Semantic Linear Genetic Programming for Symbolic Regression

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

***Abstract*—Symbolic regression (SR) is an important problem with many applications, such as automatic programming tasks and data mining. Genetic programming (GP) is a commonly used technique for SR. In the past decade, a branch of GP that utilizes the program behavior to guide the search, called semantic GP (SGP), has achieved great success in solving SR problems. However, existing SGP 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 GP (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 SR problems and can achieve a much smaller program size.**

***Index Terms*—Genetic programming (GP), mutate-and-divide propagation (MDP), symbolic regression (SR).**

I. Introduction

SYMBOLIC regression (SR) is a task to synthesize arithmetic formulas 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 maintaining [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 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., "$\text{Input}_1 + \text{Input}_2$") 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 GP (SGP) shows a faster training efficiency and better test performance than many other nonsemantic GP methods [16], [17]. SGP has become one of the most effective methods to solve SR 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 backpropagating 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 overfit 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 overfitting 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 a 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 have 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

Manuscript received January 12, 2022; revised April 29, 2022; accepted June 4, 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 62076098; in part by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant 2017ZT07X183; and in part by Guangdong Natural Science Foundation Research Team under Grant 2018B030312003. This article was recommended by Associate Editor N. Zhang. *(Corresponding authors: Jinghui Zhong; Yi Mei.)*

Zhixing Huang and Yi Mei are with the School of Engineering and Computer Science, Victoria University of Wellington, Wellington 6012, New Zealand (e-mail: zhixing.huang@ecs.vuw.ac.nz; yi.mei@ecs.vuw.ac.nz).

Jinghui Zhong is with the School of Computer science and Engineering, South China University of Technology, Guangzhou 510006, China (e-mail: jinghuizhong@scut.edu.cn).

Digital Object Identifier 10.1109/TCYB.2022.3181461

information and propose a new semantic linear GP (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 (MDP) operator. The major contributions of this article 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 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 subprograms which can be further updated by MDP. Thus, all instructions in the program can be updated according to the semantic error.

The remainder of this article is organized as follows. The related work is introduced in Section II. Then, the proposed SLGP is described in detail in Section III. Afterward, the experimental results, discussions, and analyses are given in Section IV. Finally, the conclusions are drawn in Section VI.

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 that represents programs by computation nodes in a 2-D grid system. In this article, a new kind of linear GP is developed. In literature, linear GP has been successfully applied to classification [30]–[32] and SR [25], [33], but none of the existing work consider the semantic information in linear GP individuals. In addition, 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 representations, 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 relief the time consumption in fitness evaluation. Apart from these reproduction mechanisms, 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). The 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.* [16] in the *semantic-aware crossover*, in which a semantic equivalence is checked to find subtrees with distinct semantics before subtree crossover.

These genetic operators can search subtrees 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, Uy *et al.* [45] proposed a *semantic similarity distance* to measure the semantic distance between subtrees and extended the semantic-aware crossover into a *semantic similarity-based crossover* which swaps subtrees with the most similar but different semantics to maintain the effectiveness of offspring. Based on the idea of the semantic similarity-based crossover, Uy *et al.* [46] further proposed an updated version with an adaptive similarity threshold to reduce the computation burden in selecting proper subtrees and proposed another improved version with the most similar subtree selection [47]. A semantic similarity-based mutation was also proposed by Uy *et al.* [48].

C. Direct Semantic Genetic Programming

Geometric SGP (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 are 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 recombining 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 the 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. Hara *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 contains 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 *et al.* [18] and Vanneschi *et al.* [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 contain 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 a 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 perpendicular crossover and random segment mutation to identify the new desired semantics based on the semantic angle information. These two methods can improve the generalization 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 overfitting 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 first defines the inverse operations for each function primitives and establishes a subtree library to collect the subtrees 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, the RDO method searches a "best fit" subtree from its subtree library to approximate the desired semantics. Since the RDO method is directly oriented by the target semantics and it replaces subtrees to approximate desired semantics, rather than concatenating existing expressions directly, the RDO method is more efficient than conventional GSGP and relieves the bloat effect.

RDO has been extended by some methods. For example, Nhat *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]. Ffrancon and Schoenauer [62] combined the memetic algorithm with RDO by searching the most suitable internal node as the subroot to perform crossover. Currently, backpropagating semantic error to obtain the subtree 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 (ADGSGP) which successfully improves the generalization and further relieves the bloat effect. Pawlak and Krawiec [63] also adopted the 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 formulas.

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 $\mathbf{f} = [f_1, \ldots, f_{|\mathbf{f}|}]$, 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* $[V_0, V_1, \ldots]$ 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 $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]$ and K registers, we consider the data stored during the computation as a *semantic matrix* $\mathbf{S}_{N\times K}$, where $s_{i,j}$ indicates the value of register V_j given the inputs of the case $\mathbf{x}_i$. Given an input data $\mathbf{X}$ and semantic matrix $\mathbf{S}_{\text{in}}$ of an instruction f, we denote the output semantic matrix produced by f as $\mathbf{S}_{\text{out}} = f(\mathbf{S}_{\text{in}}, \mathbf{X})$. Take the first instruction "$f_1 : V_0 = \sin(I_0 + I_1)$" in Fig. 1 as an example. If there are four

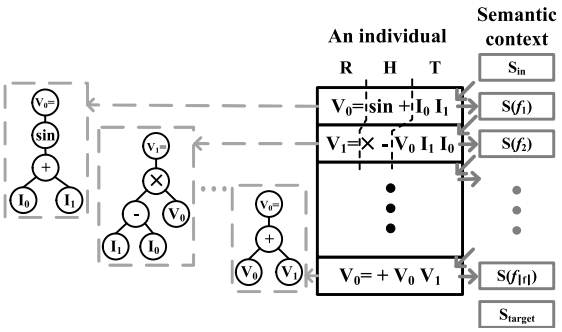


Fig. 1. Example of SLGP individual.

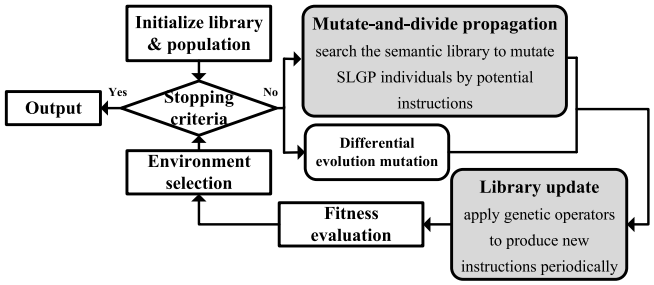


Fig. 2. Flowchart of SLGP. The key steps are highlighted in dark boxes.

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 $\mathbf{0}_{4\times 2}$, that is, 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

$$\mathbf{S}_1 = f_1\left(\mathbf{0}_{4\times 2}, \begin{bmatrix} 1, 3 \\ 2, 5 \\ 3, 4 \\ 4, 7 \end{bmatrix}\right) = \begin{bmatrix} 0.07, 0 \\ 0.12, 0 \\ 0.12, 0 \\ 0.19, 0 \end{bmatrix}.$$

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

B. SLGP Framework

The flowchart of the proposed SLGP[1] is shown in Fig. 2 and its pseudocode is shown in Algorithm 1.[2] Given the training data $[\mathbf{X}, \mathbf{y}] \in \mathbb{R}^{N\times(M+1)}$ with N instances/cases and M variables ($\mathbf{x}_i = [x_{i1}, \ldots, x_{iM}]$ is the ith case, and y_i is its target output), SLGP finds the best SR model $\mathbf{f}^*$ that maps from a M-dimensional input vector to a real-valued output.

[1]The source code of SLGP can be downloaded from https://github.com/Zhixing1020/SemanticLGP_for_SR.

[2]$\texttt{rand}(a, b)$ 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 this article.

Algorithm 1: Procedure of SLGP

Input: The training data $[\mathbf{X}, \mathbf{y}] \in \mathbb{R}^{N\times(M+1)}$
Output: The SR model $\mathbf{f}^* : \mathbb{R}^M \to \mathbb{R}$

Set $gen = 0$, NNE $= 0$;
Initialize the population $\mathbf{F}$ and semantic library Ł;
for *individual* $\mathbf{f} \in \mathbf{F}$ **do**
  $fit(\mathbf{f}), \mathbf{S}(\mathbf{f}) = \texttt{evaluate}(\mathbf{f}, \mathbf{X}, \mathbf{y})$;
for *instruction* $f \in \Phi(Ł)$ **do**
  for $\mathbf{S} \in \mathbb{S}_{in}(Ł)$ **do**
    Calculate $f(\mathbf{S})$, $\mathbb{S}_{\text{out}}(Ł) = \mathbb{S}_{\text{out}}(Ł) \cup f(\mathbf{S})$;
    NNE $=$ NNE $+ \texttt{size}(f)$;
while NNE $<$ NNE$_{\max}$ **do**
  Set offspring population $\mathbf{F}' = \emptyset$;
  for *individual* $\mathbf{f} \in \mathbf{F}$ **do**
    if $\texttt{rand}(0, 1) < 0.5$ **then**
      $\ell = \texttt{rand\_int}(1, |\mathbf{f}|)$;
      $\mathbf{f}', \mathbf{S}' = \texttt{MDP}(\mathbf{f}, \mathbf{S}(\mathbf{f}), \ell, Ł)$;
    else
      Reproduce $\mathbf{f}'$ by the DE mutation [64] (considering $\mathbf{f}$ as one of its parents);
    $fit(\mathbf{f}'), \mathbf{S}(\mathbf{f}') = \texttt{evaluate}(\mathbf{f}', \mathbf{X}, \mathbf{y})$;
    $\mathbf{F}' = \mathbf{F}' \cup \mathbf{f}'$;
  $\mathbf{f}^* = \arg\min_{\mathbf{f}\in\mathbf{F}\cup\mathbf{F}'} fit(\mathbf{f})$;
  $\mathbf{F} = \{\mathbf{f}^*\}$; // elitism
  Fill in $\mathbf{F}$ by tournament selection from $\mathbf{F} \cup \mathbf{F}'$;
  if mod $(gen, v) = 0$ **then**
    $\texttt{LibraryUpdate}(Ł, n_u, \sigma)$;
  $gen = gen + 1$;
return $\mathbf{f}^*$.

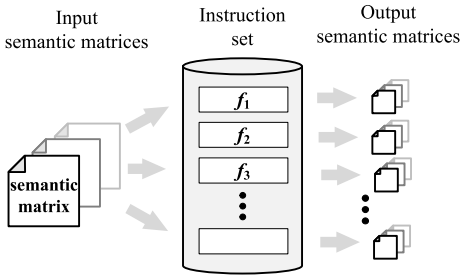


Fig. 3. Design of the semantic library in SLGP.

At the beginning, a population $\mathbf{F}$ of SR models and a *semantic library* Ł are randomly initialized. The semantic library consists of three main components: 1) a set of *input semantic matrices* $\mathbb{S}_{\text{in}}(Ł)$; 2) a set of instructions $\Phi(Ł)$; and 3) the output semantic matrix of each instruction given each input semantic matrix, that is, $\mathbb{S}_{\text{out}}(Ł) = \{f(\mathbf{S}_{\text{in}}) | f \in \Phi(Ł), \mathbf{S}_{\text{in}} \in \mathbb{S}_{\text{in}}(Ł)\}$. Hereafter, we simplify $f(\mathbf{S}_{\text{in}}, \mathbf{X})$ as $f(\mathbf{S}_{\text{in}})$, since the training data $\mathbf{X}$ are the same for all possible programs. Fig. 3 illustrates the semantic library. The input semantic matrices $\mathbb{S}_{\text{in}}(Ł)$ consists of a fixed number of randomly sampled semantic matrices within the given data range. The output semantic matrices $\mathbb{S}_{\text{out}}(Ł)$ are calculated based on $\mathbb{S}_{\text{in}}(Ł)$ for every instruction. Instructions $\Phi(Ł)$ are initialized randomly to fill up $\Phi(Ł)$. If $f(\mathbf{S}_{\text{in}})(f \in \Phi(Ł))$ is not unique in $\mathbb{S}_{\text{out}}(Ł)$, f will be removed from $\Phi(Ł)$. 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.

Algorithm 2: `evaluate`$(\mathbf{f}, \mathbf{X}, \mathbf{y})$

Input: The evaluated individual $\mathbf{f} = [f_1, \ldots, f_{|\mathbf{f}|}]$, training data $[\mathbf{X}, \mathbf{y}]$
Output: The fitness $fit(\mathbf{f})$, its semantic context $\mathbf{S}(\mathbf{f})$

Set $\mathbf{S}_0 = \mathbf{0}_{N\times K}$, $\mathbf{S}_{\text{target}} = [\mathbf{y}, \mathbf{0}_{N\times(K-1)}]$;
for $i = 1 \to |\mathbf{f}|$ **do**
  $\mathbf{S}(f_i) = \mathbf{S}_i = f_i(\mathbf{S}_{i-1}, \mathbf{X})$;
  $\text{NNE} = \text{NNE} + \texttt{size}(f_i)$;
$\mathbf{y}_{\text{out}} = [\mathbf{S}_{|\mathbf{f}|}]_{*,0}$;
$fit(\mathbf{f}) = \texttt{RMSE}(\mathbf{y}_{\text{out}}, \mathbf{y})$;
return $fit(\mathbf{f})$, $\mathbf{S}(\mathbf{f}) = [\mathbf{S}_0, \mathbf{S}(f_1), \ldots, \mathbf{S}(f_{|\mathbf{f}|}), \mathbf{S}_{\text{target}}]$;

In each generation, an offspring population is generated. Each offspring is generated from one individual in the population, by the newly proposed *MDP* operator with 50% probability, and by the differential evolution (DE) genetic operators [64] with the other 50% probability. The MDP operator will be described in more detail in Section III-D. Then, the new population is selected from the current population and the offspring population. Specifically, the best individual is copied into the new population (elitism). The remaining individuals are then selected by the tournament selection from the union of the two populations. Different from traditional GP methods, SLGP uses a random parent selection and tournament environment selection. This configuration is recommended by [65] and adopted by some existing GP methods [64]. The semantic library is updated every ν generation. The entire algorithm stops when the number of node evaluations (NNE) reaches the maximum $\text{NNE}_{\max}$. Note that NNE is increased by $\text{NNE} = \text{NNE} + \texttt{size}(f)$ every time evaluating an output semantic matrix of f (i.e., $f(\mathbf{S})$), where $\texttt{size}(f)$ is the number of primitives in f.

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 $[\mathbf{y}, \mathbf{0}_{N\times(K-1)}]$, that is, 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 $(i = 1, \ldots, |\mathbf{f}|)$ is calculated by executing the instructions sequentially. The finally output is the first column (V_0) of the final semantic matrix, that is, $\mathbf{y}_{\text{out}} = [\mathbf{S}_{|\mathbf{f}|}]_{*,0}$. Then, the fitness is calculated as the *root-mean-squared error (RMSE)* between $\mathbf{y}_{\text{out}}$ and $\mathbf{y}$, which is a common performance metric in many SR literatures [21], [40], [64]

$$\texttt{RMSE}(\mathbf{y}_{\text{out}}, \mathbf{y}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(y_{\text{out},i} - y_i\right)^2}. \tag{1}$$

D. Mutate-and-Divide Propagation

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

Algorithm 3: `MDP`$(\mathbf{f}, \mathbf{S}(\mathbf{f}), \ell, Ł)$

Input: A sequence of instructions $\mathbf{f} = [f_1, \ldots, f_{|\mathbf{f}|}]$, its semantic context $\mathbf{S}(\mathbf{f}) = [\mathbf{S}_{\text{in}}, \mathbf{S}(f_1), \ldots, \mathbf{S}(f_{|\mathbf{f}|}), \mathbf{S}_{\text{target}}]$, a step length ℓ, and a semantic library Ł
Output: The new sequence of instructions $\mathbf{f}'$, the desired input semantic matrix $\mathbf{S}'_{\text{desire}}$ of $\mathbf{f}'$

$i_1 = \max\{0, \texttt{rand\_int}(|\mathbf{f}| - \ell, |\mathbf{f}| - 1)\}$;
$i_2 = \texttt{rand\_int}(i_1 + 1, |\mathbf{f}|)$;
$\mathbf{f}_1 = [f_1, \ldots, f_{i_1}]$, $\mathbf{S}_1 = [\mathbf{S}(f_1), \ldots, \mathbf{S}(f_{i_1})]$;
$\mathbf{f}_2 = [f_{i_1+1}, \ldots, f_{i_2-1}]$, $\mathbf{S}_2 = [\mathbf{S}(f_{i_1+1}), \ldots, \mathbf{S}(f_{i_2-1})]$;
$\mathbf{f}_3 = [f_{i_2+1}, \ldots, f_{|\mathbf{f}|}]$, $\mathbf{S}_3 = [\mathbf{S}(f_{i_2+1}), \ldots, \mathbf{S}(f_{|\mathbf{f}|})]$;
$\mathbf{S}'_{\text{in}} = \mathbf{S}_{\text{in}}$;
if $i_1 > 0$ **then** $\mathbf{S}'_{\text{in}} = \mathbf{S}(f_{i_1})$
$f', \mathbf{S}'_{\text{desire}}, \mathbf{S}'_{\text{out}} = \texttt{LibrarySearch}(\mathbf{S}'_{\text{in}}, \mathbf{S}_{\text{target}}, Ł)$;
if $\mathbf{f}_3 \neq [\,]$ **then**
  $\mathbf{f}'_3, \mathbf{S}'_3 = \texttt{MDP}(\mathbf{f}_3, [\mathbf{S}'_{\text{out}}, \mathbf{S}_3, \mathbf{S}_{\text{target}}], \ell, Ł)$;
if $\mathbf{f}_2 \neq [\,]$ **then**
  $\mathbf{f}'_2, \mathbf{S}'_{\text{desire}} = \texttt{MDP}(\mathbf{f}_2, [\mathbf{S}'_{\text{in}}, \mathbf{S}_2, \mathbf{S}'_{\text{desire}}], \ell, Ł)$;
if $\mathbf{f}_1 \neq [\,]$ **then**
  $\mathbf{f}'_1, \mathbf{S}'_{\text{desire}} = \texttt{MDP}(\mathbf{f}_1, [\mathbf{S}_{\text{in}}, \mathbf{S}_1, \mathbf{S}'_{\text{desire}}], \ell, Ł)$;
$\mathbf{f}' = [\mathbf{f}'_1, \mathbf{f}'_2, f', \mathbf{f}'_3]$;
return $\mathbf{f}', \mathbf{S}'_{\text{desire}}$;

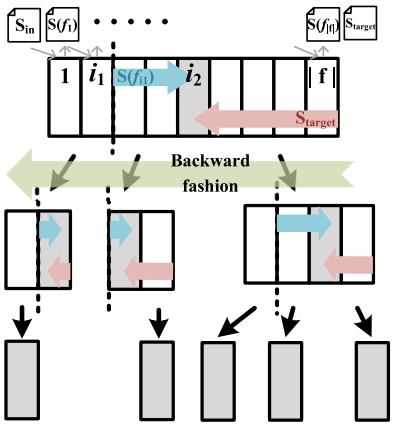


Fig. 4. Recursion of the MDP. Dark boxes highlight the mutated instructions by `LibrarySearch`$(\cdot)$ for each recursion. Blue and red arrows, respectively, indicate the input and output semantic matrices for `LibrarySearch`$(\cdot)$.

First, two indices i_1 and i_2 $(0 \leq i_1 < i_2 \leq |\mathbf{f}|)$ are randomly sampled based on the step length ℓ. Specifically, the instruction f_{i_1} provides its output semantic matrix $\mathbf{S}(f_{i_1})$ to f_{i_2}, and f_{i_2} is the instruction to be *mutated*. The two instructions split the instruction sequence into three subsequences: 1) $\mathbf{f}_1 = [f_1, \ldots, f_{i_1}]$; 2) $\mathbf{f}_2 = [f_{i_1+1}, \ldots, f_{i_2-1}]$; and 3) $\mathbf{f}_3 = [f_{i_2+1}, \ldots, f_{|\mathbf{f}|}]$. The corresponding sequences of semantic matrices $\mathbf{S}_1$, $\mathbf{S}_2$, and $\mathbf{S}_3$ are obtained accordingly. The instruction f_{i_2} is going to be mutated, thus is not included in the subsequences.

Fig. 4 shows a schematic diagram of MDP. To mutate f_{i_2}, the goal is to find the best instruction $f' \in \Phi(Ł)$ so that it can take the semantic matrix $\mathbf{S}'_{\text{in}}$ produced by f_{i_1} (or the input semantic matrix $\mathbf{S}_{\text{in}}$ if $i_1 = 0$) as inputs and approximate the target semantic matrix $\mathbf{S}_{\text{target}}$. This is done by the function `LibrarySearch`$(\cdot)$, which will be introduced in Section III-E. Note that using different f_{i_1} to provide $\mathbf{S}'_{\text{in}}$ rather

than only considering $\mathbf{S}_{\text{in}}$ 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 Ł, its desired input matrix $\mathbf{S}'_{\text{desire}}$ to produce the target semantic matrix, and its estimated output semantic matrix $\mathbf{S}'_{\text{out}}$ if given the input semantic matrix $\mathbf{S}'_{\text{in}}$. Ideally, if we find an instruction so that $f'(\mathbf{S}'_{\text{in}}) = \mathbf{S}_{\text{target}}$, then we have found the optimal sequence $\mathbf{f}^* = [\mathbf{f}_1, f']$. However, due to the limited storage of the semantic library, it is hardly possible to find the ideal instruction. Instead, we usually have $\mathbf{S}'_{\text{desire}} \neq \mathbf{S}'_{\text{in}}$ and $\mathbf{S}'_{\text{out}} \neq \mathbf{S}_{\text{target}}$. Therefore, it is necessary to recursively mutate the instructions in $\mathbf{f}_1$, $\mathbf{f}_2$, and $\mathbf{f}_3$ 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 $\mathbf{f}_3$ by MDP, so that it takes the expected output semantic matrix $\mathbf{S}'_{\text{out}}$ of f' and targets to output the target semantic matrix $\mathbf{S}_{\text{target}}$. Then, we modify $\mathbf{f}_2$ by MDP, so that it takes the output semantic matrix of f_{i_1} and outputs the desired semantic matrix of f'. It also updates the desired input semantic matrix $\mathbf{S}'_{\text{desire}}$. Finally, we modify $\mathbf{f}_1$ by MDP to take the overall input semantic matrix and output $\mathbf{S}'_{\text{desire}}$.

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

E. Semantic Library Search

The semantic library search `LibrarySearch` $(\mathbf{S}_{\text{in}}, \mathbf{S}_{\text{target}}, Ł)$ aims to search in the semantic library Ł to find the instruction f' whose behavior is closest to the semantic mapping $\mathbf{S}_{\text{in}} \rightarrow \mathbf{S}_{\text{target}}$, that is, it takes $\mathbf{S}_{\text{in}}$ and produces $\mathbf{S}_{\text{target}}$. This can be formulated as the following problem:

$$f' = \arg\min_{f \in \Phi(Ł)} \texttt{SAE}\big(f(\mathbf{S}_{\text{in}}), \mathbf{S}_{\text{target}}\big) \tag{2}$$

where `SAE` is the sum of absolute error between two matrices, calculated as follows:

$$\texttt{SAE}(\mathbf{A}, \mathbf{B}) = \frac{1}{N \cdot K} \sum_{i=1}^{N} \sum_{j=1}^{K} |a_{i,j} - b_{i,j}|. \tag{3}$$

The calculation of $f(\mathbf{S}_{\text{in}})$ 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 proposed to estimate $f(\mathbf{S}_{\text{in}})$ based on the input–output mapping stored in the semantic library. Specifically, we first select the input semantic matrix $\mathbf{S}^* \in \mathbb{S}_{\text{in}}(Ł)$ that is closest to $\mathbf{S}_{\text{in}}$ (i.e., $\mathbf{S}^* = \arg\min_{\mathbf{S} \in \mathbb{S}_{\text{in}}(Ł)} \texttt{SAE}(\mathbf{S}, \mathbf{S}_{\text{in}})$). Then, we conduct a projection as follows:

$$\tilde{f}(\mathbf{S}_{\text{in}}) = f\big(\mathbf{S}^*\big) + \nabla f\big(\mathbf{S}^*\big) \circ \big(\mathbf{S}_{\text{in}} - \mathbf{S}^*\big) \tag{4}$$

where

$$\nabla f\big(\mathbf{S}^*\big) = \frac{1}{|\mathbb{S}_{\text{in}}(Ł)| - 1} \sum_{\substack{\mathbf{S} \in \mathbb{S}_{\text{in}}(Ł) \\ \mathbf{S} \neq \mathbf{S}^*}} \big[f(\mathbf{S}) - f\big(\mathbf{S}^*\big)\big] \oslash \big[\mathbf{S} - \mathbf{S}^*\big].$$

The Hadamard product $\mathbf{A} \circ \mathbf{B}$ and Hadamard division $\mathbf{A} \oslash \mathbf{B}$ between two matrices are defined as follows:

$$\mathbf{A} \circ \mathbf{B} = \big[a_{i,j} \times b_{i,j}\big], \mathbf{A} \oslash \mathbf{B} = \big[a_{i,j} / b_{i,j}\big].$$

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

$$\overline{\Delta \mathbf{S}} = \frac{1}{|\mathbb{S}_{\text{in}}(Ł)| - 1} \sum_{\substack{\mathbf{S} \in \mathbb{S}_{\text{in}}(Ł) \\ \mathbf{S} \neq \mathbf{S}^*}} \big[\mathbf{S} - \mathbf{S}^*\big].$$

Thus, we have

$$\nabla f\big(\mathbf{S}^*\big) = \left(\sum_{\substack{\mathbf{S} \in \mathbb{S}_{\text{in}}(Ł) \\ \mathbf{S} \neq \mathbf{S}^*}} \big[f(\mathbf{S}) - f\big(\mathbf{S}^*\big)\big] \right) \oslash \left(\sum_{\substack{\mathbf{S} \in \mathbb{S}_{\text{in}}(Ł) \\ \mathbf{S} \neq \mathbf{S}^*}} \big[\mathbf{S} - \mathbf{S}^*\big] \right).$$

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 τ times. For each instruction, we estimate its output semantic matrix based on (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 (4) might not be accurate. To reduce the bias caused by the inaccurate estimation, we introduce a probability ε 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`$(\cdot)$ during the evolutionary process, we define the discard probability as $\varepsilon(f) = n(f)/(\nu \times \mathit{popsize})$, where ν is the frequency to update the library, and *popsze* 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 θ 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 $\tilde{f}'(\mathbf{S}_{\text{in}})$ have 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:

$$\tilde{f}^{-1}\big(\mathbf{S}_{\text{target}}\big) = \mathbf{S}^* + \nabla^{-1} f\big(\mathbf{S}^*\big) \circ \big(\mathbf{S}_{\text{target}} - f\big(\mathbf{S}^*\big)\big) \tag{5}$$

where

$$\nabla^{-1} f\big(\mathbf{S}^*\big) = \left(\sum_{\substack{\mathbf{S} \in \mathbb{S}_{\text{in}}(Ł) \\ \mathbf{S} \neq \mathbf{S}^*}} \big[\mathbf{S} - \mathbf{S}^*\big] \right) \oslash \left(\sum_{\substack{\mathbf{S} \in \mathbb{S}_{\text{in}}(Ł) \\ \mathbf{S} \neq \mathbf{S}^*}} \big[f(\mathbf{S}) - f\big(\mathbf{S}^*\big)\big] \right).$$

Algorithm 4 shows the pseudocode of the library search.

Algorithm 4: `LibrarySearch`($\mathbf{S}_{\text{in}}, \mathbf{S}_{\text{target}}, Ł$)

Input: The input semantics $\mathbf{S}_{\text{in}}$, the target semantics $\mathbf{S}_{\text{target}}$, the semantic library Ł
Output: The selected instruction $f' \in Ł$, its desired input semantics $\mathbf{S}'_{\text{desire}}$, its estimated output semantics $\mathbf{S}'_{\text{out}}$

$\mathbf{S}^* = \arg\min_{\mathbf{S} \in \mathbb{S}_{\text{in}}(Ł)} \texttt{SAE}(\mathbf{S}, \mathbf{S}_{\text{in}})$;
Set $\delta' = +\infty$;
for $i = 1 \to \tau$ **do**
  Randomly select $f \in Ł$;
  Let $n(f)$ be the number of times that f has been selected during the evolutionary process;
  $\varepsilon(f) = \min\left\{\frac{n(f)}{v \times popsize}, \theta\right\}$;
  if $\texttt{rand}(0,1) < 1 - \varepsilon(f)$ **then**
    Estimate $\tilde{f}(\mathbf{S}_{\text{in}})$ by Eq. (4);
    $\delta(f) = \texttt{SAE}(\tilde{f}(\mathbf{S}_{\text{in}}), \mathbf{S}_{\text{target}})$;
    if $\delta(f) < \delta'$ **then**
      $f' = f$, $\delta' = \delta(f)$;
$n(f') = n(f') + 1$;
Estimate $\tilde{f}'^{-1}(\mathbf{S}_{\text{target}})$ by Eq. (5);
return $f', \tilde{f}'^{-1}(\mathbf{S}_{\text{target}}), \tilde{f}'(\mathbf{S}_{\text{in}})$;

F. Library Update

The semantic library is updated periodically (i.e., every v generation) by adding/changing the instructions and their semantic matrices, as shown in Algorithm 5. First, a number of n_u new instructions are generated from the instructions in $\Phi(Ł)$. For generating each new instruction, two parents p_1 and p_2 are selected from $\Phi(Ł)$ 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`$(\cdot)$) 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 $\Phi(Ł)$, then it is added into $\Phi(Ł)$, and its output semantic matrices of $\mathbb{S}_{\text{in}}(Ł)$ are added into $\mathbb{S}_{\text{out}}(Ł)$. After the insertion, if $\Phi(Ł)$ exceeds the size limit ρ, 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 σ, that is, $n(f) = n(f) \times \sigma$.

Algorithm 5: `LibraryUpdate`($Ł, n_u, \sigma$)

Input: Semantic library $Ł = \langle \mathbb{S}_{\text{in}}(Ł), \Phi(Ł), \mathbb{S}_{\text{out}}(Ł) \rangle$, number of new instructions n_u, decay factor σ

for $i = 1 \to n_u$ **do**
  Randomly select instructions $p_1, p_2 \in \Phi(Ł)$;
  for $k = 1 \to 2$ **do**
    for $j = 2 \to 20$ **do**
      Randomly select $f' \in \Phi(Ł)$;
      if $\texttt{rand}(0,1) < 1 - \varepsilon(f')$ **then**
        if $n(f') > n(p_k)$ **then**
          $p_k = f'$;
  Generate a new instruction f_i by applying genetic operators to p_1 and p_2;
  if $f_i \notin \Phi(Ł)$ **then**
    $\Phi(Ł) = \Phi(Ł) \cup f_i$, $n(f_i) = 0$;
    for $\mathbf{S} \in \mathbb{S}_{in}(Ł)$ **do**
      Calculate $f_i(\mathbf{S})$, $\mathbb{S}_{\text{out}}(Ł) = \mathbb{S}_{\text{out}}(Ł) \cup f_i(\mathbf{S})$;
      NNE = NNE + $\texttt{size}(f_i)$;
    if $|\Phi(Ł)| > \rho$ **then**
      Randomly select $r \in \Phi(Ł)$;
      for $j = 2 \to 20$ **do**
        Randomly select $f' \in \Phi(Ł)$;
        if $\texttt{rand}(0,1) < 1 - \varepsilon(f')$ **then**
          if $n(f') < n(r)$ **then**
            $r = f'$;
      Remove r from $\Phi(Ł)$;
      for $\mathbf{S} \in \mathbb{S}_{in}(Ł)$ **do**
        Remove $r(\mathbf{S})$ from $\mathbb{S}_{\text{out}}(Ł)$;
for $f \in \Phi(Ł)$ **do**
  $n(f) = n(f) \times \sigma$;

IV. Experiment Setup

A. Datasets

The experiments consist of 14 SR problems, including four real-world datasets. These test problems are selected from the recent studies [21], [63], [66]. 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 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 terms 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 RMSE of the outputs (or the target registers) is less than 10^{-4} (also called "success" in this article).

B. Comparison Methods

To validate the performance of SLGP, four recently published methods are compared with SLGP. They are SL-GEP [64], GP with ϵ-lexicase selection (EPLEX) [67], competent GSGP (C-GSGP) [63], and ADGSGP [21]. All of these algorithms are state-of-the-art GP variants in solving SR problems. SL-GEP adopts a gene expression chromosome representation and an automatically defined function structure to encode programs and proposes a DE-based genetic operator to evolve programs. EPLEX proposes a new selection

TABLE I
TEST PROBLEMS

Problem	Function	#Features	Data range	#Points (Train,Test)
Benchmark problems				
Koza2	$f(x) = x^5 - 2x^3 + x$	1	[-1,1]	(20,1000)
Nguyen4	$f(x) = x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	[-1,1]	(20,1000)
Nguyen5	$f(x) = \sin(x^2)\cos(x) - 1$	1	[-1,1]	(20,1000)
Nguyen7	$f(x) = \ln(x+1) + \ln(x^2+1)$	1	[0,2]	(20,1000)
Keijzer11	$f(x,y) = xy + \sin((x-1)(y-1))$	2	[-1,1]	(100,900)
R1	$f(x) = \frac{(x+1)^3}{x^2-x+1}$	1	[-2,2]	(20,1000)
ModQua	$f(x) = 4x^4 + 3x^3 + 2x^2 + x$	1	[-2,2]	(20,1000)
Nonic	$f(x) = \sum_{i=1}^{9} x^i$	1	[-2,2]	(20,1000)
Hartman	$f(x) = -\exp(-\sum_i^4 x_i^2)$	4	[0,2]	(100,900)
RatPol3D	$f(x) = 30\frac{(x_0-1)(x_2-1)}{x_1^2(x_0-10)}$	3	x_0:[0.05,2] x_1:[1,2] x_2:[0.05,2]	(300,2700)
Real-world problems				
Airfoil	unknown	5	-	(1127,376)
BHouse	unknown	13	-	(380,126)
Tower	unknown	25	-	(3749,1250)
CCN	unknown	124	-	(1661,554)

[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.

paradigm for GP methods that enables more near-elite individuals to survive in the next population. Both of these techniques effectively relieve the bloat effect and meanwhile achieve a high success rate in SR problems. C-GSGP and ADGSGP are the two latest SGP methods. C-GSGP first 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, 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 outperforms 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 eight new parameters: 1) the number of registers K; 2) the maximal number of instructions NI; 3) the maximal length of instruction arithmetical part HL; 4) the size limit of the semantic library ρ; 5) the number of samples in the semantic library search τ; 6) library update period ν; 7) the number of new instructions during library update n_u; and 8) decay factor σ. 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 a 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.

TABLE II
DIFFERENT SETTINGS OF PARAMETERS

Parameter	Test values
K	**1**, 2, 3
NI	3, **5**, 7, 10
HL	3, **5**, 7, 9
ρ	2000, 5000, 7000, **10000**
τ	100, 300, **500**, 700
ν	10, **20**, 50, 100
n_u	500, 700, **1000**, 1200
σ	0.4, 0.6, **0.8**, 1

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 a 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 $\{+, -, \times, \div, \sin, \cos, \ln(|\cdot|), \exp\}$.[3] The terminal set contains all features of input data and ephemeral random constant ranging from 0 to 1. As ADGSGP only uses $\{+, -, \times, \div\}$ as function set, we propose two versions of SLGP. One uses only four functions as a function set, the same as ADGSGP (called SLGP-4op), and the other with eight functions (called 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 many unnecessary instructions and, thus, imposes a negative effect on the performance of SLGP. A similar pattern can

[3] The $\div$ of SLGP is protected by analytic quotient. The protected division of other methods follows their original design.

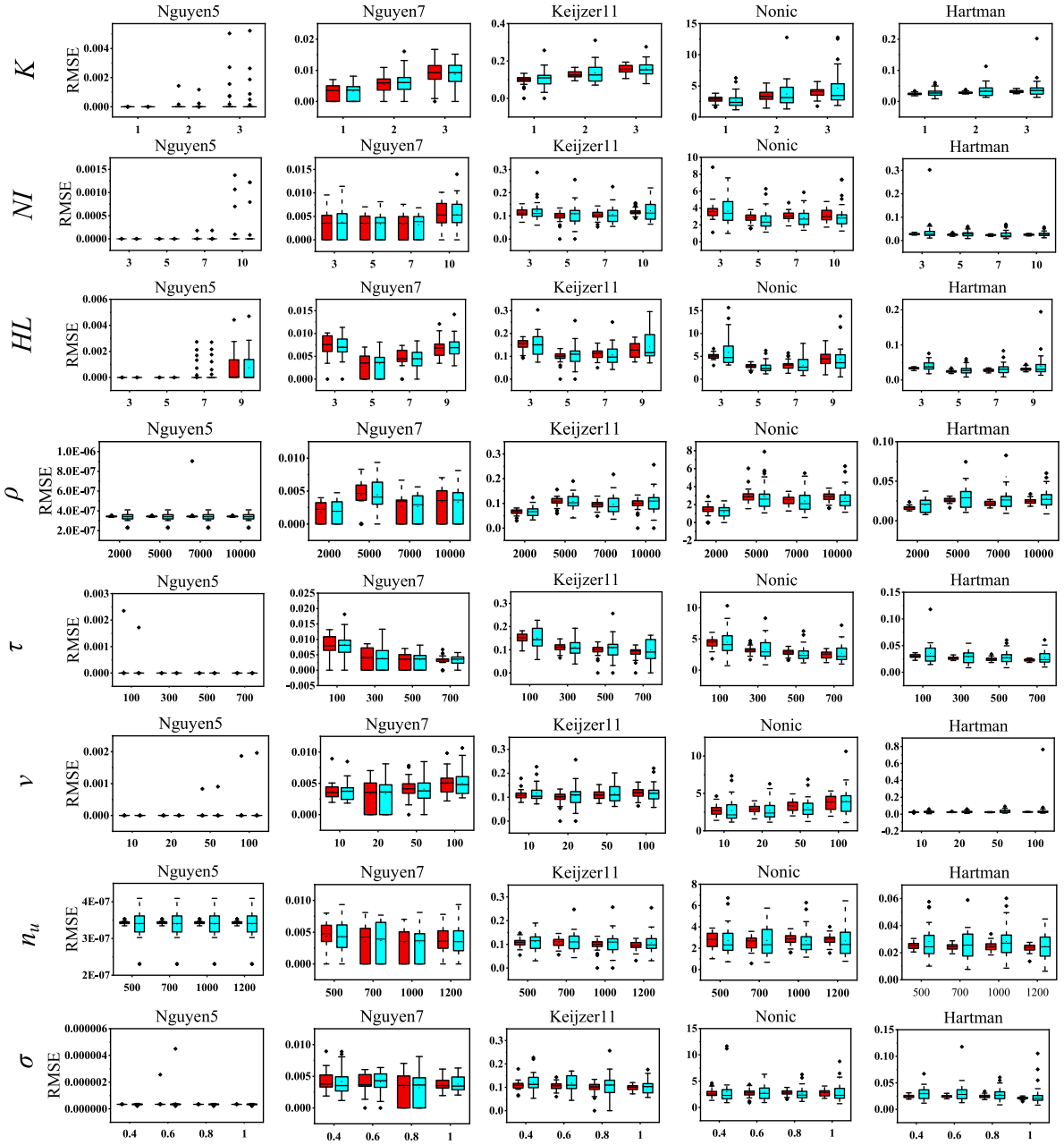


Fig. 5. Training and test error box plots of different parameter settings on five problems. The left red box plots are the distribution of training error while the blue ones are the distribution of test error. The diamonds are outliers.

also be seen for the instruction head length. For the library size limit ρ, SLGP gains significantly better performance in some cases when ρ 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 τ, on the one hand, it should not be too small compared with the library size (e.g., $\tau = 100$ versus $\rho = 10\,000$). 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 τ may make the algorithm too greedy to jump out of local optima. For example, the test error of $\tau = 700$ for problem Nguyen7 is slightly larger than setting τ to 300 or 500. For ν, n_u, and σ, different settings show a statistically similar training and test performance in most of the problems. From the parameter sensitivity analysis, we conclude the settings of SLGP parameters, as shown in Table III, in the subsequent experiments.

TABLE III
RECOMMENDED SETTINGS OF PARAMETERS

Parameters	K	NI	HL	ρ	τ	ν	n_u	σ
Values	1	5	5	2000	500	20	1000	0.8

V. RESULTS AND DISCUSSION

Based on the parameter setting analysis, we conduct more comprehensive experiments with the state-of-the-art algorithms. The experimental results are compared in terms of the

TABLE IV
SUCCESS RATE AND THE AVERAGE TEST PERFORMANCE (RMSE AND R^2)

Problem		SL-GEP		EPLEX		C-GSGP		ADGSGP		SLGP-4op		SLGP-8op	
		Suc.	Avg.	Suc.	Avg.	Suc.	Avg.	Suc.	Avg.	Suc.	Avg.	Suc.	Avg.
Koza2	RMSE	0.00	0.06(≈≈)	0.00	0.75(−−)	0.13	1.1×10^{5}(−−)	0.07	0.01(−−)	0.67	0.00(+)	0.27	0.00
	R^2	0.13	0.998(−−)	0.47	−1.34(−−)	0.33	−7.76(−−)	0.93	1.000(−−)	1.00	1.000(+)	1.00	1.000
Nguyen4	RMSE	0.93	0.00(+≈)	0.00	0.11(−−)	0.07	1.8×10^{5}(−−)	0.00	0.02(−−)	1.00	0.00(−)	0.77	0.00
	R^2	0.97	1.000(≈≈)	0.53	0.981(−−)	0.79	0.914(−−)	0.93	1.000(−−)	1.00	1.000(+)	1.00	1.000
Nguyen5	RMSE	0.13	0.01(−−)	0.00	0.01(−−)	0.21	0.02(−−)	0.00	0.00(−−)	0.73	0.00(−)	0.40	0.00
	R^2	0.67	0.998(−−)	0.80	0.981(−−)	0.69	0.854(≈−)	0.93	1.000(−−)	1.00	1.000(−)	0.97	0.983
Nguyen7	RMSE	0.13	0.01(−−)	0.00	0.01(−≈)	0.32	1.1×10^{5}(−−)	0.00	0.00(++)	0.00	0.00(+)	0.00	0.00
	R^2	0.97	1.000(−−)	0.97	1.000(−≈)	0.61	-2.6×10^{5}(−≈)	1.00	1.000(++)	1.00	1.000(+)	1.00	1.000
Keijzer11	RMSE	0.00	0.17(−−)	0.00	0.09(≈≈)	0.00	2.1×10^{5}(≈≈)	0.00	0.06(≈+)	0.00	0.07(+)	0.00	0.08
	R^2	0.00	0.819(−−)	0.03	0.930(≈≈)	0.00	0.035(−−)	0.00	0.974(≈+)	0.00	0.970(+)	0.00	0.955
R1	RMSE	0.00	0.32(−−)	0.00	0.21(−≈)	0.00	3.3×10^{5}(−−)	0.00	0.05(−+)	0.00	0.04(+)	0.00	0.12
	R^2	0.07	0.992(−−)	0.67	0.990(−≈)	0.53	-2.6×10^{143}(−≈)	1.00	1.000(−+)	1.00	1.000(+)	0.83	0.999
ModQua	RMSE	0.27	1.94(≈≈)	0.00	2.05(−−)	0.03	1.2×10^{6}(−−)	0.00	0.03(−+)	0.60	0.00(+)	0.07	0.11
	R^2	0.53	0.882(−−)	0.30	0.978(−−)	0.24	-4.6×10^{283}(−−)	1.00	1.000(−+)	1.00	1.000(+)	0.97	1.000
Nonic	RMSE	0.03	26.58(−≈)	0.00	363.14(−−)	0.00	1.2×10^{6}(−−)	0.00	5.33(−+)	0.00	0.32(+)	0.00	14.28
	R^2	0.33	0.957(−≈)	0.10	−76.34(−−)	0.07	-5.4×10^{296}(−−)	0.70	0.999(−+)	1.00	1.000(+)	0.33	0.988
Hartman	RMSE	0.00	0.11(−≈)	0.00	0.67(−≈)	0.00	2.26(−+)	0.00	0.12(−≈)	0.10	0.03(+)	0.00	25.52
	R^2	0.00	−4.491(−≈)	0.00	−1254.5(−≈)	0.00	−5855.1(−−)	0.00	−21.26(−≈)	0.00	0.912(+)	0.00	-1.6×10^{6}
RatPol3D	RMSE	0.00	0.49(−−)	0.00	1.4×10^{106}(−−)	0.00	2.1×10^{4}(≈−)	0.00	0.04(−−)	0.00	0.03(−)	0.00	0.02
	R^2	0.00	−5.715(−−)	0.03	-1.7×10^{214}(−−)	0.00	-3.4×10^{11}(−−)	0.13	0.992(−−)	0.03	0.998(−)	0.50	0.998
Airfoil	RMSE	0.00	5.23(−−)	0.00	3.89(≈+)			0.00	3.98≈≈	0.00	3.96(+)	0.00	4.05
	R^2	0.00	0.418(−−)	0.00	0.677(≈+)			0.00	0.661(≈≈)	0.00	0.667(≈)	0.00	0.649
Bhouse	RMSE	0.00	5.5(−−)	0.00	4.52(−≈)	0.00	4.27(≈+)	0.00	4.24≈≈	0.00	4.23(≈)	0.00	4.32
	R^2	0.00	0.663(−−)	0.00	0.772(−≈)	0.00	0.760(≈≈)	0.00	0.798(≈≈)	0.00	0.800(≈)	0.00	0.791
Tower	RMSE	0.00	49.75(−−)	0.00	32.59(−≈)			0.00	33.76(−−)	0.00	30.78(+)	0.00	32.17
	R^2	0.00	0.671(−−)	0.00	0.859(−≈)			0.00	0.849(−−)	0.00	0.874(+)	0.00	0.863
CCN	RMSE	0.00	0.29(≈≈)	0.00	0.4(≈≈)	0.00	0.25(≈≈)	0.00	0.25(≈≈)	0.00	0.67(≈)	0.00	0.36
	R^2	0.00	0.875(+≈)	0.00	0.818(≈≈)	0.00	0.977(++)	0.00	−95.05(≈≈)	0.00	0.439(≈)	0.00	0.853
win-draw-lose(RMSE,R^2)													
vs. SLGP-4op		1-3-10, 1-1-12		0-3-11, 0-3-11		0-4-8, 1-2-9		1-4-9, 1-4-9					
vs. SLGP-8op		0-6-8, 0-4-10		1-7-6, 1-7-6		2-2-8, 1-3-8		5-4-5, 5-4-5		9-2-3, 9-3-2			

[1] "+" means the algorithm significantly outperforms SLGP methods, "−" indicates the comparison method performs significantly worse than SLGP, and "≈" represents no significant difference between the algorithms. The similar notations are also adopted in Table VII.
[2] Some entries have $R^2 \approx 1.000$ and Suc.< 1.00. 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.

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 [4] value of all methods are shown in Table IV. Specifically, the success 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 familywise 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 this algorithm and SLGP-4op (SLGP-8op). In addition, SLGP-4op is compared with SLGP-8op, resulting in a single symbol behind SLGP-4op. At the bottom of the table, we give the win–draw–lose summary of the number of problems that the algorithm performs significantly better than, comparable to, and worse than SLGP-4op and SLGP-8op.

There are several conclusions that 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 has a draw on seven 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[5] 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 V. Based on the results, SLGP-4op has the best average ranking among the six methods. ADGSGP and SLGP-8op

[4] $R^2 = 1 - ([\mathtt{MSE}(\hat{y}, y)]/[\mathtt{VAR}(y)])$ 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 VAR(y) is the variance of y.

[5] The given code of C-GSGP can be downloaded from its original paper.

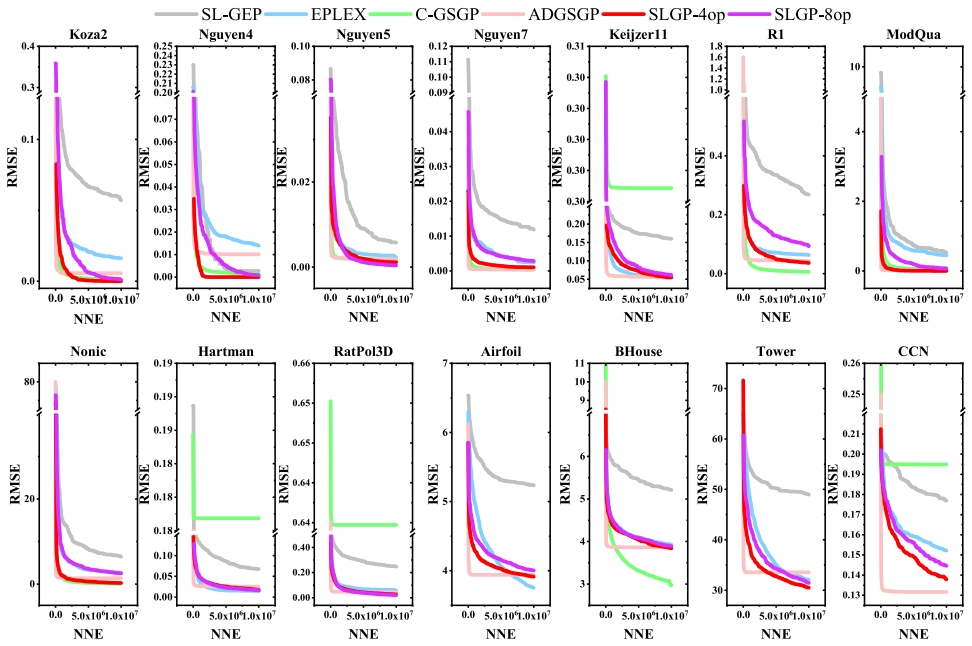


Fig. 6. Training convergence curves of the compared algorithms on the test problems.

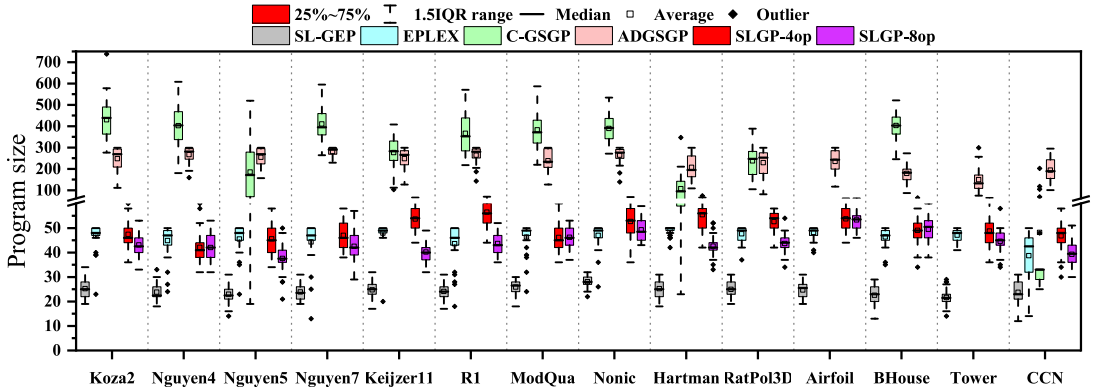


Fig. 7. Program size of the outputted solutions.

TABLE V
FRIEDMAN'S TEST AND ITS POST-HOC ANALYSIS

Algorithm	SL-GEP	EPLEX	C-GSGP	ADGSGP	SLGP-4op	SLGP-8op
mean rank	4.36	4.43	4.36	2.79	2.00	3.07
p-value	7.75E-05					
vs. SLGP-4op	0.013	0.009	0.013	1.00		
vs. SLGP-8op	1.00	0.82	1.00	1.00	1.00	

are the second tier and have a similar mean ranking. SL-GEP, EPLEX, and C-GSGP have a 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 converges 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,

TABLE VI
AVERAGE TRAINING TIME (IN SECONDS) ON ALL THE PROBLEMS

Problem	SL-GEP	EPLEX	C-GSGP	ADGSGP	SLGP -4op	SLGP -8op
Koza2	3.1	18.8	165.8	**1026.3**	125.2	178.9
Nguyen4	0.8	19.0	179.7	**1423.6**	17.42	140.8
Nguyen5	3.0	17.5	233.2	**1454.4**	209.0	238.0
Nguyen7	3.1	18.7	170.8	**1353.2**	204.4	259.5
Keijzer11	4.1	40.3	635.8	**1705.7**	445.8	615.2
R1	3.2	18.6	171.0	**1507.5**	188.7	237.4
ModQua	2.5	19.4	143.7	**1177.9**	171.2	215.5
Nonic	2.5	18.5	124.4	**1226.6**	173.4	215.3
Hartman	4.0	43.6	725.3	**1810.4**	474.4	495.9
RatPol3D	6.3	111.0	1800.7	**1850.2**	1141.6	1275.1
Airfoil	17.2	430.0	N/A	2379.8	4388.4	**4770.0**
Bhouse	7.5	168.5	1797.9	1491.2	1660.1	**1799.4**
Tower	56.1	1243.1	N/A	**18600.6**	16822.1	18396.4
CCN	30.6	739.1	**29407.0**	2928.5	6159.5	7219.6

TABLE VII
TEST RMSE FOR THE COMPONENT ANALYSIS

Problem	SLGP\MDP	SLGP\LibUpd	SLGP
Koza2	0.91≈	0.89≈	0.88
Nguyen4	0.07−	0.06−	0.00
Nguyen5	0.01−	0.01−	0.00
Nguyen7	0.01−	0.00−	0.00
Keijzer11	0.17−	0.12−	0.07
R1	0.33−	0.22−	0.08
ModQua	1.39−	0.70−	0.19
Nonic	46.85−	21.38−	5.38
Hartman	0.06−	0.05−	0.02
RatPol3D	0.17−	0.13−	0.03
Airfoil	4.86−	4.65−	4.05
Bhouse	4.92−	4.71−	4.32
Tower	48.6−	43.27−	32.17
CCN	0.37≈	0.37≈	0.36
SLGP\MDP → SLGP\LibUpd → SLGP			

[1] The "$A \rightarrow B$" means "A is significantly worse than B" based on a Friedman's test and Bonferroni method with a significance level of 0.05.

the program size of the two SLGP methods maintains roughly at the level from 40 to 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 $\Phi(Ł)$ and N_s input semantic matrices in $\mathbb{S}_{\text{in}}(Ł)$, then Ł contains $M \times N_s + N_s$ semantic matrices, $M \times N_s$ for $\mathbb{S}_{\text{out}}(Ł)$, and N_s for $\mathbb{S}_{\text{in}}(Ł)$. Since each semantic matrix has $N \times K$ floating point numbers (i.e., N input cases and K available registers), Ł 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, Ł 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 is shown in Table VI, where the largest training time is marked in bold.

In general, the SGP methods (i.e., C-GSGP, ADGSGP, and SLGP) have a much longer training time than those without semantic information (i.e., SL-GEP and EPLEX). The largest training time can be seen in different SGP methods. Among these SGP 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 times than the other SGP methods.

D. Component Analysis

SLGP has two new components: 1) MDP and 2) 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 called SLGP\MDP. The second one arbitrarily updates the library by generating random instructions and replacing random existing instructions in the library, called 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 14 SR problems is 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 of the linear GP.

These results imply two main reasons for the superior performance of SLGP. First, 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. Second, 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 article proposed a new SLGP to solve SR 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 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 method and an update method are also developed to further improve the learning ability of SLGP. This article performs a comprehensive experiment to compare SLGP with four recently proposed GP methods on 14 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 the future. For example, the time consumption of SLGP is much larger than the nonsemantic GP methods. It is the 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, the adaptive parameter controlling strategies can be introduced to SLGP to further enhance the performance.

REFERENCES

[1] J. Luna, J. Romero, C. Romero, and S. Ventura, "On the use of genetic programming for mining comprehensible rules in subgroup discovery," *IEEE Trans. Cybern.*, vol. 44, no. 12, pp. 2329–2341, Dec. 2014.

[2] A. Lensen, B. Xue, and M. Zhang, "Genetic programming for evolving a front of interpretable models for data visualization," *IEEE Trans. Cybern.*, vol. 51, no. 11, pp. 5468–5482, Nov. 2021.

[3] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 415–432, Jun. 2018.

[4] M. Filipovic, G. Sladic, B. Milosavljevic, and G. Milosavljevic, "Applying SMT algorithms to code analysis," in *Proc. Int. Conf. Appl. Internet Inf. Technol.*, 2016, pp. 163–170.

[5] S. Nguyen, M. Zhang, and K. C. Tan, "Surrogate-assisted genetic programming with simplified models for automated design of dispatching rules," *IEEE Trans. Cybern.*, vol. 47, no. 9, pp. 2951–2965, Sep. 2017.

[6] F. Zhang, Y. Mei, S. Nguyen, and M. Zhang, "Evolving scheduling heuristics via genetic programming with feature selection in dynamic flexible job-shop scheduling," *IEEE Trans. Cybern.*, vol. 51, no. 4, pp. 1797–1811, Apr. 2021.

[7] S. Asha and R. R. Hemamalini, "Synthesis of adder circuit using cartesian genetic programming," *Middle-East J. Sci. Res.*, vol. 23, no. 6, pp. 1181–1186, 2015.

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

[9] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Stat. Comput.*, vol. 4, no. 2, pp. 87–112, 1994.

[10] C. G. Johnson, "Deriving genetic programming fitness properties by static analysis," in *Proc. Eur. Conf. Genet. Program.*, 2002, pp. 298–307.

[11] S. Ruberto, L. Vanneschi, and M. Castelli, "Genetic programming with semantic equivalence classes," *Swarm Evol. Comput.*, vol. 44, pp. 453–469, Feb. 2019.

[12] W. La Cava and J. H. Moore, "Semantic variation operators for multidimensional genetic programming," Apr. 2019, *arXiv:1904.08577*.

[13] M. Castelli, L. Vanneschi, and S. Silva, "Semantic search-based genetic programming and the effect of intron deletion," *IEEE Trans. Cybern.*, vol. 44, no. 1, pp. 103–113, Jan. 2014.

[14] M. Graff, E. S. Tellez, S. Miranda-Jiménez, and H. J. Escalante, "EvoDAG: A semantic genetic programming Python library," in *Proc. IEEE Int. Autumn Meeting Power Electron. Comput.*, 2016, pp. 1–6.

[15] S. Ruberto, V. Terragni, and J. Moore, "SGP-DT: Semantic genetic programming based on dynamic targets," in *Proc. Eur. Conf. Genet. Program.*, 2020, pp. 167–183.

[16] Q. U. Nguyen, X. H. Nguyen, and M. O'Neill, "Semantic aware crossover for genetic programming: The case for real-valued function regression," in *Proc. Eur. Conf. Genet. Program.*, 2009, pp. 292–302.

[17] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Semantics-based crossover for program synthesis in genetic programming," in *Proc. Int. Conf. Artif. Evol.*, 2018, pp. 58–71.

[18] S. Ruberto, L. Vanneschi, M. Castelli, and S. Silva, "ESAGP: A semantic GP framework based on alignment in the error space," in *Proc. Eur. Conf. Genet. Program.*, 2014, pp. 150–161.

[19] T. P. Pawlak, B. Wieloch, and K. Krawiec, "Semantic backpropagation for designing search operators in genetic programming," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 326–340, Jun. 2015.

[20] A. Moraglio, K. Krawiec, and C. G. Johnson, "Geometric semantic genetic programming," in *Proc. Parallel Problem Solving Nat. XII*, 2012, pp. 21–31.

[21] Q. Chen, B. Xue, and M. Zhang, "Improving generalization of genetic programming for symbolic regression with angle-driven geometric semantic operators," *IEEE Trans. Evol. Comput.*, vol. 23, no. 3, pp. 488–502, Jun. 2019.

[22] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," *Adv. Genet. Program.*, vol. 1, pp. 311–331, Aug. 1994.

[23] P. Nordin, "Evolutionary program induction of binary machine code and its applications," Ph.D. dissertation, Dept. Comput. Sci., Univ. Dortmund, Dortmund, Germany, 1997.

[24] M. Brameier and W. Banzhaf, *Linear Genetic Programming*. New York, NY, USA: Springer, 2007.

[25] L. F. Dal Piccol Sotto and V. V. De Melo, "A probabilistic linear genetic programming with stochastic context-free grammar for solving symbolic regression problems," in *Proc. Genet. Evol. Comput. Conf.*, 2017, pp. 1017–1024.

[26] T. Hu, J. L. Payne, W. Banzhaf, and J. H. Moore, "Robustness, evolvability, and accessibility in linear genetic programming," in *Proc. Eur. Conf. Genet. Program.*, 2011, pp. 13–24.

[27] T. Hu, J. L. Payne, W. Banzhaf, and J. H. Moore, "Evolutionary dynamics on multiple scales: A quantitative analysis of the interplay between genotype, phenotype, and fitness in linear genetic programming," *Genet. Program. Evolv. Mach.*, vol. 13, no. 3, pp. 305–337, 2012.

[28] T. Hu, W. Banzhaf, and J. H. Moore, "Robustness and evolvability of recombination in linear genetic programming," in *Proc. Eur. Conf. Genet. Program.*, 2013, pp. 97–108.

[29] J. F. Miller, "An empirical study of the efficiency of learning Boolean functions using a cartesian genetic programming approach," in *Proc. Genet. Evol. Comput. Conf.*, vol. 2, pp. 1135–1142, 1999.

[30] M. Brameier and W. Banzhaf, "A comparison of linear genetic programming and neural networks in medical data mining," *IEEE Trans. Evol. Comput.*, vol. 5, no. 1, pp. 17–26, Feb. 2001.

[31] C. Fogelberg, "Linear genetic programming for multi-class classification problems," Ph.D. dissertation, Dept. School Math. Stat. Comput. Sci., Victoria Univ. Wellington, Wellington, New Zealand, 2005.

[32] C. Downey, M. Zhang, and W. N. Browne, "New crossover operators in linear genetic programming for multiclass object classification," in *Proc. Annu. Genet. Evol. Comput. Conf.*, 2010, pp. 885–892.

[33] L. F. D. P. Sotto and V. V. de Melo, "Studying bloat control and maintenance of effective code in linear genetic programming for symbolic regression," *Neurocomputing*, vol. 180, pp. 79–93, Mar. 2016.

[34] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA, USA: MIT Press, 1994.

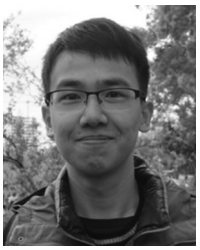
[35] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Extending program synthesis grammars for grammar-guided genetic programming," in *Proc. Parallel Problem Solving Nat. XV*, 2018, pp. 197–208.

[36] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill, "A multi-level grammar approach to grammar-guided genetic programming: The case of scheduling in heterogeneous networks," *Genet. Program. Evol. Mach.*, vol. 20, no. 2, pp. 245–283, 2019.

[37] A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete, and D. Whitley, "Linear combination of distance measures for surrogate models in genetic programming," in *Proc. Parallel Problem Solving Nat. XV*, 2018, pp. 220–231.

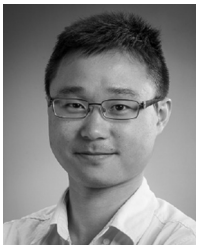
[38] Q. Chen, B. Xue, and M. Zhang, "Instance based transfer learning for genetic programming for symbolic regression," in *Proc. IEEE Congr. Evol. Comput.*, 2019, pp. 3006–3013.

[39] Q. Chen, B. Xue, and M. Zhang, "Genetic programming for instance transfer learning in symbolic regression," *IEEE Trans. Cybern.*, vol. 52, no. 1, pp. 25–38, Jan. 2022.
[40] J. Liang and Y. Xue, "An adaptive GP-based memetic algorithm for symbolic regression," *Appl. Intell.*, vol. 50, no. 11, pp. 3961–3975, 2020.
[41] L. Vanneschi, M. Castelli, and S. Silva, "A survey of semantic methods in genetic programming," *Genet. Program. Evol. Mach.*, vol. 15, no. 2, pp. 195–214, 2014.
[42] L. Beadle and C. G. Johnson, "Semantically driven crossover in genetic programming," in *Proc. IEEE Congr. Evol. Comput.*, Jun. 2008, pp. 111–116.
[43] L. Beadle and C. G. Johnson, "Semantically driven mutation in genetic programming," in *Proc. IEEE Congr. Evol. Comput.*, May 2009, pp. 1336–1342.
[44] L. Beadle and C. G. Johnson, "Semantic analysis of program initialisation in genetic programming," *Genet. Program. Evol. Mach.*, vol. 10, no. 3, pp. 307–337, 2009.
[45] N. Q. Uy, N. X. Hoai, M. Oneill, R. I. Mckay, and E. Galvanlopez, "Semantically-based crossover in genetic programming: Application to real-valued symbolic regression," *Genet. Program. Evol. Mach.*, vol. 12, no. 2, pp. 91–119, 2011.
[46] N. Q. Uy, N. X. Hoai, M. Oneill, R. I. Mckay, and D. N. Phong, "On the roles of semantic locality of crossover in genetic programming," *Inf. Sci.*, vol. 235, pp. 195–213, Jun. 2013.
[47] N. Q. Uy, E. Murphy, M. O'Neill, and N. X. Hoai, "Semantic-based subtree crossover applied to dynamic problems," in *Proc. 3rd Int. Conf. Knowl. Syst. Eng.*, 2011, pp. 78–84.
[48] N. Q. Uy, N. X. Hoai, and M. O'Neill, "Examining the landscape of semantic similarity based mutation," in *Proc. Genet. Evol. Comput. Conf.*, 2011, pp. 1363–1370.
[49] J. McDermott, A. Agapitos, A. Brabazon, and M. O'Neill, "Geometric semantic genetic programming for financial data," in *Proc. Eur. Conf. Appl. Evol. Comput.*, 2014, pp. 215–226.
[50] M. Castelli, L. Manzoni, L. Vanneschi, S. Silva, and A. Popovic, "Self-tuning geometric semantic genetic programming," *Genet. Program. Evol. Mach.*, vol. 17, pp. 55–74, Mar. 2016.
[51] A. Hara, J. Kushida, and T. Takahama, "Deterministic geometric semantic genetic programming with optimal mate selection," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, 2016, pp. 3387–3392.
[52] A. Hara, J.-I. Kushida, and T. Takahama, "Time series prediction using deterministic geometric semantic genetic programming," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, 2019, pp. 1945–1949.
[53] L. Oliveira, F. Otero, and G. Pappa, "A dispersion operator for geometric semantic genetic programming," in *Proc. Genet. Evol. Comput. Conf.*, 2016, pp. 773–780.
[54] L. Vanneschi, M. Castelli, K. Scott, and A. Popovic, "Accurate high performance concrete prediction with an alignment-based genetic programming system," *Int. J. Concr. Struct. Mater.*, vol. 12, no. 1, p. 72, 2018.
[55] M. Castelli, L. Vanneschi, S. Silva, and S. Ruberto, "How to exploit alignment in the error space: Two different GP models," in *Proc. Genet. Program. Theory Pract. XII*, 2015, pp. 133–148.
[56] L. Vanneschi, K. Scott, and M. Castelli, "A multiple expression alignment framework for genetic programming," in *Proc. Eur. Conf. Genet. Program.*, 2018, pp. 166–183.
[57] Q. Chen, B. Xue, Y. Mei, and M. Zhang, "Geometric semantic crossover with an angle-aware mating scheme in genetic programming for symbolic regression," in *Proc. Eur. Conf. Genet. Program.*, 2017, pp. 229–245.
[58] Q. Chen, M. Zhang, and B. Xue, "Geometric semantic genetic programming with perpendicular crossover and random segment mutation for symbolic regression," in *Proc. Simulat. Evol. Learn.*, 2017, pp. 422–434.
[59] T. P. Pawlak, "Geometric semantic genetic programming is overkill," in *Proc. Eur. Conf. Genet. Program.*, 2016, pp. 246–260.
[60] H. Q. Nhat, S. Chand, H. K. Singh, and T. Ray, "Genetic programming with mixed-integer linear programming-based library search," *IEEE Trans. Evol. Comput.*, vol. 22, no. 5, pp. 733–747, Oct. 2018.
[61] M. Virgolin, T. Alderliesten, and P. A. Bosman, "Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression," in *Proc. Genet. Evol. Comput. Conf.*, 2019, pp. 1084–1092.
[62] R. Ffrancon and M. Schoenauer, "Memetic semantic genetic programming," in *Proc. Genet. Evol. Comput. Conf.*, 2015, pp. 1023–1030.
[63] T. P. Pawlak and K. Krawiec, "Competent geometric semantic genetic programming for symbolic regression and boolean function synthesis," *Evol. Comput.*, vol. 26, no. 2, pp. 177–212, 2018.
[64] J. Zhong, Y.-S. Ong, and W. Cai, "Self-learning gene expression programming," *IEEE Trans. Evol. Comput.*, vol. 20, no. 1, pp. 65–80, Feb. 2016.
[65] H. Xie and M. Zhang, "Balancing parent and offspring selection in genetic programming," in *Proc. 22nd Adv. Artif. Intell. Aust. Joint Conf.*, 2009, pp. 454–464.
[66] Z. Huang, J. Zhong, L. Feng, Y. Mei, and W. Cai, "A fast parallel genetic programming framework with adaptively weighted primitives for symbolic regression," *Soft Comput.*, vol. 24, pp. 7523–7539, May 020.
[67] W. L. Cava, T. Helmuth, L. Spector, and J. H. Moore, "A probabilistic and multi-objective analysis of lexicase selection and ϵ-Lexicase selection," *Evol. Comput.*, vol. 27, no. 3, pp. 377–402, 2018.



Zhixing Huang received the B.S. and M.S. degrees from the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China, in 2018 and 2020, respectively. He is currently pursuing the Ph.D. degree with the School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand.

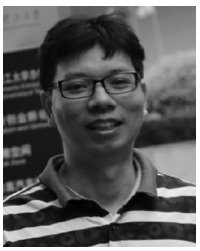
His research interests include evolutionary computation (e.g., genetic programming and its applications), combinatorial optimization, and program synthesis.



Yi Mei (Senior Member, IEEE) received the B.Sc. and Ph.D. degrees from the University of Science and Technology of China, Hefei, China, in 2005 and 2010, respectively.

He is currently a Senior Lecturer with the School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand. His research interests include evolutionary scheduling and combinatorial optimization, machine learning, genetic programming, and hyper-heuristics. He has over 150 fully referred publications, including the top journals in EC and Operations Research, such as IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, IEEE TRANSACTIONS ON CYBERNETICS, *Evolutionary Computation*, *European Journal of Operational Research*, and *ACM Transactions on Mathematical Software*.

Dr. Mei is a reviewer of over 50 international journals. He is an editorial board member/associate editor of four International Journals, and a Guest Editor of a special issue of the Genetic Programming and Evolvable Machine journal. He serves as a Vice Chair of the IEEE CIS Emergent Technologies Technical Committee, and a member of Intelligent Systems Applications Technical Committee.



Jinghui Zhong (Senior Member, IEEE) received the Ph.D. degree from the School of Information Science and Technology, Sun YAT-SEN University, Guangzhou, China, in 2012.

He is currently a Professor with the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China. From 2013 to 2016, he worked as a Postdoctoral Research Fellow with the School of Computer Engineering, Nanyang Technological University, Singapore. His research interests include evolutionary computation, machine learning, and agent-based modeling.