Semantic Linear Genetic Programming for Symbolic Regression

Zhixing Huang, Yi Mei, Senior Member, IEEE, and Jinghui Zhong, Senior Member, IEEE

Abstract—Symbolic regression is an important problem with many applications such as automatic programming tasks and data mining. Genetic programming is a commonly used technique for symbolic regression. In the past decade, a branch of genetic programming that utilizes the program behavior to guide the search, called semantic genetic programming, has achieved great success in solving symbolic regression problems. However, existing semantic genetic programming methods only focus on the tree-based chromosome representation, and usually encounter the bloat issue and unsatisfactory generalization ability. To address these issues, we propose a new semantic linear genetic programming (SLGP) algorithm. In SLGP, we design a new chromosome representation to encode the programs and semantic information in a linear fashion. To utilize the semantic information more effectively, we further propose a novel semantic genetic operator, namely, mutate-and-divide propagation, to recursively propagate the semantic error within the linear program. The empirical results show that the proposed method has better training and test errors than the state-of-the-art algorithms in solving symbolic regression problems, and can achieve a much smaller program size.

Index Terms—Symbolic Regression, Genetic Programming, Mutate-and-divide Propagation.

I. INTRODUCTION

SYMBOLIC regression (SR) is a task to synthesize arithmetic formulae automatically, which is an important ability for computers to perform data mining. It aims to construct a formula, based on a given finite set of primitives, to implement the transformation between input and output data to fit the training data as much as possible. SR has various prospective applications such as knowledge discovery [1], [2], software development and maintenance [3], [4], industrial procedure design [5], [6], and circuit design [7].

Genetic programming (GP) [8], [9] is a major technique to solve SR. It encodes a program into a syntax tree or a series of instructions, and employs various genetic operators (e.g., mutation and crossover) to generate new programs to search the solution space. After generations of the evolution, GP outputs the best program.

Considering semantic information during the GP evolution [10] has become a popular method to improve the search effectiveness of GP [11]–[15]. Semantic information is a kind of phenotype of GP individuals. Different from syntactic information (i.e., tree structure), semantic information represents GP program behavior which is resistant to the bias of introns in the genome. The semantic information of an operation (e.g., “Input1 + Input2”) is defined as the execution output by applying the operation to the input cases. Since there are usually multiple input cases, the semantic information of an operation is usually denoted as an output vector. Once a program is executed, the semantic information of different operations in the program compose the behavior trace (intermediate outputs) of the program, which is also called the semantic context. By considering the semantic information besides conventional syntactic structures during evolution, semantic genetic programming (SGP) shows a faster training efficiency and better test performance than many other non-semantic GP methods [16], [17]. SGP has become one of the most effective methods to solve symbolic regression problems [18], [19].

There are a lot of different ways to utilize the semantic information up to date, including regarding the semantic space as a unimodal cone [20], aligning semantic error vectors [18], and back-propagating the semantic error [19]. However, all these SGP methods are implemented based on tree-based GP. Though they are shown to be quite effective in approximating the target output, they often suffer a severe bloat effect, and over-fit to the training data. There have been some methods aiming at improving the generalization ability of SGP. For example, Chen et al. [21] proposed an angle-awareness geometric operator to produce smaller programs to approximate the target semantics. Besides, many tree-based SGP methods try to set a small number of generations to prevent the over-fitting of tree-based programs. But limited by the exponential size increase of tree-based programs, these methods do not obtain a satisfactory improvement.

On the other hand, linear GP [22], [23] is reported to have much simpler and more compact program representation than tree-based GP [24], [25]. The program structures of linear GP can also be analyzed more efficiently and completely than tree-based GP and has been a popular system for evolution analysis [26]–[28]. However, to the best of our knowledge, no study has considered semantic information in linear GP yet. To utilize the potential advantages of linear GP to overcome the limitations of SGP, we combine linear GP with the semantic information and propose a new semantic linear genetic programming (SLGP) algorithm. It is worth mentioning that the
linear representation can facilitate us to design more effective semantic operators for SGP. Based on the linear representation, we develop a very effective mutate-and-divide propagation operator. The major contributions of this paper are as follows: 1) We introduce the chromosome representation of linear GP into SGP. The chromosome representation in our method encodes a program into a number of instructions in a linear fashion. With the linear representation, the program can be updated flexibly, and we can make a simple but effective control to variation step size. To reduce the bloat effect, we limit both the number and the length of the instructions in a program. 2) To effectively utilize the semantic information and update programs based on this linear chromosome representation, a novel mutate-and-divide propagation (MDP) scheme is proposed. MDP adopts a divide-and-conquer strategy. In each reproduction, MDP is recursively applied to mutate an instruction based on the semantic error, and further divide the program into two sub-programs which can be further updated by MDP. Thus, all instructions in the program can be updated according to the semantic error.

The rest of the paper is organized as follows. The related work is introduced in Section II. Then, the proposed SLGP is described in detail in Section III. Afterwards, the experimental results, discussions, and analyses are given in Section IV. Finally, the conclusions are drawn in Section V.

II. RELATED WORK
A. Genetic Programming For Symbolic Regression
GP is a prevalent method to solve SR problems. Up to now, GP has been developed extensively to pursue a better performance. For example, from the perspective of chromosome representation, tree-based GP [8], linear GP [22], and Cartesian GP [29], are proposed to fit different tasks. Tree-based GP encodes a program or formula into a syntax tree, and it is one of the most common representations in SR problems. On the contrary, linear GP represents a program by a sequence of assembly instructions, and Cartesian GP is a graph-based GP which represents programs by computation nodes in a two-dimensional grid system. In this paper, a new kind of linear GP is developed. In literature, linear GP has been successfully applied to classification [30], [32] and symbolic regression [25], [33], but none of the existing work consider the semantic information in linear GP individuals. Additionally, some embedded or component-based representations, like automatically defined function [34], are also designed to address the bloat effect in GP.

Based on these chromosome representation, the reproduction of GP also underwent a lot of improvement. For example, Forstenlechner et al. [35] and Saber et al. [36] proposed a grammar-guided GP to utilize the grammar constraints in producing offspring. Auger et al. [37] combined GP with surrogate models to relieve the time consumption in fitness evaluation. Apart from these reproduction mechanism, transfer learning [38], [39] and memetic algorithm [40] are also introduced into GP. Among all these variants, SGP is one of the most important branches of GP and it has two main approaches to utilize the semantic information [41] (i.e., the indirect and direct paradigm). Indirect semantic paradigm acts on the syntax of GP individuals and indirectly promotes semantic behaviors based on survival criteria. Direct semantic paradigm, on the other hand, acts directly on the semantics of GP individuals by genetic operators.

B. Indirect Semantic Genetic Programming
The indirect SGP utilizes the semantic information in two main ways. First, it improves the semantic diversity of the population by guaranteeing the distinct semantic behavior among individuals. Second, it improves the semantic locality of offspring and thereby can smoothen the convergence, by forcing GP to search the neighbor semantic areas of best-so-far individuals.

For improving semantic diversity, Beadle and Johnson [42]–[44] proposed the semantic-driven genetic operators (i.e., mutation, crossover, and initialization) which force the genetic operators to produce solutions with distinct semantics. The offspring will be discarded if their semantics are duplicated with that of the existing solutions. The similar idea is also adopted by Nguyen et al. in the semantic aware crossover [16], in which a semantic equivalence is checked to find sub-trees with distinct semantics before sub-tree crossover.

These genetic operators can search sub-trees with unequal semantics and encourage the population to search new solution space, but cannot guarantee the effectiveness of offspring, and thereby weaken the convergence ability. To produce more effective offspring, the semantic locality is developed. Based on the semantic locality, Nguyen et al. [45] proposed a semantic similarity distance to measure the semantic distance between sub-trees and extended the semantic aware crossover into a semantic similarity-based crossover which swaps sub-trees with the most similar but different semantics to maintain the effectiveness of offspring. Based on the idea of the semantic similarity-based crossover, Nguyen et al. [46] further proposed an update version with adaptive similarity threshold to reduce the computation burden in selecting proper sub-trees, and proposed another improved version with most similar sub-tree selection [47]. A semantic similarity-based mutation was also proposed by Nguyen et al. [48].

C. Direct Semantic Genetic Programming
Geometric semantic genetic programming (GSGP) [20] is a representative work of the direct SGP. GSGP is designed based on a geometric theory [20]. The theory sees the fitness landscape of any problem as a conic landscape. The tip of the landscape is the minimum error, which means the semantics of a GP individual is perfectly matched with the target semantics. If the geometric crossover is the only search operator used, the offspring should lie in a convex hull formed by the semantics of the parent population.

To locate the target semantics, GSGP tries to shrink the convex hull by re-combining the existing solutions and forces the population to converge to the target semantics.

Based on the rationale of GSGP, more methods have been proposed. To accelerate the shrinkage process of the convex
hull. McDermott et al. [49] introduced the idea of memetic algorithm into GSGP by finding the optimal mutation step each time the one-tree GSGP mutation operator performs, and Castelli et al. [50] introduced a crossover and mutation rate adaptation mechanism into GSGP. Harra et al. [51], [52] also proposed a parent selection method to find pairs of parents whose straight line connection is closest to the target semantics, and thus accelerate the shrink process. Since the crossover operator in conventional GSGP can only produce offspring whose semantics is in the convex hull and it is not guaranteed that the convex hull definitely contain the target semantics, Oliveira et al. [53] proposed a dispersion operator to disperse individuals in the dense space around the target semantics by a multiplicative factor to improve the effectiveness of GSGP.

Nevertheless, the solutions of GSGP often have an unmanageable size and the shrinkage process of the convex hull is not efficient enough. To solve the above issues, the angle aware operators of GSGP are proposed. For example, Ruberto and Vanneschi et al. [18], [54] proposed a new type of GSGP, namely, error space alignment GP (ESAGP). Rather than shrinking the convex hull formed by population semantics, ESAGP defines error vector and error space, and aims at searching individuals with optimally aligned or coplanar error vectors. To release the limitation of the prefixed attractor in ESAGP, Castelli et al. [55] proposed a pair optimization GP whose individuals contains more than one program, and regarded the angle between the programs as fitness. An improved ESAGP called nest align GP [56] is also proposed to avoid programs with small error vector angle but huge error vector magnitude by a nested tournament selection. The angle awareness idea is also adopted by Chen et al. [57], [58]. In [57], They proposed an angle aware geometric crossover to utilize parents with a large angle to produce offspring. In [58], They proposed a perpendicularly crossover and random segment mutation to identify the new desired semantics based on the semantic angle information. These two methods can improve the generalisation of conventional GP methods and reduce the bloat effect.

Although GSGP can approximate the target semantic more effectively than standard GP methods, there is a crucial fact that programs of GSGP are actually linear combinations of random parts [59]. Moreover, these programs can be constructed more effectively by a simple linear combination algorithm with a guarantee of zero error within a polynomial time. The rationale of linear combination is not only the essential reason that GSGP suffers from a severe bloat effect, but also leads to overfit issues.

Using approximate methods to utilize semantic information is a good way to avoid linearly combining random programs, such as random desired operator (RDO) [19]. RDO firstly defines the inverse operations for each function primitives and establish a sub-tree library to collect the sub-trees with distinct semantics. Given a target semantics, RDO propagates the semantic error from the root to a random internal node based on the predefined inverse operation. Then, RDO method searches a “best-fit” sub-tree from its sub-tree library to approximate the desired semantics. Since RDO method is directly oriented by the target semantics and it replaces sub-trees to approximate desired semantics, rather than concatenating existing expressions directly, RDO method is more efficient than conventional GSGP and relieves the bloat effect.

RDO has been extended by some methods. For example, Quang et al. [60] developed RDO to solve the ephemeral constant problems by mixed-integer linear programming. The cooperation between RDO and linear scaling techniques is also investigated by Virgolin et al. [61]. Franson et al. [62] combined the memetic algorithm with RDO by searching the most suitable internal node as the sub-root to perform crossover. Currently, backpropagating semantic error to obtain the sub-tree has been applied in many state-of-the-art SGP methods. For example, Chen et al. [21] combined the RDO with ESAGP in offspring generation, to propose an angle-driven GSGP which successfully improves the generalization and further relieves the bloat effect. Pawlak et al. [63] also adopted RDO operator in their newly proposed geometric mutation and crossover operator to generate offspring. Despite the success of RDO methods, existing RDO methods still suffer from a bloat issue even under the input-output data from a simple formula.

III. SEMANTIC LINEAR GENETIC PROGRAMMING

SLGP is based on a number of newly proposed concepts about semantics, such as semantic matrix and semantic context. Therefore, we will start with introducing the individual representation and new semantic concepts in SLGP. Then, we will describe the algorithm in detail, including the overall framework and the components.

A. Individual Representation and Semantics

As a linear GP, each individual in SLGP is represented as a sequence of instructions $f = [f_1, ..., f_{|f|}]$, where each instruction is a list of functions and terminals. Specifically, each instruction consists of three parts “R | H | T”, where “R” contains a receiving register (e.g., “V_0 = ”). “H” is an arithmetical part consisting of both functions and terminals, and “T” is an operand part consisting of only terminals. The final output of the program is the value of the first register $V_0$. An example is shown in Fig. 1 in which the first instruction “$V_0 = \sin(I_0 + I_1)$” is decoded as “$V_0 = \sin(I_0 + I_1)$”, where $I_0$ and $I_1$ indicate the $x_0$ and $x_1$ values from the training data.

For SLGP individual representation, a number of registers $[V_0, V_1, ..., V_N]$ is defined to store the data during the computation. For example, two registers $V_0$ and $V_1$ are used in the individual in Fig. 1. Given $N$ training cases $X = [x_1, ..., x_N]$ and $K$ registers, we consider the data stored during the computation as a semantic matrix $S_{N \times K}$, where $s_{i,j}$ indicates the value of register $V_j$ given the inputs of the case $x_i$. Given an input data $X$ and semantic matrix $S_{in}$ of an instruction $f$, we denote the output semantic matrix produced by $f$ as $S_{out} = f(S_{in}, X)$. Take the first instruction “$f_1 : V_0 = \sin(I_0 + I_1)$” in Fig. 1 as an example. If there are four input cases (e.g., [(1, 3), (2, 5), (3, 4), (4, 7)]) and two registers, the first instruction accepts two matrices, one for semantic matrix and the other for input data. Specifically, the semantic matrix
is initially set to $O_{x \times 2}$, i.e., the two registers are initialized to zero for each independent execution. After the first instruction that changes $V_0$ based on the input values of the four cases respectively, the semantic matrix becomes

$$S_1 = f_1 \left( 0_{4 \times 2}, \begin{bmatrix} 1.3 \\ 2.5 \\ 3.4 \\ 4.7 \end{bmatrix} \right) = \begin{bmatrix} 0.07, 0 \\ 0.12, 0 \\ 0.12, 0 \\ 0.19, 0 \end{bmatrix}.$$  

A semantic context of an individual $f = [f_1, \ldots, f_I]$ is defined as a sequence of semantic matrices $S(f) = [S_{in}, S(f_1), \ldots, S(f_I), S_{target}]$, where $S_{in}$ stands for the input semantic matrix (initial register values) of the individual, $S(f_i)$ is the semantic matrix after executing the instruction $f_i$, and $S_{target}$ indicates the target semantic matrix that the individual aims to output.

B. SLGP Framework

The flowchart of the proposed SLGP\footnote{The source code of SLGP can be downloaded from \url{http://github.com/Zhixing1020/SemanticLGP_for_SR}.} is shown in Fig. 2 and its pseudo code is shown in Algorithm \footnote{\texttt{rand}(a, b) means a random sample from uniform distribution from a to b, and \texttt{rand\_int}(a, b) means a random integer from [a, b]. The similar notations are used in the rest of the paper.} Given the training data $[X, y] \in \mathbb{R}^{N \times (M+1)}$ with $N$ instances/cases and $M$ variables ($x_i = [x_{i1}, \ldots, x_{iM}]$ is the $i$th case, and $y_i$ is its target output), SLGP finds the best SR model $f^*$ that maps from a $M$-dimensional input vector to a real-valued output.

At the beginning, a population $F$ of SR models and a semantic library $L$ are randomly initialized. The semantic library consists of three main components: (1) a set of input semantic matrices $S_{in}(L)$, (2) a set of instructions $\Phi(L)$, and (3) the output semantic matrix of each instruction given each input semantic matrix, i.e., $S_{out}(L) = \{ f(S_{in}) | f \in \Phi(L), S_{in} \in S_{in}(L) \}$. Hereafter, we simplify $f(S_{in}, X)$ as $f(S_{in})$, since the training data $X$ is the same for all possible programs. Fig. 3 illustrates the semantic library. The input semantic matrices $S_{in}(L)$ consists of a fixed number of randomly sampled semantic matrices within the given data range. The output semantic matrices $S_{out}(L)$ are calculated based on $S_{in}(L)$ for every instruction. Instructions $\Phi(L)$ are initialized randomly to fill up $\Phi(L)$. If $f(S_{in})$ is not unique in $S_{out}(L)$, $f$ will be removed from $\Phi(L)$. Then, each initial individual is evaluated by the training data, and its semantic context is also calculated. The evaluation will be described in Section III-C.

In each generation, an offspring population is generated. Each offspring is generated from one individual in the population, by the newly proposed mutate-and-divide propagation (MDP) operator with 50% probability, and by the differential evolution (DE) genetic operators \footnote{Fig. 3. The design of the semantic library in SLGP.} with the other 50%
that we denote \( f \) instructions in an individual during the recursion. Let a sequence of instructions \( f \) in \( \ell \) length denote a subsequence of \( f \) as \( f_{\ell} \) and a semantic library \( \mathcal{L} \) as inputs, and returns a new sequence of instructions \( f' \) and its desired output semantic matrix \( \mathcal{S}_{\text{desire}} \) to produce its target output semantic matrix. Note that we denote \( f \) as a sequence of instructions rather than an individual, since MDP can be applied to any sub-sequence of instructions in an individual during the recursion.

First, two indices \( i_1 \) and \( i_2 \) (\( 0 \leq i_1 < i_2 \leq |f| \)) are randomly sampled based on the step length \( \ell \). Specifically, the instruction \( f_{i_1} \) provides its output semantic matrix \( \mathcal{S}(f_{i_1}) \) to \( f_{i_2} \), and \( f_{i_2} \) is the instruction to be mutated. The two instructions split the instruction sequence into three sub-sequences \( f_1 = [f_1, \ldots, f_{i_1}] \), \( f_2 = [f_{i_1+1}, \ldots, f_{i_2-1}] \), and \( f_3 = [f_{i_2+1}, \ldots, f_{|f|}] \). The corresponding sequences of semantic matrices \( \mathcal{S}_1 \), \( \mathcal{S}_2 \) and \( \mathcal{S}_3 \) are obtained accordingly. The instruction \( f_{i_2} \) is going to be mutated, thus is not included in the subsequences.

**Algorithm 2: evaluate(\( f, X, y \))**

**Input:** The evaluated individual \( f = [f_1, \ldots, f_{|f|}] \), training data \( [X, y] \)

**Output:** The fitness \( f(it) \), its semantic context \( S(f) \)

1. Set \( S_0 = 0_{N \times K} \), \( S_{\text{target}} = [y, 0_{N \times (K-1)}] \).
2. for \( i = 1 \rightarrow |f| \) do
   3. \( S(f_i) = S_i = f_i(S_{i-1}, X) \);
   4. \( \text{NNE} = \text{NNE} + \text{size}(f_i) \);
5. \( y_{\text{out}} = |S(f_i)|_{i,o} \);
6. \( f(it) = \text{RMSE}(y_{\text{out}}, y) \);
7. \( \text{return } f(it), S(f) = [S_0, S(f_1), \ldots, S(f_{|f|}), S_{\text{target}}] ; \)

**C. Fitness Evaluation**

The fitness evaluation is described in Algorithm 2. The initial semantic matrix is set to zero (all the registers are initialized to be zero), and the target semantic matrix is set to \([y, 0_{N \times (K-1)}]\), i.e., the first register equals the target output, and all the other registers take zero values. Then, the output semantic matrix of each instruction \( f_i \) (\( i = 1, \ldots, |f| \)) is calculated by executing the instructions sequentially. The finally output is the first column \( (v_0) \) of the final semantic matrix, i.e., \( y_{\text{out}} = |S(f)|_{i,o} \). Then, the fitness is calculated as the root-mean-squared error (RMSE) between \( y_{\text{out}} \) and \( y \), which is a common performance metric in many symbolic regression literatures [21], [40], [64]:

\[
\text{RMSE}(y_{\text{out}}, y) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_{\text{out}, i} - y_i)^2} \quad (1)
\]

**D. Mutate-and-divide Propagation**

The MDP process is described in Algorithm 3. It takes a sequence of instructions \( f \), its semantic context \( S(f) \), a step length \( \ell \) and a semantic library \( \mathcal{L} \) as inputs, and returns a new sequence of instructions \( f' \) and its desired output semantic matrix \( \mathcal{S}_{\text{desire}} \) to produce its target output semantic matrix. Note that we denote \( f \) as a sequence of instructions rather than an individual, since MDP can be applied to any sub-sequence of instructions in an individual during the recursion.

First, two indices \( i_1 \) and \( i_2 \) (\( 0 \leq i_1 < i_2 \leq |f| \)) are randomly sampled based on the step length \( \ell \). Specifically, the instruction \( f_{i_1} \) provides its output semantic matrix \( S(f_{i_1}) \) to \( f_{i_2} \), and \( f_{i_2} \) is the goal to find the best instruction \( f' \in \Phi(\mathcal{L}) \) so that it can take the semantic matrix \( \mathcal{S}_{\text{in}} \) produced by \( f_1 \) (or the input semantic matrix \( \mathcal{S}_{\text{in}} \) if \( i_1 = 0 \)) as inputs, and approximate the target semantic matrix \( \mathcal{S}_{\text{target}} \). This is done by the function LibrarySearch(\( \cdot \)) which will be introduced in Section III-E. Note that using different \( f_{i_2} \) to provide \( \mathcal{S}_{\text{in}} \) rather than only considering \( \mathcal{S}_{\text{in}} \) encourages \( f_{i_2} \) to be mutated into more different instructions, which improves the exploration ability. The library search returns the best instructions from the semantic library \( \mathcal{L} \), its desired input matrix \( \mathcal{S}_{\text{in}} \), to produce the target semantic matrix, and its estimated output semantic matrix \( \mathcal{S}_{\text{out}} \) if given the input semantic matrix \( \mathcal{S}_{\text{in}} \). Ideally, if we find an instruction so that \( f'(\mathcal{S}_{\text{in}}) = \mathcal{S}_{\text{target}} \), then we have found the optimal sequence \( f' = [f_1, f'_2] \). However, due to the limited storage of the semantic library, it is hardly possible to find the ideal instruction. Instead, we usually have \( \mathcal{S}_{\text{desire}} \neq \mathcal{S}_{\text{in}} \) and \( \mathcal{S}_{\text{out}} \neq \mathcal{S}_{\text{target}} \). Therefore, it is necessary to recursively mutate the instructions in \( f_1, f_2 \) and \( f_3 \) to minimize the deviation from the expected behavior.

Since the MDP of the former subsequence requires the desired input semantic matrix of the latter subsequence, we conduct the propagation in a backward order. Specifically, we first modify \( f_3 \) by MDP, so that it takes the expected output semantic matrix \( \mathcal{S}_{\text{out}} \) of \( f' \) and targets to output the target semantic matrix \( \mathcal{S}_{\text{target}} \). Then, we modify \( f_2 \) by MDP, so that it takes the output semantic matrix of \( f_1 \) and outputs the desired semantic matrix of \( f' \). It also updates the desired input semantic matrix \( \mathcal{S}_{\text{desire}} \). Finally, we modify \( f_1 \) by MDP to take the overall input semantic matrix and output \( \mathcal{S}'_{\text{desire}} \).

After the recursive mutation, we finally obtain the new sequence \( f' \) by concatenating the modified subsequences, and its desired input semantic matrix \( \mathcal{S}_{\text{desire}} \).

**E. Semantic Library Search**

The semantic library search LibrarySearch(\( \mathcal{S}_{\text{in}}, \mathcal{S}_{\text{target}}, \mathcal{L} \)) aims to search in the semantic library \( \mathcal{L} \) to find the instruction \( f' \) whose behavior is closest to the semantic mapping \( f_{i_1} \rightarrow \mathcal{S}_{\text{target}} \), i.e., it takes \( \mathcal{S}_{\text{in}} \) and produces \( \mathcal{S}_{\text{target}} \). This can be formulated as the following problem:

\[
f' = \arg \min_{f \in \Phi(\mathcal{L})} \text{SAE}(f(\mathcal{S}_{\text{in}}), \mathcal{S}_{\text{target}}), \quad (2)
\]

where \( \text{SAE} \) is the sum of absolute error between two matrices, calculated as follows:

\[
\text{SAE}(A, B) = \frac{1}{N \cdot K} \sum_{i=1}^{N} \sum_{j=1}^{K} |a_{i,j} - b_{i,j}|. \quad (3)
\]

The calculation of \( f(\mathcal{S}_{\text{in}}) \) requires to execute the instruction \( f \), which may have a high time complexity in practice (e.g., if the instruction has a for loop). To address this issue, we
Algorithm 3: MDP(f, S(f), ℓ, L)

Input: A sequence of instructions \( f = \{f_1, \ldots, f_\ell\} \), its semantic context \( S(f) = \{S(f_1), \ldots, S(f_\ell), S_{\text{target}}\} \), a step length \( \ell \), and a semantic library \( L \).

Output: The new sequence of instructions \( f' \), the desired input semantic matrix \( S'_{\text{desire}} \) of \( f' \).

1. \( i_1 = \max(0, \text{rand_int}(\ell - 1)) \);
2. \( i_2 = \text{rand_int}(i_1 + 1, \ell) \);
3. \( f_1 = \{f_1, \ldots, f_{i_1}\}, S_1 = [S(f_1), \ldots, S(f_{i_1})] \);
4. \( f_2 = \{f_{i_1+1}, \ldots, f_{i_2-1}\}, S_2 = [S(f_{i_1+1}), \ldots, S(f_{i_2-1})] \);
5. \( f_3 = \{f_{i_2}, \ldots, f_\ell\}, S_3 = [S(f_{i_2}), \ldots, S(f_\ell)] \);
6. \( S'_0 = S_{\text{in}} \);
7. if \( i_2 > 0 \) then \( S'_1 = S(f_1) \);
8. \( f', S'_{\text{desire}}, S'_{\text{out}} = \text{LibrarySearch}(S'_0, S_{\text{target}}, L) \);
9. if \( f_3 \neq f' \) then
10. \( f', S'_{\text{desire}}, S'_{\text{out}} = \text{MDP}(f_3, S'_{\text{out}}, S_3, S_{\text{target}}, \ell, L) \);
11. if \( f_2 \neq f' \) then
12. \( f'_1, S'_{\text{desire}} = \text{MDP}(f_2, S'_1, S_2, S'_{\text{desire}}, \ell, L) \);
13. if \( f_1 \neq f'_1 \) then
14. \( f', S'_{\text{desire}} = \text{MDP}(f_1, S'_1, S_1, S'_{\text{desire}}, \ell, L) \);
15. return \( f', S'_{\text{desire}} \).

proposed to estimate \( f(S_{\text{in}}) \) based on the input-output mapping stored in the semantic library. Specifically, we first select the input semantic matrix \( S^* \in S_{\text{in}}(L) \) that is closest to \( S_{\text{in}} \) (i.e., \( S^* = \arg \min_{S \in S_{\text{in}}(L)} SAE(S, S_{\text{in}}) \)). Then, we conduct a projection as follows:

\[
\hat{f}(S_{\text{in}}) = f(S^*) + \nabla f(S^*) \circ (S_{\text{in}} - S^*),
\]

where

\[
\nabla f(S^*) = \frac{1}{|S_{\text{in}}(L)| - 1} \sum_{S \in S_{\text{in}}(L) \neq S^*} [f(S) - f(S^*)] \circ [S - S^*].
\]

The Hadamard product \( A \circ B \) and Hadamard division \( A \odot B \) between two matrices are defined as

\[
A \circ B = [a_{i,j} \times b_{i,j}], \quad A \odot B = [a_{i,j} / b_{i,j}]\text{.}
\]

Note that there are a large number of divisions in \( \nabla f(S^*) \), which is time-consuming especially when semantic matrices are relatively large. To further save computation time of the rough estimation, we replace \( [S - S^*] \) into its mean value \( \Delta S \), where

\[
\Delta S = \frac{1}{|S_{\text{in}}(L)| - 1} \sum_{S \in S_{\text{in}}(L) \neq S^*} [S - S^*].
\]

Thus we have

\[
\nabla f(S^*) = (\sum_{S \in S_{\text{in}}(L) \neq S^*} [f(S) - f(S^*)]) \odot (\sum_{S \in S_{\text{in}}(L) \neq S^*} [S - S^*]).
\]

In practice, it is too time consuming to exhaustively search in the library to find the best instruction due to the large number of instructions stored in the library. To address this issue, we simply adopt the Monte Carlo method to approximately find the best instruction. Specifically, we randomly sample an instruction from the library for \( \tau \) times. For each instruction, we estimate its output semantic matrix based on Eq. (4). If the estimated output semantic matrix is closer to the current best instruction, it replaces the current best instruction.

Note that the output estimation Eq. (4) might not be accurate. To reduce the bias caused by the inaccurate estimation, we introduce a probability \( \varepsilon \) to directly discard a randomly sampled instruction, if it has been selected many times before. Specifically, let \( n(f) \) be the number of times that \( f \) has been selected by LibrarySearch(·) during the evolutionary process, we define the discard probability as \( \varepsilon(f) = n(f)/\nu \times \text{popsize} \), where \( \nu \) is the frequency to update the library, and \( \text{popsize} \) is the population size. In other words, a larger \( n(f) \) leads to a higher probability to be discarded. In addition, we set an upper bound \( \theta \) to the probability, so that a good instruction always has some chance to be selected. In this case, we set \( \theta = 0.95 \), so that the discard probability can be no larger than 95%.

Finally, after the best instruction \( f' \) and its estimated output \( \hat{f}'(S_{\text{in}}) \) has been found, we need to calculate its desired input semantic matrix to produce the target output. We can follow the same inverse projection process as follows:

\[
\hat{f}'(S_{\text{in}}) = S^* + \nabla^{-1} f(S^*) \circ (S_{\text{target}} - f(S^*)),
\]

where

\[
\nabla^{-1} f(S^*) = (\sum_{S \in S_{\text{in}}(L) \neq S^*} [S - S^*]) \odot (\sum_{S \in S_{\text{in}}(L) \neq S^*} [f(S) - f(S^*)]).
\]

Algorithm 4 shows the pseudo code of the library search.

F. Library Update

The semantic library is updated periodically (i.e., every \( \nu \) generations) by adding/changing the instructions and their semantic matrices. First, a number of \( n_u \) new instructions are generated from the instructions in \( \Phi(L) \). For generating each new instruction, two parents \( p_1 \) and \( p_2 \) are selected from \( \Phi(L) \) by a size-20 tournament selection. It is expected that if an instruction has been selected more often, it is more likely to generate useful new instructions. However, we also need to reduce the bias caused by the inaccurate selection. Thus,
each candidate instruction $f'$ in the tournament pool has a probability of $\varepsilon(f')$ to be discarded directly. Among the remaining candidate instructions, the one with the largest $n(f')$ (number of times it has been selected by LibrarySearch(·)) is selected as the parent. Then, the new instruction is generated by applying genetic operators to the parents.

If the generated new instruction is not a duplicate in $\Phi(\mathcal{L})$, then it is added into $\Phi(\mathcal{L})$, and its output semantic matrices of $\mathcal{S}_{in}(\mathcal{L})$ are added into $\mathcal{S}_{out}(\mathcal{L})$. After the insertion, if $\Phi(\mathcal{L})$ exceeds the size limit $\rho$, then an instruction is selected by the size-20 tournament selection to be removed. Here, we aim to remove the instructions that have been rarely selected while reducing the selection bias. Thus, each candidate instruction $f'$ has a probability of $\varepsilon(f')$ to be discarded directly. Among the remaining ones, the one with the smallest $n(f')$ is selected. In the end, to increase exploration, $n(f)$ for each instruction in the library is decayed by the decay factor $\sigma$, i.e., $n(f) = n(f) \times \sigma$.

IV. EXPERIMENT SETUP

A. Datasets

The experiments consist of fourteen SR problems, including four real-world datasets. These test problems are selected from the recent studies [21], [63], [65]. They have various properties, such as high dimensionality, large real constant value, and different value domains. The details of the datasets are listed in Table I. For each method, 30 independent runs with different random seeds are performed.

In our experiment, we adopt the number of node evaluation (NNE) as the metric of computation resources. NNE counts the number of node (terminal and function) evaluations in a program for evaluating one semantic matrix. It is increased by the program size in each fitness evaluation of a program. Thus, a long/large program requires more NNE than a short/small program. NNE can help us make a fair comparison in term of computation resources, especially when the program size is significantly different and there exist a lot of additional node evaluations in genetic operators. The maximal number of NNE is set to $10^7$. An algorithm can also stop early when the root mean square error (RMSE) of the outputs (or the target registers) is less than $10^{-4}$ (also called “success” in our paper).

B. Comparison methods

To validate the performance of SLGP, four recently published methods are compared with SLGP. They are SL-GEP [64], GP with $\varepsilon$-lexicase selection (EPLEX) [67], competent GSGP (C-GSGP) [63], and angle-driven GSGP (ADGSGP) [21]. All of these algorithms are state-of-the-art GP variants in solving symbolic regression problems. SL-GEP adopts a gene expression chromosome representation and an automatically defined function structure to encode programs, and proposes a differential evolution-based genetic operator to evolve programs. EPLEX proposes a new selection paradigm for GP methods which enables more near-elite individuals to survive in the next population. Both of these techniques effectively relieves the bloat effect and meanwhile achieves a high success rate in SR problems. C-GSGP and ADGSGP are two latest SGP methods. C-GSGP firstly defines the principles of operator effectiveness based on semantic diversity. For example, a mutation is effective if and only if the produced offspring has distinct semantics with the ones of its parents. Based on the effectiveness principles, C-GSGP further proposed four semantic genetic operators (i.e., competent initialization.

---

### Algorithm 4: LibrarySearch($\mathcal{S}_{in}, \mathcal{S}_{target}, \mathcal{L}$)

**Input**: The input semantics $\mathcal{S}_{in}$, the target semantics $\mathcal{S}_{target}$, the semantic library $\mathcal{L}$

**Output**: The selected instruction $f' \in \mathcal{L}$, its desired input semantics $\mathcal{S}_{out}'$, its estimated output semantics $\mathcal{S}_{out}$

1. $\mathcal{S}' = \arg \min_{\mathcal{S} \in \mathcal{S} \in (\mathcal{L})} \text{SAE}(\mathcal{S}, \mathcal{S}_{in});$
2. Set $\delta' = +\infty;$
3. for $i = 1 \rightarrow \tau$
   4. Randomly select $f \in \mathcal{L};$
   5. Let $n(f)$ be the number of times that $f$ has been selected during the evolutionary process;
   6. $\varepsilon(f) = \min \left\{ \frac{n(f)}{n_{\text{popsize}}, \theta} \right\};$
   7. if rand$(0, 1) < 1 - \varepsilon(f)$ then
      8. Estimate $f(\mathcal{S}_{in})$ by Eq. (4);
      9. $\delta(f) = \text{SAE}(f(\mathcal{S}_{in}), \mathcal{S}_{target});$
     10. if $\delta(f) < \delta'$ then
      11. $f' = f$, $\delta' = \delta(f);$
12. $n(f') = n(f') + 1;$
13. Estimate $f'^{-1}(\mathcal{S}_{target})$ by Eq. (5);
14. return $f'$, $f'^{-1}(\mathcal{S}_{target})$, $f'(\mathcal{S}_{in});$

### Algorithm 5: LibraryUpdate($\mathcal{L}, n_{\text{in}}, \sigma$)

**Input**: Semantic library $\mathcal{L} = (\mathcal{S}_{in}(\mathcal{L}), \Phi(\mathcal{L}), \mathcal{S}_{out}(\mathcal{L}))$, number of new instructions $n_{\text{in}}$, decay factor $\sigma$

1. for $i = 1 \rightarrow n_{\text{in}}$
   2. Randomly select instructions $p_1, p_2 \in \Phi(\mathcal{L});$
   3. for $k = 1 \rightarrow 2$
      4. Randomly select $f' \in \Phi(\mathcal{L});$
      5. if rand$(0, 1) < 1 - \varepsilon(f')$ then
         6. if $n(f') > n(p_k)$ then
            7. $p_k = f' ;$
5. Generate a new instruction $f_i$ by applying genetic operators to $p_1$ and $p_2$;
   6. if $f_i \notin \Phi(\mathcal{L})$ then
      7. $\Phi(\mathcal{L}) = \Phi(\mathcal{L}) \cup f_i$, $n(f_i) = 0;$
   8. for $\mathcal{S} \in \mathcal{S}_{in}(\mathcal{L})$
      9. Calculate $f_i(\mathcal{S})$, $\mathcal{S}_{out}(\mathcal{L}) = \mathcal{S}_{out}(\mathcal{L}) \cup f_i(\mathcal{S});$
     10. $\text{NNE} = \text{NNE} + \text{size}(f_i);$ if $|\Phi(\mathcal{L})| > \rho$ then
   11. Randomly select $r \in \Phi(\mathcal{L});$
   12. for $j = 1 \rightarrow 20$
      13. Randomly select $f' \in \Phi(\mathcal{L});$
      14. if rand$(0, 1) < 1 - \varepsilon(f')$ then
         15. if $n(f') < n(r)$ then
            16. $r = f' ;$
   17. Remove $r$ from $\Phi(\mathcal{L});$
   18. for $\mathcal{S} \in \mathcal{S}_{out}(\mathcal{L})$
      19. Remove $r(\mathcal{S})$ from $\mathcal{S}_{out}(\mathcal{L});$
   20. for $f \in \Phi(\mathcal{L})$
      21. $n(f) = n(f) \times \sigma;$

Table I: The Test problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Function</th>
<th>#Features</th>
<th>Data range</th>
<th>#Points (Train, Test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Korz2</td>
<td>$f(x) = x^5 - 2x^3 + x$</td>
<td>1</td>
<td>[-1,1]</td>
<td>(20,1000)</td>
</tr>
<tr>
<td>Nguyen4</td>
<td>$f(x) = x^6 + x^5 + x^4 + x^3 + x^2 + x$</td>
<td>1</td>
<td>[-1,1]</td>
<td>(20,1000)</td>
</tr>
<tr>
<td>Nguyen5</td>
<td>$f(x) = \sin(x^2)\cos(x) - 1$</td>
<td>1</td>
<td>[-1,1]</td>
<td>(20,1000)</td>
</tr>
<tr>
<td>Nguyen7</td>
<td>$f(x) = \ln(x + 1) + \ln(x^2 + 1)$</td>
<td>1</td>
<td>[0,2]</td>
<td>(20,1000)</td>
</tr>
<tr>
<td>Keijzer11</td>
<td>$f(x_1, x_2) = 20(x_1 - 1)^2 + 10(x_2 - 1)^2$</td>
<td>2</td>
<td>[1,1]</td>
<td>(100,900)</td>
</tr>
<tr>
<td>R1</td>
<td>$f(x) = \frac{(x + 1)^3}{x^2 + 1}$</td>
<td>1</td>
<td>[-2,2]</td>
<td>(20,1000)</td>
</tr>
<tr>
<td>ModQua</td>
<td>$f(x) = 4x^4 + 3x^3 + 2x^2 + x$</td>
<td>1</td>
<td>[-2,2]</td>
<td>(20,1000)</td>
</tr>
<tr>
<td>Nonic</td>
<td>$f(x) = \sum_{i=1}^{9} x_i$</td>
<td>1</td>
<td>[-2,2]</td>
<td>(20,1000)</td>
</tr>
<tr>
<td>Hartman</td>
<td>$f(x) = -\exp(-\sum_{i=1}^{4} x_i^2)$</td>
<td>4</td>
<td>[0.2]</td>
<td>(100,900)</td>
</tr>
<tr>
<td>RatPol3D</td>
<td>$f(x) = 30 \left(\frac{5x_0 - 1}{6x_0 - 1}\right) x_1(1.2 - x_1)$</td>
<td>3</td>
<td>[0.05,2]</td>
<td>(300,2700)</td>
</tr>
</tbody>
</table>

Real-world problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Function</th>
<th>#Features</th>
<th>Data range</th>
<th>#Points (Train, Test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airfoil</td>
<td>unknown</td>
<td>5</td>
<td>-</td>
<td>(1127,376)</td>
</tr>
<tr>
<td>BHouse</td>
<td>unknown</td>
<td>13</td>
<td>-</td>
<td>(380,126)</td>
</tr>
<tr>
<td>Tower</td>
<td>unknown</td>
<td>25</td>
<td>-</td>
<td>(3749,1250)</td>
</tr>
<tr>
<td>CCN</td>
<td>unknown</td>
<td>124</td>
<td>-</td>
<td>(1661,554)</td>
</tr>
</tbody>
</table>

1. The notation "#" means "the number of".
2. The training instances for benchmark problems are sampled uniformly from the given interval. The test instances are sampled randomly from the same interval. The training and test instances in real-world problems are split randomly.

The different settings of parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Test values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>$NI$</td>
<td>3, 5, 7, 9</td>
</tr>
<tr>
<td>$HL$</td>
<td>2000, 5000, 7000, 10000</td>
</tr>
<tr>
<td>$\rho$</td>
<td>10, 50, 100</td>
</tr>
<tr>
<td>$\tau$</td>
<td>0.4, 0.6, 0.8, 1</td>
</tr>
</tbody>
</table>

competent tournament selection, competent mutation, and competent crossover) and replaces the corresponding operators of SGP. According to the experiment results, the joint method of these competent operators significantly outperform many other SGP methods, and has a smaller program size. On the other hand, ADGSGP incorporates RDO with the idea of error-space angle alignment, and proposes perpendicular crossover and random segment mutation to reproduce offspring. The empirical results also show that ADGSGP can find programs with significantly lower training and test error. The detailed parameter settings of these comparison methods are all set as their recommended values.

C. Parameter Settings

The proposed SLGP contains a number of parameters. Due to the new individual representation and components, SLGP has 8 new parameters: the number of registers $K$, maximal number of instructions $NI$, maximal length of instruction arithmetical part $HL$, size limit of the semantic library $\rho$, number of samples in the semantic library search $\tau$, library update period $\nu$, number of new instructions during library update $n_u$, and decay factor $\sigma$. To investigate the effect of these parameters, a parameter analysis experiment is conducted. First, we give these parameters a default setting based on the recommended settings of other GP and SGP methods. Then, for each SLGP new parameter, we select several other values around its default value, and compare among their performance. In the experiment, five SR problems (i.e., Nguyen5, Nguyen7, Keijzer11, Nonic, and Hartman) are adopted. These SR problems are selected because they have different number of features and data ranges, and thus different levels of difficulty. For each candidate value of each parameter, we run SLGP with this candidate parameter value, while the other parameters are fixed to the default values. Table II shows the tested parameter values, where the default values are in bold.

For the remaining GP common parameters, we follow the suggestions in [64]. Specifically, we set the population size to 50. When filling the new population by tournament selection, the tournament size is set to 2. This way, we can achieve good balance between exploration and exploitation.

The function and terminal sets are designed based on our benchmark problems. To ensure sufficiency, the function set is $\{+, -, x, \div, \sin, \cos, \ln(\cdot), \exp\}$. The terminal set contains all input data and ephemeral random constant ranging from 0 to 1. As ADGSGP only uses $\{+, -, x, \div\}$ as function set, we propose two versions of SLGP. One uses only four functions as function set, same as ADGSGP (named SLGP-4op), and the other with eight functions (named SLGP-8op).

For the parameter sensitivity analysis, Fig. 5 shows the box plots of the training and test error obtained by SLGP with different settings shown in Table II. From the figure, an ascending trend is seen for the number of registers $K$ in most of the problems. As the number of registers increases, the search space enlarges and thus SLGP requires more computation time to find satisfactory solutions. It can be observed that when the maximal number of instructions becomes larger (from 5 to 10), the performance of SLGP for difficult tasks (e.g., Keijzer11, Nonic, and Hartman) tends to be similar. However, for the simple tasks (e.g., Nguyen5 and Nguyen7), a large number of instructions may allow too much unnecessary instructions, and thus imposes a negative effect on the performance of SLGP. The similar pattern can also be seen for the instruction head length. For the library size limit $\rho$, SLGP gains the significantly better performance in some cases when $\rho$ is set to 2000. It is believed that a small library can converge to effective instructions faster than a large library. For the number of samples during library search $\tau$, on one hand, it should not be too small compared with the library size (e.g., $\tau = 100$ vs $\rho = 10000$). Otherwise, the tournament selection tends to be too random to find effective instructions. On the other hand, for some problems like Nguyen7, a very large $\tau$ may make the algorithm too greedy to jump out of local minima.
optima. For example, the test error of \( \tau = 700 \) for problem Nguyen7 is slightly larger than setting \( \tau \) to 300 or 500. For \( \nu, n_u, \) and \( \sigma \), different settings show a statistically similar training and test performance in most of problems. From the parameter sensitivity analysis, we conclude the settings of SLGP parameters, as shown in Table III, in the subsequent experiments.

V. RESULTS AND DISCUSSIONS

Based on the parameter setting analysis, we conduct more comprehensive experiments with the state-of-the-art algorithms. The experiment results are compared in terms of the success rate, training and test performance, and the program size of the finally outputted program.

A. Success Rate and Learning Efficiency

The success rate, the test RMSE and \( R^2 \) value of all methods are shown in Table IV. Specifically, the success

\[ R^2 = 1 - \frac{\text{MSE}(\hat{y}, y)}{\text{VAR}(y)} \]

where MSE is the mean square error, \( \hat{y} \) and \( y \) are the estimated output from the model and the ground truth of target output respectively, and \( \text{VAR}(y) \) is the variance of \( y \).

---

TABLE III

<table>
<thead>
<tr>
<th>Parameters</th>
<th>( K )</th>
<th>( NI )</th>
<th>( HL )</th>
<th>( \rho )</th>
<th>( \nu )</th>
<th>( n_u )</th>
<th>( \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>2000</td>
<td>500</td>
<td>20</td>
<td>1000</td>
</tr>
</tbody>
</table>
rate of RMSE is the proportion of independent runs whose training RMSE is smaller than $10^{-4}$. The success rate of $R^2$ is the proportion of runs whose test $R^2$ value is larger than 0.999. These results are collected from 30 independent runs of each method, with different random seeds. The Wilcoxon test with a family-wise false discovery rate correction (by the Benjamini and Hochberg method) and a significance level of 0.05 is also conducted on test RMSE and $R^2$ respectively. For each compared algorithm on each problem, the “Avg.” is followed by two symbols from “+”, “−”, and “≈”, where the first (second) “+” indicates that this algorithm is significantly better than SLGP-4op (SLGP-8op), “−” indicates that this algorithm is significantly worse than SLGP-4op (SLGP-8op), and “≈” means that there is no statistical difference between the algorithms. The similar notations are also adopted in Table IV.

Some entries have “++” or “−−” or “≈≈” in their comparison pairs. This is caused by 1 or 2 runs with $R^2$ slightly smaller than 0.999 (unsuccessful) and the remaining 28 or 29 runs with $R^2 > 0.999$ (successful), leading to the mean $R^2$ over 30 runs rounding to 1.000.

There are several conclusions can be drawn from Table IV. First, the SLGP methods have higher success rates than the other methods in terms of both RMSE and $R^2$ in most of the problems. Second, in terms of test error, the two SLGP methods significantly outperform or are at least competitive with other methods in most of the problems. For example, SLGP-8op wins the comparison with EPLEX on 6 of the 14 problems and have a draw on 7 problems in terms of RMSE and $R^2$ value. Taking a closer look on the four SGP methods (i.e., C-GSGP, ADGSGP, and the two SLGP versions), it can be found that the SLGP methods also effectively relieve the pathological prediction of existing SGP methods. It should be noted that some results (e.g., Airfoil and Tower) of C-GSGP are empty since its given code cannot be found that the SLGP methods also effectively relieve the pathological prediction of existing SGP methods. It should be noted that some results (e.g., Airfoil and Tower) of C-GSGP are empty since its given code cannot be applied to those real-world problems, because of the huge memory consumption of its semantic library. Besides, Table IV also shows that by adopting a smaller function set, the performance of SLGP can be significantly enhanced in some problems such as ModQua and Tower.

To have a more comprehensive comparison on the overall performance, a Friedman’s test with a significance level of 0.05 is applied to the test RMSE of all the methods, as shown in Table VI. Based on the results, SLGP-4op has the best average ranking among the six methods. ADGSGP and SLGP-8op are the second tier and have a similar mean ranking. SLGP, EPLEX, and C-GSGP have large rank about 4.4. The $p$-value of Friedman’s test is 7.75E-05, which implies there is a significant difference among the compared algorithms. A post-hoc analysis with Bonferroni correction is conducted, which shows that SLGP-4op significantly outperforms most of the other methods.

Overall, SLGP has a competitive effectiveness with other state-of-the-art methods.
To validate the training efficiency, the average training convergence curves of all methods are shown in Fig. 6. The curves record the fitness of the best individual in different NNE times. Overall, the figure shows that SLGP-4op is among the fastest algorithm that converge to the smallest RMSE for most problems. Specifically, the SLGP methods converge much faster and deeper than the GP methods without semantic information (i.e., SL-GEP and EPLEX). It shows that the introduction of semantic information can help GP methods to improve training efficiency. Besides, although the two existing SGP methods drop down faster and deeper than the two SLGP variants on some problems such as R1 and BHouse, they have overall worse test performance than SLGP based on Table IV. In a nutshell, the success rate, test performance, and the training convergence curves verify the high learning efficiency of SLGP.

### B. Comparison on Program Size

An overall program size discussion is performed here. The program size of solutions is defined as the number of nodes, including functions and terminals. Fig. 7 shows the box plots of all outputted solutions’ program size. As shown in Fig. 7, the program size of the two SLGP methods maintain roughly at the level from 40-60, which is much lower than the ones of the two state-of-the-art SGP methods whose program size distributions are larger than 100 in most problems. Though the solutions of the SLGP methods are slightly larger than those of SL-GEP which controls the bloat effect by a nested reused structure, we still believe SLGP is much more effective than SL-GEP since they achieve significantly better success rate and generalization ability than SL-GEP in all test problems. To conclude, SLGP has a competitive and even better learning performance than the state-of-the-art SGP methods with a much smaller program size.

### C. Computation Overhead

1) **Memory overhead:** The semantic library is the main consumption of memory during training. If there are $M$ instructions in $\Phi(L)$ and $N_s$ input semantic matrices in $S_{in}(L)$, then $L$ contains $M \times N_s + N_s$ semantic matrices, $M \times N_s$ for $S_{in}(L)$ and $N_s$ for $S_{in}(L)$. Since each semantic matrix has $N \times K$ floating point numbers (i.e., $N$ input cases and $K$ available registers), $L$ contains $N \times K \times (M + 1) \times N_s$ floating point numbers. Take the largest Tower dataset in our experiment as an example, assume $N_s = 5$ and each floating point number occupies 8 bytes, $L$ in SLGP consumes about 300 MB memory based on the recommended settings.

2) **Time complexity:** Utilizing semantic information in GP search leads to additional computation overhead. To investigate the time efficiency of SLGP, we compare its training time with the other algorithms. Note that the time comparison here is not a rigorous one since the compared methods are implemented in different programming languages (e.g., C++, Java and Python) on different platforms (e.g., Windows 10 and Linux). The time comparison here serves as a primary investigation of time complexity. The average training time of the methods on all the problems are shown in Table VI, where the largest training

![Image of the table](image_url)
time is marked in bold. In general, the semantic GP methods (i.e., C-GSGP, ADGSGP, SLGP) have much longer training time than those without semantic information (i.e., SL-GEP and EPLEX). The largest training time can be seen in different semantic GP methods. Among these semantic GP methods, ADGSGP has the largest training time for the benchmark problems. The training time of SLGP also increases with the number of training instances. But in most of the real-world problems, SLGP methods can have similar or shorter training time than the other semantic GP methods.

D. Component Analysis

SLGP has two new components, MDP and library update. To verify the effectiveness of each component, we conduct a component analysis here. We compare SLGP with two other variants. The first one removes the MDP component (i.e., 100% conventional genetic operators), which is named SLGP\MDP. The second one arbitrarily updates the library by generating random instructions and replacing random existing instructions in the library, named SLGP\LibUpd. By comparing SLGP with SLGP\MDP, we can verify the effectiveness of MDP. By comparing SLGP with SLGP\LibUpd, we can verify the effectiveness of the library update. The test RMSE of the compared algorithms on the fourteen SR problems are shown in Table VII. A Wilcoxon test is also applied.

The table shows that SLGP performs significantly better than both SLGP\MDP and SLGP\LibUpd. This demonstrates the effectiveness of both the MDP and library update component. In addition, we can see that SLGP\LibUpd showed significantly better performance than SLGP\MDP. This means that even without library update, MDP can greatly improve the performance the linear GP.

These results imply two main reasons for the superior performance of SLGP. Firstly, MDP can learn the behaviors of different instructions. Mutating instructions based on their behaviors (i.e., semantic information) is more effective than mutating instructions randomly. Secondly, including more effective instructions into the semantic library (i.e., library update) is a useful strategy to strengthen the performance of MDP. To conclude, both MDP and library update components play significant roles in SLGP.

VI. CONCLUSION AND FUTURE WORK

This paper proposed a new semantic linear genetic programming (SLGP) to solve symbolic regression problems. To the best of our knowledge, SLGP is the first SGP method implemented on linear chromosome representation. Based on the linear chromosome representation, a novel and effective semantic operator, called mutate-and-divide propagation (MDP) is designed. MDP utilizes a divide-and-conquer strategy to recursively mutate the instructions of the program based on the semantic information. Besides MDP, a library search

![Fig. 7. The program size of the outputted solutions.](image-url)

### Table VI

<table>
<thead>
<tr>
<th>Problem</th>
<th>SL-GEP</th>
<th>EPLEX</th>
<th>C-GSGP</th>
<th>ADGSGP</th>
<th>SLGP-4op</th>
<th>SLGP-8op</th>
</tr>
</thead>
<tbody>
<tr>
<td>Koza2</td>
<td>3.1</td>
<td>18.6</td>
<td>16.0</td>
<td>19.6</td>
<td>15.6</td>
<td>12.5</td>
</tr>
<tr>
<td>Nguyen4</td>
<td>0.8</td>
<td>19.0</td>
<td>17.9</td>
<td>14.3</td>
<td>17.4</td>
<td>14.0</td>
</tr>
<tr>
<td>Nguyen5</td>
<td>3.0</td>
<td>17.5</td>
<td>23.2</td>
<td>145.4</td>
<td>209.0</td>
<td>238.0</td>
</tr>
<tr>
<td>Nguyen7</td>
<td>3.1</td>
<td>18.7</td>
<td>170.8</td>
<td>135.3</td>
<td>204.4</td>
<td>259.5</td>
</tr>
<tr>
<td>Keijzer11</td>
<td>4.1</td>
<td>40.3</td>
<td>635.8</td>
<td>1705.7</td>
<td>445.6</td>
<td>615.2</td>
</tr>
<tr>
<td>R1</td>
<td>3.2</td>
<td>18.6</td>
<td>171.0</td>
<td>1507.5</td>
<td>188.7</td>
<td>237.4</td>
</tr>
<tr>
<td>ModQua</td>
<td>2.5</td>
<td>19.4</td>
<td>143.7</td>
<td>1177.9</td>
<td>171.2</td>
<td>215.5</td>
</tr>
<tr>
<td>Nonic</td>
<td>2.5</td>
<td>18.5</td>
<td>1244</td>
<td>12266</td>
<td>173.4</td>
<td>215.3</td>
</tr>
<tr>
<td>Hartman</td>
<td>4.0</td>
<td>43.6</td>
<td>725.3</td>
<td>1810.4</td>
<td>474.4</td>
<td>495.9</td>
</tr>
<tr>
<td>RatPol3D</td>
<td>6.3</td>
<td>111.0</td>
<td>1800.7</td>
<td>1850.2</td>
<td>1141.6</td>
<td>1275.1</td>
</tr>
<tr>
<td>Airfoil</td>
<td>17.2</td>
<td>430.0</td>
<td>N/A</td>
<td>2379.8</td>
<td>4388.4</td>
<td>4770.0</td>
</tr>
<tr>
<td>BHouse</td>
<td>7.5</td>
<td>168.5</td>
<td>1797.9</td>
<td>1491.2</td>
<td>1660.1</td>
<td>1799.4</td>
</tr>
<tr>
<td>Tower</td>
<td>56.1</td>
<td>1243.1</td>
<td>N/A</td>
<td>18600.6</td>
<td>1682.2</td>
<td>18396.4</td>
</tr>
<tr>
<td>CCN</td>
<td>30.6</td>
<td>739.1</td>
<td>29407.0</td>
<td>2928.5</td>
<td>6159.5</td>
<td>7219.6</td>
</tr>
</tbody>
</table>

### Table VII

<table>
<thead>
<tr>
<th>Problem</th>
<th>SLGP</th>
<th>MDP</th>
<th>SLGP\LibUpd</th>
<th>SLGP\MDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Koza2</td>
<td>0.91</td>
<td>≈</td>
<td>0.89</td>
<td>0.88</td>
</tr>
<tr>
<td>Nguyen4</td>
<td>0.07</td>
<td>0.06</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Nguyen5</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Nguyen7</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Keijzer11</td>
<td>0.17</td>
<td>0.12</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>R1</td>
<td>0.33</td>
<td>0.22</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>ModQua</td>
<td>0.13</td>
<td>0.10</td>
<td>0.19</td>
<td>0.19</td>
</tr>
<tr>
<td>Nonic</td>
<td>0.46</td>
<td>0.21</td>
<td>5.38</td>
<td>5.38</td>
</tr>
<tr>
<td>Hartman</td>
<td>0.06</td>
<td>0.05</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>RatPol3D</td>
<td>0.17</td>
<td>0.13</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>Airfoil</td>
<td>4.86</td>
<td>4.65</td>
<td>4.05</td>
<td>4.05</td>
</tr>
<tr>
<td>BHouse</td>
<td>4.92</td>
<td>4.71</td>
<td>4.32</td>
<td>4.32</td>
</tr>
<tr>
<td>Tower</td>
<td>48.6</td>
<td>41.27</td>
<td>32.17</td>
<td>32.17</td>
</tr>
<tr>
<td>CCN</td>
<td>0.37</td>
<td>0.37</td>
<td>0.36</td>
<td>0.36</td>
</tr>
</tbody>
</table>

1 The “A → B” means “A is significantly worse than B” based on a Friedman’s test and Bonferroni method with a significance level of 0.05.
method and an update method are also developed to further improve the learning ability of SLGP. The paper performs a comprehensive experiments to compare SLGP with four recently proposed GP methods on fourteen popular benchmark problems including real-world problems. The empirical results show that SLGP not only has a higher success rate and significantly better (or at least competitive) generalization ability than (with) other state-of-the-art methods in most problems, but also produces solutions with a much smaller program size than existing tree-based SGP methods whose solutions are often too complex. The experiment verifies the promising learning ability of the SLGP.

There are some potential directions to further improve SLGP in future. For example, the time consumption of SLGP is much larger than the non-semantic GP methods. It is potential to improve the time efficiency of SLGP by surrogate models. Besides, the robustness of SLGP should be further investigated. Despite the recommended parameter settings, SLGP is less competitive when some parameters have different settings. Thus, adaptive parameter controlling strategies can be introduced to SLGP to further enhance the performance.

REFERENCES


