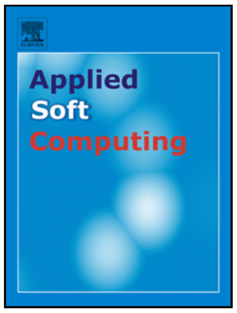


Title: Statistical Genetic Programming for Symbolic Regression

Author: Maryam Amir Haeri Mohammad Mehdi Ebadzadeh Gianluigi Folino

PII: S1568-4946(17)30393-9
DOI: http://dx.doi.org/doi:10.1016/j.asoc.2017.06.050
Reference: ASOC 4319

To appear in: *Applied Soft Computing*

Received date: 10-6-2016
Revised date: 29-4-2017
Accepted date: 25-6-2017

Statistical Genetic Programming for Symbolic Regression

Maryam Amir Haeri[a], Mohammad Mehdi Ebadzadeh[a], Gianluigi Folino[b]

[a]*Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran*
[b]*ICAR-CNR, Rende, Italy*

Abstract

In this paper, a new Genetic Programming (GP) algorithm for symbolic regression problems is proposed. The algorithm, named Statistical Genetic Programming (SGP), uses statistical information—such as variance, mean and correlation coefficient—to improve GP. To this end, we define *well-structured trees* as a tree with the following property: Nodes which are closer to the root have a higher correlation with the target. It is shown experimentally that on average, the trees with structures closer to well-structured trees are smaller than other trees. SGP biases the search process to find solutions whose structures are closer to a well-structured tree. For this purpose, it extends the terminal set by some small well-structured subtrees, and starts the search process in a search space that is limited to *semi–well-structured trees* (i.e., trees with at least one well-structured subtree). Moreover, SGP incorporates new genetic operators, i.e., *correlation-based mutation* and *correlation-based crossover*, which use the correlation between outputs of each subtree and the targets, to improve the functionality. Furthermore, we suggest a *variance-based editing* operator which reduces the size of the trees. SGP uses the new operators to explore the search space in a way that it obtains more accurate and smaller solutions in less time.

SGP is tested on several symbolic regression benchmarks. The results show that it increases the evolution rate, the accuracy of the solutions, and the generalization ability, and decreases the rate of code growth.

Keywords: Genetic Programming, Symbolic Regression, Well-Structured Subtree, Semi–Well-Structured Tree, Well-Structuredness Measure, Correlation Coefficient.

1. Introduction

In recent years, the problem of improving genetic programming (GP) has attracted many researchers. GP has three clearly identified challenges, *code growth*, *huge search space*, and *problem difficulty*.

- **Code growth (bloat):** Uncontrollable growth of the average tree size, without noticeable improvement in fitness, is named *bloat*. This phenomenon has two drawbacks: firstly, the evolution of large programs is computationally expensive; secondly, increasing the complexity of programs may decrease the ability of generalization.

  Several theories have been proposed on the causes of the bloat, including the *removal bias theory* [68], *replication accuracy theory* [45], *nature of program search space theory* [37, 39], and *crossover bias theory*[17, 57].

- **Huge Search Space:** The GP search space is huge due to its programs being variable-length. There are many *functionally-equivalent* solutions (i.e., with the same fitness value) in the search space. In other words, in the GP space, there exist many trees with the same phenotype[1] [18]. Despite the existence of many "redundant" solutions (i.e., solutions with the same phenotype but different genotypes) in the GP search space, most of the

Email addresses: haeri@aut.ac.ir (Maryam Amir Haeri), ebadzadeh@aut.ac.ir (Mohammad Mehdi Ebadzadeh), folino@icar.cnr.it (Gianluigi Folino)

[1]The attributes of a program are either structural (encoding related) or functional (behavioral). The structural attributes, also called the *genotype*, refers to the internal code of a program. On the other hand, the functional attributes, also called the *phenotype*, refers to the observable behavior of a program [11]. Most of the time, the phenotype of a GP tree is defined as its fitness value (the fitness is defined over the whole training data instances).

solution trees are large. According to the *nature of program search space theory* [37, 39], there are more large solutions than the smaller ones. However, large solutions are more complicated (less smooth) and have poorer generalization. Thus, limiting the search space helps to find simpler (smaller) solutions, in less time.

- **Problem Difficulty:** One of the challenges in GP is its application to difficult problems. There are two important questions in this context. 1) Why some problem instances are hard for GP, and 2) how hard a particular problem for GP is.

  One way to indicate the hardness of a problem is by visualizing the relationship between genotypes and fitnesses. This is called the "fitness landscape metaphor", where each genotype is sketched as a mountain whose height represents its fitness. Similar genotypes are sketched closer together.

  While sketching the fitness landscape is an effective way to indicate the hardness of a problem, in most problems, it is impossible to sketch the fitness landscape. As an example, consider high-dimensional problems, where fitness landscape is of no use [53]. Therefore, several researchers [30, 70, 60, 73, 55] suggested some measures of problem hardness based on fitness landscape concepts such as *Fitness Distance Correlation* (FDC) [70] and *Negative Slope Coefficient* (NSC) [60]. FDC is based on the correlation coefficient of the fitness and the distance to the goal. The major drawback of this measure is that the genotype of the goal is required beforehand. NSC is based on the relationship between the fitness values of individuals and the fitness values of their neighbors. This measure is predictive and does not need the genotype of the global optimum.

  Daida *et al.* [15] in their investigations, found that the expressiveness of the common notions of the fitness landscape or intrinsic properties of a problem domain is not enough to make the problem GP-hard. They showed experimentally that the standard GP does not properly search all possible tree structures, or all regions of the search space. Hence, there are several structures which are unlikely to be generated by GP. They also showed that it is hard for GP to generate trees with "high depth and small size," or trees with "shallow depth and large size."

This paper suggests a new GP approach, called *Statistical Genetic Programming* (SGP), which tries to overcome the aforementioned problems, especially the bloat phenomenon and huge search space by utilizing statistical information. A preliminary version of this approach was published in [4]. Using extra information of the population can help GP to have a better search process and find better solutions.

The basic idea evolves over the notion of well-structured trees, which satisfies the property that nodes closer to the root are more correlated to the target. Our experimental results show that, on average, trees whose structures are closer to well-structured ones are smaller than others. Based on this observation, SGP tries to find solutions which are closer to well-structured trees, in order to find smaller and more accurate solutions.

To search the space and generate new trees, two new genetic operators are defined: correlation-based crossover and correlation-based mutation. In contrast to the standard GP crossover, the correlation-based crossover does not choose the crossover points uniformly at random. It tries to find better subtrees to exchange. Therefore, it has more chance to generate offsprings better than the parents. Correlation-based mutation is based on the same idea, and does not choose the mutant subtree uniformly at random. Rather, it is more likely that an ineffective subtree is exchanged by a new "good" subtree.

Finally, SGP defines a simple editing operator for decreasing the rate of code growth. It replaces each invariant subtree with a number, effectively reducing the size of the generated trees.

Experimental results show that SGP can reduce the code growth, and conduct the search process to find smaller and more accurate solutions. The results show that in the case of difficult problems, it significantly outperforms the standard GP. Furthermore, we compare SGP with several "semantic-based" GP systems, and show that it is better than or comparable to these systems (depending on the problem and criteria—see Section 4.7).

The rest of this paper is organized as follows: Section 2 briefly discusses the previous works that tried to overcome the GP issues. Section 3 describes the proposed GP algorithm (SGP) in detail. In Section 4, the experimental results are presented. Section 5 is devoted to conclusion. Finally, Section 6 proposes possible ways to extend it in the future.

2. Related Works

Section 1 introduced the problems associated with GP. This section reviews several works, which try to alleviate or resolve some of those problems. Moreover, our method can be categorized as a semantic GP, several of semantic GP methods are reviewed here.

2.1. Code Growth and Bloat Control Methods

There are several approaches to bloat control, some of which are listed below.

1. **Code Editing:** Koza [34] presented a new operator called the *editing operator*, which is performed periodically on the trees and removes the subtrees not affecting the final solution. Zhang *et al.* [77] introduced a method for online simplification of GP programs during the evolution. They used multiple rules to simplify GP programs, as in algebraic expression simplification. The program trees are traversed in a bottom-up fashion by a post-fix order and the simplification rules are applied. Naoki *et al.* [47] suggested another algebraic simplification method for symbolic regression problems, called equivalent decision simplification. In their approach, a suitable set of simple trees (S_{Simple}) is determined; all subtrees of a target tree are checked for being equivalent to a tree of S_{Simple}. If any subtree is equivalent to a tree of S_{Simple} and is larger than it, it is replaced by the simple tree.
2. **Pseudo Hillclimbing:** Langdon and Poli [38] proposed a method that controls the size of trees when a new individual is created. If an offspring has better fitness or is smaller than its parents, it remains in the population; otherwise, it is removed and the crossover operator is performed again. Pseudo hillclimbing can slow the growth rate, but it leads to the increase of the running time of the GP algorithm.
3. **Parsimony Pressure:** In this technique, which is suggested by Luke and Panait [41, 40] a penalty term is added to the fitness function. This term penalizes large trees. Generally, the penalty term compromises between the fitness and growth. The most reported problems of this method are the difficulty in choosing a proper weight for the penalty term, and misleading the main search problem.
Poli and McPhee [59] overcame this problem by deriving parsimony coefficient based on the size evolution equation [58]. In this method, called covariant parsimony pressure, the parsimony coefficient is $c = \frac{\text{Cov}(l,f)}{\text{Var}(l)}$. The coefficient is recalculated in every generation, and it guarantees that the expectation of the program size in the next generation is identical to the current generation. Moreover, they demonstrated a way to generate c such that the average program size obeys any given function of time.
4. **Size or Depth Limits:** Some variations of GP [32] use size or depth limits to control the size of individuals. Although this strategy can control the bloat, if the discovered solution is larger than the limits, it cannot find the solution. To overcome this problem, some methods [63, 64] utilize *dynamic* limits instead of *static* ones. Dynamic limits are generally chosen in each generation and are derived from the function of the best individual size.
5. **Prune and Plant:** This method, proposed by Alfaro-Cid *et al.* [1], selects a tree and replaces one of its subtrees by a terminal node. Then the subtree is planted in the population as a new individual. Therefore, the average size of the population reduces.
6. **Tarpeian:** The tarpeian bloat control method which has been suggested by [43] is based on the size evolution equation [58]. This method tries to decrease the selection pressure of the large individuals. Hence, the fitness of individuals which are larger than the average size of the population are set to the minimum fitness by a probability p_t. Tarpeian bloat control decreases the probability of selection of larger solutions, and so it can control the rate of code growth.
7. **Multi-objective GP:** Another method for controlling the bloat phenomenon is using multi-objective GP. Ekart and Nemeth [21] suggested using a variant of Pareto tournament selection in which an individual is selected in the case which it is not dominated by a set of randomly chosen individuals. If the condition is not satisfied, another individual is selected from the population.
Occasionally, it is possible that small individuals take over the population in the early generations. To circumvent this, one can modify the Pareto criterion by allowing non-dominated solutions which are somewhat larger, yet their fitness is not worse. Moreover, Bleuler *et al.* [9] utilized multi-objective evolutionary technique, SPEA2, to prevent bloat. Two other researches using a variation of multi-objective GP to increase the code growth include [16, 8].

2.2. Problem Difficulty

Daida *et al.* [14] demonstrated that usually the solutions to harder problem instances discovered by GP are larger than those of the easier problem instances. In addition, GP needs more time to solve difficult problems. Therefore, they tried to answer the question "What makes a problem GP-hard?". In [15], they hypothesized that the structural mechanisms are the causes of problem difficulty. They showed that the GP evolutionary search which utilizes the tree-representation is unable to explore all tree shapes effectively. More clearly, it is quite difficult for tree-based GP to find solutions which need a tree representation with a full or very narrow tree shape. Thus, they could explain why in the tree-based GP, the problem difficulty cannot be considered only from the fitness landscape viewpoint.

According to their experiments, it is hard for GP to build some structures. They divided the search space of the GP tree-structure into four regions, presented in order of increasing difficulty: (1) easy, (2) transitional, (3) hard, and (4) out-of-bounds. The trees in the first region are easily attainable by GP with binary function set, while the trees in the fourth region cannot be attained by any binary tree, either because the trees are very narrow (trees with high depth and few nodes), or very full (trees with low depth and many nodes). If the solution of one problem is in the third region (called the "hard" region), GP cannot find the solution easily. For further information, please consult [15].

Sprogar *et al.* [69] stated that when GP is poor in finding some particular structures, the GP search should be guided to replace the irrelevant structures with critical ones. Thus, they proposed an approach to provide important structures for the GP progress. They applied an incremental learning to the evolution of GP trees, which converts a difficult problem into a set of easier ones.

As some problems may be hard for GP several researchers try to predict the performance of GP for problems.

Trujillo *et al.* [71] suggested a method for predicting problem difficulty of GP for classification problems. They extracted some statistical and complexity descriptors from the problem dataset, and estimated the performance of GP over this data. They utilized a standard linear regression and GP-based regression approaches to build the predictive function. Thus, they generate a function of descriptors that predict the performance of GP over the classification problems.

Graff and Poli [25, 26] estimated the Performance of *Evolutionary Program-induction Algorithms* (EPA) based on the problem distances, for symbolic regression problems. In their approach, they predict the performance of GP based on the similarity of the problem and a set of reference problem. First, a set of reference problems is selected. Then, by the performance of those reference problems, the performance of a GP system is approximated using a linear model. The linear model parameters are estimated using multivariate linear regression.

Moreover, they proposed an improvement of their approach [24]. They used some problem difficulty indicators for predicting the GP performance instead of using the difference between the problem and a set of reference problems. The new method does not need to identify a set of reference problem or to select a distance measure.

Gustafson *et al.* [28] studied the relationship between problem difficulty and code growth. They explained that, in difficult problems, there are fewer good solutions. This leads to high entropy, which increases selection pressure, resulting in low diversity (in terms of size, shape and contents). The chain of cause-and-effect just mentioned induces a high rate of bloat. Therefore, it is suggested that code growth can be controlled by keeping the selection pressure low.

2.3. GP Search Space

Some researchers have tried to analyze the search space of GP [18, 75, 56]. For example, Ebner [18] compared the search spaces of GP and GA. He reported that, because of the high redundancy in the GP search space, finding a solution is not as difficult as it seems. He also suggested that exploiting a proper representation technique—such that from any individual with a distinct phenotype, any individual with a random phenotype can be obtained within a limited number of steps—can help the GP process. Moreover, considering their experimental results, they suggested that the individuals with identical phenotypes should be scattered evenly in the GP search space.

Nguyen *et al.* [51] suggested a guideline for selecting the function set based on analyzing the fitness landscape. They used information content and the autocorrelation function for analyzing the fitness landscape of each function set. They showed experimentally that the performance of the function set and the autocorrelation function value are strongly related. Thus, they suggested using autocorrelation function for selecting the function set for a particular problem.

2.4. Semantic GP

Several approaches tried to enhance the search process of GP by using the concept of *semantics*. In the field of GP, the semantics of a program is defined as its behavior with respect to the input values of training data. In other words, the semantics of a GP program is the vector of its outputs for the fitness cases of the problem [50]. Most of semantic-based approaches modify genetic operators in order to maintain semantic diversity or improve searching the fitness landscape. Traditional GP operators ignore the meaning (semantic) of trees (programs) and explore the search space by manipulating their syntax. In contrast, semantic genetic operators use several pieces of information about the phenotype to improve the GP operators for creating better individuals. Several semantic-based GP algorithms are explained below.

Nguyen *et al.* [49] defined semantic equivalence and semantic similarity of two subtrees. They proposed two crossover operators based on these definitions. They called their crossover operators *semantic aware crossover* (SAC) and *semantic similarity-based crossover* (SSC). The main idea of these crossover operators is that the semantics of the two exchanged subtrees should be different, but not wildly different.

Moraglio *et al.* [46] introduced *Geometric Semantic Genetic Programming* (GSGP). They suggested exact geometric operators, which directly search the semantic space. For symbolic regression problems, an offspring of the geometric semantic crossover is the weighted average of the parents' semantics. The syntactic representation of this offspring is a tree constructed on top of the parent trees, in a way that the resulting program (function) is the weighted average of the parents' programs (functions). Since GSGP crossover offspring includes both parents as subtrees, the bloat phenomenon is escalated by this approach. Thus, they suggested using automatic GP simplification to reduce the code growth.

Castelli *et al.* [13] proposed a semantic GP system which builds, maintains, and updates the GP generations by utilizing semantic distribution. Their method biases the search process toward GP solutions which are semantically similar to the best ones achieved by now. Their approach leads to an increase in the average size of programs. Therefore, they used intron deletion to decrease the code growth. They showed that after intron deletion, the method can outperform bacterial and standard GP.

Krawiec *et al.* [35] suggested an approach to approximate geometric crossover, where several children are generated by each crossover, and the offspring having the highest semantic similarity with the parents goes to the next generation. Later, they improved their idea and suggested crossover operators with semantic backpropagation [36]. They called their new operator Approximately Geometric Semantic Crossover (AGX). The main purpose of AGX is to reduce the code growth of the geometric semantic crossover. In AGX, the subtrees of the parents are replaced with those leading to the construction of offerings being semantically intermediate to the parents. To this end, the semantic midpoint of the parents is computed first. Then, the crossover point of each parent is chosen. Next, the backpropagation procedure is performed to calculate the semantics of a subtree ST, such that if the parent subtree is replaced with ST, the semantics of the offsprings are approximately equal to the midpoint. This approach utilizes a library of subtrees with different semantics, and finds a subtree in the library as close to the semantics of ST as possible, and substitutes this subtree with the subtrees chosen in the parents.

Pawlak *et al.* [54] introduced another semantic crossover, called Random Desired Operator (RDO). This operator uses the backpropagation idea of AGX. In RDO operator, first a parent is selected by a selection method, and then a random subtree ST is picked. Then the backpropagation is performed to find the semantics of the subtree which, if replaced with ST, produces an offspring whose semantics match the target outputs (target semantic). These two operators can outperform canonical GP and geometric semantic crossover. Although these approaches(AGX and RDO) can control the bloat effectively, the main issue with them is the computational complexity, as the backpropagation procedure is very time consuming.

In this paper, we utilize the statistical information of the outputs of each subtree to improve the GP. As the output vector of each subtree represents its semantics, our approach is somehow similar to the semantic methods. In Section 4, we compare our approach with two recent semantic GP systems.

3. Statistical Genetic Programming

In this section, the main formulation of the SGP algorithm is defined. First, in Section 3.1 the statistical information of GP trees used in SGP is introduced. Afterwards, in Section 3.2, a measure of the contribution of each subtree to the solution is discussed. In Section 3.3, we define the notion of well-structured tree and semi–well-structured tree and demonstrate a salient property of well-structured trees. Then, we describe the technique of using semi–well-structured subtrees in generating the initial population and limiting the search space of GP. Sections 3.4 and 3.5 introduce the correlation-based crossover and mutation, while Section 3.6 describes the variance-based editing strategy. Finally, Section 3.7 is devoted to the computational overheads of SGP and to the reusability of statistical information of GP trees.

3.1. Statistical Information of GP Trees

GP is one of the algorithms used to solve the symbolic regression problem. However, GP suffers from issues such as bloat. We put forward the notion of Statistical GP (SGP), which tries to alleviate the issue by guiding the GP process using several sample statistics. A *sample statistic* is a measure characterizing a sample, which is often used to estimate the real value of the same measure for the entire population. There are several classes of sample statistics: sample mean and median, sample variance and standard deviation, sample quantiles, order statistics (e.g., minimum and maximum), sample moments (e.g., skewness and kurtosis). There also exist several statistics to measure the relation between two sets of samples, the most famous of which is the correlation coefficient.

SGP incorporates the simplest statistics, which can efficiently represent the behavior of the data. These consist of the sample mean, variance, and the correlation coefficient.

We use statistical information to guide the search process for *symbolic regression* problems. Consider a training data with n input variables $x_1, \ldots, x_n$. The training data has M instances (fitness cases), where the j^{th} instance is denoted by $X_j = (x_{j1}, \ldots, x_{jn})$. The target output for the j^{th} instance is $y_j = f(X_j)$, where f is the target function to be approximated.

Let $\mathcal{T}$ be a *terminal set* containing the input variables as well as several real numbers, and $\mathcal{F}$ be a *function set* containing several mathematical functions such as addition, multiplication, protected division, and trigonometric functions. Let T be a tree whose leaves are picked from $\mathcal{T}$, and whose internal nodes are chosen from $\mathcal{F}$. Assign a unique number to all nodes of T, and let g_i denote the function computed by the subtree of T whose root is the node i.

SGP computes the following values, for each node i of all trees in its population trees:

$$E[g_i] = \frac{1}{M} \sum_{j=1}^{M} g_i(X_j) \tag{1}$$

where $E[g_i]$ denotes the expectation (i.e., average) of the output of g_i over all sample instances $X_1, \ldots, X_M$.

$$E[g_i^2] = \frac{1}{M} \sum_{j=1}^{M} g_i^2(X_j) \tag{2}$$

where $E[g_i^2]$ denotes the expectation (i.e., average) of the squared output of g_i over all sample instances $X_1, \ldots, X_M$.

$$E[g_i \cdot f] = \frac{1}{M} \sum_{j=1}^{M} g_i(X_j) \cdot f(X_j) \tag{3}$$

where $E[g_i \cdot f]$ denotes the expectation (i.e., average) of the product of the output of g_i and the target output (f), over all sample instances $X_1, \ldots, X_M$.

The mean (m_i), sample variance (σ_i^2) and correlation coefficient (ρ_i) of each node with f can be computed as follows:

$$m_i = E[g_i] \tag{4}$$

$$\sigma_i^2 = \frac{M}{M-1} \left(E[g_i^2] - E[g_i]^2 \right) \tag{5}$$

$$\rho_i = \frac{E[g_i \cdot f] - E[g_i] \cdot E[f]}{\sigma_i \cdot \sigma_f} \tag{6}$$

Notice that in the above formula, $E[g_i]$, $E[g_i^2]$ and $E[g_i \cdot f]$ are computed based on equations (1) to (3). Furthermore, $E[f]$ (average of the target output over all sample instances), $E[f^2]$ (average of the squared target output over all sample instances), and σ_f^2 (variance of the target output) are determined based on the following formula:

$$E[f] = \frac{1}{M} \sum\nolimits_{j=1}^{M} f(X_j) \tag{7}$$

$$E[f^2] = \frac{1}{M} \sum\nolimits_{j=1}^{M} f^2(X_j) \tag{8}$$

$$\sigma_f^2 = \frac{M}{M-1} \left(E[f^2] - E[f]^2 \right) \tag{9}$$

In order to have unbiased statistics for computing σ_i^2 and σ_f^2, we used the *sample variance* (rather than the *empirical variance*).

Therefore, for each node of a GP tree the average and standard deviation of its outputs over the fitness cases and the correlation coefficient between its outputs and the target outputs are computed.

An example of a GP tree and its statistical information is shown in Figure 1. Suppose that we have a symbolic regression problem with regression data (fitness cases) as shown in Figure 1, and let the depicted tree represents an individual of a GP population. Each node of the tree implies a function; for instance, the function of node n_6 is $g_6 = 0.6 * x_1$. The "tree function" is the function implied by the root of the tree. Considering the regression data, one can compute statistical information—mean or variance—for each node of the tree (i.e. the function implied by it). For instance, for node n_4 ($g_4 = x_2 + x_1 * 6$) and for the given regression data, the mean of output of g_4 is equal to $E[g_4] = 0.68$. All relevant data are tabulated in Figure 1. Another potentially useful piece of statistical information is the correlation coefficient of the outputs of each node function with the desired output values (output values of regression function f). This measure can indicate the relation between the function of each node and the desired function, and shows how much a subtree is effective in modeling the desired function.

3.2. Contribution of Each Subtree in GP Trees

In solving symbolic regression problems with GP, a solution tree represents a function that models the target, and each of its subtrees represents a function that contributes to modeling the target. In a solution tree, by moving from the leaves toward the root, the solution gradually becomes completed. Higher subtrees contain the lower ones. If the tree does not contain subtrees with no contribution to modeling the target, higher subtrees will have more contribution to modeling the target than the lower ones.

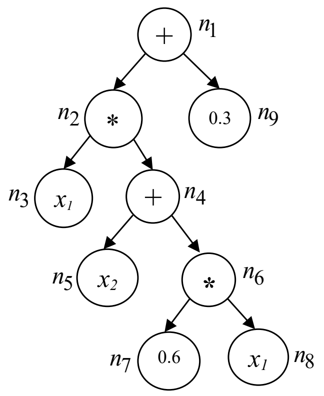
In GP trees, the contributions of different subtrees might be different. Some subtrees can have more contribution to modeling the target than the others. Moreover, some trees may have lots of subtrees with no contribution. A proper measure indicates how much a subtree contributes to generating the desired function. Such measure should increase by moving from the leaves to the root of a solution without any subtrees with no contribution to modeling the target. We call this measure the *contribution measure*.

Finding a proper contribution measure is rather difficult. The desired function is not available in a real problem, and just some training data are available. Hence, the measure should indicate how much a subtree contributes to modeling the target, based solely on the desired outputs.

The first idea for a contribution measure is based on the fitness, as it indicates how much a GP tree learns the behavior of the training data. However, the fitness may not be a good contribution measure: Consider the desired function $f(x_1, x_2) = x_1^2 + x_2 + 1000$, where $x_1, x_2 \in [-1, 1]$. A subtree with function $g(x_1, x_2) = x_1^2$ contributes toward constructing f, but its fitness is very low. In other words, although a subtree may have a good relationship with the target (or have a positive effect on modeling it), the fitness of that subtree may not be considerable.

We need a measure that indicates the dependency between the outputs of a subtree with the target outputs. If a subtree has a high dependency with the target outputs, it has a contribution to modeling the target. *Correlation coefficient* is a measure of linear dependency. In this paper, we investigate correlation coefficient as a contribution measure. The notion of correlation coefficient was previously used in several works in the GP domain. Burke *et al.* [11] employed correlation for evaluating diversity measures. Altenberg *et al.* [3] utilized the notion of correlation in the definition of building blocks. Some papers such as [10] and [62] used the correlation coefficient between the outputs of a tree and the targets as the fitness function. Smits *et al.* [67] suggested using correlation coefficient between the output and input variables as a criterion for feature selection in GP.

Regression Data		
x_1	x_2	$f(x_1,x_2)$
0.1	0.3	1.38
0.4	0.5	1.93
0.2	0.2	1.42
0.8	0.6	3.42
0	0.9	1.65



Node	Function
n_1	$g_1(x_1, x_2) = x_1(x_2 + 0.6x_1) + 0.3$
n_2	$g_2(x_1, x_2) = x_1(x_2 + 0.6x_1)$
n_3	$g_3(x_1, x_2) = x_1$
n_4	$g_4(x_1, x_2) = x_2 + 0.6x_1$
n_5	$g_5(x_1, x_2) = x_2$
n_6	$g_6(x_1, x_2) = 0.6x_1$
n_7	$g_7(x_1, x_2) = 0.6$
n_8	$g_8(x_1, x_2) = x_1$
n_9	$g_9(x_1, x_2) = 0.3$

x_1	x_2	$g_1(x_1,x_2)$	$g_2(x_1,x_2)$	$g_3(x_1,x_2)$	$g_4(x_1,x_2)$	$g_5(x_1,x_2)$	$g_6(x_1,x_2)$	$g_7(x_1,x_2)$	$g_8(x_1,x_2)$	$g_9(x_1,x_2)$
0.1	0.3	0.336	0.036	0.1	0.36	0.3	0.06	0.6	0.1	0.3
0.4	0.5	0.596	0.296	0.4	0.74	0.5	0.24	0.6	0.4	0.3
0.2	0.2	0.364	0.064	0.2	0.32	0.2	0.12	0.6	0.2	0.3
0.8	0.6	1.164	0.864	0.8	1.08	0.6	0.48	0.6	0.8	0.3
0	0.9	0.3	0	0	0.9	0.9	0	0.6	0	0.3

Node	Statistics		
	$m_i = E[g_i]$	$\sigma_i^2 = E[g_i^2] - E[g_i]^2$	$\rho_i = \dfrac{E[g_i.f] - E[g_i]E[f]}{\sigma_{g_i}\,\sigma_f}$
n_1	0.552	0.130496	0.980357
n_2	0.252	0.130496	0.980357
n_3	0.3	0.1	0.926148
n_4	0.68	0.111	0.793819
n_5	0.5	0.075	0.324068
n_6	0.18	0.036	0.926148
n_7	0.6	0	0
n_8	0.3	0.1	0.926148
n_9	0.3	0	0

Figure 1: An example of a GP tree and its statistical information.

It seems that the correlation coefficient between the outputs of the desired function (target outputs) and the outputs of each subtree can be considered as a proper contribution measure. Assume there is a high correlation between the outputs of a subtree and the outputs of the desired function. This means that by changes of inputs, both of the desired function and the function corresponding to the subtree change similarly. In other words, they have the same behavior. Thus, the subtree is not independent of the desired function: it contributes to modeling a part of the desired function, and it is useful for generating the solution.

If the solution completely fits to the training data, the correlation between the outputs of the function of the tree root with the desired outputs is 1. By moving from the leaves of the solution tree to its root, the solution function is gradually completed. It seems reasonable that in a solution not having subtrees with no contribution to modeling the target, by moving from the leaves to the root, the correlation coefficient of the function outputs of each node and the desired outputs (outputs of f) increases.

Therefore, in this paper, we choose the correlation coefficient as a contribution measure. In the rest of the paper,

Table 1: Test Problems

Benchmark Number	Benchmark Function	Function Formula	Domain	Number of Instances
1	$f_1(x_1, x_2, x_3, x_4, x_5)$	$\frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$	$x_i \in [0, 6]$	100 random points
2	$f_2(x_1, x_2)$	$\frac{(x_1-3)^4+(x_2-3)^3+(x_2-3)}{(x_2-2)^4+10}$	$x_1, x_2 \in [0, 6]$	50 random points
3	$f_3(x)$	$0.3x\sin(2\pi x)$	$x \in [-2, 2]$	40 random points
4		Toxicity problem	626 input variables	234
5		Biovalibility problem	241 input variables	359
6		Concrete problem	8 input variables	1030

the usefulness of this measure is evaluated experimentally. Thereafter, an improvement of GP is proposed which utilizes the correlation coefficient.

3.3. Well-Structured Tree and Search Space Restriction

After choosing correlation coefficient as a contribution measure, we defined the notion of *Well-Structured Tree* and *Semi–well-Structured Tree* as follows:

Definition 1 (Well-Structured Tree). *A tree is called* well-structured *if the absolute value of the correlation of its nodes with f (the desired function) is non-decreasing along all paths from the leaves to the root.*
A Well-Structured Subtree (WSS) *is defined likewise.*

Definition 2 (Semi–well-Structured Tree). *A tree is called* semi well-structured *if it has at least one well-structured subtree.*

To estimate how close a given tree is to a well-structured tree, the *Well-Structuredness Measure* is defined as follows:

Definition 3 (Well-Structuredness Measure (WSM)). *Well-structuredness measure of a tree T is computed by the following formula:*

$$WSM(T) = \frac{1}{L-1}\sum_{i=2}^{L}\left(\left|\rho(f, g_{\text{parent of } i})\right| - \left|\rho(f, g_i)\right|\right) . \tag{10}$$

Since WSM is in the range $[-1, +1]$*, we may normalize*[2] *WSM in the range* $[0, 1]$*, by adding 1 to it, and dividing the result by 2.*

Here, ρ is the correlation coefficient, and L is the size of the tree. Moreover, i denotes the index of the tree nodes. The index of the root is $i = 1$.

The WSM of a GP tree is the average of the differences between correlation coefficients of the outputs of each parent node and its child with the desired outputs, along the tree. According to this definition, if a tree is well-structured, all the terms composing the sum are non-negative and so have an increasing effect on the WSM of that tree. Furthermore, if the absolute value of ρ of a parent node with the desired outputs is less than the absolute value of ρ of its child with the desired outputs, then WSM is decreased. Therefore, this measure can estimate how close a tree is to a well-structured tree.

If a tree contains subtrees with less or no contribution to modeling the target, it needs some other subtrees to model the target. Such subtrees will increase the size of the solution tree. On the contrary, if a tree contains lots of subtrees with no or less contribution, its structure gets far from a well-structured tree. Intuitively, and on average, solutions whose structures are closer to a well-structured tree should be smaller. To evaluate this hypothesis, and verify whether the correlation coefficient is a proper measure of contribution, some experiments were conducted.

[2]Normalization is performed to make the WSM charts more appealing, and so as the visual comparison is simplified (c.f. Figure 2).

Table 2: GP Settings

Population Size	500
Function Set	$\{+, -, \times, \div\}$
Fitness Function	Negative of Mean Squared Error (-MSE)
Initial Population Method	Ramped-half-and-half
Selection Method	Tournament Selection
Tournament Size	5
Crossover Rate	90%
Mutation Rate	5%
Maximum Tree Depth in Initial Population	6
Maximum Generations	200
Number of Runs	50 independent runs for each test

The relation among fitness, size and well-structuredness is investigated by using six symbolic regression problems. The results were extracted from 50 independent runs for each problem. Information about the problems, and the settings of the GP parameters are available in Table 1 and Table 2 respectively. More information about the benchmarks is available in Section 4.1. To estimate how close the structure of a tree is to a well-structured tree, the WSM measure was used. The size, WSM and fitness of each tree are recorded. Using GP, various solutions were generated for each of the problems. The solutions are the best individuals of each generation of each run. In other words, intermediate solutions are also used in the experiment to have various types of solutions. Since, the number of GP runs is 50 and each run contains 200 generations, for each problem $10,000$ solutions were gathered. To extract meaningful results from this data, size and fitness are sorted and grouped into five bins. Trees which are in bin 1 of the fitness have the lowest values in fitness, and those are in the bin 5 have the highest values. Moreover, as for the size, trees which are in bin 1 are the smallest and those are in the bin 5 are the largest. Additionally, the structures of the trees which have greater WSM are closer to well-structured trees.

Figure 2 illustrates the bar charts of the tree WSM vs. tree fitness vs. tree size, for the six benchmarks. The results of this experiment lead to the following three conclusions:

1. Among the solutions of GP with similar fitness, those trees whose structures are closer to a well-structured tree are smaller. If the tree contains subtrees with little or no contribution to modeling the target, the WSM will increase. On the other hand, these types of subtrees increase the size of the tree, because they do not have the ability to model the target function, and thus the tree requires more subtrees to model the target. Hence, for trees with the same fitness, the one with less subtrees with little contribution has greater WSM, and is smaller.
   This valuable property can be used in GP to find smaller solutions. Typically, SGP search space is very huge and quite redundant. There are a lot of solutions with the same phenotype, but they have different structures (different genotypes). It seems that it is possible to conduct GP to find solutions which are well-structured or closer to a well-structured tree, in order to have smaller solutions.
2. Among the trees of the same size, those with greater fitness values have greater WSM. If we consider trees with the same size, the one with greater fitness has subtrees which better model the target. That is, it contains more subtrees with contribution toward the target, and by moving from its leaves to its root, the target function is approximated better. Thus, this tree is closer to a well-structured tree and has greater WSM.
   This indicates that the correlation coefficient between the outputs of two functions can be considered as a contribution measure.
3. Moreover, the WSM's of the best solutions (the solutions with great fitness and small size) are greater than the other trees. This shows it is possible to utilize the correlation coefficient of the desired outputs and the tree (subtree) outputs, to find optimal (based on both the size and the fitness) solutions. In our experiments, WSM has its greatest value when the size has its smallest value and fitness has the greatest value. In other words, based on the experimental results, the best solutions have the greatest WSM values. Hence, trees whose structures are closer to well-structured trees have better fitness and smaller size. If a solution tree has both proper fitness and proper size, it models the target function without many redundant and ineffective codes.

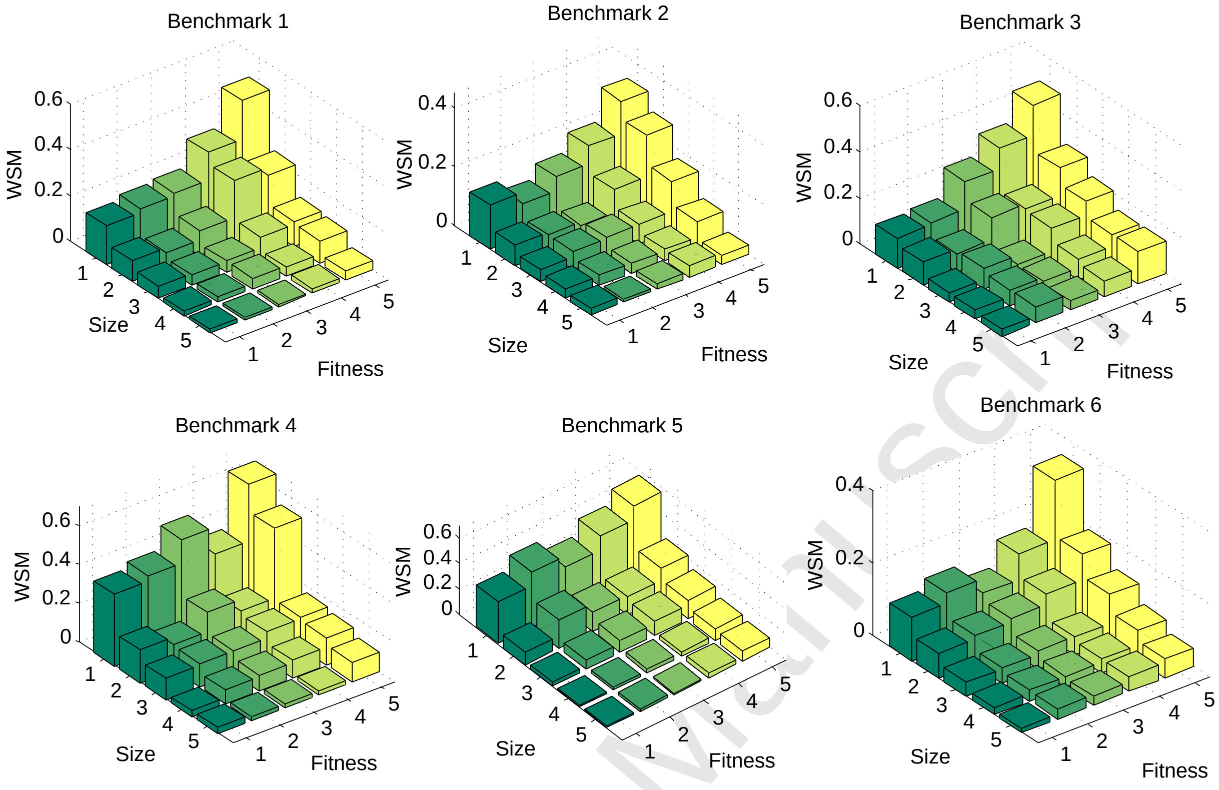


Figure 2: The effects of well-structuredness on the tree size. WSM is normalized in the range [0, 1].

Based on the above experiment, to find smaller solution trees, SGP uses the following approach: It limits the GP search space to semi–well-structured trees, and inclines the GP search process towards finding well-structured solutions (or solutions which are closer to them).

To limit the GP search space to semi–well-structured trees, SGP constructs a pool of WSSs and adds them to the terminal set. The WSSs are created in the manner described in the pseudo-code of Algorithm 1.

For generating initial population with semi–well-structured trees, SGP uses an *extended* terminal set[3] containing some random constants, the problem variables and WSSs: $\mathcal{T} = \{\{\text{Random Constants}\}, \{x_1, \ldots, x_n\}, \{\text{WSSs}\}\}$. The initial population is generated using the ramped-half-and-half technique. All trees in the initial population are constructed in this manner.

Based on the definition, a semi–well-structured tree is a tree which has at least one WSS. In order to limit the search space to semi–well-structured trees, we use the WSSs in the terminal set and check that each tree has at least one WSS in it. In other words, if in generating a tree, we utilize at least one WSS, the tree will become semi–well-structured. When we put the WSSs in the terminal set, we can ignore the semi-well-structuredness check: If the terminal set has enough WSSs and the size of each initial tree is considerable, the probability of generating a tree with no WSSs will be quite low. Moreover, in most cases, the number of WSSs is greater than the number of variables, and thus each initial tree contains many WSSs. Hence, it is highly likely that the SGP initial population trees contain a lot of WSSs.

Using WSSs to generate the initial population has two advantages. Firstly, it results in search space restriction, because the search process is limited to semi–well-structured trees. Secondly, it can be helpful to decrease the average

[3]The terminal set of the standard GP does not include WSSs. Here, we use the term *extended* terminal set to distinguish the SGP terminal set, which includes WSSs as well.

```
   input  : Problem variables, limit $l$
   output: Well-structured subtrees (WSSs)
1  Enter each variable $x_i$ to an empty queue called WSS-Queue as a subtree $ST_i$.
   while Max tree size in the WSS-Queue is less than the limit $l$ do
2      for each two subtrees $ST_i$ and $ST_j$ in the WSS-Queue do
3          construct subtrees $A, B, C, D$:
4          $A = ST_i + ST_j$
5          $B = ST_i \times ST_j$
6          $C = ST_i - ST_j$
7          $D = ST_i \div ST_j$
8      end
9      if the new constructed subtrees are Well-structured then
10         add them to the WSS-Queue
11     end
12 end
13 Remove variables $x_i$ from the WSS-Queue.
```

Algorithm 1: Pseudo-code of generating WSSs.

program size and find smaller solutions—because using WSSs to generate population leads to constructing trees with greater WSM, and for a distinct fitness, the results (see Figure 2) show that trees with greater WSM are smaller.

While the SGP initial population is limited to semi–well-structured trees, this limit is not a very hard one, and thus the initial search space of SGP is complete. By the complete space, we mean a search space that contains a solution. In other words, even if the SGP search space is limited to the semi–well-structured tree, it will certainly contain at least one solution (and most likely, much more than one solution). SGP merely changes the bias of the search to find trees which are closer to a well-structured tree (or finding smaller solutions, on average). In Section 4.5, we show that, this limitation does not decrease the diversity of the population. In the following a new crossover and a new mutation are introduced. These operators can search this limited space efficiently. The new operators help SGP to preserve the population diversity, and increase the rate of constructive operations.

3.4. Correlation-Based Crossover

The correlation can be utilized to improve the functionality of the genetic operators like crossover and mutation. Here, a new crossover operator called *Correlation-Based crossover* (CB crossover) is proposed. In the CB crossover, the nodes which are more correlated to f have a greater chance to be selected as the crossover point, as follows: The subtrees of each parent take part in a Q-tournament, and the winner subtree of one parent is swapped with the winner subtree of the other one.

"Q tournament" refers to a selection model in evolutionary computing. In a Q-tournament selection, Q individuals are chosen at random, and compete with each other in a tournament.

In CB crossover, we used the above approach as follows: Let T_1 and T_2 be two trees. For each tree, Q random nodes are selected, and the nodes compete in a tournament. The winner of this tournament is the node with the highest absolute correlation with f. The winner is selected as the crossover point. Let n_1 be the winner node of the Q tournament for T_1, and define n_2 similarly for T_2. The CB crossover operator substitutes the subtree of T_1 rooted at n_1 with the subtree of T_2 rooted at n_2.

The selection pressure can be tuned by changing the parameter Q. If Q is larger, nodes with lower absolute correlation have a smaller chance to be selected. On the other hand, when Q equals to one (just one node is selected for the tournament), a random node is selected, and the CB crossover becomes identical to the standard crossover.

As the size of the trees is varied, the parameter Q is considered proportional to the size of the tree. In Section 4.1, we experimentally determined the optimum value for Q.

The CB crossover operator does not select the swamping subtree from the parent blindly (the Q-tournament helps in increasing the probability of selecting a subtree with a good correlation with f). This leads to the relocation of good

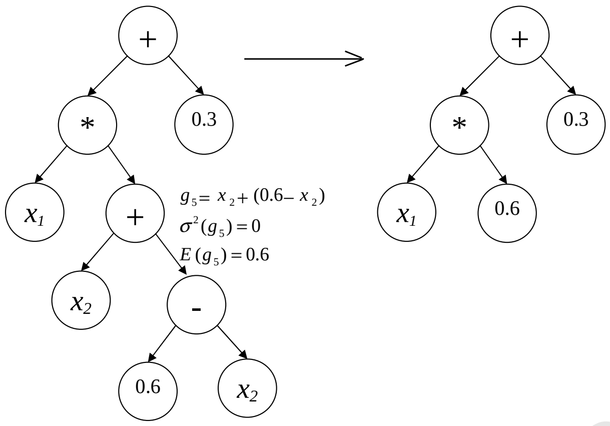


Figure 3: Variance-based editing operator

subtrees in the population. In Section 4, it will be shown that this operator is more effective than the standard subtree crossover, because the probability of selecting a crossover point in the ineffective part of the code is decreased. Thus, the chance of generating offsprings with greater fitness values than their parents is increased.

3.5. *Correlation-Based Mutation*

A new mutation operator named *Correlation-Based Mutation* (CB mutation) is defined in SGP approach. In CB mutation, the probability of choosing each node for mutation is inversely proportional to its absolute correlation. The subtree corresponding to the chosen node is replaced by a random subtree. The random subtree is constructed in the manner of generating a random tree for initial population from the extended terminal set. It helps to substitute ineffective and less-effective subtrees by a semi–well-structured subtree. In Section 4 it will be shown that this mutation operator is more efficient than the standard mutation.

3.6. *Variance-Based Editing*

Here, a new editing method called Variance-Based editing (VB editing) is introduced. VB editing eliminates introns and subtrees that only construct numeric constants. Indeed, one of the problems of the GP is *code bloat*, i.e. producing code which is slow, long, or wasteful of resources. More precisely, code bloat is a considerable increase in the average code size of the population with no significant change of the fitness. A method for bloat reduction is called *editing*. To edit the trees, variance and mean of each node are used. Every subtree whose variance of its root is zero is replaced with the mean of its root. Most of the subtrees of GP trees are introns or just for constructing a numeric constant. The variance of these subtrees is equal to zero. Thus, this editing operator can restrict the code growth significantly.

Figure 3 illustrates the operation of VB editing. A subtree with zero output variance—by considering the fitness cases (training data) as the inputs—means that it generates a constant value for all training data. This constant value is exactly equal to the mean of the outputs of that subtree. Thus, it is possible to replace the subtree with the mean of its output. For example, in Figure 3, The function of subtree $g_5 = x_2 + (0.6 - x_2)$ for different inputs generates 0.6. The mean of the output of this subtree over training data is 0.6. Thus, this subtree can be replaced by the constant node 0.6.

3.7. *Computational Overheads and Reusability*

The pseudo-code of SGP is presented in Algorithm 2. At first, SGP generates the initial population. To generate the initial population, SGP uses a terminal set containing WSSs, problem variables and random numbers. The WSSs are generated based on Algorithm 1. After generating the initial population, the statistical information of each node of each population tree should be computed. The fitness of each tree is also computed in this phase. Moreover, each

```
input  : Dataset, GP Parameters
output: The best individual
1  Generating WSSs;
2  Add WSSs to the terminal set $T$.
3  Generating initial random population
4  for each node of each GP individual do
5      Calculate mean, variance, and correlation coefficient with the desired
       outputs.
6  end
7  for each GP individual do
8      Compute the fitness of the individual.
9      Edit the individual by VB editing.
10 end
11 while # total generations < Max_Generation do
12     Operator ← SelectGeneticOperator(Crossover, Mutation, Reproduction)
13     if Operator = Crossover then
14         $P_1$ and $P_2$ ← Parent Selection(Population)
15         $c$ ← CB-Crossover ($P_1, P_2$)
16         Calculate statistical information of the new individual $c$.
17         Evaluate the fitness of the new individual.
18         Edit $c$ by the variance based editing.
19     end
20     if Operator = Mutation then
21         $P_1$ ← Parent Selection(Population)
22         $c$ ← CB-Mutation ($P_1$)
23         Calculate statistical information of the new individual $c$.
24         Evaluate the fitness of the new individual.
25         Edit $c$ by the variance based editing.
26     end
27     Survival selection
28 end
```

Algorithm 2: Pseudo-code of the SGP.

tree of the initial population is edited by the variance-based editing operator. Then in each generation some trees are selected for crossover and mutation. When the offsprings of these operators join to the population, the statistical information of these new individuals should be computed, and these new individuals should be edited by the VB editing operator.

Although SGP computes some additional information during the evolution, these computations do not considerably increase overheads, and they are not very time consuming. Most of the statistical information is reusable, and those in need of update can be computed simultaneously and in parallel while updating the fitnesses. In the initial step, the statistics of each tree in the initial population are calculated. In other generations, the statistics of each tree are needed to be recalculated only after crossover and mutation operators. In such cases, most of the statistics of each tree are reusable and only some of them should be recalculated.

For example, consider the trees of Figure 4; after the crossover, for each offspring, it is only needed to recalculate the statistics of the nodes, which are in the path from the crossover point to the root. In Figure 4 only the highlighted nodes need the recalculation of their statistics. The statistics of the other nodes are reusable. Also, after the application of the mutation operator, only the statistics of the new subtree and the nodes which are in the path from the mutation point to the root, need recalculation.

Furthermore, these calculations are needed when the recalculation of fitness is required. Computing the fitness and the statistical information of each tree, require the same tree traversing. Thus, the statistical information can be

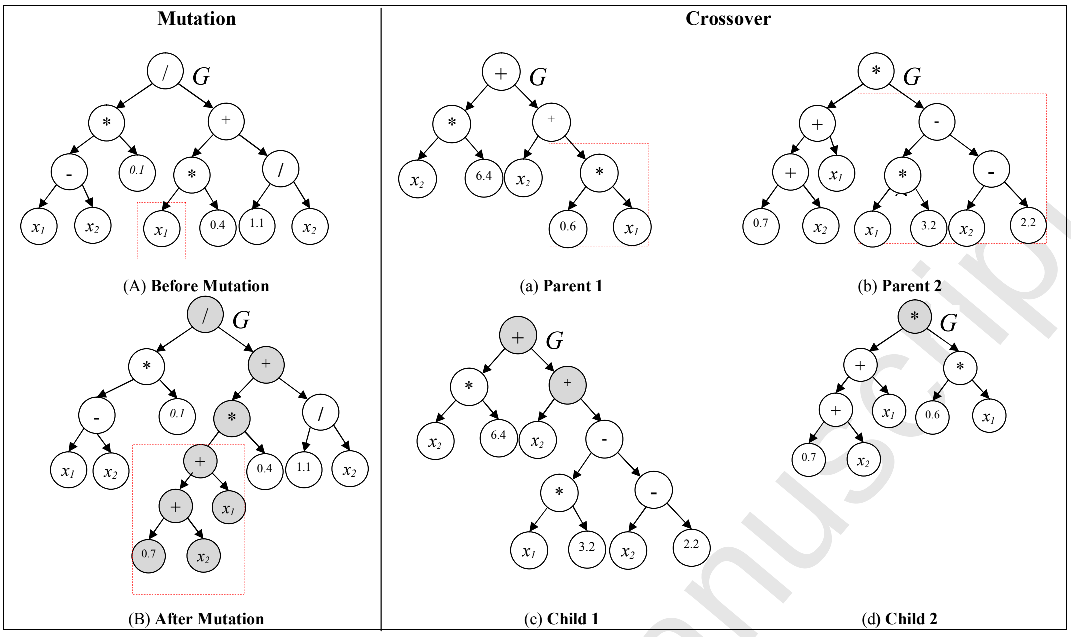


Figure 4: Reusability of statistical information of GP trees during the evolution. (Only the statistics of highlighted nodes should be updated after the genetic operations. Statistical information of other nodes is reusable.)

computed simultaneously with the computation of the fitness values. According to that, it is very easy to compute fitness and statistics of the tree in parallel.

Moreover, in Section 4, we show that the statistical GP finds the solution in fewer generations than GP. This reduction of computations is more than the overheads of SGP. In Section 4.6 it will be shown that the execution time of SGP is smaller than the execution time of the standard GP when both of them find the same solutions (by "same solution" we mean solutions with the same value of fitness).

4. Experimental Results

This section is devoted to testing the performance of SGP and the effects of the new genetic operators. Specifically, we investigate the effects of SGP from several perspectives. Section 4.2 illustrates the effects of WSSs and the newly defined operators (CB crossover, CB mutation, and VB editing) on the GP performance. Section 4.3 and Section 4.4 examine the bloat controlling ability and generalization of SGP respectively. Section 4.5 is devoted to the diversity of SGP. Section 4.6 investigates the execution time of SGP. Section 4.7 compares SGP with two important semantic-based GP methods. Finally, Section 4.8 tests SGP on a tunably difficult problem.

4.1. Test Problems and GP Parameter Settings

SGP was tested by six real valued symbolic regression problems. The benchmark problems are presented in Table 1.

The first three benchmarks are artificial test problems which were selected from [44, 74, 48, 49, 31].

The fourth benchmark is a real-life problem called "Toxicity" which is also known as LD50. It was previously used in [5, 65, 66, 7, 72, 12]. The Toxicity dataset has 234 instances and 626 input variables (features). The features of each instance are molecular descriptors identifying a drug, and its target output is the median lethal dose (*toxicity*) of that drug. More details of this problem are available in [5].

Benchmark 5 (Bioavailability) is from real-life as well, and was previously used in [5, 65, 66, 7, 72, 12]. The Bioavailability dataset has 260 instances and 241 features. The features of each instance are molecular descriptors

Table 3: GP Settings

Population Size	200
Function Set	$\{+, -, \times, \div\}$
Fitness Function	Root Mean Squared Error (RMSE)
Initial Population Method	Ramped-half-and-half
Selection Method	Tournament Selection
Tournament Size	5
Crossover Rate	90%
Mutation Rate	5%
Maximum Tree Depth in Initial Population	6
Number of Generations	100
Number of Runs	50 independent runs for each test

that identify a drug, and its target output is the percentage of the drug effectively reaching systemic blood circulation (see [5] for more information).

The sixth benchmark is "Concrete Compressive Strength". This dataset is one of the datasets of the *UCI Machine Learning Repository* [76], was previously used by [7, 29, 6]. Concrete dataset includes 1030 instances and 8 input variables. The variables indicate the composition of a concrete. The target output is a highly non-linear function of the inputs and it is the quantitative value of compressive strength of a concrete.

For benchmarks 4, 5, and 6, fifty percent of each dataset is used for training the GP systems.

The function set is $\mathcal{F} = \{+, -, \times, \div\}$. Note that "÷" represents *protected division*[4]. The terminal set consists of 100 *Ephemeral Random Constants* (ERC) [33] chosen in the range $[0, 1]$, and the benchmark function variables. For SGP the WSSs are also added to the terminal set. Parameters of the basic GP are tuned as shown in Table 3. The fitness function is Root Mean Squared Error (RMSE).[5] Note that here, we do not put any limit on the size of programs (i.e., maximum depth or number of nodes), because we want to investigate the pure effect of SGP on the code growth. Moreover, the population size is set to 200, not to a high population size: It is important to show that while SGP limits the search space, it can preserve the diversity and reach accurate solutions, even when the population size is not big. The parameter settings of Table 3 are used for the experiments of Section 4.2 through Section 4.6.

SGP has the following extra parameters:

- **Depth limit for generating WSSs (l):** l is a limit on the depth of the trees in the WSS pool. This parameter is set based on experimental results over a validation set. To investigate a proper value for this parameter, we considered the validation error (see Table 8 for the validation sets of the benchmarks) and the time required to generate the WSS pool. If l is set to 4, SGP can reach accurate solutions without much computational cost for all six benchmarks.

  Table 4 shows the "average validation error" and "the average time of WSS pool generation" for various choices of l. By increasing l, the validation error of SGP decreases and the running time increases. For $l = 4$ (and to some extent $l = 3$), we obtained a good trade-off between the validation error and the running time.

  For $l < 3$, the WSS trees are too small to positively affect the final results. For most problems, $l > 4$ does not reduce the error much (due to the loss of diversity), but running time increases noticeably.

- **Tournament size for CB crossover (Q):** In CB crossover, the subtrees of each parent compete, and the one with a higher correlation coefficient with f wins. The winner subtree is replaced with the winner of the other parent.

[4]Protected division returns one when the denominator is zero.

[5]In this paper, we utilized RMSE or $-$MSE as the fitness functions, which are basically the same (RMSE is the square root of MSE). RMSE is more common in the literature for these benchmarks, but in Section 3.3, we opted for $-$MSE as the fitness is "better" for greater values of $-$MSE. Furthermore, the range of $-$MSE values was more suitable for quantization into bins.

In Section 4.8, however, we used the *Sum of Squared Errors* for the raw fitness function, as we wanted to compare our approach with previous works, especially with [28].

Table 4: Comparison of average validation error and average time for constructing the WSS pool, based on different choices of depth limit l. Note that the column "NO WSS" refers to the canonical GP, in which no WSS is constructed, and hence the WSS construction time is 0.

(a) Average validation error based on RMSE							
Test Problem	No WSS	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$	$l = 6$
Benchmark 1	0.95	0.58	0.57	0.54	0.52	0.52	0.53
Benchmark 2	6.73	5.61	3.32	3.15	2.90	2.82	2.80
Benchmark 3	0.392	0.297	0.208	0.201	0.203	0.201	0.210
Benchmark 4	2618.10	2563.71	2421.09	2335.83	2316.09	2297.72	2290.01
Benchmark 5	46.74	38.59	34.99	34.05	33.65	34.36	35.62
Benchmark 6	23.81	21.33	18.65	15.01	14.91	14.37	14.41
(b) Average time (in seconds) for constructing WSS pool							
Test Problem	No WSS	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$	$l = 6$
Benchmark 1	-	0.8	1.52	2.72	6.05	14.64	22.12
Benchmark 2	-	0.3	0.6	1.02	1.82	2.52	3.67
Benchmark 3	-	0.2	0.28	0.36	0.46	0.59	0.79
Benchmark 4	-	3.3	6.98	11.45	20.77	38.45	73.2
Benchmark 5	-	2.11	4.07	8.33	14.49	29.91	69.03
Benchmark 6	-	0.95	2.19	3.51	6.95	14.62	23.40

Table 5: Average validation error (based on RMSE) for various choices of Q. Notice that Q is the number of nodes, denoted here as the percentage of the size of the tree. For instance, if the tree has 200 nodes and the percentage is 5%, then $Q = 200 \times 5\% = 10$.

Test Problem	Percentage of tree nodes								
	Standard crossover	2%	5%	10%	15%	20%	25%	30%	35%
Benchmark 1	0.95	0.81	0.67	0.65	0.66	0.67	0.71	0.84	0.89
Benchmark 2	6.73	5.53	4.61	4.13	3.99	3.91	0.382	4.93	5.72
Benchmark 3	0.392	0.341	0.277	0.251	0.244	0.243	0.253	0.285	0.293
Benchmark 4	2618.10	2503.3	2415.6	2389.1	2453.2	2481.5	2462.2	2498.1	2505.6
Benchmark 5	46.74	40.70	37.59	36.00	35.85	35.69	36.77	37.85	39.44
Benchmark 6	23.81	20.41	16.33	15.65	15.87	16.60	18.55	21.27	21.74

This competition is implemented by Q-tournament selection, where Q denotes the number of nodes chosen for competing in the tournament.

To find the best value for the parameter Q, different values were tested for each benchmark. Table 5 presents the validation error of SGP with different values for parameter Q. The results showed that choosing Q somewhere between 5% to 20% of the tree size leads to good results for all six benchmarks. In our experiments, the parameter Q is set to 10% of the tree size.

- **Percentage of selected features for generating WSSs:** To generate the WSS pool, we used Algorithm 1. This algorithm constructs and evaluates a tree for all pairs of variables. This can be time-consuming for high-dimensional problems (such as benchmarks 4 and 5), where there are many pairs of variables. In such cases, SGP uses feature selection to make computations efficient. In this paper, a feature selection method based on the mutual information [22] is used.

  To investigate the number of features required for generating WSSs, a trade-off between "the time of generating the WSS pool" and "the validation error" is considered. Table 6 depicts this trade-off for different percentages of selected features (for benchmarks 4 and 5). A good trade-off is achieved if we select about 5% of features.

  Note that the extended terminal set includes variables in addition to WSSs. Thus, while feature selection is used for generating WSSs, all the variables can be used in the population trees.

Table 6: Comparison of average validation error and average time of constructing the WSS pool, for various percentages of the selected features. Note that the column "NO WSS" refers to the canonical GP, in which no WSS is constructed, and hence the WSS construction time is 0.

(a) Average validation error based on RMSE							
Test Problem	0% (NO WSS)	2%	5%	7%	10%	13%	15%
Benchmark 4	2618.10	2443.71	2316.09	2395.83	2293.45	2301.72	2353.01
Benchmark 5	46.74	36.09	33.65	33.35	33.14	33.90	34.71
(b) Average time (in seconds) for constructing WSS pool							
Test Problem	0% (NO WSS)	2%	5%	7%	10%	13%	15%
Benchmark 4	-	6.87	20.77	30.69	49.58	90.45	194.7
Benchmark 5	-	6.11	14.49	22.71	34.89	57.22	89.10

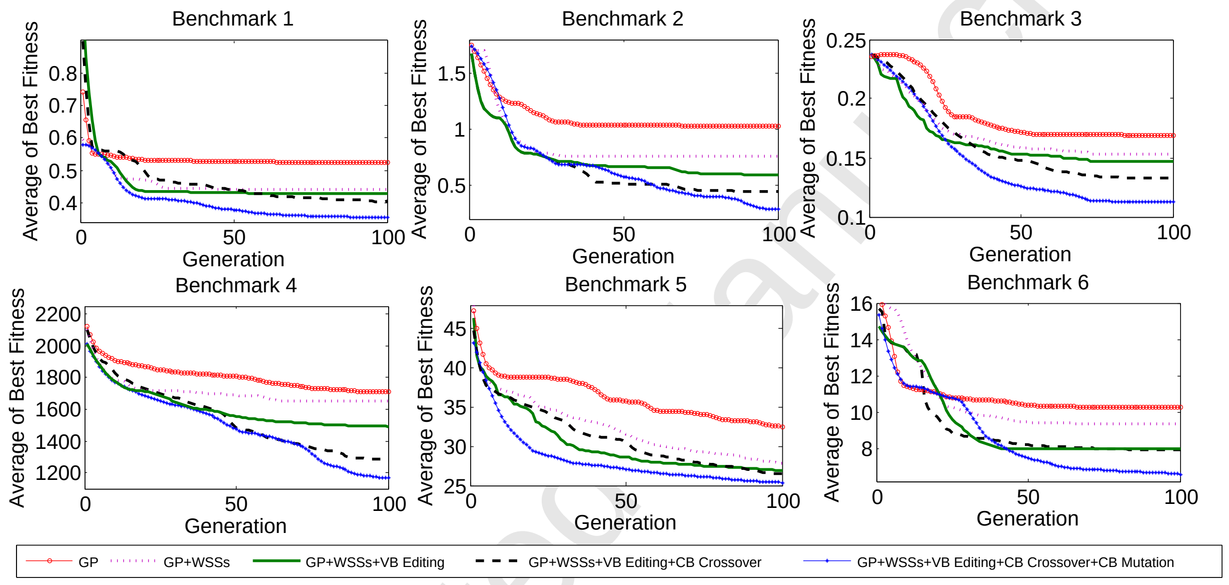


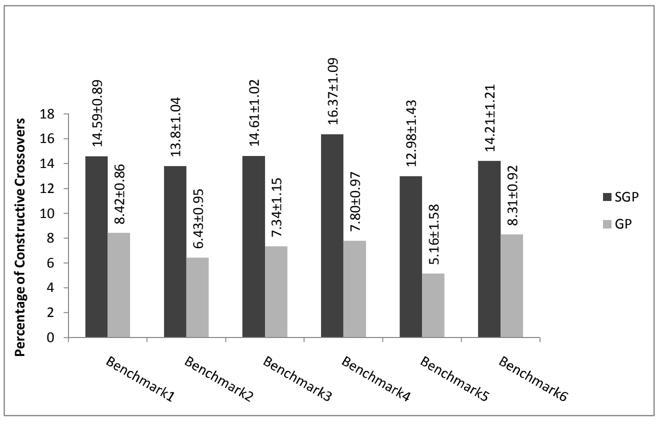
Figure 5: The effects of each SGP level on the GP performance.

4.2. *The Impact of SGP Modifications*

Define the "base GP" as the standard GP with the settings reported in Table 3. To show the effect of each SGP modification (moving-point) over the *base GP*, let us examine the results in a step-by-step manner. At each step, one of the modifications is added to the previous step, and the resulting accuracy is analyzed.

1. **Generating and employing WSSs**. In the first step, the WSS pool is constructed using Algorithm 1, and added to the extended terminal set.
2. **Variance-based editing:** In the second step, the variance-based editing operator is added to the previous level.
3. **CB crossover**. The third incremental modification of GP is adding the CB crossover. This is further elaborated in Section 4.2.1.
4. **CB mutation**. Finally, in the fourth step, the CB mutation operator is performed instead of standard mutation. Further details can be found in Section 4.2.2.

Figure 5 exhibits the results of experiments on the benchmark functions. In these charts, the performance of each step of SGP is compared with other steps, as well as with the simple GP (base GP). As the figure shows, by adding each SGP step to the GP, the algorithm attains more accurate solutions. The reason for the improvement in the accuracy of solutions and speed of the algorithm is threefold: (1) limiting the search space to semi–well-structured trees,



Test Problem	P-value of Welch's t-test
Benchmark 1	1.97E-57
Benchmark 2	4.41E-59
Benchmark 3	6.42E-55
Benchmark 4	1.88E-63
Benchmark 5	1.91E-45
Benchmark 6	6.18E-46

Figure 6: Comparing the rate of constructive crossover in GP and SGP.

(2) deleting the introns, and (3) exploring this limited space using correlation-based operators with more constructive rate.

In the next two subsections, the constructive rates of correlation-based operators are compared with the standard ones.

4.2.1. Constructive Crossover Rate

The most important advantage of CB crossover is that it decreases the rate of destructive and neutral crossovers, while increasing the rate of constructive ones. Neutral crossover is a crossover operation, which results in generating an offspring whose fitness is identical to the fitness of their parents. Constructive crossover is a crossover operation that generates an offspring with a fitness value greater than the parents. Moreover, destructive crossover is a crossover that results in offspring with a fitness value worse than the parents [52]. In CB crossover, the subtrees, which are more correlated with f, are more likely to be selected as swapping genetic material. Thus, the probability of selecting the swapping genetic material from ineffective codes (introns) and the probability of neutral crossovers are decreased. Moreover, CB crossover leads to relocating more correlated subtrees with f in the population, which are hopefully placed in the right positions. Therefore, CB crossover increases the rate of constructive crossover. Based on [49], if the two subtrees chosen as the crossover points have similar semantics, the results of crossover operators are improved. In the standard subtree crossover, two subtrees are chosen randomly to be substituted with each other. In the process, a good subtree with high contribution may change with a low-contribution subtree. In this case, the crossover is destructive. However, in the CB crossover, the chance that the crossover points are both subtrees with high contribution is more. Thus, the probability of changing a high-contribution subtree with a low-contribution one is less, and therefore the chance of destructive crossover is lower.

In Figure 6 the percentage of the constructive crossover in the SGP is compared with the percentage of the constructive crossover in GP for the six benchmarks.

To evaluate the significance of the difference in the average percentages of the constructive crossover between SGP and canonical GP, a two-tailed Welch's t-test with the confidence level of 0.95 ($\alpha = 0.05$) was used. The Welch's t-test is an adaptation of the Student's t-test useful when the variance of the two samples is not necessarily equal. Here, the null and the alternative hypotheses are as follows:

- H_0: the averages of the percentage of the constructive crossover of SGP and of canonical GP are the same.
- H_1: the averages of the percentage of the constructive crossover of SGP and of canonical GP are different.

The P-values of the Welch's t-test are shown in Figure 6. As can be seen in the figure, the rate of constructive crossover in SGP is significantly greater than that of canonical GP.

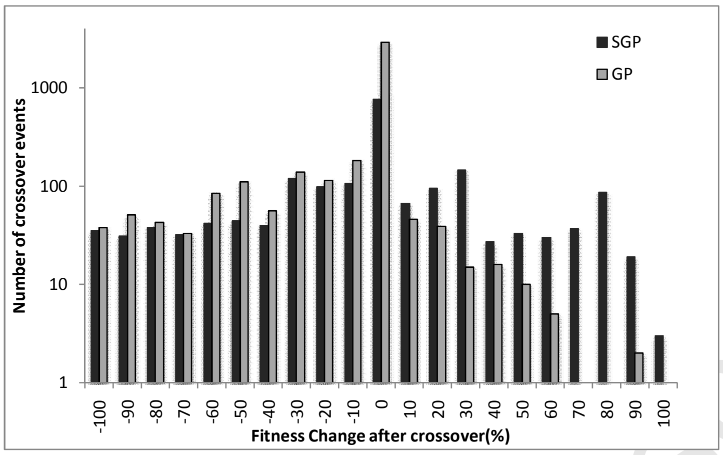
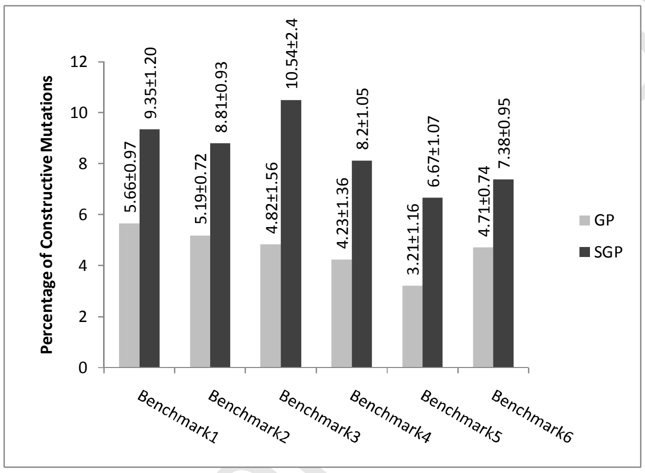


Figure 7: Typical proportion of the destructive, neutral and constructive crossovers for early generations. The right area of zero is devoted to the constructive crossovers, and the left one is devoted to the destructive crossovers. The area over zero demonstrates the neutral crossovers.



Test Problem	P-value of Welch's t-test
Benchmark 1	3.04E-30
Benchmark 2	4.24E-38
Benchmark 3	6.31E-24
Benchmark 4	5.90E-29
Benchmark 5	4.82E-28
Benchmark 6	8.78E-28

Figure 8: Comparison of constructive mutation rate of GP and SGP.

For more details, Figure 7 displays the histogram of the relative proportions of the three different types of crossover for GP and SGP (for benchmark 2). The x-axis represents the percentage of change ($\Delta f_{\text{percent}}$) in the fitness after performing crossover.

$$\Delta f_{\text{percent}} = 100 \times \frac{\Delta f_{\text{before}} - \Delta f_{\text{after}}}{f_{\text{before}}} \tag{11}$$

Note that the optimal fitness is 0 and the worst fitness is ∞. The area right of zero is devoted to constructive crossovers and the left one is devoted to destructive crossovers. As can be seen in the histogram, the number of neutral and destructive crossovers in SGP is less than those of GP. Additionally, the number of constructive crossovers of SGP is more than that of GP.

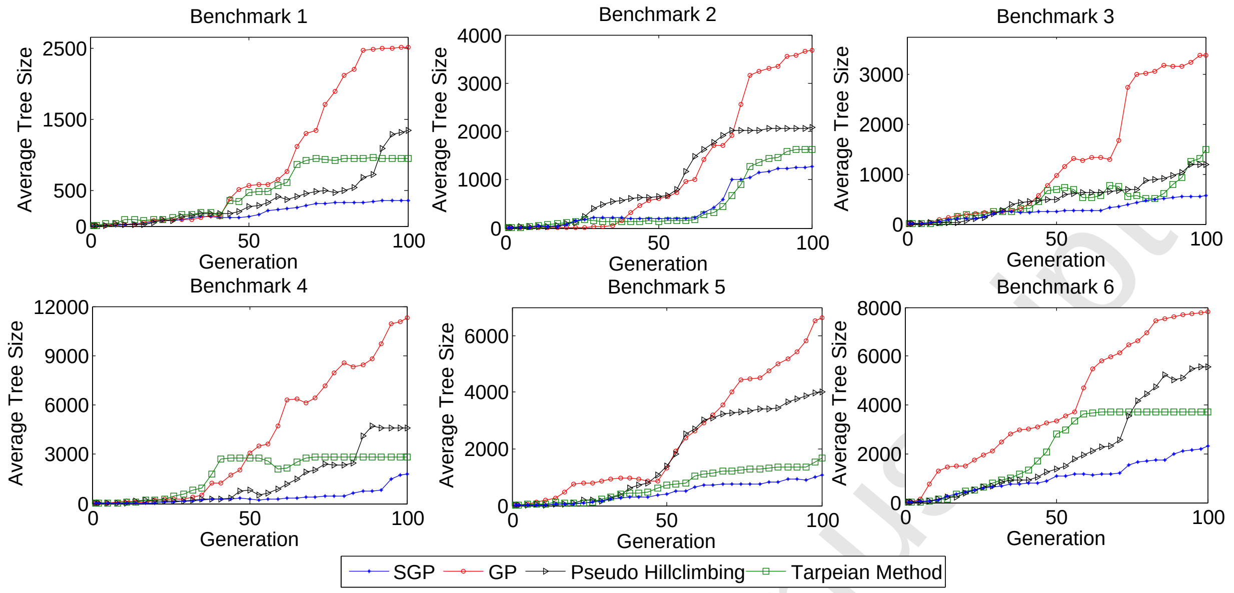


Figure 9: Comparison of GP, SGP, and two bloat controlling methods in their ability of controlling bloat.

4.2.2. Constructive Mutation Rate

This subsection is devoted to investigating the constructive mutation rate of CB mutation. Constructive mutation is a mutation operator that leads to generate an individual with the fitness value greater than that of its parent. In Figure 8 the percentages of constructive mutation of SGP and GP are compared for the six benchmarks. The rate of constructive mutation in the SGP is almost twice the average of GP.

In CB mutation, the mutation point is selected based on the correlation of each node with f, so the probability of selecting the mutation point from ineffective codes is decreased, while the probability of generating an offspring having better fitness than its parent is increased. In the standard mutation, a random subtree of the tree is replaced by a random subtree. Thus, a good subtree may be replaced by a random subtree, which destructs the solution tree. However, in CB mutation, subtrees which have no contribution have more chance to be mutated. Therefore, the probability of changing a high-contribution subtree a random subtree is decreased, and so is the probability of destructive mutation.

Therefore, CB mutation can improve the performance of SGP by increasing the probability of constructive mutations. Moreover, in CB mutation the replaced subtree is semi–well-structured, so it can lead to generating trees which are closer to well-structured trees. As mentioned before, the trees which are nearer to the structure of a well-structured tree are smaller.

4.3. SGP and Bloat

One of the important advantages of SGP is its bloat controlling ability. Here, the bloat of SGP in comparison with GP and two other bloat controlling methods is investigated. SGP can control the bloat by means of two methods: (1) using VB editing (2) limiting the search space to semi–well-structured trees and finding solutions with greater WSM. Figure 9 displays, for GP and SGP, the *average tree size of the population* vs. *the number of generations*. Generally, the average tree size of SGP increases much more slowly than that of GP. In Figure 9, SGP is also compared with two bloat control methods, the Tarpeian method and pseudo hillclimbing. Based on the results of [2], Tarpeian method is one of the most effective bloat control methods. As in [2], the parameter p_t of this method was set to 0.3. Pseudo hillclimbing is another bloat control method which can decrease the rate of code growth [38, 42].

As the figure shows, SGP can control the rate of code growth more effectively than these methods. The average and the standard deviation of the average tree size of population at the end of the evolution for these GP systems are

Table 7: Average tree size.

	SGP	GP		Pseudo HillClimbing Method		Tarpeian Method	
Test Problem	**Average Tree Size**	**Average Tree Size**	**P-value of Welch's t-test**	**Average Tree Size**	**P-value of Welch's t-test**	**Average Tree Size**	**P-value of Welch's t-test**
Benchmark 1	**359.8 ± 78.1**	2513.3 ± 561.3	9.88811E-32	1323.6 ± 427.5	2.07349E-21	958.2 ± 103.3	3.87E-52
Benchmark 2	**1250.5 ± 241.4**	3650.6 ± 804.5	7.2646E-28	2071.1 ± 348.9	2.081E-23	1633.3 ± 267.1	2.79E-11
Benchmark 3	**560.75 ± 82.3**	3370 ± 629.2	8.0931E-35	1189.5 ± 331.7	2.0073E-18	1495.8 ± 221.6	5.82E-37
Benchmark 4	**1773.3 ± 311.2**	11280.6 ± 2008.9	2.7383E-36	4629.4 ± 875.8	1.8471E-30	2851.5 ± 439.5	1.96E-24
Benchmark 5	**1080.1 ± 186.1**	6623.4 ± 1065.7	1.451E-38	4000.7 ± 734.2	9.7684E-34	1670.1 ± 366.3	1.38E-15
Benchmark 6	**2301.4 ± 304.6**	7801.9 ± 1471.9	8.2508E-32	5549.2 ± 921.5	5.4795E-32	3705.2 ± 647.8	1.02E-21

shown in Table 7. Moreover, the p-values of the Welch's *t*-test are shown in this table. The Welch's *t*-test examines the significance of the difference between the average tree size of the population of SGP and other GP systems.

Note that here, we do not put any limit on the size of programs (i.e., maximum depth or number of nodes), because we want to investigate the pure effect of SGP on the code growth. As the results of Table 7 show, if the GP does not have any control on the program size, the size of its programs increase dramatically. In contrast, SGP can reduce the code growth effectively without putting any limit on the program size. In Section 4.7, we put depth limit on the GP systems. By looking at the results of that experiment, it is possible to see that if we use a size limit for GP and SGP programs, the size of both GP systems becomes smaller. However, the size of SGP programs is still smaller than that of GP programs.

4.4. Generalization Ability

Generalization ability is one of the most important aspects in measuring the performance of a learning algorithm.

We used two methodologies for comparison GP and SGP generalization, one with *two* datasets (training and test), and one with *three* datasets (training, validation and test). These Methodologies are as follows:

1. **Using two datasets: training and test.** Each algorithm is trained for several generations (e.g., 100 generations), and the final solution of each algorithm (e.g., the solution of the generation 100) is evaluated on the test data. In this case, the canonical GP has usually poor results, as the bloat occurs, the solutions become gradually more complicated, and the generalization ability decreases.
2. **Using three datasets: training, validation and test.** By using a validation set, it is possible to increase the generalization ability of GP solutions. We use the validation set in order to stop the evolution process: When the GP starts to overfit (i.e., for several consecutive generations, the validation error increases), we can stop the evolution. In other words, when the validation error of the GP solution increases while the training error decreases, it means that the GP solution starts to lose its generalization. So the evolution should be stopped, and the final solution of GP is the best (based on training data) individual of the last generation before the overfitting occurs.

The two-dataset methodology was used as it is the most common approach in many papers, and we followed this methodology to make our results comparable with the ones in the literature. However, [23] suggests to increase the generalization ability of GP by just using a validation set, and stop the evolution process when the overfitting is detected by the validation set. To investigate whether the generalization ability of SGP is better than GP *even in the case that the validation set error is used to stopped GP system from overfitting*, we followed the three-dataset methodology.

In the second methodology, we stop the evolution if the validation error increases for 10 consecutive generations. The size of the validation and test sets for each benchmark is outlined in Table 8. The benchmarks, training instances and the domain of each benchmark were previously described in Table 1, all the parameter settings of this experiment are similar to the previous experiments (Table 3). In Table 9, the averages and standard deviations of the test error (RMSE) for SGP and canonical GP are compared. In both cases, we compared the values for the two-dataset and three-dataset methodologies.

Table 8: The validation set and test set used for each benchmark problem.

Benchmark Function	Validation Set	Test Set
Benchmark 1	100 random points	500 random points
Benchmark 2	50 random points	1157 points $x_1, x_2 = (-0.25 : 0.2 : 6.35)$
Benchmark 3	50 random points	2000 points $x = [-2 : 0.001 : 2]$
Benchmark 4	20% of the data	30% of the data
Benchmark 5	20% of the data	30% of the data
Benchmark 6	20% of the data	30% of the data

Table 9: The average and standard deviation of SGP test error in comparison with GP.

Test Problem	**GP Test Error** Two-Dataset Methodology	**SGP Test Error** Two-Dataset Methodology	**GP Test Error** Three-Dataset Methodology	**SGP Test Error** Three-Dataset Methodology	**P-value of Welch's t-test** Three-Dataset Methodology
Benchmark 1	0.97 ± 0.13	0.67±0.070	0.59±0.021	0.52±0.034	1.94E-20
Benchmark 2	6.7 ± 1.21	2.65± 0.81	3.81±0.88	2.38±0.46	1.00E-12
Benchmark 3	0.403 ± 0.066	0.193 ±0.039	0.221±0.021	0.176±0.018	1.02E-15
Benchmark 4	2652.12 ± 134.3	2262.53 ±101.2	2167.54±85.37	2003.41±67.56	3.46E-18
Benchmark 5	45.16 ± 2.95	35.01 ± 2.60	36.1±2.32	33.27±1.21	5.40E-11
Benchmark 6	23.45 ± 2.61	14.97 ± 1.75	15.81±1.93	12.93±1.54	9.47E-20

The results of Table 9 indicate that the test error of SGP is less than that of the canonical GP for both methodologies. SGP finds smaller solutions, so the probability of finding simpler solutions (those with less functional complexity) is greater than the canonical GP. It is important to note that the canonical GP has poor generalization when the two-dataset methodology is used. In contrast, SGP reaches a solution with acceptable generalization in this case.

Two types of complexities are defined for a GP model [74]:

1. *Compactness of the genotype (structural complexity):* Specifies how compact the model expression is.
2. *Smoothness of the phenotype (functional complexity):* Determines how smooth the associated function surface is.

A solution is said to be more general if it does not *overfit* the training data. Overfitting is closely related to *functional complexity*. While overfitting can be due to the *bloat* phenomena, the latter can occur because of the introns (ineffective codes), which does not cause overfitting. In this case, bloat increases the structural complexity, but not the functional complexity.

Larger trees are often more complex, both structurally and functionally. SGP tries to find solutions that are smaller than those found by GP. This is done by removing introns, as well as by finding simpler (having less functional complexity) solutions. While the former does not directly influence the generalization ability of SGP, the latter does.

4.5. Diversity of SGP

In [4], we started to investigate the diversity of SGP. The results showed that SGP has higher phenotypic and genotypic diversities. However, a new comparison is necessary, as SGP is modified with the addition of WSSs. To measure the phenotypic and genotypic diversities, the corresponding entropies were utilized using the same method of [4].

4.5.1. Diversity Measures

This section presents the diversity measures which are used in this paper.

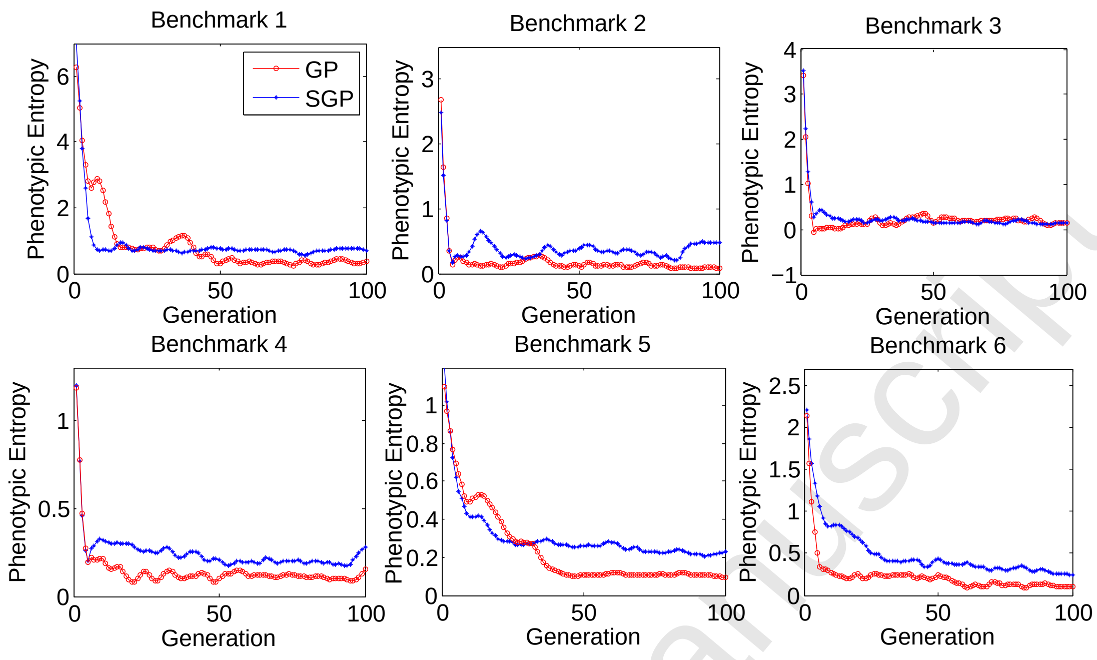


Figure 10: Phenotypic diversity of SGP in comparison with GP.

Phenotypic diversity is related to different fitness values in the population. In this paper, phenotypic entropy is utilized as a phenotypic diversity measure. The phenotypic entropy of the population P can be calculated as follows [61]:

$$H_p(P) = -\sum_{j=1}^{N} p_j \log(p_j) \tag{12}$$

where N is the number of different fitness values in the population P, and p_j is the fraction $\frac{n_j}{N}$ of the population P that has fitness j.

As fitness in our case is a continuous quantity, we have to discretize the fitness values first. To this end, we used an adaptive procedure, in which the ranges are determined *on the fly* while the fitness values become known gradually. In practice, for each generation, the first computed fitness value becomes the representative for the first range. Subsequently, we compute the following quantity for each fitness range i:

$$\delta_i = \left| \frac{\text{new fitness value} - \text{avg. fitness in the range } i}{\text{avg. fitness in the range } i} \right| \tag{13}$$

and if it is less than a predefined threshold τ, we put the new fitness value into that range (in case of ties, the i having the minimum δ_i wins). Otherwise, if no such i is found, a new range is created. In the experiments, τ is set to 0.02.

In order to measure the genotypic diversity, the genotypic entropy is used in the paper. Genotypic diversity is related to different structures in the population. A tree distance measure is needed to keep into account the different structures. We use the tree edit distance measure, as defined by Ekárt and Németh [19].

The distance between two trees T_1 and T_2 can be computed as follows:

$$\text{dist}(T_1, T_2) = \begin{cases} d(a,b) & \text{if neither } T_1 \text{ nor } T_2 \text{ has any child} \\ d(a,b) + K \times \sum_{l=1}^{m} \text{dist}(s_l, t_l) & \text{otherwise} \end{cases} \tag{14}$$

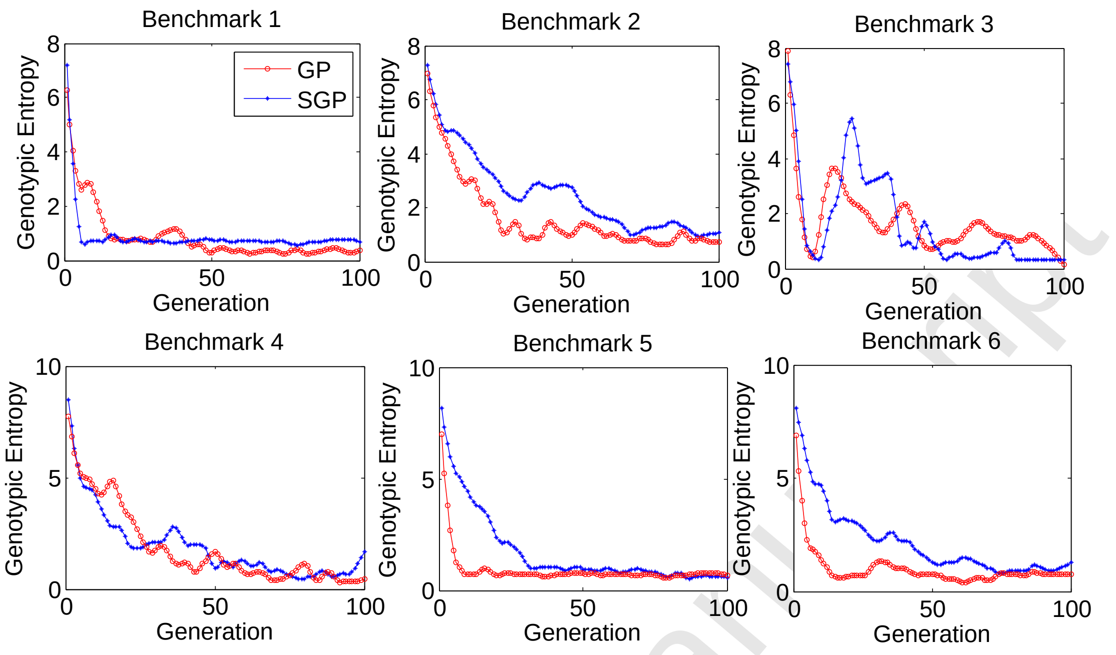


Figure 11: Genotypic diversity of SGP in comparison with GP.

where a and b are the roots of T_1 and T_2, respectively. T_1 and T_2 have m possible subtrees s and t. The parameter K is set to $\frac{1}{2}$. The a-to-b distance $d(a, b)$ is 0 if the nodes a and b are equal, and 1 if they are different. The edit distance is calculated for each individual against the best individual in the run so far (note that it is different from the best individual in the current population).

As in the case of the phenotypic entropy, genotypic entropy is computed as follows:

$$H_{ge}(P) = -\sum_{j=1}^{N} \text{ge}_j \log(\text{ge}_j) \tag{15}$$

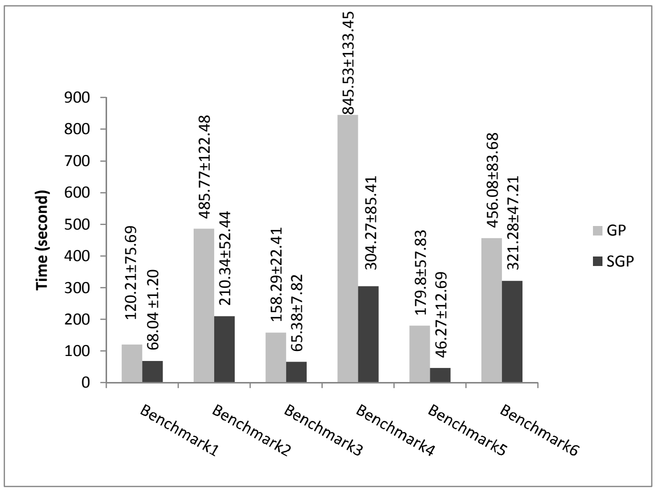
where ge_j is the fraction of the population that has a given distance from the best individual in the run so far.

4.5.2. Diversity results

Figure 10 and Figure 11 show the phenotypic and genotypic diversities of SGP and GP for the mentioned benchmarks. The results demonstrate that the phenotypic and genotypic diversities of SGP are no less than those of the GP. Thus, limiting the GP search space to the semi–well-structured tree does not lead to the loss of diversity, and the diversity of SGP is still no less than that of GP.

In SGP, the search space is limited to semi–well-structured trees. This can possibly *reduce* the diversity. However, the *phenotypic* diversity of SGP is not less than that of GP due to the following reasons:

1. The CB crossover decreases the rate of neutral crossover while increasing the constructive crossover. Thus, in SGP, the probability of generating an offspring which is better than (phenotypically different from) its parent is higher than in the standard GP.
2. The CB mutation decreases the rate of neutral mutation while increasing the rate of constructive mutation. Thus, similar to CB crossover, it increases the phenotypic diversity.
3. VB editing is effective in eliminating the introns. This can help in significantly decreasing the rate of neutral genetic operations. Neutral operators generate offsprings with no phenotypic changes.



Test Problem	P-value of Welch's t-test
Benchmark 1	2.82E-05
Benchmark 2	1.97E-22
Benchmark 3	3.79E-36
Benchmark 4	2.11E-39
Benchmark 5	5.06E-22
Benchmark 6	1.96E-15

Figure 12: Running time of SGP and GP to reach the same solution.

Additionally, he *genotypic* diversity of SGP is not less than that of GP due to the following reason:

In a well-structured tree, higher nodes are more correlated to the regression function f. SGP guides the search process toward finding trees whose structures are close well-structured trees. Hence, in CB crossover, higher nodes of the tree have more chance to be selected as the crossover point. In the measure of genotypic diversity, higher nodes of a tree are more influential, as the coefficient K (in Equation 14) is less than 1 (here it is 0.5).

4.6. Running Time

Here, we compare the running time of GP and SGP. Although SGP calculates extra information, it improves the speed of GP in the following sense: SGP finds solutions with the same fitness faster than GP. Figure 12 compares the average and standard deviation of running time of GP and SGP to reach a solution with the same fitness. As the table demonstrates, SGP finds the solution more rapidly than the canonical GP. Note that for this experiment, the reusability of information and the parallelization, which were mentioned in Section 3.7 are not used. Thus, if reusability and parallelization were utilized in SGP, the performance of SGP still will improve. Moreover, to obtain fair comparison, the time spent on generating WSSs is included in the SGP running time.

4.7. Comparison with Semantic-Based GP Systems

In this subsection, SGP is compared with semantic-based GPs. SGP uses the correlation between the outputs of each subtree and the target outputs. Semantic based GP methods also utilize the outputs of each subtree as the semantics of that subtree. Thus, these approaches are somehow similar. Several of the recent semantic methods have been mentioned in Section 2. Here, SGP is compared with two of the best semantic methods, based on the training and test error, the running time, and the code growth. These methods are the AGX (Approximately Geometric Semantic Crossover) and RDO (Random Desired Operator) which are completely investigated in [54].

For this experiment, the parameter settings are similar to those of [54]: We consider two optimum[6] library sizes for AGX and RDO. One of them contains trees with maximum depth equals to 3 ($h = 3$), while the other contains trees with maximum depth of 4 ($h = 4$). If AGX or RDO uses the former library, we call them AGX$_3$ and RDO$_3$; otherwise, they are called AGX$_4$ and RDO$_4$, respectively. The population size is 500, the function set is $\mathcal{F} = \{+, -, \times, \div\}$, the tournament size is 7. The fitness function is RMSE. Moreover, the method for generating the initial population is "ramped-half-and-half," and the maximum tree depth in initial population is 6. Additionally, for all GP methods, the crossover rate and the mutation rate are 90% and 10%, respectively. Similar to [54], for reducing the code growth for

[6]Optimum, as reported by the authors of [54].

Table 10: Comparing SGP with AGX, RDO and canonical GP. The best method for each benchmark is highlighted.

	Method	Training Error (RMSE)	Test Error (RMSE) Two-Dataset Methodology	Test Error (RMSE) Three-Dataset Methodology	Tree Size	Running Time
Benchmark 1	GP	0.49 ± 0.21	1.04 ± 0.15	0.56 ± 0.05	301.2 ± 87.9	151.5 ± 30.2
	SGP	0.28 ± 0.12	0.71 ± 0.09	0.51 ± 0.03	126.5 ± 35.2	134.4 ± 27.5
	AGX_3	0. 29± 0.13	0.78 ± 0.15	0.53 ± 0.03	86.7 ± 17.0	581.3 ± 57.4
	RDO_3	0.28 ± 0.08	0.71 ± 0.09	0.52 ± 0.05	109.0 ± 15.6	481.1 ± 62.0
	AGX_4	0.25 ± 0.09	0.62 ± 0.07	0.52 ± 0.03	97.7 ± 21.8	621.3 ±76.3
	RDO_4	0.26 ± 0.10	0.68 ± 0.07	0.51 ± 0.04	139.0 ± 20.9	521.1 ± 80.9
Benchmark 2	GP	0.78 ± 0.24	6.40 ± 1.23	3.72 ± 0.91	587.6 ± 94.7	351.5 ± 51.9
	SGP	0.23 ± 0.10	2.79 ± 0.65	2.39 ± 0.52	220.7 ± 68.3	330.9 ± 55.7
	AGX_3	0.37 ± 0.13	5.05 ± 1.01	3.25 ± 0.31	43.4 ± 26.2	671.2 ± 88.0
	RDO_3	0.27 ± 0.14	4.61 ± 1.11	3.11 ± 0.46	160.0 ± 37.8	711.1 ± 74.7
	AGX_4	0.26 ± 0.09	4.05 ± 0.87	2.95 ± 0.38	266.8 ± 41.4	822.9 ± 81.2
	RDO_4	0.22 ± 0.10	3.31 ± 0.52	2.22 ± 0.36	65.5 ± 21.0	893.0 ± 77.9
Benchmark 3	GP	0.152 ± 0.023	0.370 ± 0.052	0.201 ± 0.033	332.1 ± 43.7	183.6 ± 51.3
	SGP	0.091 ± 0.010	0.181 ± 0.017	0.164 ± 0.013	141.3 ± 37.1	170.4 ± 47.8
	AGX_3	0.080 ± 0.012	0.236 ± 0.016	0.172 ± 0.012	88.7 ± 26.6	420.3 ± 90.3
	RDO_3	0.068 ± 0.005	0.195 ± 0.009	0.110 ± 0.004	136.5 ± 40.6	480.5 ± 81.0
	AGX_4	0.053 ± 0.004	0.173 ± 0.011	0.092 ± 0.006	221.5 ± 48.0	516.2 ± 74.7
	RDO_4	0.037 ± 0.005	0.163 ± 0.008	0.081 ± 0.004	265.3 ± 37.7	631.0 ± 93.7
Benchmark 4	GP	1701.1 ± 45.3	2738.0 ± 157.2	2174.4 ± 79.7	856.3 ± 90.7	521.4 ± 96.2
	SGP	1116.5 ± 89.3	2311.4 ± 91.6	2000.2 ± 81.0	339.2 ± 55.1	465.3 ± 40.2
	AGX_3	1234.3 ± 55.3	2509.5 ± 103.4	2164.3 ± 93.2	50.1 ± 13.5	502.4 ± 160.1
	RDO_3	1337.6 ± 76.9	2477.4 ± 135.2	2171.7 ± 64.2	163.0 ± 44.0	901.5 ± 80.5
	AGX_4	1234.3 ± 55.3	2430.2 ± 103.4	2078.3 ± 81.1	362.8 ± 76.1	1105.4 ± 117.0
	RDO_4	1337.6 ± 76.9	2398.9 ± 135.2	2103.7 ± 92.0	495.8 ± 88.3	1301.3 ± 123.5
Benchmark 5	GP	29.06 ± 2.12	42.19 ± 3.16	35.80 ± 2.42	613.7 ± 72.1	260.1 ± 65.8
	SGP	25.40 ± 2.05	33.24 ± 2.69	32.61 ± 1.35	212.5 ± 43.2	279.8 ± 77.2
	AGX_3	26.97 ± 1.78	37.15 ± 2.81	34.08 ± 1.62	150.4 ± 39.8	489.7 ± 53.0
	RDO_3	28.65 ± 1.90	36.52 ± 2.60	33.10 ± 1.51	142.5 ± 22.5	521.6 ± 86.9
	AGX_4	23.37 ± 1.53	36.81 ± 3.03	33.08 ± 1.33	301.3 ± 61.3	645.0 ± 75.1
	RDO_4	26.61 ± 1.75	34.72 ± 2.91	32.57 ± 1.41	346.8 ± 41.5	688.9 ± 69.0
Benchmark 6	GP	9.02 ± 1.50	24.57 ± 2.35	16.05± 1.64	829.0 ± 114.5	539.0 ± 53.9
	SGP	6.52 ± 1.16	13.32 ± 1.58	12.52 ± 1.49	294.9 ± 77.8	464.2 ± 44.6
	AGX_3	7.51 ± 1.27	17.65 ± 1.80	13.92 ± 1.57	151.3 ± 32.1	834.6 ± 79.1
	RDO_3	7.82 ± 1.04	20.3 ± 2.47	14.56 ± 1.21	136.5 ± 25.5	981.3 ± 80.4
	AGX_4	7.01 ± 1.17	15.44 ± 1.80	13.62 ± 1.57	187.5 ± 14.8	1147.0 ± 103.2
	RDO_4	6.82 ± 1.22	14.15 ± 2.47	13.65 ± 1.21	478.4 ± 57.9	1401.2 ± 125.5

all GP systems, two simple methods are used: First, for selecting the crossover point, a version of crossover called (Koza-I) [33] is used which selects the crossover points with probability of 0.9 at the non-terminal nodes and with probability of 0.1 at the leaves. Second, a maximum depth limit of 17 is used. The experiments were performed over 30 independent runs for each GP system and each problem. The benchmark problems are identical to those of Table 1. The training, validation and test sets are from Table 8.

The results of the experiment are presented in Table 10. The first column of Table 10 is the name of the method being compared. We also computed the results for the canonical GP with this parameter settings as the reference method. The second column is the training error. The third and fourth columns are the test error, computed based on the two-dataset and three-dataset methodologies, respectively, as described in Section 4.4. The fifth column is the tree size of the final solution. The sixth column is the running time in seconds. The reported values in columns 2 through 6 have the format $\mu \pm \sigma$, where μ is the mean of the value and σ is its standard deviation (over 30 independent runs).

Column 1 shows that the training error of the SGP is comparable with those of accurate but complicated methods, such as RDO_4 and AGX_4, which use a larger library and therefore have better training data. In Benchmarks 4 and 6, SGP even outperforms other methods.

Columns 2 and 3 compare the generalization ability of the methods. As Section 4.4 explains, in the two-dataset methodology (column 2), we stop the evolution if the validation error increases for 10 consecutive generations. For column 2, SGP outperforms other methods for Benchmarks 2, 4, 5, and 6. For column 3, SGP outperforms other methods for Benchmarks 1, 4, and 6. In both columns and for other benchmarks, SGP has very similar results to the best method.

For each method, considerable difference between these two columns can indicate that the method loses its generalization ability (overfitted) after some generations. Generally speaking, the canonical GP has the worst generalization ability, and the generalization abilities of AGX_4 and RDO_4 are higher than those of the AGX_3 and RDO_3, respectively. In most cases, among the *geometric semantic methods*, RDO_4 has the lowest test error (better generalization). SGP has the least difference between columns 2 and 3, showing that, compared to other GP systems in Table 10, the test error of SGP increases minimally during the evolution. Thus, the overfitting rate of SGP is less than the other GP methods.

Column 4 reports the tree size of each method. Methods such AGX and RDO utilize an *active bloat control*: In their crossover operation, a random tree of the parent is replaced by a tree whose depth is at most 3 (or 4). This can control the bloat very effectively, and sometimes results in very small solutions, especially for the methods using the smaller library. The average tree size of the SGP is quite comparable with AGX_4 and RDO_4, and are sometimes even better (smaller). The SGP results are larger than the best method (either AGX_3 or RDO_3); however, as we saw previously, SGP often computes the solution with less error and with better generalization ability. Furthermore, as we will see for the next column, SGP computes the solutions much faster.

Column 6 reports the running time of the GP methods. Based on the results, SGP is always the winner, except for Benchmark 5, where the result of SGP is comparable to that of the winner (canonical GP). Although SGP computes some statistics and generates a WSS pool, these computations are not very time consuming (we discussed about the computational complexity of SGP in Section 3.7). Additionally, as SGP can control the bloat, computing the fitness of the trees are faster and this also help SGP to reduce its time complexity. Therefore, the running time of SGP is often better than that of the canonical GP.

Notice that SGP is 2 to 4 times faster than AGX or RDO. The reason is due to the time-consuming backpropagation procedure used in AGX and RDO: The average of total running time devoted to backpropagation (inversion) process in some cases is even more than twice the evolution time of the canonical GP. Although RDO or AGX can find small solutions, they need lots of time to do so. While they can control the code growth effectively, they miss one of the important reasons of avoiding code growth: Reducing the time complexity.

4.8. *Applying SGP to a Tunably Difficult Problem*

As mentioned in Section 1, finding the solutions of some problems is hard for GP [15]. In this section, it is investigated how SGP deals with difficult problems. Hence, SGP is applied to random polynomial problems, which were shown to be tunably difficult for GP [20, 27, 28]. The polynomials used are shown in the following:

polynomial of degree 3:

$$= (x + 0.44) \times (x + 0.54) \times (x + 0.27)$$

polynomial of degree 7:

$$= (x + 0.44) \times (x + 0.54) \times (x + 0.27) \times (x + 0.04) \times (x + 0.41) \times (x - 0.43) \times (x - 0.71)$$

polynomial of degree 11:

$$= (x + 0.44) \times (x + 0.54) \times (x + 0.27) \times (x + 0.04) \times (x + 0.41) \times (x - 0.43) \times (x - 0.71) \times$$

Table 11: GP Settings

Population Size	500
Number of Generations Size	100
Function Set	$\{+, -, \times, \div\}$
Raw Fitness	Sum of Squared Error
Initial Population Method	Ramped-half-and-half
Tournament Size	4
Crossover Rate	90%
Mutation Rate	5%
Maximum Tree Depth in initial population	4
Maximum Tree Depth	10
Number of Runs	50

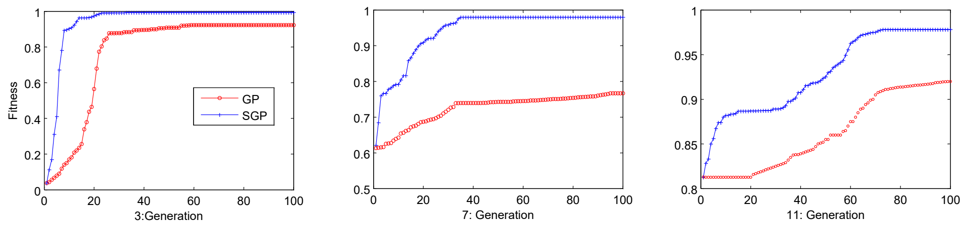


Figure 13: Fitness versus generation for polynomial problem instances for GP and SGP.

$(x + 0.82) \times (x - 0.63) \times (x - 0.75) \times (x - 0.91)$

The polynomial of degree 3 is easy for GP. The polynomial of degree 7 and 11 are moderate and difficult respectively [20, 27, 28]. Gustafson *et al.* [28] investigated the behavior of GP when it faces difficult problems. They applied GP to the aforementioned polynomial problems, and reported that when the degree of the polynomial increases, and the problem becomes harder, the rate of fitness improvement decreases. Moreover, in harder problems, the rate of code growth is higher and the population has deeper trees.

In this work, we repeated the same experiments conducted by Gustafson *et al.* in [28] for both the canonical GP and SGP in order to examine the behavior of SGP when applied to difficult problems. The same settings of the above cited paper are used. The parameter settings of GP and SGP are demonstrated in Table 11. These settings are also used for SGP algorithm. In Figure 13 and Figure 14, the adjusted fitness ($\frac{1}{1+\text{RawFitness}}$) is reported.

As depicted in Figure 14 for all three polynomial problems, the accuracy of the solutions found by SGP is more than the solutions found by GP. Furthermore, SGP finds the solutions in earlier generations. Finally, the SGP solutions are smaller than the GP solutions. Although in the experiments of this section there is a depth limit on the individuals and their depth cannot exceed 10, the average tree nodes of the population of SGP is less than the GP.

In the case of the easy problem, polynomial problem of degree 3, both GP and SGP can find accurate solutions, but SGP can find fine-tuned solutions in almost all runs, so the variance of the best of run fitness is lower in SGP. In other words, the probability of finding fine-tuned solutions is greater in SGP. Moreover, on average SGP is faster in finding the solution according to Figure 13 and Figure 14.

In the case of polynomial of degree 7, the moderate problem, the accuracy of SGP solution is significantly greater than the GP solution. Furthermore, SGP reaches the solution faster than the GP. In this case, SGP on average, reaches the solution after 34 generations; however, GP reaches the solution after 88 generations.

In the case of difficult problem (the polynomial of degree 11), SGP can find accurate solutions (with average adjusted fitness of 0.978), however GP cannot reach fine-tuned solutions, and it falls into a local optimum and converges prematurely. SGP uses CB crossover and CB mutations, also it deletes ineffective subtrees, so it can explore the search space more efficiently and can reach more accurate solutions even for difficult problem instances. Furthermore, VB editing operator helps SGP to decrease the size of solutions.

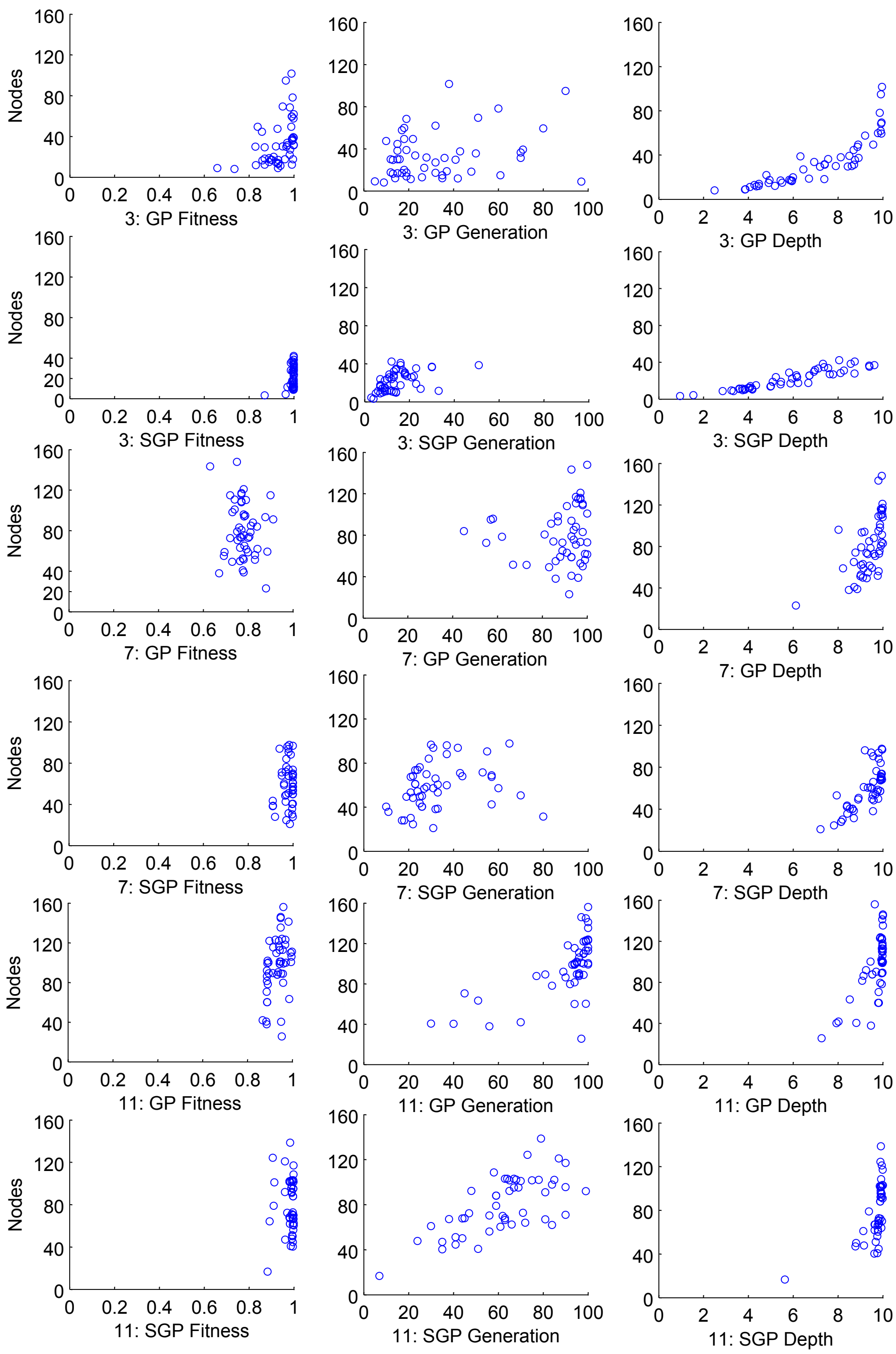


Figure 14: Results of GP and SGP algorithms for polynomial problem instances. In the left column, the average size of the population which contains the best of run individual versus the best of run fitness is sketched. In the middle column the average size is plotted against the generation in that the best solution is found. Finally, in the right column the average size versus the average depth of population is outlined. Each point belongs to an independent run.

5. Conclusion

This paper proposed a new approach to GP called Statistical Genetic Programming (SGP). GP search space contains lots of redundant (with the same phenotype) solutions with different structures and sizes (with different genotypes). SGP improves GP and addresses issues such as bloat and huge search space. To this end, SGP computes several statistics for each node of each GP tree. The statistics are the mean and variance of the output of each node, and the correlation coefficient between the node and target outputs. They are used by SGP in different phases of GP.

The approach is based on the concept of well-structured and semi–well-structured trees. In practice, a well-structured tree is a tree in which, the absolute value of the correlation of its nodes with the desired function is non-decreasing along all paths from the leaves to the root. A semi–well-structured tree is one which contains at least one WSS. The well-structuredness measure is defined as a measure that can estimate how close a tree to a well-structured tree is.

It is important to find smaller solutions, and this work showed experimentally that on average, the solution trees whose structures are closer to well-structured trees are smaller. Thus, it biases the search space towards finding such solutions. For this purpose, SGP generates a pool of low-depth WSSs, and adds them to the terminal set. By utilizing this terminal set, SGP initial population is semi-well structured, and the trees in this population contain lots of WSSs, which can help to find solutions closer to a well-structured tree.

After generating the initial population, SGP incorporates new genetic operators such as CB crossover, CB mutation, and VB editing to evolve the individuals. Correlation-based crossover selects crossover points by a chance proportional to the correlation coefficient of each node with the desired outputs. CB mutation chooses a node for mutation with a probability inversely proportional to its absolute correlation with the target. The subtree corresponding to the chosen node is replaced by a semi–well-structured tree. The VB editing method is a simple way to delete the introns, and exchange huge subtrees (which only generate a constant) with one node. Thus, VB editing can reduce the code growth effectively.

Finally, for evaluating SGP, several widely-used benchmarks were used. SGP was evaluated from different aspects, such as training error, test error, code growth, diversity, and running time. The results demonstrated that SGP can obtain more accurate and general solutions in less time, and control the bloat effectively.

6. Future Work

Although this work has the advantages mentioned in Section 5, it has some issues that can be considered for the future work:

- Not all small and fitted solutions have the property that, by moving from the leaves to the root, the correlation coefficient with the target increases. For some problems, there are good solution trees which are very small, yet do not have this property. However, we showed experimentally that, if we consider all solution trees in the GP search space, on average the solution trees with structures closer to a well-structured tree are smaller. Thus, similar to almost all bloat-controlling methods for GP, SGP tries to find smaller solutions, but does not guarantee to find the optimal solution. For future work, it is possible to investigate a property which holds for all optimal solution trees.

- This work is limited to symbolic regression problems, and it cannot be applied to some other problems, such as the classification. The reason is as follows: SGP is mainly based on using the correlation coefficient between the target output and the output of each subtree. However, in classification problems, the target output is the class label, and it is not possible to compute the correlation coefficient of the output of each subtree and the target output. For future work, it is possible to consider a similar approach using different information measures to improve GP for other types of problems, especially classification.

- An interesting idea is to study the robustness of SGP against different types of noise. As a preliminary investigation, we added 10% and 20% Gaussian noise to the targets (outputs of the training data), and evaluated the error over the test data. We performed the experiment over the 6 benchmarks of the paper, for both GP and SGP. For 10% Gaussian noise, neither GP nor SGP is disturbed much. However, for 20% Gaussian noise, SGP significantly performed better, especially for the hard problems such as Benchmark 4.

For better handling of the noise, it is possible to introduce some new operators, such as the ϵ-VB editing operator as follows: Every subtree whose variance of its root is less than ϵ is replaced with the mean of its root. Investigating the robustness of SGP and the performance of such operators against various types of noise is a good idea for the future works.

References

[1] E. Alfaro-Cid, A. Esparcia-Alcázar, K. Sharman, F. F. d. Vega, and J. J. Merelo. Prune and plant: A new bloat control method for genetic programming. In *HIS '08: Proceedings of the 2008 8th International Conference on Hybrid Intelligent Systems*, pages 31–35, Washington, DC, USA, 2008. IEEE Computer Society.

[2] E. Alfaro-Cid, J. J. Merelo, F. F. de Vega, A. I. Esparcia-Alcázar, and K. Sharman. Bloat control operators and diversity in genetic programming: A comparative study. *Evol. Comput.*, 18(2):305–332, 2010.

[3] L. Altenberg et al. The evolution of evolvability in genetic programming. *Advances in genetic programming*, 3:47–74, 1994.

[4] M. Amir Haeri, M. M. Ebadzadeh, and G. Folino. Statistical genetic programming: The role of diversity. In V. Snel, P. Krmer, M. Kppen, and G. Schaefer, editors, *Soft Computing in Industrial Applications*, volume 223 of *Advances in Intelligent Systems and Computing*, pages 37–48. Springer International Publishing, 2014.

[5] F. Archetti, S. Lanzeni, E. Messina, and L. Vanneschi. Genetic programming for human oral bioavailability of drugs. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 255–262, New York, NY, USA, 2006. ACM.

[6] A. Ashour, L. Alvarez, and V. Toropov. Empirical modelling of shear strength of rc deep beams by genetic programming. *Computers & structures*, 81(5):331–338, 2003.

[7] R. Azad and C. Ryan. Variance based selection to improve test set performance in genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1315–1322. ACM, 2011.

[8] K. Badran and P. I. Rockett. The roles of diversity preservation and mutation in preventing population collapse in multiobjective genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1551–1558. ACM, 2007.

[9] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler. Multiobjective genetic programming: Reducing bloat using spea2. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 536–543. IEEE, 2001.

[10] C. J. Burgess and M. Lefley. Can genetic programming improve software effort estimation? a comparative evaluation. *Information and Software Technology*, 43(14):863–873, 2001.

[11] E. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *Trans. Evol. Comp*, 8(1):47–62, 2004.

[12] M. Castelli, L. Manzoni, S. Silva, and L. Vanneschi. A quantitative study of learning and generalization in genetic programming. *Genetic Programming*, pages 25–36, 2011.

[13] M. Castelli, L. Vanneschi, and S. Silva. Semantic search-based genetic programming and the effect of intron deletion. *IEEE transactions on cybernetics*, 44(1):103–113, 2014.

[14] J. M. Daida, R. R. Bertram, S. A. Stanhope, J. C. Khoo, S. A. Chaudhary, O. A. Chaudhri, and J. A. I. Polito. What makes a problem gp-hard? analysis of a tunably difficult problem in genetic programming. *Genetic Programming and Evolvable Machines*, 2(2):165–191, 2001.

[15] J. M. Daida, H. Li, R. Tang, and A. M. Hilss. What makes a problem gp-hard? validating a hypothesis of structural causes. In *GECCO'03: Proceedings of the 2003 international conference on Genetic and evolutionary computation*, pages 1665–1677, Berlin, Heidelberg, 2003. Springer-Verlag.

[16] E. D. De Jong and J. B. Pollack. Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines*, 4(3):211–233, 2003.

[17] S. Dignum and R. Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1588–1595, New York, NY, USA, 2007. ACM.

[18] M. Ebner. On the search space of genetic programming and its relation to nature's search space. In *Proceedings of the 1999 Congress on Evolutionary Computation*, CEC '99, 1999.

[19] A. Ekárt and S. Németh. Maintaining the diversity of genetic programs. *Genetic Programming*, pages 122–135, 2002.

[20] A. Ekárt and S. Németh. Maintaining the diversity of genetic programs. In J. Foster, E. Lutton, J. Miller, C. Ryan, and A. Tettamanzi, editors, *Genetic Programming*, volume 2278 of *Lecture Notes in Computer Science*, pages 122–135. Springer Berlin / Heidelberg, 2002.

[21] A. Ekárt and S. Z. Nemeth. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73, 2001.

[22] P. A. Estévez, M. Tesmer, C. A. Perez, and J. M. Zurada. Normalized mutual information feature selection. *Neural Networks, IEEE Transactions on*, 20(2):189–201, 2009.

[23] C. Gagné, M. Schoenauer, M. Parizeau, and M. Tomassini. Genetic programming, validation sets, and parsimony pressure. In *European Conference on Genetic Programming*, pages 109–120. Springer, 2006.

[24] M. Graff, H. J. Escalante, J. Cerda-Jacobo, and A. Avalos Gonzalez. Models of performance of time series forecasters. *Neurocomputing*, 122:375–385, 2013.

[25] M. Graff and R. Poli. Practical model of genetic programming's performance on rational symbolic regression problems. In *Genetic Programming*, pages 122–133. Springer, 2008.

[26] M. Graff and R. Poli. Practical performance models of algorithms in evolutionary program induction and other domains. *Artificial Intelligence*, 174(15):1254–1276, 2010.

[27] S. Gustafson, E. K. Burke, and N. Krasnogor. On improving genetic programming for symbolic regression. In *Congress on Evolutionary Computation*, pages 912–919, 2005.

[28] S. Gustafson, A. Ekárt, E. Burke, and G. Kendall. Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 5(3):271–290, 2004.

[29] N. T. Hien and X. H. Nguyen. Learning in stages: A layered learning approach for genetic programming. In *RIVF*, pages 1–4, 2012.

[30] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 184–192, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[31] M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *Genetic Programming*, pages 70–82. Springer, 2003.

[32] C. Kennedy and C. Giraud-Carrier. A depth controlling strategy for strongly typed evolutionary programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, page 879885. Morgan Kaufmann, 1999.

[33] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[34] J. R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT Press, Cambridge, MA, USA, 1994.

[35] K. Krawiec and P. Lichocki. Approximating geometric crossover in semantic space. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 987–994. ACM, 2009.

[36] K. Krawiec and T. Pawlak. Approximating geometric crossover by semantic backpropagation. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 941–948. ACM, 2013.

[37] W. B. Langdon and R. Poli. Fitness causes bloat. In *Soft Computing in Engineering Design and Manufacturing*, pages 23–27. Springer-Verlag, 1997.

[38] W. B. Langdon and R. Poli. Genetic programming bloat with dynamic fitness. In *Proceedings of the First European Workshop on Genetic Programming*, pages 97–112, London, UK, 1998. Springer-Verlag.

[39] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. *The evolution of size and shape*, pages 163–190. MIT Press, Cambridge, MA, USA, 1999.

[40] S. Luke and L. Panait. Fighting bloat with nonparametric parsimony pressure. In *PPSN VII: Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, pages 411–421, London, UK, 2002. Springer-Verlag.

[41] S. Luke and L. Panait. Lexicographic parsimony pressure. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[42] S. Luke and L. Panait. A comparison of bloat control methods for genetic programming. *Evol. Comput.*, 14(3):309–344, 2006.

[43] S. Mahler, D. Robilliard, and C. Fonlupt. Tarpeian bloat control and generalization accuracy. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 141–141. Springer Berlin / Heidelberg, 2005.

[44] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, et al. Genetic programming needs better benchmarks. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798. ACM, 2012.

[45] N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 303–309, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[46] A. Moraglio, K. Krawiec, and C. G. Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature-PPSN XII*, pages 21–31. Springer, 2012.

[47] M. Naoki, B. Mckay, N. Xuan, E. Daryl, and S. Takeuchi. A new method for simplifying algebraic expressions in genetic programming called equivalent decision simplification. In *Proceedings of the 10th International Work-Conference on Artificial Neural Networks: Part II: Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, IWANN '09, pages 171–178, Berlin, Heidelberg, 2009. Springer-Verlag.

[48] Q. U. Nguyen, N. T. Hien, N. X. Hoai, and M. O'Neill. Improving the generalisation ability of genetic programming with semantic similarity based crossover. In *Genetic Programming*, volume 6021 of *Lecture Notes in Computer Science*, pages 184–195. Springer Berlin / Heidelberg, 2010.

[49] Q. U. Nguyen, N. X. Hoai, M. O'Neill, R. I. McKay, and E. Galván-López. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

[50] Q. U. Nguyen, T. A. Pham, X. H. Nguyen, and J. McDermott. Subtree semantic geometric crossover for genetic programming. *Genetic Programming and Evolvable Machines*, 17(1):25–53, 2016.

[51] Q. U. Nguyen, C. D. Truong, X. H. Nguyen, and M. O'Neill. Guiding function set selection in genetic programming based on fitness landscape analysis. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 149–150. ACM, 2013.

[52] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, 1995.

[53] M. O'Neill, L. Vanneschi, S. Gustafson, and W. Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):339–363, Sept. 2010.

[54] T. P. Pawlak, B. Wieloch, and K. Krawiec. Semantic backpropagation for designing search operators in genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(3):326–340, 2015.

[55] S. Picek and M. Golub. The new negative slope coefficient measure. In *Proceedings of the 10th WSEAS international conference on evolutionary computing*, pages 96–101, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

[56] R. Poli and M. Graff. There is a free lunch for hyper-heuristics, genetic programming and computer scientists. In *Proceedings of the 12th European Conference on Genetic Programming*, EuroGP '09, pages 195–207, Berlin, Heidelberg, 2009. Springer-Verlag.

[57] R. Poli, W. B. Langdon, and S. Dignum. On the limiting distribution of program sizes in tree-based genetic programming. In *Proceedings of the 10th European conference on Genetic programming*, EuroGP'07, pages 193–204, Berlin, Heidelberg, 2007. Springer-Verlag.

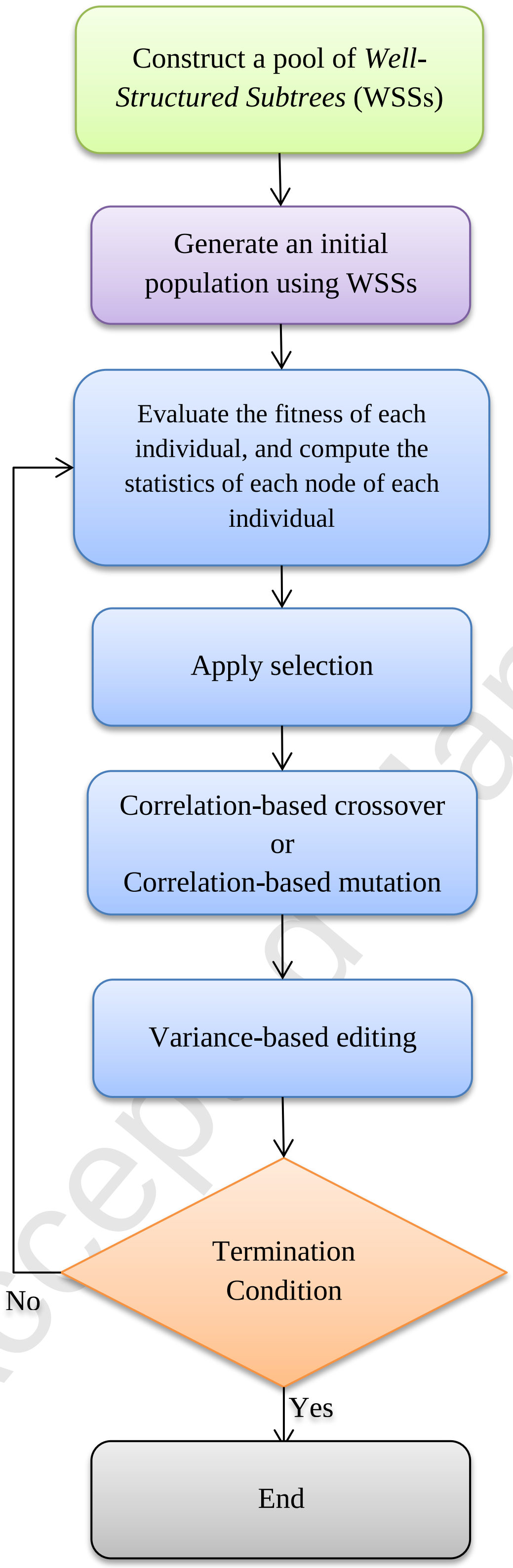
[58] R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part ii. *Evol. Comput.*, 11:169–206, May 2003.

[59] R. Poli and N. F. McPhee. Parsimony pressure made easy. In *Proceedings of the 10th annual conference on Genetic and evolutionary*

computation, GECCO '08, pages 1267–1274, New York, NY, USA, 2008. ACM.
[60] R. Poli and L. Vanneschi. Fitness-proportional negative slope coefficient as a hardness measure for genetic algorithms. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1335–1342, New York, NY, USA, 2007. ACM.
[61] J. Rosca. Entropy-driven adaptive representation. In *Proceedings of the workshop on genetic programming: From theory to real-world applications*, volume 9, pages 23–32. Tahoe City, California, USA, 1995.
[62] D. A. Savic, G. A. Walters, and J. W. Davidson. A genetic programming approach to rainfall-runoff modelling. *Water Resources Management*, 13(3):219–231, 1999.
[63] S. Silva and J. Almeida. Dynamic maximum tree depth: a simple technique for avoiding bloat in tree-based gp. In *GECCO'03: Proceedings of the 2003 international conference on Genetic and evolutionary computation*, pages 1776–1787, Berlin, Heidelberg, 2003. Springer-Verlag.
[64] S. Silva and E. Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179, 2009.
[65] S. Silva and L. Vanneschi. Operator equalisation, bloat and overfitting: a study on human oral bioavailability prediction. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1115–1122. ACM, 2009.
[66] S. Silva and L. Vanneschi. State-of-the-art genetic programming for predicting human oral bioavailability of drugs. *Advances in Bioinformatics*, pages 165–173, 2010.
[67] G. Smits, A. Kordon, K. Vladislavleva, E. Jordaan, and M. Kotanchek. Variable selection in industrial datasets using pareto genetic programming. In *Genetic Programming Theory and Practice III*, pages 79–92. Springer, 2006.
[68] T. Soule and J. A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *In 1998 IEEE International Conference on Evolutionary Computation*, pages 781–186. IEEE Press, 1998.
[69] M. Sprogar and V. Podgorelec. Incremental approach to structurally difficult problems in genetic programming. *Elektronika ir Elektrotechnika*, 20(5):154–157, 2014.
[70] M. Tomassini, L. Vanneschi, P. Collard, and M. Clergue. A study of fitness distance correlation as a difficulty measure in genetic programming. *Evol. Comput.*, 13:213–239, June 2005.
[71] L. Trujillo, Y. Martínez, E. Galván-López, and P. Legrand. Predicting problem difficulty for genetic programming applied to data classification. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1355–1362. ACM, 2011.
[72] L. Trujillo, S. Silva, P. Legrand, and L. Vanneschi. An empirical study of functional complexity as an indicator of overfitting in genetic programming. *Genetic Programming*, pages 262–273, 2011.
[73] L. Vanneschi, M. Tomassini, P. Collard, and S. Vrel. Negative slope coefficient: A measure to characterize genetic programming fitness landscapes. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekrt, editors, *Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin / Heidelberg, 2006.
[74] E. J. Vladislavleva, G. F. Smits, and D. Den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *Trans. Evol. Comp.*, 13:333–349, April 2009.
[75] J. R. Woodward. Modularity in genetic programming. In *Proceedings of the 6th European conference on Genetic programming*, EuroGP'03, pages 254–263, Berlin, Heidelberg, 2003. Springer-Verlag.
[76] I. C. Yeh. Concrete Compressive Strength Data Set. http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength. [Online; accessed 03-July-2013].
[77] M. Zhang, P. Wong, and D. Qian. Online program simplification in genetic programming. In *Proceedings of the 6th international conference on Simulated Evolution And Learning*, SEAL'06, pages 592–600, Berlin, Heidelberg, 2006. Springer-Verlag.

Highlights

- A new genetic programming algorithm called Statistical Genetic Programming (SGP) is proposed.
- Well-structured and semi–well-structured trees are defined.
- SGP uses statistical information to generate some well-structured subtrees and uses these subtrees to generate the initial population.
- Correlation-based crossover and correlation-based mutation operators are defined and used to explore the search space.
- Variance-based editing is defined for removing the introns.


Construct a pool of *Well-Structured Subtrees* (WSSs)
Generate an initial population using WSSs
Evaluate the fitness of each individual, and compute the statistics of each node of each individual
Apply selection
Correlation-based crossover or Correlation-based mutation
Variance-based editing
Termination Condition
No
Yes
End