International Journal of Computational Intelligence and Applications
© World Scientific Publishing Company

Semantic Search Techniques for Learning Smaller Boolean Expression Trees in Genetic Programming

Nicholas C. Miller* and Philip K. Chan†

Department of Computer Sciences
Florida Institute of Technology
150 W. University Blvd.
Melbourne, FL 32901, USA

Received October 27, 2013
Revised June 15, 2014

One sub-field of Genetic Programming (GP) which has gained recent interest is semantic GP, in which programs are evolved by manipulating program *semantics* instead of program syntax. This paper introduces a new semantic GP algorithm, called SGP+, which is an extension of an existing algorithm called SGP. New crossover and mutation operators are introduced which address two of the major limitations of SGP: large program trees and reduced accuracy on high-arity problems. Experimental results on "deceptive" Boolean problems show that programs created by the SGP+ are 3.8 times smaller while still maintaining accuracy as good as, or better than, SGP. Additionally, a statistically significant improvement in program accuracy is observed for several high-arity Boolean problems.

Keywords: Genetic programming; semantic search; Boolean.

1. Introduction

Genetic programming, popularized by John Koza [1], 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? The relationship between program syntax and program behavior (i.e. semantics) is a complex one - even minor changes to program syntax can have drastic changes to program semantics. There has been an increased interest in semantic GP in recent years as an alternative to syntax-based GP representations [2], where the goal is to perform a more direct search in semantic space. Semantic GP can be used to solve "deceptive" problems, which are problems with a particularly complicated syntax-semantic mapping which causes traditional GP to fall short. One promising approach - the

*nmiller2011@my.fit.edu
†pkc@cs.fit.edu

SGP algorithm - was presented by Moraglio et al. [3], but has the limitation that the programs are very large.

The goal of this paper is to address that limitation - namely, to create semantically-evolved programs that are smaller overall and are at least as accurate as the programs created by SGP on both deceptive and non-deceptive problems. As a by-product of the smaller size, the programs should also execute faster.

A new algorithm, SGP+, is proposed which evolves smaller programs by introducing new crossover and mutation operators that take advantage of programs discovered in *prior* generations. A random program archive (RPA) is maintained that contains a history of the semantics of prior programs, and these are used to direct evolution more quickly towards the target, which results in smaller programs.

Experimental results on deceptive synthetic Boolean problems indicate that SGP+ creates significantly smaller programs which are 3.8 times smaller than SGP programs, on average. The accuracy of these programs is as good as SGP, and in several cases the accuracy was improved by a statistically significant amount.

The paper is organized as follows: Section 2 introduces key concepts and ideas necessary to understand semantic GP algorithms; Section 3 discusses prior research into the sub-field of semantic GP; Section 4 presents the SGP+ algorithm; Section 5 presents experimental procedures and observations; finally, Section 6 presents conclusions and areas of future work.

2. Background

The problem can be stated as follows: given a set of n input-output pairs (or instances) $T = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_n}, y_n)\}$, find a model, or hypothesis, $h : \mathbf{X} \rightarrow Y$ that interpolates all known input-output pairs:

$$\forall(\boldsymbol{x_i}, y_i) \in T, h(\boldsymbol{x_i}) = y_i \tag{1}$$

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 primarily for simplicity of implementation and analysis, but the ideas apply equally well to other domains (e.g. regression). The space of hypothesis functions, H, is the space of all possible Boolean expression trees. An example of a Boolean expression tree is shown in Figure 1.

The *semantics* of an expression tree h' can be expressed as a vector $\boldsymbol{Y'}$ corresponding to the output of the tree for each $\boldsymbol{x_i} \in T$. If visualized in truth table form, each row represents an instance in T and the final column represents the semantics.

The notion of semantic space is 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

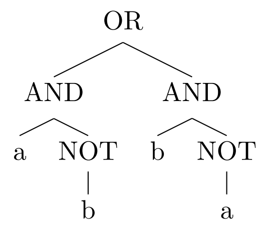


Fig. 1. Example of an XOR expression tree in H

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 expression tree that can produce the semantics represented by a point. Using the example from Figure 1, a program with equivalent semantics would be (OR (AND (NOT A) B) (AND (NOT B) A)) and would be represented by the same point in semantic space.

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.

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. 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.

3. Related Work

The idea of *semantic novelty* is introduced by Lehman [4], which is used to find novel, or unique, programs during the evolution of a genetic program in an effort to promote diversity and reduce overfitting in deceptive problem domains. It is a divergent search, as there is no objective other than to produce novel programs that have maximal distance to their k nearest neighbors in semantic space. Combining novelty with objective fitness (multi-objective search) can often produce better results overall [5,6]. A common pitfall in GP is insufficient population diversity. One strategy shown to improve diversity is to ensure there is a sufficient amount of semantic diversity in the initial population, and to also ensure that the diversity is maintained using a diversity-promoting crossover operator [7].

An investigation into the nature of GP in semantic space is investigated by McPhee et al. [8]. They find that the majority of crossover operations (about 60%) produce programs that are *semantically identical* to the parent programs, resulting in no movement towards the final objective. Various methods have been proposed for utilizing program semantics in genetic programming to counteract these inefficiencies. For example, Phong et al. [9] create a semantically-aware crossover operator in order to approximate the Gaussian Q-Function, for which no closed form currently exists. Similar crossover operators are proposed by Uy et al. [10,11]. Krawiec et al. [12] investigate the semantic similarities and differences between a random sampling of programs. The key result found is that as tasks become more complex, they also become more modular. This means that an algorithm that can compose a solution from sub-solutions, or conversely, that can decompose a problem into smaller sub-problems, may be able to tackle more complex problems.

The idea of program composition is investigated by Krawiec [13] by performing a GP search in the embedded semantic space of small, depth-limited tree-based programs. The search space is arranged such that nearby programs have similar semantics. This creates a smoother fitness landscape, which is easier to search. The results indicate that the space can be exploited in a compositional manner to build larger compound programs, resulting in more effective search in the larger space of programs. Further research by Krawiec [14,15,16] focuses on module identification and module exploitation in GP using a monotonicity metric. Problem decomposition in GP is also used by Kattan et al. [17] with promising results in the symbolic regression domain.

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* is discussed by Krawiec [18] where the goal is to find an approximately medial (in semantic space) crossover. Several other geometric crossovers have also been proposed [19,20]. A geometric crossover operator is used in the SGP algorithm [3], but with the relaxed contraint that the offspring is not necessarily medial. To provide some intuition about the nature of the growth of a tree in the SGP algorithm, a few example generations are provided. In generation 0, the population is initialized with random programs, as depicted in Figure 2. The internal nodes of these randomly generated programs are chosen from the function set {AND, OR, NAND, NOR}. At generation 1, crossover and mutation are performed on the initial programs from generation 0. This is depicted in Figure 3.

Note that programs in this generation are composed via the IF function. In fact,

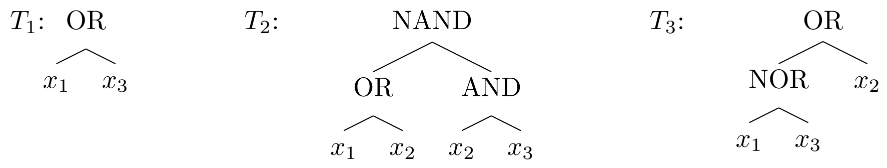


Fig. 2. Generation 0: The “primordial soup” from which to evolve and compose expression trees

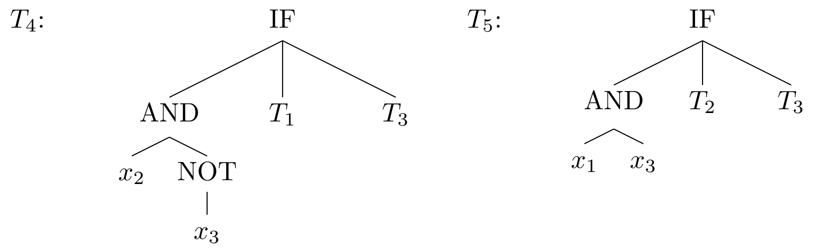


Fig. 3. Generation 1: Composition of programs from generation 0

this function is used to compose programs for all future generations. This function is important because it allows for semantically intermediate offspring. Depending on the first input of the IF node, it will either choose the output of the program at the second input or the third input. This means that the semantics at the output of the IF node are a mix of the outputs from the sub-nodes. Because programs are composed of programs from previous generations, offspring trees will always be larger than their parents.

The key result of the SGP paper [3] is that crossover and mutation operators are proven to be geometric and also are not reliant on a generate-and-test methodology (i.e. they search 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.

4. Approach (Improved Semantic-GP)

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 operations by converging to a solution more quickly. The general strategy will be to choose parents whose crossover is 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.

4.1. *Semantic Crossover*

In the SGP algorithm, the selection of parents for crossover is done using normal GP selection methods (e.g. tournament 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. Because the focus is on the Boolean domain, hypotheses exist in semantic Hamming space. However, for the purposes of visualization, Euclidean space is utilized to demonstrate geometric relationships between programs. 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 4. In this example, parents

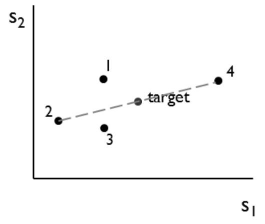


Fig. 4. Example of choosing parents which straddle the target in semantic space. In this case parents 2 and 4 are chosen over parents 1 and 3, despite being further away from the target.

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 is referred to as *divergence from geometricity* [18], and is calculated using the triangle inequality:

$$d_G(t, p_1, p_2) = ||t, p_1|| + ||t, p_2|| - ||p_1, p_2|| \tag{2}$$

Here, $||a, b||$ represents the Hamming distance between programs a and b. If the target semantics lie on the line segment between two parents then d_G is 0, so 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 is chosen using tournament selection.

Prior research by Krawiec shows that a medial geometric crossover has optimal *expected* fitness [18]. However, given two parents and a known target, the medial point may not be optimal. This is illustrated in Figure 5. 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. One of the primary differences between SGP and SGP+ is the location of the crossover offspring in semantic space. Figure 6 illustrates this difference. Both algorithms utilize geometric crossover, but the offspring produced by SGP+ is closer to the target.

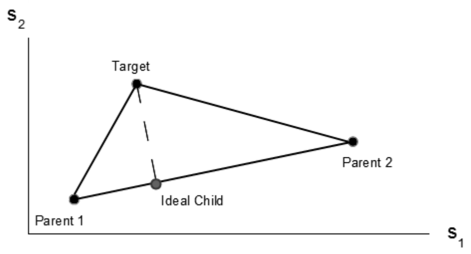


Fig. 5. Example of an ideal geometric crossover in 2D Euclidean semantic space

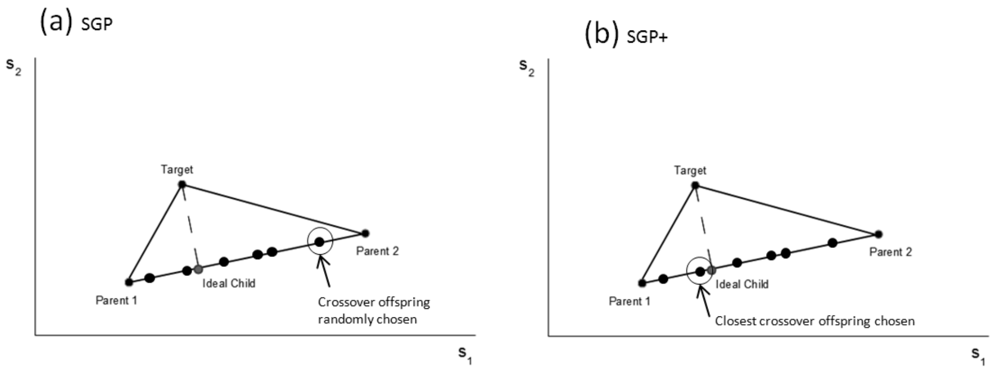


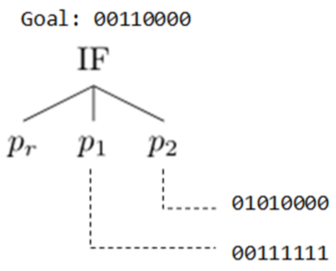
Fig. 6. Each black dot represents a potential offspring program. (a) SGP crossover is geometric (on the line segment between parents), but the position along the line is randomly chosen. (b) SGP+ chooses the offspring that is closest to the target.

However, we can't just "choose" the ideal child. Recall that crossover is performed via the IF operator, where the first parent (p_1) represents the true branch and the second parent (p_2) represents the false branch. The conditional part (p_r) determines the geometric location of the child between the parents (a.k.a crossover mask). In the extreme cases, the offspring are identical to one of the parents, which can occur if the semantics of p_r are $(0 \dots 0)$ or $(1 \dots 1)$. Therefore, the problem becomes finding an appropriate conditional input such that the child produced is closest to the ideal child. In SGP, this conditional input is chosen as a random program, but in SGP+, it is chosen from an *archive* of previously seen programs with known semantics.

4.1.1. *Random Program Archive*

A *random program archive* (RPA) is maintained which contains a history of programs observed during evolution. The fitness of each of these programs has been previously evaluated, so the semantics are fully known. This is useful because these known programs can be used as inputs to the crossover and mutation operators so that the semantics of the produced child are more desirable than randomly produced children.

As an example, suppose crossover is to be performed between two parents and the ideal child has semantics of (00110000), as depicted in Figure 7.



p_r	out	$\|\|out, Goal\|\|$
00110011	01110011	3
01011100	00011100	3
01010101	00010101	3
11110000	00110000	0

Fig. 7. Selecting a p_r from the RPA. If the last p_r is chosen, then the Hamming distance to the ideal child is 0.

For this example, there are only four known programs in the RPA. The output column represents the output of the IF node if that p_r were chosen as the conditional input. When a p_r bit is 1, the corresponding bit in p_1 appears at the output. Similarly, if p_r is 0, then the corresponding bit in p_2 appears at the output. The p_r that minimizes the Hamming distance to the ideal child is chosen.

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 offspring can be produced. In an effort to increase the number of p_r crossover masks, an archiving step is added to the main generational loop. A random program archive is maintained that will initially contain many small random programs. At each generation, a randomly chosen subset of the population is added to the archive. This p_r archive is similar to the archive described by Lehman [4] in that the archive contains a history of programs observed throughout *all* generations. For two given parents, the RPA is searched for a particular ideal crossover mask, and the program p_r that minimizes distance $||IF(p_r, p_1, p_2), Goal||$ is chosen.

There are two main side-effects to using a random program archive. The first is that the archive increases linearly in size with each generation, resulting in longer p_r search times for each crossover. The second side-effect is that the p_r programs inserted into the overall expression 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 does not need to be re-evaluated each time it is encountered. When an expression tree is evaluated, the output of sub-trees is saved, so each p_r will only be evaluated once.

4.2. *Semantic 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+ identifies the first semantic bit difference between the program to mutate and the target and sets that bit to match. This is equivalent to taking a step in a single dimension in semantic space towards the target. Figure 8 illustrates the conceptual difference between mutation in SGP and SGP+ in 2-D semantic Euclidean space.

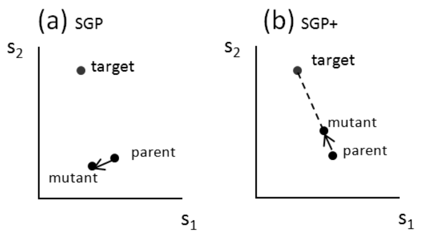


Fig. 8. (a) Mutation in SGP takes a single step in a random direction. (b) SGP+ mutation takes a step in the direction of the target semantics.

As an example, suppose the target semantics are (01010000) and the program (01010111) is randomly chosen to be mutated. The first bit that differs is the sixth bit, and we'd like to be able to clear that bit in the mutant in order to match the target. To isolate that bit, we need a program with semantics (00000100). Instead of searching the RPA for this program, we construct it as a Boolean conjunction of inputs. In this case, the program (AND (AND x_1 (NOT x_2)) x_3) has semantics (00000100) and is used to clear the bit in the mutant as depicted in Figure 9.

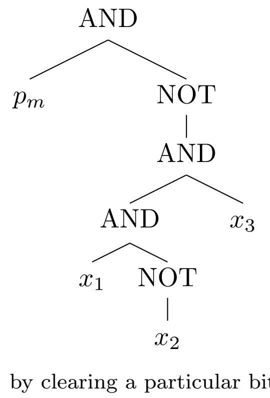


Fig. 9. Mutating p_m by clearing a particular bit to match the target.

Given that we can mutate individual bits to match corresponding bits in the target, the question arises: Why don't we 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 expression 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 expression tree has depth $O(k)$. If k is large, then not only is the tree overly complex, but the evaluation of the tree takes 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).

4.3. *SGP+ Algorithm*

The structure of the SGP+ algorithm is nearly identical to SGP as described by Moraglio et al. [3]. There are three primary differences: a new Semantic-Crossover operator, a new Semantic-Mutation operator, and the addition of a random program archive (RPA). The revised algorithm is detailed in Algorithm 4.1. The random

Algorithm 4.1 Improved Semantic Genetic Boolean Programming algorithm

```
Param: Train - A set of input-output pairs $(\boldsymbol{x_i}, y_i)$
Param: popSize - The size of the population
Param: maxGens - The maximum number of generations to evolve
Param: mutRate - Mutation rate, range [0.0, 1.0]
Param: rpaRate - Rate to add programs to the RPA, range [0.0, 1.0]
Param: funcSet - The set of functions to use as internal nodes in initial population
Return: An expression tree that interpolates all input-output pairs in Train
 1: function SGP+(Train, popSize, maxGens, mutRate, rpaRate, funcSet)
 2:     Initialize P with popSize randomly generated expression trees
 3:     Initialize RPA with random minterm programs
 4:     perfectFitness ← size(Train)
 5:     gen ← 0
 6:     while gen < maxGens and best.fitness < perfectFitness do
 7:         gen ← gen +1
 8:         nextP ← {}
 9:         $\forall p \in P$, evaluate fitness w.r.t. Train
10:         best ← $\operatorname{argmax}_{p \in P}$ {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
```

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. 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 details of the crossover and mutation sub-procedures are ommitted due to space limitations, but the idea is to choose offspring that are closest to the target in semantic space by utilizing the RPA as discussed in sections 4.1 and 4.2.

5. Evaluation and Results

There are two main criteria for evaluation of algorithms: program size, which is the number of internal nodes in tree, and accuracy. Accuracy is defined as the percentage of correctly classified instances in the training set:

$$Acc(h', T) = \frac{|\{h'(\boldsymbol{x_i}) = y_i \mid (\boldsymbol{x_i}, y_i) \in T\}|}{|T|} \tag{3}$$

where h' is the model returned by the algorithm and T is the training set.

The parameters for the genetic-programming based algorithms are described in Table 1. The datasets used are Boolean functions with varying number of inputs.

Table 1. Parameters for GP-based algorithms

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

Two functions (PARITY and MUX) are deceptive in nature, which highlights the difference between standard and semantic GP. Non-deceptive problems are also considered, in order to observe how well the algorithms deal with "easy" problems. The PARITY* problems compute odd parity, the MUX6 problem computes the multiplexer function with 4 input bits and two select bits, the OR* problems simply compute the OR function, the COMP* problems compute the comparator function, and the RANDOM* problems are completely randomized functions, which may or may not be deceptive. For instance, RANDOM5 randomly chooses one of the possible 2^{32} 5-input functions.

5.1. *Results*

First, metrics during evolution are observed during the training/learning process on the PARITY5 problem.

A comparison of program accuracy and program size per generation is provided in Figure 10. The SGP+ data cuts off at generation 7 because the problem was solved

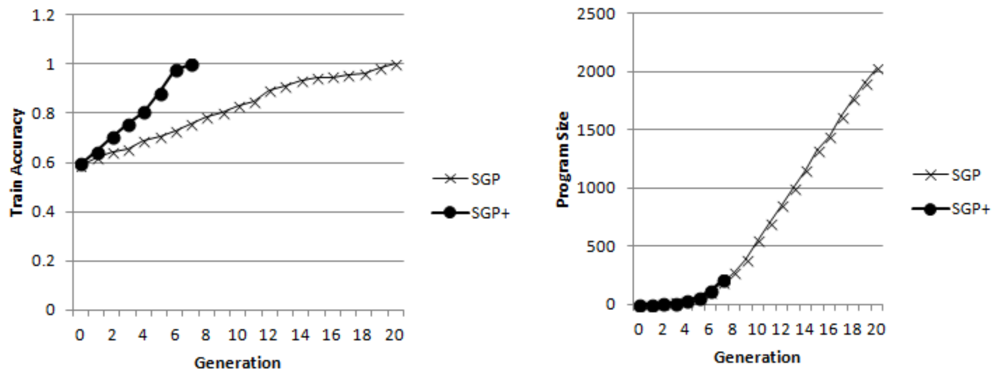


Fig. 10. (Left) Program accuracy vs. Generation. (Right) Program size vs. Generation. Comparison on the PARITY5 problem.

at or before generation 7 in all 10 runs. 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.

Figure 10 also 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 is much smaller than the SGP program due to achieving perfect training set accuracy in an earlier generation.

Accuracy A comparison of program accuracy over all Boolean problems is provided in Table 2. An additional column is included for the standard GP algorithm. Results in this column were originally reported by Moraglio [3]. Fields marked with a dash were not reported and are omitted. The final column is a two-sample T-test to determine whether SGP+ is significantly more accurate than SGP at the 95% confidence level. P-values below 0.05 (highlighted) indicate SGP+ is significantly more accurate. 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. 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 performs well for simple non-deceptive problems and low-arity deceptive problems. However, the larger PARITY7 and PARITY8 prove difficult for SGP, and the evolution is cut off at the maximum of 50 generations before

Table 2. A comparison of program accuracy for each algorithm. The GP and SGP columns contain results originally reported by Moraglio et al.

Problem	GP		SGP		SGP+		SGP+	T-test
	mean	std	mean	std	mean	std	Improvement (%)	p
PARITY5	0.529	0.024	**1.000**	0.000	**1.000**	0.000	0.0%	1.0000
PARITY6	0.505	0.007	0.998	0.005	**1.000**	0.000	0.2%	0.9342
PARITY7	0.501	0.002	0.888	0.013	**1.000**	0.000	**12.6%**	**0.0163**
PARITY8	0.501	0.002	0.748	0.012	**0.997**	0.010	**33.3%**	**0.0007**
MUX6	0.708	0.033	**1.000**	0.000	**1.000**	0.000	0.0%	1.0000
OR5	-	-	**1.000**	0.000	**1.000**	0.000	0.0%	1.0000
OR6	-	-	**1.000**	0.000	**1.000**	0.000	0.0%	1.0000
OR7	-	-	**1.000**	0.000	**1.000**	0.000	0.0%	1.0000
OR8	-	-	0.999	0.002	**1.000**	0.000	0.1%	0.9480
COMP6	0.802	0.038	**1.000**	0.000	**1.000**	0.000	0.0%	1.0000
COMP8	0.803	0.028	0.962	0.014	**1.000**	0.000	4.0%	0.3605
RAND5	0.822	0.066	**1.000**	0.000	**1.000**	0.000	0.0%	1.0000
RAND6	0.836	0.066	**1.000**	0.000	**1.000**	0.000	0.0%	1.0000
RAND7	0.851	0.053	0.930	0.018	**1.000**	0.000	7.5%	0.1520
RAND8	0.896	0.053	0.832	0.012	**1.000**	0.000	**20.2%**	**0.0013**

a solution is found. The accuracy of SGP+ is nearly perfect in all cases (a solution is found in 10 out of 10 runs), with the exception of one run of the PARITY8 problem on SGP+. Most of the differences in training set accuracy are insignificant, due to each of the algorithms solving the problem in all 10 out of 10 runs with perfect accuracy. However, SGP+ is significantly more accurate on high-arity deceptive problems (PARITY7, PARITY8, RAND8). The "SGP+ Improvement" column shows that the average improvement is 22.03% for these types of problems (simple average of the three highlighted numbers).

The accuracy improvement for high-arity deceptive problems can be attributed to the fact that SGP+ performs a more direct search in semantic space (i.e. less randomness), so that convergence can occur in earlier generations. The movement within the search space is greedy in the sense that all offspring are created so that they have the highest chance of minimizing the distance to the target. Typically, less randomness in a GP algorithm is undesirable as it can lead to overcrowding due to lack of diversity, but with SGP+ the diversity is maintained in the random program archive. The RPA maintains diversity because the programs it contains are chosen without bias to the target. As the problem size gets larger, the greedy search performed by SGP+ allows it to converge on the target with higher probability than SGP, because it effectively ignores large portions of the semantic search space that do not look promising.

Program Size Next, a comparison of program size is provided in Table 3. The T-test column has p values of a T-test between SGP and SGP+. Values of p below 0.05 (highlighted) indicate that SGP+ generated a significantly smaller program. In all cases, SGP+ produces a significantly smaller program than SGP.

The CNF/DNF column represents the number of internal nodes that are needed to represent the conjunctive normal form (CNF) or disjunctive normal form (DNF) - whichever is smaller - using only the allowed functions, namely {AND, OR, NAND, NOR}. As an example, the DNF representation of OR5 is (A OR B OR C OR

Table 3. Comparison of program size for each algorithm

Problem	SGP		SGP+		T-test	CNF/DNF
	mean	std	mean	std	p	
PARITY5	2717.3	355.3	386.0	70.9	**8.66E-20**	79
PARITY6	6045.1	463.8	1580.8	93.1	**8.34E-22**	191
PARITY7	8087.9	193.9	3293.9	207.9	**1.01E-22**	447
PARITY8	9488.1	182.9	5723.0	297.5	**1.99E-21**	1023
MUX6	4072.3	494.6	386.7	89.5	**5.80E-21**	191
OR5	184.3	150.9	24.2	0.8	**2.38E-11**	4
OR6	341.9	257.8	24.2	0.6	**5.56E-13**	5
OR7	1524.7	1337.4	25.2	1.7	**7.92E-16**	6
OR8	3583.6	2586.8	26.0	1.8	**6.47E-18**	7
COMP6	3581.7	370.7	386.4	105.4	**8.35E-21**	191
COMP8	9281.0	298.9	2126.9	269.4	**1.31E-23**	1023
RAND5	1597.7	436.4	100.3	40.1	**7.69E-18**	79
RAND6	5609.0	804.1	1001.3	145.9	**6.94E-21**	191
RAND7	8031.5	249.9	2201.7	123.5	**1.25E-23**	447
RAND8	9504.2	200.9	4892.0	333.5	**5.16E-22**	1023

D OR E), which has equivalent S-expression (OR (OR (OR (OR A B) C) D) E), which requires 4 internal nodes. To calculate the size of the CNF/DNF programs, if the truth table contains half ones and half zeros (which we can expect for the PARITY, MUX, COMP, and RAND problems), 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. This column represents a desirable program size to achieve in the context of genetic programming.

As expected, SGP+ generates programs larger than those represented by CNF/DNF for two primary reasons - node selection within the expression tree is partially randomized, and nodes are always added to, and never removed from, the tree. Hence, there is room for improvement. However, one benefit of SGP+ is that it can be adapted to other problem domains, whereas CNF/DNF is strictly for Boolean problems. For example, to adapt to the regression domain, the internal nodes of the random programs can be replaced with arithmetic operators {+, -, *, /}. In the SGP paper [3], it is proven that geometric crossover is possible in the regression domain if the offspring program T_3 is constructed from parent programs T_1 and T_2 and random program T_R as $T_3 = (T_1 * T_R) + ((1-T_R) * T_2)$. This is very similar to the IF construction in the Boolean domain in that T_R is determining the location of the offspring program somewhere between the two parents. Similarly, mutation can be accomplished with $T_3 = T_1 + m_s * (T_{R1} - T_{R2})$, where T_{R1} and T_{R2} are random real functions and m_s is the step size. To adapt these crossover and mutation operators to SGP+ an RPA can be created, similar to the Boolean domain, which tracks a history of program semantics that can be utilized to perform a more ideal crossover and mutation.

6. Conclusions and Future Work

A new algorithm, SGP+, was proposed that directly searches the semantic space and is an improvement over the existing SGP algorithm, creating programs that are significantly smaller in size over all tested problems. Additionally, classification accuracy has a significant 22.03% improvement for high-arity deceptive Boolean problems, such as the 8-input odd parity problem.

Future work includes extension to the regression domain and implementing steps to reduce overfitting and model complexity. One potential solution to overfitting is to separate the training data into training and validation sets. Another option is to grow the tree top down. SGP+ is created bottom-up in that the initial population of programs resides at the leaf-level and later generations are composed of programs from the next-lowest level. If this were reversed, then the first generation of programs would be at the root of the tree. This might be useful because the majority of overfitting occurs in later generations at the leaf level, which means tree pruning techniques could be applied.

In conclusion, semantic genetic programming is good for solving certain types of deceptive problems, but further improvements would be necessary to make it a strong learning algorithm in general.

References

1. J. R. Koza, *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems.* Stanford University, Department of Computer Science, 1990.
2. M. O'Neill, L. Vanneschi, S. Gustafson, and W. Banzhaf, "Open issues in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, pp. 339–363, Sept. 2010.
3. A. Moraglio, K. Krawiec, and C. Johnson, "Geometric semantic genetic programming," in *Parallel Problem Solving from Nature - PPSN XII*, vol. 7491 of *Lecture Notes in Computer Science*, pp. 21–31, Springer Berlin / Heidelberg, 2012.
4. J. Lehman and K. O. Stanley, "Efficiently evolving programs through the search for novelty," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, (New York, NY, USA), pp. 837–844, ACM, 2010.
5. G. Cuccu and F. Gomez, "When novelty is not enough," in *Applications of Evolutionary Computation*, vol. 6624 of *Lecture Notes in Computer Science*, pp. 234–243, Springer Berlin / Heidelberg, 2011.
6. J.-B. Mouret and S. Doncieux, "Using behavioral exploration objectives to solve deceptive problems in neuro-evolution," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 627–634, ACM, 2009.
7. D. Jackson, "Promoting phenotypic diversity in genetic programming," in *Parallel Problem Solving from Nature, PPSN XI*, vol. 6239 of *Lecture Notes in Computer Science*, pp. 472–481, Springer Berlin / Heidelberg, 2010.
8. N. McPhee, B. Ohs, and T. Hutchison, "Semantic building blocks in genetic programming," in *Genetic Programming*, vol. 4971 of *Lecture Notes in Computer Science*, pp. 134–145, Springer Berlin / Heidelberg, 2008.
9. D. N. Phong, N. Q. Uy, N. X. 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*, pp. 1 –6, june 2012.
10. N. Q. Uy, N. X. Hoai, M. O'Neill, and B. McKay, "Semantics based crossover for boolean problems," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, (New York, NY, USA), pp. 869–876, ACM, 2010.
11. N. Q. Uy, N. X. Hoai, M. ONeill, R. McKay, and E. Galván-López, "Semantically-

based crossover in genetic programming: application to real-valued symbolic regression," *Genetic Programming and Evolvable Machines*, vol. 12, no. 2, pp. 91–119, 2011.
12. K. 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, (New York, NY, USA), pp. 783–790, ACM, 2012.
13. K. 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, (New York, NY, USA), pp. 1379–1386, ACM, 2011.
14. K. Krawiec and B. Wieloch, "Functional modularity for genetic programming," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, (New York, NY, USA), pp. 995–1002, ACM, 2009.
15. K. Krawiec and B. Wieloch, "Analysis of semantic modularity for genetic programming," *Foundation Of Computing And Decision Sciences*, vol. 34, no. 4, p. 265, 2009.
16. K. Krawiec and B. Wieloch, "Automatic generation and exploitation of related problems in genetic programming," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1–8, IEEE, 2010.
17. A. Kattan, A. Agapitos, and R. Poli, "Unsupervised problem decomposition using genetic programming," in *Genetic Programming* (A. Esparcia-Alczar, A. Ekrt, S. Silva, S. Dignum, and A. Uyar, eds.), vol. 6021 of *Lecture Notes in Computer Science*, pp. 122–133, Springer Berlin Heidelberg, 2010.
18. K. Krawiec, "Medial crossovers for genetic programming," in *Genetic Programming*, vol. 7244 of *Lecture Notes in Computer Science*, pp. 61–72, Springer Berlin / Heidelberg, 2012.
19. K. Krawiec and T. Pawlak, "Locally geometric semantic crossover," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pp. 1487–1488, ACM, 2012.
20. K. Krawiec and P. Lichocki, "Approximating geometric crossover in semantic space," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 987–994, ACM, 2009.