It was expected that the more programs that are added to the RPA, the longer the training time will be, because the RPA must be searched every time a crossover operation occurs. So it is not surprising to see the training time increase proportionally with the RPA rate. It also appears that increasing this rate has a favorable impact on the program size and program evaluation time, presumably due to finding better crossover masks because of the increased diversity in the RPA. The ideal value appears to be 0.45, meaning that with a population size of 200, 90 programs will be randomly chosen to be added to the RPA at each generation. Programs will only be added to the RPA if they are not already present, so increasing the RPA rate even further will most likely not improve the evaluation metrics.

Table 4.4: SGP+ Parameter Sensitivity - Initial RPA Size

Init RPA	Program Size	Train Time (s)	Eval Time (s)
0.1	461.7	1.064	0.207
0.2	518.4	1.100	0.232
0.3	454.3	1.045	0.201
0.4	482.5	1.075	0.215
0.5	475.8	1.105	0.212
0.6	471.8	1.051	0.209
0.7	488.5	1.080	0.221
0.8	493.6	1.108	0.220
0.9	459.7	1.056	0.210
1.0	483.7	1.064	0.213
1.1	436.4	1.052	0.190
1.2	429.4	1.035	**0.186**
1.3	449.2	1.045	0.196
1.4	**425.4**	**1.012**	**0.186**
1.5	473.2	1.056	0.215
1.6	462.8	1.059	0.207
1.7	445.7	1.031	0.195
1.8	465.3	1.093	0.204
1.9	451.8	1.069	0.197
2.0	471.8	1.056	0.209

Table 4.5: SGP+ Parameter Sensitivity - RPA rate

RPA rate	Program Size	Train Time (s)	Eval Time (s)
0.00	658.2	1.145	0.279
0.05	524.3	**1.086**	0.213
0.10	568.4	1.185	0.237
0.15	550.5	1.230	0.227
0.20	499.8	1.234	0.204
0.25	460.6	1.242	0.186
0.30	485.9	1.302	0.198
0.35	483.1	1.340	0.197
0.40	521.1	1.431	0.216
0.45	**454.3**	1.377	**0.182**
0.50	459.5	1.364	0.189

The results for the parent pool size sensitivity are provided in Table 4.6. Recall from the

Table 4.6: SGP+ Parameter Sensitivity - Parent Pool Size

Parent Pool Size	Program Size	Train Time (s)	Eval Time (s)
2	664.2	**1.170**	0.285
3	582.8	1.288	0.243
4	541.6	1.332	0.223
5	432.3	1.325	0.175
6	400.2	1.497	0.160
7	394.3	1.684	0.158
8	329.1	1.784	0.129
9	361.9	2.130	0.146
10	358.8	2.359	0.143
11	324.1	2.618	0.130
12	332.6	3.143	0.135
13	323.4	3.363	0.130
14	**296.2**	3.704	**0.121**
15	306.8	4.156	0.125

SGP+ algorithm that parents are chosen from the parent pool, and that all pairs of parents in the pool are considered. Therefore, it is expected that the training time should increase with the the number of parents in the pool. Additionally, it is expected that the larger parent pools should reduce the program size and evaluation time, as it will be more likely to find the "ideal" parents for crossover, resulting in offspring that are closer to the target semantics. The experimental results match these expectations. In the ideal case, we would allow for a pool size of 200 consisting of everyone in the population, though this would increase the training time drastically. Therefore, a pool size of 7 was chosen a compromise between program size and training time.

4.3.2.2 SDPS Parameter Sensitivity

The next experiment will be a parameter sensitivity test on the PARITY5 problem for the parameters that are specific to the SDPS algorithm. There is only one parameter to consider, and that is the choice of the function set. Three choices will be considered - use only the base function set of {AND,OR,NAND,NOR}, use all 2-input functions, or use all 3-input functions. The results of this experiment are provided in Table 4.7. The function set represents what kinds of functions can be present at the internal nodes of the program trees. It is expected that having a larger number of functions to choose from will allow for a more expressive, and therefore shorter, tree. There are a total $2^4 = 16$ possible 2-input functions and $2^8 = 256$ possible 3-input functions. As expected, the shortest program trees are created when there are

Table 4.7: SDPS Parameter Sensitivity - Function Set

Function Set	Program Size	Train Time (s)	Eval Time (s)
{AND,OR,NAND,NOR}	61	**0.536**	0.003
All 2-input	40	0.600	0.002
All 3-input	**17**	7.321	**0.001**

a larger number of functions to choose from. Additionally, the evaluation time is shorter due to the shorter program size. To ensure a fair comparison with other algorithms, only the 4 base functions will be used in subsequent experiments.

4.3.2.3 SGP Metrics During Evolution

For the evolutionary GP algorithms, metrics will be observed during the training/learning process on the PARITY5 problem. The evaluation time will not be considered since that metric only applies after the model has been learned.

A comparison of program size per generation is provided in Figure 4.1. The SGP+ data

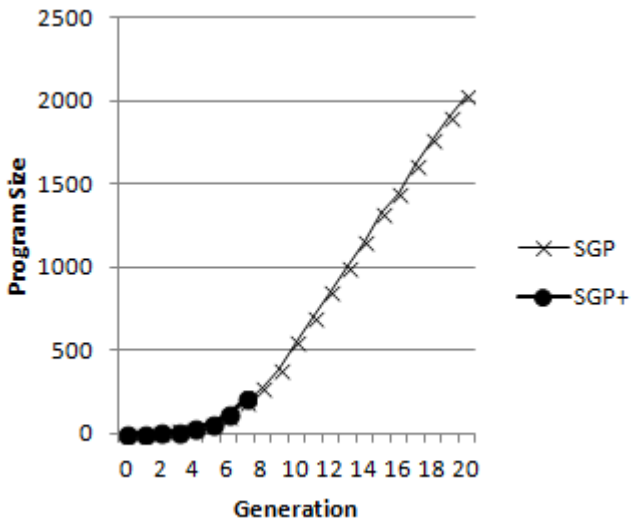

Figure 4.1: Program size vs. Generation for SGP and SGP+ on the PARITY5 problem

cuts off at generation 7 because the problem was solved at or before generation 7 in all 10 runs. This figure shows that the program size per generation is roughly equal for both algorithms. This is expected because the crossover and mutation operators increment the program size by equal amounts each generation in both SGP and SGP+. However, after training is complete

the size of the SGP+ program will be much smaller than the SGP program due to achieving perfect training set accuracy in an earlier generation.

A comparison of training set accuracy per generation is provided in Figure 4.2. This figure

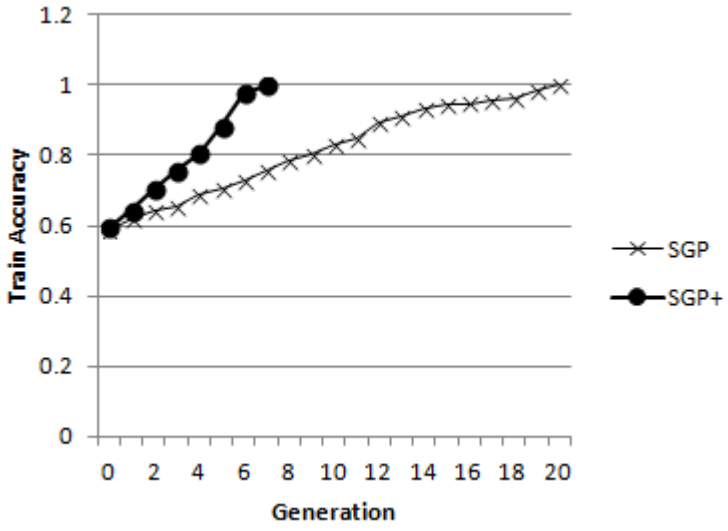

Figure 4.2: Training Set Accuracy vs. Generation for SGP and SGP+ on the PARITY5 problem

shows that accuracy per generation increases more for the SGP+ algorithm than it does for the SGP algorithm. This is primarily due to the SGP+ crossover operator taking *larger* and *more directed* steps in semantic space towards the target, compared to SGP. Additionally, the SGP+ mutation operator helps by taking a step in the direction of the target semantics instead of a step in a random direction. This improved accuracy per generation is essential to keeping the program size small since the size grows exponentially with the number of generations.

A comparison of training time per generation is provided in Figure 4.3. This figure shows that SGP is performing better than SGP+ with respect to training time per generation. This is expected, because the SGP+ algorithm sacrifices time each generation to perform a more selective search for parents that have a high chance of producing offspring that are close to the target in semantic space. Even with this sacrifice, the overall training time of the SGP+ algorithm is less than SGP due to finding a program with perfect accuracy in an earlier generation (0.916 seconds vs. 1.056 seconds).

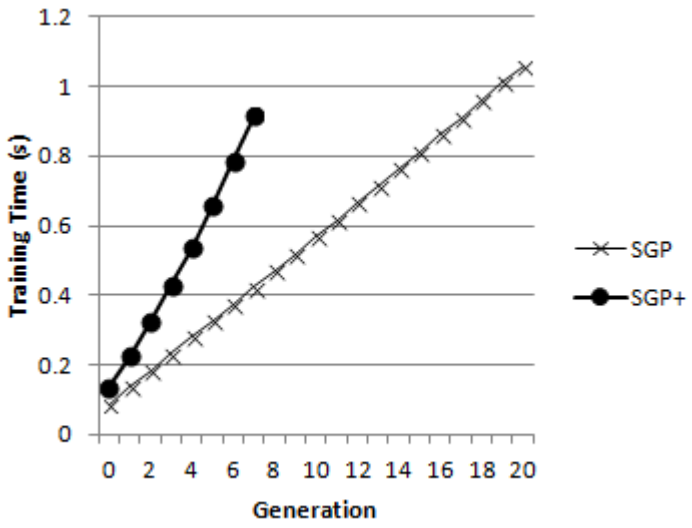

Figure 4.3: Training Time vs. Generation for SGP and SGP+ on the PARITY5 problem

4.3.2.4 Program Accuracy

An overall comparison of all algorithms over all metrics is provided in the next few sections. First, a comparison of program accuracy on the training set is provided in Table 4.8. An additional column is added for the standard GP algorithm. Results in this column were originally reported in [11], Table 1. Fields marked with a dash were not reported and are omitted. The first observation is that the GP algorithm does very poorly on the PARITY problems. The target semantics for these problems are half 1s and half 0s, so an accuracy of 50% could easily be achieved by creating a trivial program that outputs all 0s or all 1s. Standard GP does not do much better than 50% accuracy, so it seems ill-suited for that type of problem. In general, it is expected that standard GP does poorly on any kind of deceptive problem, hence the desire for semantic-based algorithms.

The SGP algorithm does fairly well for simple non-deceptive problems, and even deceptive problems with a low number of inputs. However, the larger PARITY7 and PARITY8 proved difficult for SGP, and the evolution was cut off at the maximum of 50 generations before a solution could be found. The accuracy of SGP+ and SDPS was nearly perfect in all cases (found a solution in 10 out of 10 runs), with the exception of one run of the PARITY8 problem on SGP+. These results show that SGP+ and SDPS can more accurately fit training data in cases where SGP+ and standard GP can not.

Table 4.8: Comparison of program accuracy for each algorithm

Problem	GP		SGP		SGP+		SDPS	
	mean	std	mean	std	mean	std	mean	std
PARITY5	0.529	0.024	**1.000**	0.000	**1.000**	0.000	**1.000**	0.000
PARITY6	0.505	0.007	0.998	0.005	**1.000**	0.000	**1.000**	0.000
PARITY7	0.501	0.002	0.888	0.013	**1.000**	0.000	**1.000**	0.000
PARITY8	0.501	0.002	0.748	0.012	0.997	0.010	**1.000**	0.000
MUX6	0.708	0.033	**1.000**	0.000	**1.000**	0.000	**1.000**	0.000
OR5	-	-	**1.000**	0.000	**1.000**	0.000	**1.000**	0.000
OR6	-	-	**1.000**	0.000	**1.000**	0.000	**1.000**	0.000
OR7	-	-	**1.000**	0.000	**1.000**	0.000	**1.000**	0.000
OR8	-	-	0.999	0.002	**1.000**	0.000	**1.000**	0.000
COMP6	0.802	0.038	**1.000**	0.000	**1.000**	0.000	**1.000**	0.000
COMP8	0.803	0.028	0.962	0.014	**1.000**	0.000	**1.000**	0.000
RAND5	0.822	0.066	**1.000**	0.000	**1.000**	0.000	**1.000**	0.000
RAND6	0.836	0.066	**1.000**	0.000	**1.000**	0.000	**1.000**	0.000
RAND7	0.851	0.053	0.930	0.018	**1.000**	0.000	**1.000**	0.000
RAND8	0.896	0.053	0.832	0.012	**1.000**	0.000	**1.000**	0.000

4.3.2.5 Program Size

Next, a comparison of program size is provided in Table 4.9. Note the addition of a CNF/DNF

Table 4.9: Comparison of program size for each algorithm

Problem	CNF/DNF	SGP		SGP+		SDPS	
		mean	std	mean	std	mean	std
PARITY5	79	2717.3	355.3	386.0	70.9	**61.0**	0.0
PARITY6	191	6045.1	463.8	1580.8	93.1	**136.0**	0.0
PARITY7	447	8087.9	193.9	3293.9	207.9	**301.0**	0.0
PARITY8	1023	9488.1	182.9	5723.0	297.5	**643.0**	0.0
MUX6	191	4072.3	494.6	386.7	89.5	**18.0**	0.0
OR5	**4**	184.3	150.9	24.2	0.8	**4.0**	0.0
OR6	**5**	341.9	257.8	24.2	0.6	7.0	0.0
OR7	**6**	1524.7	1337.4	25.2	1.7	9.0	0.0
OR8	**7**	3583.6	2586.8	26.0	1.8	11.0	0.0
COMP6	191	3581.7	370.7	386.4	105.4	**16.0**	0.0
COMP8	1023	9281.0	298.9	2126.9	269.4	**37.0**	0.0
RAND5	79	1597.7	436.4	100.3	40.1	**27.0**	0.0
RAND6	191	5609.0	804.1	1001.3	145.9	**75.0**	0.0
RAND7	447	8031.5	249.9	2201.7	123.5	**91.0**	0.0
RAND8	1023	9504.2	200.9	4892.0	333.5	**267.0**	0.0

column. This column represents the number of internal nodes that would be needed to represent the CNF/DNF formula (whichever is smaller) using only the allowed functions, namely {AND, OR, NAND, NOR}. This is a "baseline" for which to compare the program sizes obtained by each of the algorithms. If this baseline number cannot be beaten, then it is likely that the model is overly complex. To calculate the size of the CNF/DNF programs, if the truth table contains half ones and half zeros, then there are 2^{n-1} clauses, each of which require $(n-1)$ nodes.

These clauses are then combined with $2^{n-1} - 1$ nodes, for a total of $(n-1)2^{n-1} + 2^{n-1} - 1 = n2^{n-1} - 1$ nodes. Note that this number does not include negation of inputs, and represents a lower/conservative bound.

It is clear that the SDPS algorithm has the smallest program size for most problems. Program size was one of the key metrics to improve in the design of the SGP+ and SDPS algorithms, and it is clear that a drastic improvement was made compared to the program sizes of SGP. This applies to both the deceptive and non-deceptive problems. For example, OR8, a simple non-deceptive problem had thousands of internal nodes in the program tree, whereas both SGP+ and SDPS had around 10 to 30.

For the deceptive problems, it seems that both SGP and SGP+ have an unusually large size. In fact, they are unable to beat the CNF/DNF baseline, which means the model could be overly complex and contain extraneous/unnecessary computations. Significant algorithmic modifications would most likely be needed to overcome this limitation.

4.3.2.6 Training and Evaluation Time

A comparison of training time is provided in Table 4.10. This is the time, in seconds, that it takes for the model to be built. There are two points to make here. The first is that as

Table 4.10: Comparison of training time (s) for each algorithm

Problem	SGP		SGP+		SDPS	
	mean	std	mean	std	mean	std
PARITY5	1.554	0.135	1.694	0.184	**0.582**	0.062
PARITY6	5.085	0.371	6.779	0.401	**3.463**	0.218
PARITY7	**12.108**	0.603	22.816	1.400	20.686	0.194
PARITY8	**24.852**	1.143	71.715	4.490	129.595	1.825
MUX6	3.790	0.439	2.701	0.356	**0.475**	0.024
OR5	0.451	0.205	0.278	0.006	**0.035**	0.002
OR6	1.068	0.391	0.489	0.013	**0.150**	0.007
OR7	3.190	1.803	0.937	0.107	**0.475**	0.012
OR8	12.348	7.871	1.785	0.181	**1.455**	0.037
COMP6	3.222	0.271	2.802	0.323	**0.340**	0.007
COMP8	23.454	0.308	23.801	2.743	**4.746**	0.132
RAND5	1.088	0.153	0.901	0.154	**0.235**	0.001
RAND6	4.670	0.567	4.366	0.483	**1.694**	0.047
RAND7	11.467	0.041	13.433	0.627	**5.097**	0.101
RAND8	**23.065**	0.185	57.409	4.205	40.004	1.525

the number of inputs are increased, SGP+ takes longer than SGP. This is due to the RPA increasing linearly in size each generation, which causes the crossover operation to take longer. The second point is that the training time for SDPS was very long for the PARITY8 and RAND8 problems. This uncovers a weakness in the ability of SDPS to handle a large number of inputs for deceptive/difficult problems. This is due to the exhaustive search that occurs at

each node decomposition.

Finally, Table 4.11 provides a comparison of program evaluation times (after the model is built). This is the time it takes to evaluate the program on all instances in the training set. SDPS was best across the board, due to the smaller program sizes. In many cases, the

Table 4.11: Comparison of evaluation time (s) for each algorithm

Problem	SGP		SGP+		SDPS	
	mean	std	mean	std	mean	std
PARITY5	1.395	0.189	0.146	0.033	**0.003**	0.000
PARITY6	8.530	1.011	2.043	0.138	**0.012**	0.000
PARITY7	31.693	1.380	13.820	0.897	**0.060**	0.021
PARITY8	104.792	3.277	78.641	4.419	**0.244**	0.029
MUX6	5.679	1.249	0.378	0.109	**0.002**	0.000
OR5	0.063	0.060	0.001	0.000	**0.000**	0.000
OR6	0.321	0.259	0.003	0.000	**0.001**	0.000
OR7	4.891	4.755	0.011	0.003	**0.002**	0.000
OR8	34.449	26.684	0.037	0.013	**0.004**	0.000
COMP6	4.446	0.523	0.377	0.121	**0.001**	0.000
COMP8	96.016	3.681	26.469	3.827	**0.013**	0.002
RAND5	0.728	0.224	0.027	0.015	**0.001**	0.000
RAND6	7.463	1.087	1.179	0.208	**0.006**	0.000
RAND7	29.377	1.033	8.726	0.588	**0.016**	0.002
RAND8	98.014	2.797	66.254	4.689	**0.111**	0.035

evaluation time for SDPS is several orders of magnitude smaller than SGP.

4.4 Classification Problems

This section will present results on several real-world Boolean classification problems from the UCI Machine Learning Repository. Additionally, the data will be split into separate training and test sets to observe the ability of each algorithm to learn a model that can generalize to unseen instances.

4.4.1 Data Set Description

A variety of different classification tasks will be attempted, some more difficult than others. Additionally, the MONK problems (described in [15]) are used because training/test set accuracy numbers exist for a large variety of other machine learning algorithms. This will allow for a ranking of SGP+ and SDPS amongst existing machine learning techniques.

4.4.1.1 Preprocessing

All data sets consist of multiple discrete attributes and a single Boolean classification. For discrete attributes that can take on multiple values (e.g. color = {RED, BLUE, GREEN}),

each attribute value was assigned a unique binary encoding (e.g. color = {(0,0), (0,1), (1,0)}). Once all attributes are converted to binary, they are concatenated together to form the input vector. No conversion is necessary for the classifications, as they are already in Boolean/binary form.

4.4.1.2 UCI Data Sets

Table 4.12 lists the UCI Boolean data sets used. The Car data set was derived from a simple

Table 4.12: UCI Boolean Data Set Description

Problem	Atrributes	Inputs	Train Instances	Test Instances
Car	6	12	1296	432
TTT	9	18	718	240
MONK1	6	10	124	432
MONK2	6	10	169	432
MONK3	6	10	122	432

hierarchical model and classifies whether a car is acceptable or unacceptable according to attributes such as price, number of doors, capacity, trunk size, and estimated safety. The TTT data set contains Tic-Tac-Toe end-games, where each attribute represents one of the nine positions on the board and the corresponding token that is placed on it, one of {X,O,B}. The output is a class favorable/unfavorable, stating whether the board position is favorable for player X. The MONK's problems were the basis of a first international comparison of learning algorithms. The results were summarized in [15]. There are three MONK's problems, each with different Boolean target concepts. For the MONK3 problem, 5% class noise was added to the training set.

4.4.2 Program Accuracy

An overall comparison of all algorithms is provided, similar to what was done with the synthetic Boolean problems. Due to excessive run times, only a single run is performed on the Car and TTT data sets. 10 runs are averaged for the three MONK data sets. First, a comparison of training and test set accuracy is provided in Table 4.13. Overall, SDPS had the best training and test set accuracy, presumably due to a smaller program tree that was able to generalize to unseen instances in the test set. The test set accuracy for MONK2 and MONK3 is close between all algorithms. The MONK3 data set had 5% noise added to the training set instances, and it seems that SGP+ and SDPS had better test set accuracy than SGP. Usually when data sets are noisy, care should be taken not to overfit the model to the training data. For this

Table 4.13: Comparison of training and test set accuracy for each algorithm over the UCI data sets

Problem	SGP		SGP+		SDPS	
	Train Acc	Test Acc	Train Acc	Test Acc	Train Acc	Test Acc
Car	0.944	0.907	**1.000**	0.938	**1.000**	**0.988**
TTT	0.786	0.707	0.969	0.682	**1.000**	**0.874**
MONK1	0.960	0.773	**1.000**	0.801	**1.000**	**1.000**
MONK2	0.846	**0.703**	**1.000**	0.692	**1.000**	0.701
MONK3	0.975	0.873	**1.000**	**0.921**	**1.000**	0.917

paper, no steps were taken to avoid overfitting (e.g. pruning), so this would be a good area for future research.

To see how SGP, SGP+, and SDPS stack up to other learning algorithms, test set accuracy on the MONK problems is be compared in Table 4.14. To ensure a fair comparison, the test set contains identical instances for all algorithms. Most of these results are extracted from [15] where they were originally reported. Note that these results are over 20 years old (1991). The intent of the comparison is to get some idea of how the semantic algorithms compare with classical machine learning algorithms, so the raw accuracy numbers (which may be outdated) are not as important as the ranking. The rank column represents the ranking from 27 (worst) to 1 (best) of each algorithm for that particular problem. All three algorithms are ranked average in most cases, with the exception of SDPS on the MONK1 problem, which achieved perfect accuracy. It is likely that accuracy improvements could be made to the SGP+ and SDPS algorithms by implementing some form of tree pruning to avoid overfitting, as most other learning algorithms listed have overfitting avoidance built in. In general, it seems like Backpropagation and its variants perform the best, though this could be specific to the MONK problems.

4.4.3 Program Size, Training Time, and Evaluation Time

Next, a comparison of size, training time, and evaluation time is provided in Table 4.15. These results are similar to what was seen on the synthetic Boolean problems. The program sizes created by SGP and SGP+ seem excessively long, which leads to long evaluation times. In general, SGP+ outperforms SGP, and SDPS outperforms both SGP and SGP+. Additionally, the inability of SDPS to handle a large number of inputs is exposed on the TTT data set (which has 18 inputs), where SDPS had the longest evaluation time. This was also observed on the synthetic PARITY8 problem.

Table 4.14: Comparison of test set accuracy from 27 learning algorithms on the MONK problems

Algorithm	**MONK1**	rank	**MONK2**	rank	**MONK3**(noisy)	rank
AQ17-DCI	**1.000**	1	**1.000**	1	0.942	12
AQ17-HCI	**1.000**	1	0.931	5	**1.000**	1
AQ17-FCLS	-	27	0.926	6	0.972	8
AQ14-NT	-	27	-	27	**1.000**	1
AQ15-GA	**1.000**	1	0.868	7	**1.000**	1
Assistant Professional	**1.000**	1	0.830	8	**1.000**	1
mFOIL	**1.000**	1	0.692	13	**1.000**	1
ID5R-1	0.817	17	0.618	24	-	27
IDL	0.972	12	0.662	21	-	27
ID5R-hat	0.903	14	0.657	22	-	27
TDIDT	0.757	21	0.667	20	-	27
ID3	0.986	11	0.679	18	0.944	11
ID3, no windowing	0.832	16	0.691	16	0.956	9
ID5R-2	0.797	19	0.692	15	0.952	10
AQR	0.959	13	0.797	9	0.870	19
CN2	**1.000**	1	0.690	17	0.891	17
CLASSWEB 0.10	0.718	22	0.648	23	0.808	21
CLASSWEB 0.15	0.657	24	0.616	25	0.854	20
CLASSWEB 0.20	0.630	25	0.572	26	0.752	22
PRISM	0.863	15	0.727	10	0.903	16
ECOBWEB leaf pred.	0.718	23	0.674	19	0.682	23
Backpropagation	**1.000**	1	**1.000**	1	0.931	13
BP + weight decay	**1.000**	1	**1.000**	1	0.972	6
Cascade Correlation	**1.000**	1	**1.000**	1	0.972	7
SGP	0.773	20	0.703	11	0.873	18
SGP+	0.801	18	0.692	14	0.921	14
SDPS	**1.000**	1	0.701	12	0.917	15

Table 4.15: Comparison of size, training time, and evaluation time for each algorithm over the UCI data sets

Problem	SGP			SGP+			SDPS		
	Size	Train (s)	Eval (s)	Size	Train (s)	Eval (s)	Size	Train (s)	Eval (s)
Car	9887	129.8	1811.6	5016	239.6	1337.2	**61**	**77.6**	**0.1**
TTT	8497	**81.6**	629.1	5703	141.0	520.7	**148**	259.4	**0.1**
MONK1	8096	12.6	28.6	1392	8.2	5.5	**20**	**2.7**	**0.0**
MONK2	8179	**16.9**	47.7	3017	19.2	22.0	**127**	19.4	**0.0**
MONK3	4387	12.1	14.1	467	4.3	1.4	**34**	**3.7**	**0.0**

4.5 Statistical Significance of Results

To understand the significance of the differences observed, a two-sample T-test is performed to compare each of the metrics on the synthetic Boolean problems. A comparison will be made between SGP/SGP+ as well as SGP+/SDPS. The p-value provided represents the two-tailed significance of the calculated t-value, rounded to four decimal places. An additional column is provided to state whether one algorithm is significantly better than the other with respect to the metric in question at a 95% confidence level.

A T-test on training set accuracy is given in Table 4.16. Most of the differences in training

Table 4.16: T-test on training set accuracy for synthetic Boolean problems

Problem	SGP/SGP+ Compare				SGP+/SDPS Compare			
	Diff	t(df=9)	p	SGP+ Better?	Diff	t(df=9)	p	SDPS Better?
PARITY5	0.000	0.000	1.0000	-	0.000	0.000	1.0000	-
PARITY6	0.002	0.085	0.9342	-	0.000	0.000	1.0000	-
PARITY7	0.112	2.947	0.0163	better	0.000	0.000	1.0000	-
PARITY8	0.249	5.036	0.0007	better	-0.003	-0.090	0.9303	-
MUX6	0.000	0.000	1.0000	-	0.000	0.000	1.0000	-
OR5	0.000	0.000	1.0000	-	0.000	0.000	1.0000	-
OR6	0.000	0.000	1.0000	-	0.000	0.000	1.0000	-
OR7	0.000	0.000	1.0000	-	0.000	0.000	1.0000	-
OR8	0.001	0.067	0.9480	-	0.000	0.000	1.0000	-
COMP6	0.000	0.000	1.0000	-	0.000	0.000	1.0000	-
COMP8	0.038	0.963	0.3605	-	0.000	0.000	1.0000	-
RAND5	0.000	0.000	1.0000	-	0.000	0.000	1.0000	-
RAND6	0.000	0.000	1.0000	-	0.000	0.000	1.0000	-
RAND7	0.070	1.565	0.1520	-	0.000	0.000	1.0000	-
RAND8	0.168	4.601	0.0013	better	0.000	0.000	1.0000	-

set accuracy were insignificant, due to each of the algorithms being able to solve the problem in all 10 out of 10 runs with perfect accuracy. However, there were some significant differences between SGP and SGP+ for the problems with a large number of inputs. This means that SGP+ and SDPS are more likely than SGP to solve problems with a large number of inputs.

A T-test on program size is given in Table 4.17. For all problems, there was a significant difference in program sizes between SGP and SGP+. This difference easily surpasses the 95% confidence level, as indicated by the near-zero p-values. This means that the improvements made in the SGP+ algorithm resulted in the creation of smaller programs. Furthermore, there was a significant difference in program sizes between SGP+ and SDPS for all problems, meaning that SDPS had even smaller programs than SGP+ (and much much smaller than SGP).

A T-test on training time is given in Table 4.18. Between SGP and SGP+, significant differences were observed, though not always for the better. In many cases, the training time on SGP+ is significantly longer than SGP. This primarily occurs on the deceptive problems. This result is not too surprising because the growing RPA in SGP+ causes longer search times

Table 4.17: T-test on program size for synthetic Boolean problems

Problem	SGP/SGP+ Compare				SGP+/SDPS Compare			
	Diff	t(df=9)	p	SGP+ Better?	Diff	t(df=9)	p	SDPS Better?
PARITY5	2331.3	338.776	0.000	better	325.0	115.793	0.000	better
PARITY6	4464.3	567.526	0.000	better	1444.8	449.215	0.000	better
PARITY7	4794.0	717.487	0.000	better	2992.9	622.711	0.000	better
PARITY8	3765.1	515.343	0.000	better	5080.0	883.571	0.000	better
MUX6	3685.6	457.494	0.000	better	368.7	116.918	0.000	better
OR5	160.1	38.996	0.000	better	20.2	67.753	0.000	better
OR6	317.7	59.291	0.000	better	17.2	66.615	0.000	better
OR7	1499.5	122.931	0.000	better	16.2	37.274	0.000	better
OR8	3557.6	209.771	0.000	better	15.0	33.541	0.000	better
COMP6	3195.3	439.323	0.000	better	370.4	108.236	0.000	better
COMP8	7154.1	900.300	0.000	better	2089.9	381.986	0.000	better
RAND5	1497.4	205.792	0.000	better	73.3	34.726	0.000	better
RAND6	4607.7	448.481	0.000	better	926.3	230.062	0.000	better
RAND7	5829.8	905.082	0.000	better	2110.7	569.789	0.000	better
RAND8	4612.2	598.544	0.000	better	4625.0	759.775	0.000	better

Table 4.18: T-test on training time for synthetic Boolean problems

Problem	SGP/SGP+ Compare				SGP+/SDPS Compare			
	Diff	t(df=9)	p	SGP+ Better?	Diff	t(df=9)	p	SDPS Better?
PARITY5	-0.140	-0.744	0.476	-	1.112	6.726	0.000	better
PARITY6	-1.694	-5.784	0.000	worse	3.316	12.644	0.000	better
PARITY7	-10.708	-22.698	0.000	worse	2.130	5.061	0.001	better
PARITY8	-46.863	-59.235	0.000	worse	-57.880	-69.098	0.000	worse
MUX6	1.089	3.664	0.005	better	2.226	10.833	0.000	better
OR5	0.173	1.130	0.288	-	0.243	8.150	0.000	better
OR6	0.579	2.733	0.023	better	0.339	7.191	0.000	better
OR7	2.253	4.891	0.001	better	0.462	4.018	0.003	better
OR8	10.563	11.168	0.000	better	0.330	2.120	0.063	-
COMP6	0.420	1.635	0.137	-	2.462	12.857	0.000	better
COMP8	-0.347	-0.596	0.566	-	19.055	33.714	0.000	better
RAND5	0.187	1.012	0.338	-	0.666	5.075	0.001	better
RAND6	0.304	0.890	0.397	-	2.672	11.011	0.000	better
RAND7	-1.966	-7.216	0.000	worse	8.336	29.310	0.000	better
RAND8	-34.344	-49.174	0.000	worse	17.405	21.813	0.000	better

in the crossover operation. This is the sacrifice that is made in SGP+ to gain smaller program sizes. Between SGP+ and SDPS, SDPS takes significantly less time to train in 13 out of 15 problems. However, in the case of PARITY8 the time was significantly more than SGP+ by a large margin (57.880 seconds on average). As mentioned before, SDPS has a weakness in that it does not deal with a large number of inputs very efficiently. Every node decomposition involves considering every pair of inputs. For problems that require many node decompositions, this can drastically increase the training time.

Finally, a T-test on evaluation time is given in Table 4.19. Similar to the results from the

Table 4.19: T-test on evaluation time for synthetic Boolean problems

Problem	**SGP/SGP+ Compare**				**SGP+/SDPS Compare**			
	Diff	t(df=9)	p	SGP+ Better?	Diff	t(df=9)	p	SDPS Better?
PARITY5	1.249	7.953	0.0000	better	0.143	2.362	0.0425	better
PARITY6	6.487	18.155	0.0000	better	2.031	16.402	0.0000	better
PARITY7	17.873	35.533	0.0000	better	13.76	43.084	0.0000	better
PARITY8	26.151	28.280	0.0000	better	78.397	111.516	0.0000	better
MUX6	5.301	13.647	0.0000	better	0.376	3.417	0.0077	better
OR5	0.062	0.759	0.4671	-	0.001	0.000	1.0000	-
OR6	0.318	1.875	0.0936	-	0.002	0.000	1.0000	-
OR7	4.880	6.712	0.0001	better	0.009	0.493	0.6339	-
OR8	34.412	19.980	0.0000	better	0.033	0.868	0.4078	-
COMP6	4.069	15.211	0.0000	better	0.376	3.243	0.0101	better
COMP8	69.547	76.144	0.0000	better	26.456	40.560	0.0000	better
RAND5	0.701	4.302	0.0020	better	0.026	0.637	0.5401	-
RAND6	6.284	16.566	0.0000	better	1.173	7.716	0.0000	better
RAND7	20.651	48.660	0.0000	better	8.71	34.018	0.0000	better
RAND8	31.760	34.824	0.0000	better	66.143	91.296	0.0000	better

T-test on program size, in most cases there are significant improvements in evaluation time. This is expected, because smaller programs can be evaluated more quickly. SGP+ evaluation time was significantly less than SGP, often by a large margin. Likewise, SDPS evaluation time was significantly less than SGP+ (and much much less than SGP) on 10 out of 15 problems.

4.6 Analysis

For the classification problems, it appears that SGP, SGP+, and GPS perform poorly on the MONK2 problem. Although other machine learning algorithms also perform worst on MONK2, it is interesting to analyze because it may uncover a weakness that could be improved upon. The definition of the MONK2 problem is that the output is a 1 if exactly two attributes take on their first attribute value. This means that the other 4 attributes must not take on their first attribute value. An example instance could be $\{\{a_1{=}00,a_2{=}00,a_3{=}1,a_4{=}10,a_5{=}01,a_6{=}1\},\text{out}{=}1\}$, where the first two attributes take on their first value. The problem statement is complex and difficult

to represent with traditional DNF or CNF. Modularity could be useful for this problem, particularly the equality function. Most attributes are encoded with two bits, so testing the equality of 2 attributes would require a 4-input equality function. If this function were allowed to used in the tree, it could make the target concept easier to learn. This was attempted with SDPS, where the equality function was added to the function library, but there was only a marginal 2% improvement in test set accuracy. The improvement was small most likely due to the choice of attribute encoding. Since multiple bits are representing a single attribute, they should stay together (e.g. attribute 1 encoded into input bits 0 and 1, so bits 0 and 1 should never occur in isolation). However, the current implementation does not handle these multi-bit inputs, so the algorithm was unable to take advantage of the equality function. To overcome this limitation, a new type of tree representation would be needed for handling multi-value discrete attributes. One such representation is provided in Figure 4.4. Note that the new multi-value nodes on the

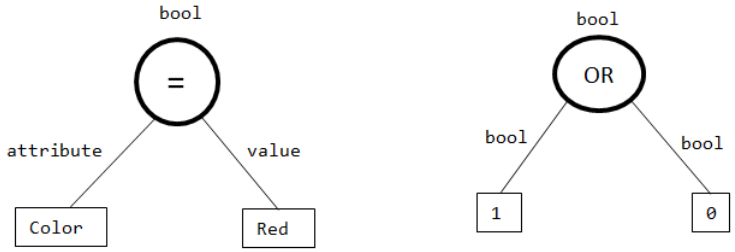

Figure 4.4: A potential solution to the multi-value discrete attribute problem. Two different types of nodes would be allowed in the tree - a multi-value node (left) and a normal Boolean node. The multi-value node would take an attribute type on the left branch (e.g. Color) and a constant value of that type (e.g. Red). In this example, the node will output a 1 if the color associated with an input is Red.

left of the figure would only be needed at the leaf level for translating discrete attributes into Boolean values. This ensures compatibility with normal Boolean nodes that are used in higher levels of the tree.

The fact that the training accuracy for SGP+ and SDPS on MONK2 is 100% while the test set accuracy is around 70% could indicate that overfitting is occurring. This could be overcome by implementing some form of pruning. It would be fairly easy to add pruning support to SDPS, as the majority of node decompositions occur at the leaves in an attempt to perfectly classify the last few instances. If the tree were pruned, training accuracy would be sacrificed and tree size would be smaller, but the program may generalize better to the test set instances as a result.

Chapter 5

Conclusions

Two new algorithms were proposed, SGP+ and SDPS, that directly search the semantic space and are improvements over the existing SGP algorithm. In particular, both SGP+ and SDPS build programs that are significantly smaller in size, with SDPS program sizes being even smaller than SGP+. For the deceptive parity problems, SGP+ programs were 3.8 times smaller than SGP and SDPS programs were 32.5 times smaller than SGP, on average. The reduction in program size also results in a significant reduction in program evaluation time, which is important when processing large amounts of data. Additionally, classification accuracy had a significant 17.6% improvement for high-arity deceptive Boolean problems, such as the 8-input odd parity problem. Finally, SDPS has shown better generalization to unseen instances (from the test set) on 4 out of 5 UCI Boolean classification problems. This improvement in model generalization is presumably due to the reduction in program size.

The proposed SGP+ algorithm included several strengths and weaknesses that were revealed in the experiments. Strengths include:

- Better at solving deceptive problems than standard GP (see Table 4.8). This is an important result of semantic algorithms in general, as standard GP algorithms appear to be ill-equipped for solving deceptive Boolean problems.
- Smaller program sizes than SGP (see Table 4.17).
- Solves high-arity deceptive problems more frequently than SGP (see Table 4.16).
- Shorter evaluation times due to smaller program sizes (see Table 4.19).
- Covers more instances per generation than SGP (see Figure 4.2).

Some of the weaknesses (or areas of improvement) for SGP+ include:

- Lots of magic numbers. This is a general problem for genetic programming, but SGP+ adds 3 more magic numbers which can affect the performance of the program.
- Long training times for problems that require many generations. This is due to the random program archive that increases linearly in size each generation.
- Higher memory usage than SGP, due to the RPA.
- Usage of random programs is counter-intuitive. Random programs were also used in SGP, so it's not new to SGP+, but it is still a problem because it is counter to the principle of Occam's Razor. It also makes the resulting program very hard for a human reader to interpret.
- Program length is still excessively long. Often the programs output by SGP+ have more nodes than a simple DNF minterm program would have, which means that there is a lot of unnecessary computation (introns) in the program.
- Inability to deal with overfitting and noisy data. The results from the UCI experiments suggest that SGP+ programs do not generalize all that well to unseen instances, which could be due to overfitting.

The strengths of the SDPS algorithm include:

- Avoids usage of random programs. Programs are therefore easier for humans to interpret and analyze.
- No magic numbers. This is a big improvement over the genetic programming algorithms. The only input that is required is the set of library functions to use at internal tree nodes.
- Smaller program sizes than both SGP and SGP+ on all deceptive Boolean problems (see Table 4.17).
- Shorter evaluation time than SGP and SGP+ on most problems.
- Algorithm is flexible enough to handle arbitrary Boolean functions as internal tree nodes. Increasing the number of functions in the library results in much improved performance (see SDPS parameter sensitivity).

Some of the weaknesses of SDPS include:

- Increased training time for large input sizes, due to exhaustive search that occurs for every node decomposition.

- Inability to deal with overfitting and noisy data. However, due to the top-down construction of the program pruning is still option, as most of the overfitting should occur at the leaves. This could be an area for future research.

- Very susceptible to poor greedy decisions near the root of the program tree. Bad decisions result in poor semantic decomposition, which can result in much larger program trees. This problem is amplified by the fact that the first few node decompositions are usually the most uncertain and therefore are the most likely to be "bad" decisions. In other words, the semantic space landscape is largely unknown near the beginning of the algorithm, and this is the time when semantic decomposition is the most critical.

The reduction in program size was the key metric to optimize in all experiments. Smaller program sizes typically resulted in shorter training times, improved classification accuracy on both seen and unseen instances, and shorter program evaluation times. With this in mind, it appears that SDPS is the most promising semantic-based algorithm, as it achieved significantly smaller program sizes than any of the other algorithms tested.

In general, semantic search appears most useful for solving deceptive problems. It succeeds in finding a solution more often than standard GP, where the search falls flat due to difficult fitness landscape. However, for general classification problems, the proposed semantic algorithms do not stack particularly well to other algorithms, such as Backpropagation and ID3. In conclusion, semantic algorithms are good for solving certain types of deceptive problems, but further improvements would be necessary to make it a strong learning algorithm in general.

5.1 Future Work

There are several areas of future research which could improve some of the weaknesses of SGP+ and SDPS previously discussed. The first would be extension to non-Boolean domains. Extension to the regression problem domain would not be too difficult, as the semantics would be replaced by real numbers instead of 1s and 0s. Also, the semantic space will be Cartesian, which means the distance metric would have to change from Hamming distance to Euclidean distance.

There are several efficiency improvements that could be made in both algorithms. The most obvious are the exhaustive search of the RPA in SGP+ during crossover operations and the exhaustive search of every combination function/inputs in SDPS. One solution would be to perform random sampling. This would reduce the training time and allow for extension to a larger number of inputs at the expense of some classification accuracy.

Currently, only single-output functions are allowed, but it would be desirable to extend to

multiple outputs. This would be similar to a circuit-based or graph-based model, where the output of a single gate can be fed to multiple locations (fan-out). One naive solution is to run the algorithm multiple times, once for each output bit. This would result in a lot of repeated computation, so a better solution would be to build support into the algorithm itself.

One of the weaknesses of SDPS was its poor greedy decisions near the root of the tree. Therefore, it would be useful to include some sort of backtracking and/or beam search to minimize this risk.

Finally, it would be desirable to harness the power of program modularity. There has been a lot of research dedicated to discovering modularity within programs, and some of those ideas could be applied to SGP+ and SDPS. As an example, sub-problems in the SDPS algorithm could be marked as modular programs and added to the function set, where they could be re-used in other parts of the tree. This would increase the node decomposition time, but may result in fewer node decompositions overall due to the increased number of function choices.

Bibliography

[1] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming.* Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

[2] Giuseppe Cuccu and Faustino Gomez. When novelty is not enough. In *Applications of Evolutionary Computation*, volume 6624 of *Lecture Notes in Computer Science*, pages 234–243. Springer Berlin / Heidelberg, 2011.

[3] David Jackson. Promoting phenotypic diversity in genetic programming. In *Parallel Problem Solving from Nature, PPSN XI*, volume 6239 of *Lecture Notes in Computer Science*, pages 472–481. Springer Berlin / Heidelberg, 2010.

[4] John R Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems.* Stanford University, Department of Computer Science, 1990.

[5] Krzysztof Krawiec. Semantically embedded genetic programming: automated design of abstract program representations. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1379–1386, New York, NY, USA, 2011. ACM.

[6] Krzysztof Krawiec. Medial crossovers for genetic programming. In *Genetic Programming*, volume 7244 of *Lecture Notes in Computer Science*, pages 61–72. Springer Berlin / Heidelberg, 2012.

[7] Krzysztof Krawiec. On relationships between semantic diversity, complexity and modularity of programming tasks. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 783–790, New York, NY, USA, 2012. ACM.

[8] Krzysztof Krawiec and Bartosz Wieloch. Functional modularity for genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 995–1002, New York, NY, USA, 2009. ACM.

[9] Joel Lehman and Kenneth O. Stanley. Efficiently evolving programs through the search for novelty. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 837–844, New York, NY, USA, 2010. ACM.

[10] Nicholas McPhee, Brian Ohs, and Tyler Hutchison. Semantic building blocks in genetic programming. In *Genetic Programming*, volume 4971 of *Lecture Notes in Computer Science*, pages 134–145. Springer Berlin / Heidelberg, 2008.

[11] Alberto Moraglio, Krzysztof Krawiec, and Colin Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31. Springer Berlin / Heidelberg, 2012.

[12] Jean-Baptiste Mouret and Stéphane Doncieux. Using behavioral exploration objectives to solve deceptive problems in neuro-evolution. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 627–634. ACM, 2009.

[13] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3):339–363, September 2010.

[14] Dao Ngoc Phong, Nguyen Quang Uy, Nguyen Xuan Hoai, and R. McKay. Evolving approximations for the gaussian q-function by genetic programming with semantic based crossover. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1 –6, june 2012.

[15] Sebastian B Thrun, Jerzy Bala, Eric Bloedorn, Ivan Bratko, Bojan Cestnik, John Cheng, Kenneth De Jong, Saso Dzeroski, Scott E Fahlman, D Fisher, et al. The monk's problems a performance comparison of different learning algorithms. 1991.