This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

Self-Learning Gene Expression Programming

Jinghui Zhong, Yew-Soon Ong, and Wentong Cai

***Abstract*—In this paper, a novel self-learning gene expression programming (GEP) methodology named SL-GEP is proposed to improve the search accuracy and efficiency of GEP. In contrast to the existing GEP variants, the proposed SL-GEP features a novel chromosome representation where each chromosome is embedded with subfunctions that can be deployed to construct the final solution. As part of the chromosome, the subfunctions are self-learned or self-evolved by the proposed algorithm during the evolutionary search. By encompassing subfunctions or any partial solution as input arguments of another subfunction, the proposed SL-GEP facilitates the formation of sophisticated, higher-order, and constructive subfunctions that improve the accuracy and efficiency of the search. Further, a novel search mechanism based on differential evolution is proposed for the evolution of chromosomes in the SL-GEP. The proposed SL-GEP is simple, generic and has much fewer control parameters than the traditional GEP variants. The proposed SL-GEP is validated on 15 symbolic regression problems and six even parity problems. Experimental results show that the proposed SL-GEP offers enhanced performances over several state-of-the-art algorithms in terms of accuracy and search efficiency.**

***Index Terms*—Even Parity Problem, Evolutionary Computation, Gene Expression Programming, Genetic Programming, Symbolic Regression Problem.**

I. Introduction

GENETIC programming (GP) is an evolutionary computation technique that has been proven to be useful for automating the design of computer programs that solve user-defined tasks [1], [2]. Since its inception by Koza in 1992 [1], many variants of GP have been developed [3]–[6]. One of the notable variants is Gene Expression Programming (GEP), which was introduced by Ferreira [5] in early 2000. The distinct feature of GEP is that it adopts a gene expression representation, which models after computer programs by using fixed-length strings instead of using parse trees as in the traditional GP. With a gene expression representation, GEP is shown to provide more concise and readable solutions than GP [7]. In the last decade, GEP has been widely used in many applications including classification problems, time series predictions and others [7]–[11]. However, due to its iterative nature, the GEP can be quite computationally intensive, especially when dealing with large scale problems. Besides, GEP contains a number of control parameters in the algorithm which require time-consuming fine tuning.

To enhance the performance of GPs on large scale and complex applications, an effective approach is to incorporate high order knowledge that can accelerate the search [12]–[18]. Considering the classical 10-bit even parity problem for example, it is extremely difficult for the traditional GP to solve this problem with only primitives "AND", "OR" and "NOT". Nevertheless, if the "XOR" function is introduced as a new primitive, the complexity of the problem can be largely reduced and thus the search process can be significantly accelerated [12]. To define such high order knowledge, however, is often domain-specific and can be non-trivial.

In the GP community, some efforts to automate the acquisition of high order knowledge have been reported. For example, Angeline and Pollack [19], as well as Kameya [16], tried to mine for promising partial solutions as high order knowledge from historical search information, and then ensure their survival during the evolutionary process. Meanwhile, Koza [15] considered the high order knowledge as forms of Automatically Defined Functions (ADFs) and proposed to evolve ADFs automatically by GP during the evolutionary process. The ADFs are subfunctions that provide subsolutions to specific subproblems. The ADFs can then be combined to provide solutions to larger problems. Thus far, ADFs have been shown to be very effective for improving the performances of GP on a series of problems [15], [20], [21]. Recently, Walker and Miller [22] also considered the use of ADFs in an Embedded Cartesian Genetic Programming (ECGP), which reportedly outperformed GP and GP with ADFs on a variety of problems.

In contrast to the previous studies, little research on incorporating high order knowledge such as ADFs to enhance GEP have been reported to date. Ferreira [23] had conducted a preliminary research on the use of ADFs to arrive at an enhanced GEP (named GEP-ADF). However, the mechanisms of GEP-ADF make it inefficient or non-scalable to large scale complex problems. Besides, GEP-ADF has many more new operators and control parameters than the traditional GEP, which makes it inconvenient for practical usage.

To address the above mentioned issues, this paper presents a novel Self-Learning GEP (SL-GEP) methodology. In the proposed SL-GEP, each chromosome is designed to comprise a main program and a set of ADFs, all of which are represented using a gene expression approach. Each ADF is then a subfunction for solving a subproblem, and is combined in the main program to solve the given problem of interest. Being part of the chromosome, the ADFs self-learn and are thus automatically designed in the SL-GEP along with the search as evolution progresses online. In the initialization step, all ADFs encoded in each chromosome have been randomly generated. Then in each generation, the evolutionary operators including mutation and crossover shall evolve the ADFs of the chromosomes accordingly. Through the selection operator, ADFs that lead to high quality solutions are more likely to

Jinghui Zhong, Yew-Soon Ong, and Wentong Cai are with the School of Computer Engineering, Nanyang Technological University, Singapore, 639798 (e-mail:jinghuizhong@gmail.com; ASYSOng@ntu.edu.sg; ASWTCAI@ntu.edu.sg;).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

survive across generations. In this manner, high quality ADFs are evolved along with the evolutionary search process. A distinct feature of our SL-GEP is that our ADFs encompass other ADFs and/or any subtree of the main program as input arguments, resulting in the formation of complex ADFs or C-ADFs. This feature offers significant flexibility over the classical GEP search as well as the GEP-ADF in the design of sophisticated and constructive ADFs that showcases enhanced accuracy and search efficiency in the search. In addition, a novel search mechanism based on differential evolution (DE) [24] is presented for the effective and efficient evolution of the solution population. The proposed SL-GEP is easy to implement, generic, and contains much fewer control parameters than GEP and GEP-ADF. In the experimental section, the proposed SL-GEP is comprehensively validated using 15 symbolic regression benchmarks and six even-parity problems. Results obtained demonstrate that the SL-GEP outperforms several state-of-the-art GP variants, in terms of accuracy and search efficiency.

The rest of the paper is organized as follows: Section II describes related background techniques. Section III describes the proposed SL-GEP. Section IV presents the experimental studies. At last, Section V draws the conclusions.

II. Preliminaries

This section provides a brief introduction and some background knowledge on the techniques considered in the paper. In general, GP has been widely used in various applications and one of the most common application areas is symbolic regression [25], [26]. Thus, a formal definition of the symbolic regression problem is presented here to aid readers in comprehending our proposed methodology better. Then the traditional gene expression approach of GEP [5] is presented, followed by a description of the extended gene expression approach that incorporates ADFs as introduced in GEP-ADF [23]. Finally, other related GEP variants are discussed.

A. Symbolic Regression Problem

Given a set of measurement data which consist of input variable values and output responses, the symbolic regression problem involves finding a mathematical formula $S(\cdot)$ to describe the relationships between input variables and the outputs. The mathematical formula $S(\cdot)$ derived can provide insights into the system that generates the data and thus can be subsequently used for predicting the output of new input variables.

Formally, the i-th sample data can be expressed as:

$$[v_{i,1}, v_{i,2}, ..., v_{i,n}, o_i] \tag{1}$$

where n is the number of input variables, $v_{i,j}$ is the j-th variable value of the i-th sample data, and o_i is the corresponding output. The formula $S(\cdot)$ is comprised of function symbols (e.g., $+, \times, sin$), variables and constants. The quality of a $S(\cdot)$ is evaluated by its fitting accuracy for given data. This is commonly achieved using the root-mean-square-error ($RMSE$):

$$f(S(\cdot)) = \sqrt{\frac{\sum_{i=1}^{N}(y_i - o_i)^2}{N}} \tag{2}$$

where y_i is the output of $S(\cdot)$ for the i-th input data, o_i is the true output of the i-th input data, and N is the number of data to be fitted. Therefore, given a function set and variable set as building blocks, the task to solve for a symbolic regression problem is to find the optimal formula $S(\cdot)^*$ that minimizes the $RMSE$ for the given data:

$$S(\cdot)^* = \arg\min_{S(\cdot)} f(S(\cdot)) \tag{3}$$

B. Traditional Gene Expression Approach of GEP

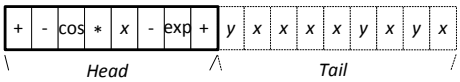

Fig. 1. The structure of traditional gene expression chromosome.

In traditional GP, the mathematical formula $S(\cdot)$ is represented by tree structures that comprise of two types of nodes: function and terminal. A function node has one or multiple children (e.g., $+$, sin), while a terminal node is a leaf node without any child (e.g., variables, constants). Instead of using parse trees as in the traditional GP, GEP uses fixed-length strings to represent mathematical formulas. As shown in Fig. 1, a chromosome is represented by a string of symbols that comprises of two parts, namely *Head* and *Tail*. Each element of *Head* is a function or a terminal, while each element of *Tail* can only represent a terminal. For example, given a function set

$$\Psi = \{+, -, *, /, cos, exp\} \tag{4}$$

and a terminal set

$$\Gamma = \{x, y\} \tag{5}$$

a typical gene expression chromosome of length 17 can be represented as:

$$X = [+, -, cos, *, x, -, exp, +, y, x, x, x, x, y, x, y, x] \tag{6}$$

Each gene expression chromosome can be converted to an equivalent expression tree (ET) using a breadth first traversal scheme. For example, the chromosome depicted in (6) can be converted to the ET form shown in Fig. 2 (a), which can be expressed mathematically as

$$(exp(x) * (x + x) - x) + cos(y - x) \tag{7}$$

The lengths of both the *Head* (h) portion and the *Tail* (l) portion are kept as fixed. In order to ensure that any chromosome can be properly converted into a valid ET, h and l are imposed with the following constraint:

$$l = h \cdot (u - 1) + 1 \tag{8}$$

where u is the maximum number of children in the functions. For example, if $u = 2$, then we have $l = h + 1$.

Note that in each chromosome, there may exist redundant elements in the *Tail* section, which are not used in the construction of ET in the current search generation. Nevertheless, it is worth noting that these unused elements may become useful in subsequent generations as the search evolves. For

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

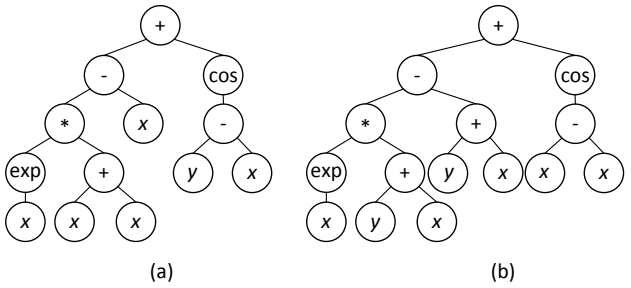

Fig. 2. Examples of two ETs. (a) the original ET. (b) the modified ET after mutating a symbol of the original ET.

instance, referring once again to (6), if the fifth symbol in this chromosome (i.e., x) evolves to $+$, the 14-th and 15-th elements (i.e., y and x) change from being deemed as redundant to become useful for constructing the converged ET, which is illustrated in Fig. 2 (b). In addition, since the number of nodes in the final ET never exceeds the predefined length of the chromosomes, the gene expression approach has a tendency of producing short programs, thus avoiding the bloating limitation of GP.

In the classic GEP, the evolutionary operators include initialization, evaluation, elitist, selection, mutation, inversion, IS-Transposition, Root-Transposition, one-point crossover and two-point crossover. The readers are referred to [5], [8] for the implementation details of GEP.

C. Extended Gene Expression Technique Used in GEP-ADF

In [23], Ferreira combined ADF with GEP to arrive at the GEP-ADF. In GEP-ADF, each chromosome consists of a number of conventional genes and a homeotic gene, as illustrated in Fig. 3. All of the conventional genes and the homeotic gene are represented using gene expression. Each conventional gene encodes a sub ET, and serves as an ADF. The homeotic gene fuses different ADFs via linking functions (e.g.,$+$, $*$), and encodes the main program that generates the final output. The function set and terminal set of ADFs are the same as in the traditional gene expression approach. The function set of the homeotic gene, on the other hand, comprises of the linking functions, while the terminal set of homeotic gene contains only ADFs.

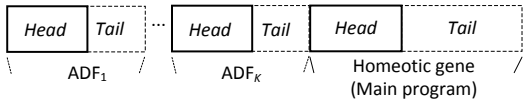

Fig. 3. The structure of the gene expression chromosome in GEP-ADF.

For example, consider the following chromosome:

$$[*,+,-,y,x,x,z,+,/,-,x,y,z,x, \\ *,*,sin,2,+,1,2,1,2,1,2] \tag{9}$$

This chromosome encodes two ADFs and one homeotic gene (i.e., the main program). An example ET of the GEP-ADF is illustrated in Fig. 4. It can be observed that both ADF_1 and ADF_2 are used twice in the homeotic gene. Further, it should

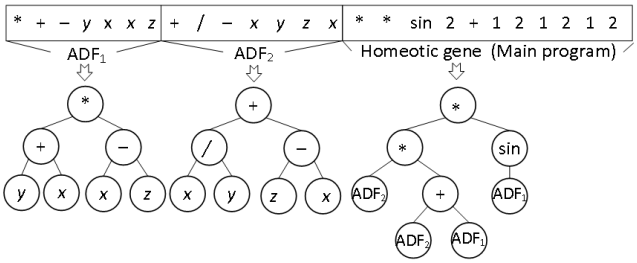

Fig. 4. An example chromosome of GEP-ADF and the corresponding ET.

be noted with this chromosome representation, ADFs can only be used as terminals of the homeotic gene and contain no input argument.

D. Other Related GEP variants

Since the first introduction of GEP, various enhanced GEP variants have been proposed. Existing works mainly focus on hybridizing new operations or adaptive parameter controlling strategies with GEP to improve the search efficiency. For example, Liu et al. [27] proposed an extended GEP which adaptively replaces worse individuals with randomly generated individuals, so as to increase the population diversity. A hybrid selection method was also adopted to improve the search efficiency. Similarly, Zhang and Xiao [28] proposed a modified GEP that dynamically changes the population size as a way to avoid local stagnation. Instead of first building a tree and then traversing the tree for fitness evaluation, Peng et al. [29] proposed a new method to evaluate fitness of gene expression chromosomes without building the expression tree. This method was reported to be capable of reducing the computational efforts of the GEP. Elena et al. [30] presented an adaptive GEP that adaptively adjusts the number of genes used in the chromosomes. Huang studied the schema theory of GEP and developed some theorems that describe the propagation of schema from one generation to another.

In contrast to previous works, this paper extends the canonical GEP with ADFs as part of the chromosome representation that self-evolve or self-learn along with search evolution, while at the same time deploying them for acceleration of the search. Further, this paper develops novel DE-based operators in the GEP. The proposed DE-based search mechanism not only contains much fewer control parameters, but also offers improved search performance.

III. Self-Learning Gene Expression Programming

This section describes the proposed SL-GEP methodology, which is an enhanced version of the classical GEP as well as the GEP-ADF. In contrast to the existing GEP approaches, SL-GEP introduces a novel chromosome representation that facilitates the formation of C-ADFs which embodies sophisticated and constructive high-order knowledge concisely. Further, a novel evolutionary search mechanism is proposed to simplify the evolution of chromosomes in SL-GEP. Since the ADFs and C-ADFs are self-learned along with the GEP search evolution, we label the proposed algorithm as Self-Learning GEP (SL-GEP).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

A. Proposed Chromosome Representation

In the proposed SL-GEP, each chromosome comprises of a "main program" and several ADFs, as illustrated in Fig. 5. The main program is the compressed expression of the solution, which provides the final output. Meanwhile, each ADF is a subfunction, which can be combined to solve the larger problem in the main program. Both the main program and ADFs are represented using a gene expression representation. But the main program and ADFs have different function sets and terminal sets. For the main program, the function set consists of not only functions (e.g., $+$, $-$, and sin) but also ADFs defined in the chromosome, while the terminal set consists of variables and constants (e.g., x, y, π). For ADFs, the function set consists of functions, while the terminal set consists of input arguments. Table I compares and contrasts the function sets and the terminal sets of the three gene expression approaches considered in this study.

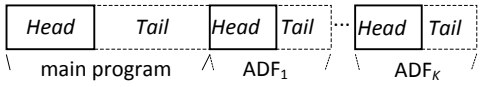

Fig. 5. The structure of the proposed chromosome representation.

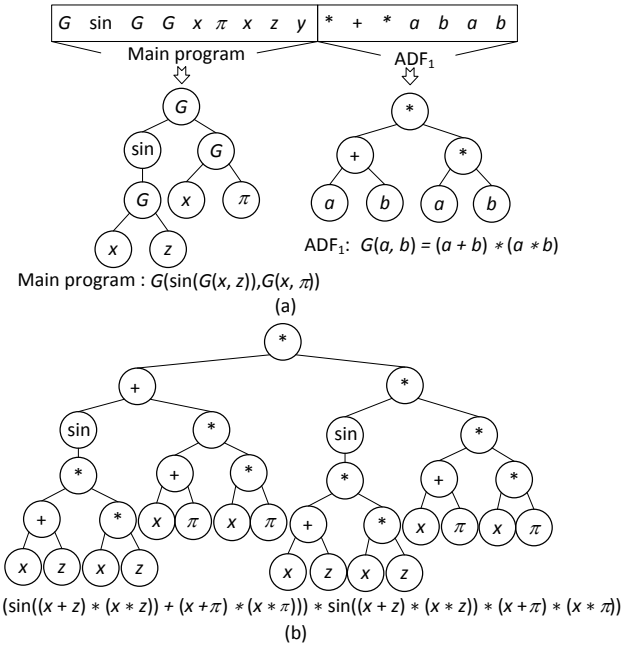

Fig. 6. An example chromosome of SL-GEP and related ETs. (a) An example chromosome of SL-GEP and the corresponding ET using ADFs. (b) The decoded ET without using ADFs.

Figure 6 shows an example chromosome of the proposed representation in SL-GEP. Suppose the value of u in (8) is 2, and h and l are set to be: $h = 4, l = 5$; the head length (h') and tail length (l') of ADFs are set to be: $h' = 3, l' = 4$. A typical chromosome with one ADF can be expressed as:

$$[G, sin, G, G, x, \pi, x, z, y, *, +, *, a, b, a, b] \tag{10}$$

TABLE I
THE FUNCTION SETS AND TERMINAL SETS IN THREE GENE EXPRESSION TECHNIQUES.

Method	Position	Function set	Terminal set
GEP	$Head$ and $Tail$	functions	variables, constants
GEP-ADF	Main Program	linking functions	ADFs
	ADFs	functions	variables, constants
SL-GEP	Main Program	functions, ADFs	variables, constants
	ADFs	functions	input arguments

where $\{+, *, sin\}$ is the function set, G is the ADF, $\{x, z, \pi\}$ is the terminal set, and $\{a, b\}$ is the set of input arguments. As shown in Fig. 6, the main program can be decoded as

$$G(sin(G(x, z)), G(x, \pi)) \tag{11}$$

and the ADF can be decoded as

$$G(a, b) = (a + b) * (a * b) \tag{12}$$

Inserting (12) into (11), we then obtain the final expression (S):

$$\begin{aligned} S =& G(sin(G(x, z)), G(x, \pi)) \\ =& G(sin((x + z) * (x * z)), (x + \pi) * (x * \pi)) \\ =& \big(sin((x + z) * (x * z)) + (x + \pi) * (x * \pi)\big) * \\ & \big(sin((x + z) * (x * z)) * (x + \pi) * (x * \pi)\big) \end{aligned} \tag{13}$$

In this way, an ET with 33 nodes can be concisely represented by two ETs with a total of only 15 nodes. Through this simple example, we hope to demonstrate the benefit of the proposed chromosome representation in terms of effectively compressing complex expressions concisely.

In addition, as can be seen in the above example, a distinct feature of the SL-GEP that differs from GEP-ADF [23] is that our approach introduces C-ADFs that have input arguments that can come in the form of variables (e.g., x, z), constants (e.g., π), ADFs, or any subtree of the main program. For example, in Fig. 6 (a), the root node of the main program is an ADF with two input arguments. The first input argument is a subtree of the main program (i.e., $sin(G(x, z))$), while the second input argument is an ADF (i.e., $G(x, \pi)$). This feature significantly improves the capabilities of chromosomes to represent sophisticated high-order expressions concisely. Specifically, the expression x^{100} can be represented as $G * G * G * G * G * G * G * G * G * G$, when $G = x^{10}$ and $*$ are defined as the ADF and linking function, respectively, in the GEP-ADF. In contrast, the same expression can be more concisely represented as $G(G(x))$ in our proposed SL-GEP by simply defining a C-ADF with $G(a) = a^{10}$. This is not only more concise but also more human readable.

B. Proposed Algorithm

The chromosome representation of SL-GEP is of fixed length and structurally similar to the traditional gene expression. Thus, the search mechanism of GEP can be easily integrated to evolve chromosomes with the proposed representation. However, the GEP contains a number of operations and control parameters that need to be carefully tuned. To

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

TABLE II
THE BASIC NOTATIONS.

Name	Summary
NP	The size of population.
X_i	The i-th target vector in the population.
Y_i	The mutant vector of X_i.
U_i	The trial vector of X_i.
D	The number of elements in each chromosome.
$x_{i,j}$	The j-th element of X_i .
$y_{i,j}$	The j-th element of Y_i .
$u_{i,j}$	The j-th element of U_i .
F	The scaling factors in mutation operation.
CR	The crossover rate.
φ	The mutation probability.
p_a	The selection probability of element a.
E_{max}	The maximum number of fitness evaluations.
c_{rate}	The probability of an element being a constant.
H_1	The $Head$ of main program in a chromosome.
T_1	The $Tail$ of main program in a chromosome.
H_2	The $Heads$ of ADFs in a chromosome.
T_2	The $Tails$ of ADFs in a chromosome.
h	The length of H_1.
l	The length of T_1.
K	The number of ADFs in each chromosome.
h'	The head length of ADFs.
l'	The tail length of ADFs.
θ	The proportion of functions and ADFs in H_1 of all chromosomes in the population.
Rand(**A**)	Randomly choose an element from **A**.
Route-wheel(**A**)	Use Route-wheel selection to choose an element from **A** based on the selection probabilities of elements in **A**.

Algorithm 1: SL-GEP.

```
Begin:
for i = 1 to NP do
    for j = 1 to D do
        x_{i,j} ← "random assignment"
Evaluate the initial population
while termination condition is not met do
    Update the frequencies of functions and variables
    for i = 1 to NP do
        /* mutation and crossover */
        F = rand(0,1); CR = rand(0,1)
        Randomly choose two individuals X_{r1} ≠ X_i, and
        X_{r2} ≠ X_i ≠ X_{r1} from current population
        Set k to be a random integer between 1 and D
        for j = 1 to D do
            Calculate the mutation probability φ by (28)
            if (rand(0,1) < CR or j=k) and
            (rand(0,1) < φ) then
                u_{i,j} ← "frequency-based assignment"
            else
                u_{i,j} = x_{i,j}
        /* Selection */
        if f(U_i) < f(X_i) then
            X_i = U_i;
End
```

solve this problem and further improve the performance, this subsection proposes a novel search mechanism based on DE operations. The procedure of the proposed SL-GEP is given in Algorithm 1, which consists of four main steps, namely initialization, mutation, crossover, and selection. The basic notations are listed in Table II.

1) Step 1 - Initialization: The first step is to generate NP random chromosomes to form an initial population. As defined in Section III-A, each chromosome can be represented by a vector of symbols labeled here as target vector:

$$X_i = [x_{i,1}, x_{i,2}, \ldots, x_{i,D}] \tag{14}$$

where i is the index of the chromosome in the population, D is the length of the chromosome, and $x_{i,j}$ is the j-th element of X_i. The value of D is computed by:

$$D = h + l + K * (h' + l') \tag{15}$$

where K is the number of ADFs in each chromosome. Note that in this paper, the maximum number of input arguments for all functions and ADFs is two, thus we have $l = h + 1$, and $l' = h' + 1$. The value of $x_{i,j}$ is assigned by a "random assignment" scheme which contains two phases. The first phase is to configure the type as a function, ADF, terminal, or input argument, randomly. Nevertheless, the assigned type must be feasible. For example, if $x_{i,j}$ belongs to the *Tail* of a main program, then $x_{i,j}$ can only be of a terminal type. Denoting the set of feasible values of the assigned type as $\boldsymbol{E}$, the second phase thus involves a random selection of a feasible value from $\boldsymbol{E}$ as the value of $x_{i,j}$.

It should be noted that the initialization step randomly initializes the chromosomes, including the ADFs. The chromosomes are then evolved iteratively via the mutation, crossover and selection evolutionary operations. Hence, the ADFs will self-learn along with the SL-GEP search, without a priori knowledge or manual configuration.

2) Step 2 - Mutation: In the second step, a mutation operation is performed on each target vector to generate a mutant vector. The traditional DE mutation is expressed as

$$Y_i = X_{r1} + F \cdot (X_{r2} - X_{r3}) \tag{16}$$

where F is the scaling factor, $r1, r2, r3$ and i are four distinct individual indices.

However, since the elements in the chromosomes are discrete symbols, the numerical operations of the traditional DE cannot be directly used. To address this issue, we decompose the DE mutation into three phases. The first phase is to obtain a difference (Δ) of two random target vectors in the current population (named *Distance Measuring*):

$$\Delta = (X_{r2} - X_{r3}) \tag{17}$$

The second phase is to scale the difference by a coefficient F (named *Distance Scaling*) :

$$\Delta' = F \cdot \Delta \tag{18}$$

The third phase is to add the scaled difference to another random target vector to create a mutant vector (named *Distance*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

Adding):

$$Y_i = X_{r1} + \Delta' \tag{19}$$

In the search space of tree structure computer programs, the above three phases are implemented as follows.

Distance Measuring: In the discrete domain, the distance between two elements can be zero (i.e., two elements are the same) or none zero. Here we use a "don't care" symbol "*" to represent a nonzero distance:

$$x_{i,j} - x_{k,j} = \begin{cases} 0, \text{if } x_{i,j} \text{ is the same as } x_{k,j} \\ *, \text{otherwise} \end{cases} \tag{20}$$

where $x_{i,j}$ and $x_{k,j}$ are the j-th element value of X_i and the j-the element value of X_k respectively. For example, $sin - exp = *$; $sin - sin = 0$. In this way, we can obtain the Δ in (17) using a binary difference vector. Each dimension of the difference vector is then computed using (20).

Distance Adding: We describe the third phase before the second phase to facilitate the understanding. The third phase adds a difference vector to a target vector, so as to create a mutant vector. Clearly, an element will remain unchanged if it is added to a zero distance:

$$x_{i,j} + 0 = x_{i,j} \tag{21}$$

If $x_{i,j}$ is added to a nonzero distance, it will evolve into a new value:

$$x_{i,j} + * = a, a \in \Omega_j \tag{22}$$

where Ω_j is the set of feasible values for the j-th dimension of X_i. The new value a is selected with a "frequency-based assignment" scheme, as described in Algorithm 2. In the "frequency-based assignment" scheme, if $x_{i,j} \in ADF$, it will be assigned in the same way as in the "random assignment" scheme. Otherwise, the type is chosen based on the frequencies of the feasible types that appear in the population. Particularly, feasible types that appear more often in the population are more likely to be selected when using the "frequency-based assignment" scheme. The selection probability of a is

$$p_a = \frac{c_a + c_0}{\sum_{b \in \Omega_j}(c_b + c_0)} \tag{23}$$

where c_a is the frequency of a in the main programs of all chromosomes in the population, and c_0 is a small constant (e.g., $c_0 = 1$). We introduce c_0 to ensure that each feasible value has at least a small selection probability even though its frequency is near to zero or equals zero.

Distance Scaling: In the second phase, the difference vector is scaled by a coefficient factor F. To achieve this, we make $x_{i,j}$ evolve into a new value at a mutation probability of φ, if it is added to a nonzero distance. The mutation probability is determined by

$$\varphi = \sum_{a \in \Omega_j} \left(\frac{F \cdot (c_a + c_0)}{\sum_{b \in \Omega_j}(c_b + c_0)}\right) = F \tag{24}$$

In this way, a small F translates to a small mutation probability, which has the effect of making the mutant vector more similar to the base target vector.

Algorithm 2: Frequency-based assignment

Input : The element to be mutated (I).
Output : The mutated element (I).

```
Begin:
if I ∈ H1 then
    if rand(0,1) < θ then
        I = Roulette-wheel(function set ⋃ ADF set).
    else
        I = Roulette-wheel(terminal set)
else if I ∈ T1 then
    I = Roulette-wheel(terminal set)
else if I ∈ H2 then
    type = Rand ({function, input argument})
    if type = function then
        I = Rand (function set)
    else
        I = Rand(input argument set)
else if I ∈ T2 then
    I = Rand(input argument set)
End
```

To summarize, the above three phases enable the DE mutation to generate mutant vectors in a discrete space. In the SL-GEP, we use the commonly used "DE/current-to-best/1" mutation scheme as expressed by

$$Y_i = X_i + F \cdot (X_{best} - X_i) + F \cdot (X_{r1} - X_{r2}) \tag{25}$$

where X_{best} is the best individual in the population. The above mechanism can be easily extended to implement the "DE/current-to-best/1". The computation process of (25) can be decomposed into two substeps, i.e., $T_i = X_i + F \cdot (X_{best} - X_i)$ and $Y_i = T_i + F \cdot (X_{r1} - X_{r2})$, where $T_i = [t_{i,1}, t_{i,2}, \ldots, t_{i,n}]$ is a temporary vector. The probabilities of $x_{i,j}$ and $t_{i,j}$ remaining unchanged in the first and second substep are $1 - F \cdot \psi(x_{pbest,j}, x_{i,j})$ and $1 - F \cdot \psi(x_{r1,j}, x_{r2,j})$ respectively, where $\psi(a, b)$ is defined as

$$\psi(a, b) = \begin{cases} 1, \text{if } a \neq b \\ 0, \text{otherwise} \end{cases} \tag{26}$$

Therefore, the probability of $x_{i,j}$ remaining unchanged after performing the two substeps is:

$$(1 - F \cdot \psi(x_{best,j}, x_{i,j})) * (1 - F \cdot \psi(x_{r1,j}, x_{r2,j})) \tag{27}$$

Hence, the mutation probability of $x_{i,j}$ is

$$\varphi = 1 - (1 - F \cdot \psi(x_{best,j}, x_{i,j})) * (1 - F \cdot \psi(x_{r1,j}, x_{r2,j})) \tag{28}$$

To improve the robustness of the algorithm and reduce the number of control parameters, we also adopt a random scheme to set the values of F in the "DE/current-to-best/1" mutation scheme:

$$F = rand(0, 1) \tag{29}$$

where $rand(a, b)$ returns a random value uniformly distributed within $[a, b]$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

3) Step 3 - Crossover: In the third step, each target vector X_i is crossover with its mutant vector Y_i to create a trial vector U_i:

$$u_{i,j} = \begin{cases} y_{i,j}, \text{if} \quad rand(0,1) < CR \quad \text{or} \quad j = k \\ x_{i,j}, \text{otherwise} \end{cases} \tag{30}$$

where CR is the crossover rate, k is a random integer between 1 and D, $u_{i,j}, y_{i,j}$ and $x_{i,j}$ are the j-th variables of U_i, Y_i and X_i respectively. Similar to F, the value of CR is set to be $CR = rand(0,1)$.

4) Step 4 - Selection: Finally, in the fourth step, the fitter solution between each pair of the target and trial vector is then chosen to form a new population:

$$X_i = \begin{cases} U_i, \text{if} \quad f\big(U_i\big) < f\big(X_i\big) \\ X_i, \text{otherwise} \end{cases} \tag{31}$$

where $f\big(X\big)$ returns the fitness value of X.

There is a repetition between the second step and the fourth step until the termination condition is met.

To summarize, the above four steps extend the DE search mechanism to evolve the encoded solutions. It should be noted that the proposed SL-GEP is a generic framework such that other state-of-the-art DEs such as JADE [31], CoDE [32] and et al. [33], [34] can be easily used as alternatives. In what follows, we study the proposed algorithm with GEP and GEP-ADF in terms of their computational complexities and control parameter sizes. According to the above procedures, the computational complexity of the proposed SL-GEP in each generation is the sum of two parts: 1) generating NP new chromosomes, 2) decoding the NP new chromosomes into the ETs and evaluating their fitness values. The frequencies of functions and variables can be updated by scanning the elements in the population with a single pass, which has a complexity of $O(NP \cdot D)$. Selecting one element based on the "frequency-based assignment" scheme is a roulette-wheel selection procedure. By using the method proposed in [35], the complexity of the above "frequency-based assignment" operation can be reduced to $O(1)$. Thus, the complexity for mutation and crossover is $O(NP \cdot D)$, and the complexity of the first part is $O(NP \cdot D)$. The complexity of the second part is problem-specific but is the same for all GEP variants. Denoting the complexity of the second part as $O(f)$, then the complexity of the proposed SL-GEP for each generation is $O(NP \cdot D) + O(f)$. Accordingly, we can obtain the complexity of GEP and GEP-ADFs as $O(NP \cdot D) + O(f)$. Thus, the complexities of our proposed SL-GEP algorithm, GEP and GEP-ADF are similar. Meanwhile, as listed in Table III, the proposed SL-GEP algorithm has only four control parameters, while the standard GEP and GEP-ADF contain eleven control parameters and twenty control parameters, respectively. From this point of view, the proposed SL-GEP algorithm contains much fewer control parameters and thus is more convenient and appropriate for practical usage.

TABLE III
NUMBER OF CONTROL PARAMETERS IN GEP, GEP-ADF, AND SL-GEP.

Algorithm	Parameters
GEP	NP, h, mutation rate (pm), inversion rate (pi), IS transposition rate (pis), RIS transposition rate ($pris$), number of genes (ng), one-point crossover rate ($pc1$), two-point crossover rate ($pc2$), gene recombination rate (pg), gene transposition rate (ptg)
GEP-ADF	$NP, h, pm, pi, pis, pris, ng, pc1, pc2, pg, ptg$, and the corresponding parameters of homeotic gene, denoted as $h', pm', pi', pis', pris', pc1', pc2', pg', ptg'$, respectively
SL-GEP	NP, h, h', k

C. Constant Creation

Numerical constant is an integral part of most mathematical formulas. Hence, the creation of numerical constants is an important part of GP and its variants towards the evolution of satisfying mathematical formulas. However, it is rather challenging for GP-like methods to precisely approximate the constant values, because the numerical constants are continuous values while the chromosome representations of GP-like methods are generally suitable for combinatorial optimization. In the literature, several methods have been proposed for solving the constant creation problem. The most common one is the "ephemeral random constant (ERC)" which was introduced by Koza [1]. In Koza's method, the ERC is treated as a special terminal symbol. The value of each ERC is assigned with a random value within a specific range when creating the initial population. Then the random ERCs are fixed and moved from one parse tree to another by the crossover operator. Ferreira proposed a new method to handle constants in GEP [36]. In Ferreira's method, an extra terminal "?" is used to represent constants in the formula. A constant pool is randomly created and assigned to each individual in the initialization. When decoding the gene expression chromosome for fitness evaluation, each "?" is assigned with a constant in the constant pool accordingly. An extra Dc domain and some special Dc operations are introduced to facilitate the constant creation process. Besides the above two methods, others such as the local search method [37], nonlinear least squares minimization [38], and EAs [39] have also been used to search for constant values.

As in GEP [8], the constant creation operation is regarded as an optional operation in the proposed SL-GEP. When constants are considered in the search process, the SL-GEP adopts the commonly used ERC to handle constants, due to its simplicity. It should be noted that other existing constant handling methods can also be considered in the proposed SL-GEP with ease. Specifically, a set of fixed random constants within a specific range (e.g., [-1,1]) have been generated in the initialization step. A new terminal symbol is introduced to represent ERCs. In the "random-assignment" and the "frequency-based assignment" procedures, each element of H_1 and T_1 has probability of c_{rate} to be assigned with a constant. When an element is to be assigned with a constant, a random constant from the constant set will be selected. The constant assigned to the element shall remain fixed for the rest of the run unless a "frequency-based assignment" mutation is performed on it.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

TABLE IV
FIFTEEN SYMBOLIC REGRESSION BENCHMARKS FOR VALIDATION.

P	Objective Function	Data set
F_1	$x^4+x^3+x^2+x$	$U[-1,1,200]$
F_2	x^5-2x^3+x	$U[-1,1,200]$
F_3	$x^6-2x^4+x^2$	$U[-1,1,200]$
F_4	x^3+x^2+x	$U[-1,1,200]$
F_5	$x^5+x^4+x^3+x^2+x$	$U[-1,1,200]$
F_6	$x^6+x^5+x^4+x^3+x^2+x$	$U[-1,1,200]$
F_7	$sin(x^2)cos(x)-1$	$U[-1,1,200]$
F_8	$sin(x)+sin(x+x^2)$	$U[-1,1,200]$
F_9	$ln(x+1)+ln(x^2+1)$	$U[0,2,200]$
F_{10}	$\sqrt{x}$	$U[0,4,200]$
F_{11}	$sin(x)+sin(y^2)$	$U[0,1,1000]$
F_{12}	$2sin(x)cos(y)$	$U[0,1,1000]$
F_{13}	$10\cdot(5+(x-3)^2+(y-3)^2+(z-3)^2+(v-3)^2+(w-3)^2)^{-1}$	$U[0.05,6.05,2000]$
F_{14}	$30\frac{(x-1)(z-1)}{y^2(x-10)}$	$x: U[0.05,2,1000]$ $y: U[1,2,1000]$ $z: U[0.05,2,1000]$
F_{15}	Unknown	4999 samples

Variable names are in order of x, y, z, v, w; $U[a,b,c]$ represents c uniform random samples from a to b; The function set of all problems is $\{+,-,\times,\div,sin,cos,e^x,ln(|x|)\}$.

TABLE V
PARAMETER SETTINGS OF THE ALGORITHMS CONSIDERED.

Algorithm		Parameter settings
GP Variants	GEP [5]	NP = 50; h = 10; pm = 0.1; pi = 0.1; $pc1$ = 0.7; $pc2$ = 0.7; pis = 0.1; $pris$ = 0.1; ng = 1
	GEP-ADF [23]	$NP = 50; pn = 1; h = h' = 10; pm = pm' = 0.044; pc1 = pc1' = 0.3; pc2 = oc2' = 0.3; pg = 0.3; pis = pis' = 0.1; pris = pris' = 0.1$
	GP [41]	$NP = 1024$; other parameters using the default settings defined in the ECJ library
	LDEP [42]	$NP = 20$; number of registers = 6; F = 0.5; CR = 0.1; constant probability = 0.05
	TreeDE [43]	$NP = 20$, $L = 6$, $x_L = 0$, $x_U = 20$, $F = 0.5$, $\lambda = 0.5$, $CR = 0.1$
Proposed methods	GEP-ADF2	NP = 50; h = 10; h' = 3; $K = 2; pm = 0.1; pi = 0.1; pc1 = 0.7; pc2 = 0.7; pis = 0.1; pris = 0.1; ng = 1$
	SL-GEP	NP =50; h = 10; h' = 3; K = 2; c_{rate} = 0.05
	SL-GEP/JADE	NP =50; h = 10; h' = 3; K = 2; p = 0.05; c = 0.1; c_{rate} = 0.05

IV. EXPERIMENTS AND COMPARISONS

This section investigates the performance of the proposed SL-GEP. First, the experiment settings, including the benchmark test problems and the performance metrics for comparison are presented. Then, the proposed SL-GEP is assessed against GEP, GEP-ADF, GP and two recently published GP variants. Finally, the impacts of important control parameters of SL-GEP are investigated.

A. Experiment Settings

This section assesses the performance of SL-GEP via solving two categories of well established problems. The first category contains 15 symbolic regression problems, as listed in Table IV. These benchmark problems are chosen from [26] and [40]. Among them, $F_1 \sim F_{14}$ are commonly used benchmark problems that have unique structural complexities with respect to the objective formulas. F_{15} emerges from an industrial problem on modeling gas chromatography measurements of the composition of a distillation tower. This problem contains 4999 records, with each record containing 25 potential input variables and one output value [1]. In Table IV, the last column describes the data set to be fitted, where $U[a,b,c]$ represents c uniform random samples from a to b. As suggested in [26], the function set of all 15 problems is set to be $\{+,-,\times,\div,sin,cos,e^x,ln(|x|)\}$.

The second category is the k-even parity problem, which requires finding a suitable boolean formula comprising k arguments. The boolean formula should return true value if and only if there are even number of arguments assigned with true values. The even parity problems are popular benchmark problems for assessing GP [26]. These are very difficult boolean optimization problems for blind random search and the traditional GP. In this paper, six even parity problems are considered in the second category, i.e., 3-parity to 8-parity problems. As suggested in [22], the function set of the six even parity problems is set to be $\{\text{AND}, \text{OR}, \text{NAND}, \text{NOR}\}$.

Seven GP variants have been chosen to pit against the proposed SL-GEP. The first algorithm is the traditional GEP [5]. The second algorithm is the GEP-ADF [23], which is a preliminary version of GEP combined with ADF. The third is the GP system developed based on the ECJ library [41]. The fourth is LDEP [42], which combines DE with linear genetic programming. The fifth is the Tree-based DE (TreeDE) [43], a discrete version of DE for solving symbolic regression problems. The sixth is a modified SL-GEP named GEP-ADF2, which replaces the proposed DE-based search mechanism with the traditional operations of GEP (e.g., selection, mutation, one-point crossover and two-point crossover). We compare SL-GEP with GEP-ADF2 to analyze the effectiveness of the proposed DE-based search mechanism. The last one is an enhanced version of SL-GEP (labeled as SL-GEP/JADE), which replaces the classical "DE/current-to-best/1" mutation strategy in SL-GEP with the "DE/current-to-pbest/1" strategy in JADE [31]. The self-adaptive parameter control strategy of JADE is also adopted in SL-GEP/JADE to adaptively adjust F and CR. We introduce SL-GEP/JADE simply to demonstrate that the proposed SL-GEP provides a generic framework where different search mechanisms can be easily accommodated.

Table V lists the parameter settings of the algorithms under consideration. The parameters of TreeDE and LDEP are configured according to what have been suggested in their original papers. As suggested in [44], the NP of GP is set to be 1024, and other parameters are set the same as the default settings in the ECJ library. The parameters of GEP and GEP-ADF are set as suggested by Ferreira in [23]. In the original paper of GEP and GEP-ADF, the author did not suggest a

[1]The data set of F_{15} can be downloaded from http://symbolicregression.com/sites/default/files/DataSets/towerData.txt.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

fixed rule for setting the head length, but suggested using different head lengths for solving different kinds of problems. As a rule of thumb, the larger number of variables a problem has, a larger head length should be used. In our experimental studies, the head length is set empirically based on the variable size of the problem. For fair comparison, we fixed the head lengths of all GEP variants to be 10 for solving small variable size problems such as $F_1 - F_{14}$, while 20 for solving large size problems like F_{15} and the six even parity problems. As for LDEP, the suggested number of registers is 6 which is infeasible for solving large scale even parity problems and F_{15}, where the numbers of variables are larger than 6. To solve this issue, the number of registers of LDEP is set to 12 for the six even parity problems and 40 for F_{15}. Our experimental studies showed that these settings produced promising results. To solve problems that require constants (i.e., F_{13} F_{14}, and F_{15}), GP used Koza's ERC that was implemented in the ECJ library, while GEP, GEP-ADF, and GEP-ADF2 used Ferreira's method as described in [8]. The LDEP used the special constant creation mechanism as proposed in its original paper, and our proposed SL-GEP and SL-GEP/JADE considered the method described in Section III-C. For all problems, each algorithm will terminate when the number of evaluations reaches a maximum of 1,000,000 (i.e., $E_{max} = 1,000,000$).

B. Performance Metrics for Comparison

For the 15 symbolic regression problems, the 10-fold cross validation method is adopted for training and testing. Specifically, for each problem, the data set is evenly divided into 10 folds. For each EA run, 9 folds are used for training while the remaining fold is used at the end to test the best solution found. There are 10 different cases, and we repeated each algorithm on each case for 10 times with different random seeds. Hence, there are altogether 100 different runs for each algorithm on each problem. During the training stage, 9 folds of data are used to evaluate the fitness of each solution, as in (2). When an algorithm converges to a solution S_i with $f(S_i) < 10^{-4}$, a successful search convergence or perfect hit is assumed. At the end of each EA run, the best solution attained is tested using (2) based on the remaining fold of data. The average testing result of the 100 EA runs is used for comparison analysis. As for the six even parity problems, the training data are all possible input assignments and the corresponding outputs. For example, the 3-parity problem contains $2^3 = 8$ different input and output pairs. These eight input and output pairs are used as the training data. The fitness value of each solution is evaluated based on (2) using all the training data. Since these is no testing data for even parity problems, for each algorithm we performed 100 independent runs on each problem. The average training performances of the 100 runs are then reported.

In the empirical studies, the first performance metric considered is the testing accuracy given by (2). This metric is regarded as most important for evaluating the performance of an algorithm. Besides, as suggested in [45], the success rate of achieving perfect hits (denoted as Suc) is adopted as the second metric. The Suc is computed by:

$$Suc = \frac{C_s}{C} \cdot 100\% \tag{32}$$

TABLE VI
COMPARISON RESULTS OF GEP, GEP-ADF2 AND SL-GEP.

Problem	GEP [5]		GEP-ADF2		SL-GEP	
	Suc	$RMSE$	Suc	$RMSE$	Suc	$RMSE$
F_1	93	0.0040 −(≈)	88	0.0036 −	100	0
F_2	8	0.0072 −(−)	25	0.0039 −	71	0.0011
F_3	11	0.0068 −(−)	46	0.0047 −	99	9.32E-6
F_4	100	0 ≈(≈)	100	0 ≈	100	0
F_5	49	0.0216 −(−)	70	0.0090 −	100	0
F_6	7	0.0519 −(−)	49	0.0188 −	91	0.0024
F_7	61	0.0071 −(≈)	27	0.0064 −	85	0.0005
F_8	99	0.0002 ≈(≈)	100	0 ≈	100	0
F_9	6	0.0148 −(−)	4	0.0105 −	27	0.0069
F_{10}	43	0.0206 −(≈)	18	0.0258 −	93	0.0016
F_{11}	100	0 ≈(≈)	99	0.0001 ≈	100	0
F_{12}	70	0.0076 −(≈)	77	0.0041 −	100	0
F_{13}	0	0.1728 −(≈)	0	0.1737 −	0	0.1591
F_{14}	0	0.2215 −(−)	0	0.1959 −	0	0.1583
F_{15}	0	56.1899−(≈)	0	55.2503−	0	52.5662
3-parity	100	0 ≈(≈)	100	0 ≈	100	0
4-parity	1	0.4323 −(−)	100	0 ≈	100	0
5-parity	0	0.5736 −(−)	100	0 ≈	100	0
6-parity	0	0.6371 −(−)	98	0.0065 ≈	100	0
7-parity	0	0.6726 −(−)	95	0.0202 ≈	100	0
8-parity	0	0.6920 −(−)	98	0.0085 ≈	100	0
−	17 (11)		12			
≈	4 (10)		9			

Symbols − and ≈ outside brackets represent that the competitor is respectively significantly worse than and similar to SL-GEP according to the Wilcoxon signed-rank test at $\alpha = 0.05$. The symbols in brackets are the corresponding results between GEP and GEP-ADF2.

where C is the number of independent runs and C_s is the number of successful runs achieving a perfect hit.

In addition, the number of fitness evaluations required to achieve a perfect hit (denoted as Run Time, RT) is adopted as the third performance metric. This metric gives an indication of the convergence speed of an algorithm. In the event an algorithm fails to achieve a perfect hit, the method described in [46] is then considered for estimating the RT:

$$RT = E_s + \frac{1 - Suc}{Suc} E_{max} \tag{33}$$

where E_s is the average number of fitness evaluations to achieve a perfect hit for the successful runs and E_{max} is the maximum number of fitness evaluations.

C. Comparisons with GEP and GEP-ADF2

Table VI summaries the Suc and $RMSE$ obtained by GEP, GEP-ADF2 and SL-GEP. The $RMSE$ is the average testing accuracy of the 100 independent runs based on 10-fold cross validation. For each problem, a Wilcoxon signed-rank test is performed to detect for significant differences between two algorithms. First, we evaluate our GEP-ADF2 against the conventional GEP [5]. It can be observed from Table VI that GEP-ADF2 significantly outperformed GEP on 11 out of the 21 problems according to the $RMSE$, and performed competitively on the remaining 10 problems. Especially, GEP-ADF2 reported much higher Suc on all six even parity problems, while GEP failed to locate the global optimum for the last four problems. This is because the solution expression of a large scale even parity problem becomes overwhelming

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

TABLE VII
EXAMPLE ADFS FOUND BY GEP-ADF2 ON F_1 TO F_6 AND THE SIX EVEN PARITY PROBLEMS.

Problem	ADFs	Main program (S)
F_1	$G_1(t_1,t_2) = t_2 + t_2 * t_1$	$G_1(x * x, G_1(x,x))$
F_2	$G_1(t_1,t_2) = t_2 * t_2 * t_1$ $G_2(t_1,t_2) = t_1 * (t_1/t_2)$	$G_2((G_2(x,x) - G_1(x,x)),x)$
F_3	$G_1(t_1,t_2) = (t_2 * t_2) * t_1$	$G_1(x/x, x - G_1(x,x))$
F_4	$G_1(t_1,t_2) = (t_1 * t_2) + t_2$	$G_1(G_1(x,x),x)$
F_5	$G_1(t_1,t_2) = t_1 + (t_2 * t_1)$	$G_1(x, G_1(G_1(x,x), x * x))$
F_6	$G_1(t_1,t_2) = t_1 * (t_2 * t_1)$ $G_2(t_1,t_2) = (t_1 * t_2) + t_1$	$G_2(G_2(x, G_2(x,x)), G_1(x,x))$
3-parity	$G_1(t_1,t_2) = (t_2 * t_1) + (t_1[!+]t_2)$	$G_1((x_1[!*]x_1), G_1(x_3,x_2))$
4-parity	$G_1(t_1,t_2) = (t_2 * t_1) + (t_1[!+]t_2)$	$G_1(G_1(x_4,x_1), G_1(x_2,x_3))$
5-parity	$G_1(t_1,t_2) = (t_1[!*]t_2)[!*](t_1 + t_2)$	$G_1(G_1(x_3,x_2),$ $G_1((x_1[!+]x_1), G_1(x_5,x_4)))$
6-parity	$G_1(t_1,t_2) = (t_1 * t_2) + (t_1[!+]t_2)$	$G_1(G_1(G_1(x_5,x_6),x_1),$ $G_1(G_1(x_3,x_4),x_2))$
7-parity	$G_1(t_1,t_2) = ((t_2[!+]t_1) + (t_1 * t_2))$	$G_1(G_1(x_5, G_1(G_1(x_4,x_1),$ $G_1(x_7,x_3))), G_1(((x_2 + x_7)$ $[!*]x_2), x_6))$
8-parity	$G_1(t_1,t_2) = ((t_1[!*]t_2) * (t_2 + t_1))$	$G_1((x_5[!*]x_5), G_1(G_1(G_1(x_4,$ $G_1(x_3,x_1)), G_1(x_7,x_8)),$ $G_1((x_6 + x_6), x_2)))$

AND, OR, NAND, and NOR are denoted as "*", "+", "[!*]", and "[!+]" respectively.

or extremely complex. Thus, such problems are very difficult for the traditional GEP to solve, especially without the use of subfunctions.

Besides the even parity problems, GEP-ADF2 also exhibited better performances on problems F_2 to F_6. Further analysis and verifications indicated that these are the kind of problems whose solutions can be compressed effectively with the use of ADFs. For example, the original expression of F_5 is"$x*x*x*x*x*x*x+x*x*x*x*x*x+x*x*x*x+x*x*x+x$", which contains 29 symbols. By using "$G(a,b) = b + b * a$" as an ADF, F_5 can be compressed as "$G(G((x*x), G(x,x)), x)$", which contains much fewer symbols. Table VII showcases several other solutions found by GEP-ADF2 on problems F_1 to F_6 and the six even parity problems. It is worth noting that the converged solutions are found to be very concise, which is made possible due to the use of ADFs. Note that GEP-ADF2 needs additional computational requirements to search for the promising ADFs. Hence, on simple problems such as F_4 and F_1, the benefit of using ADFs is not very significant. However, as the complexity of the problem increases, the benefit of ADFs becomes more evident and significant, as can be observed from the results of F_5, F_6, as well as the six even parity problems. As for $F_7 - F_{12}$, their original expressions contain only a few number of symbols. Thus, they can be expressed concisely without using ADFs. Thus, the performances of GEP-ADF2 are similar to those of GEP on these problems. These results demonstrate that the proposed GEP-ADF2 can effectively utilize the ADFs in improving the search accuracy of GEP.

Next, we evaluate SL-GEP against GEP-ADF2 to investigate the effectiveness of the proposed DE-based search mechanism for discrete problems. It can be observed that SL-GEP performed significantly better than GEP-ADF2 on 12 problems, and performed competitively on the remaining nine problems, in terms of the $RMSE$. However, according to the Suc values, SL-GEP obtained 100% success rate on 12 problems, which is superior to the GEP-ADF2. Besides, the Suc values of SL-GEP on the remaining nine problems were observed to be much better than or at least competitive to those of GEP-ADF2. These results thus indicate that the proposed DE-based search mechanism helps to further improve the performance of GEP-ADF2.

In addition, we also perform the multiple-problem Wilcoxon's test to check the behaviors of GEP, GEP-ADF2, and SL-GEP on the test suite. Table VIII summarizes the statistical results. Again, the results of the multiple-problem Wilcoxon's test indicate that the GEP-ADF2 exhibited significantly better performance than GEP, while the proposed SL-GEP exhibited significantly better performance than both GEP and GEP-ADF2, with a probability error of $p < 0.05$.

TABLE VIII
RESULTS OF THE MULTIPLE-PROBLEM WILCOXON'S TEST FOR GEP, GEP-ADF2 AND SL-GEP.

Algorithms	GEP vs SL-GEP	GEP-ADF2 vs SL-GEP	GEP vs GEP-ADF2
p-Value	9.822E-5	0.0002189	0.0005578

p-Value < 0.05 indicates that the second competitor performed significantly better than the first competitor according to the multiple-problem Wilcoxon signed-rank test at $\alpha = 0.05$, in terms of the $RMSE$.

TABLE X
RANKING OF GP, GEP-ADF, TREEDE, LDEP, SL-GEP/JADE, AND SL-GEP ACCORDING TO THE STATISTICAL TEST OF THE FRIEDMAN TEST.

Algorithm	GP	GEP-ADF	TreeDE	LDEP	SL-GEP/JADE	SL-GEP
Average Ranking	4.4524	4.9286	4.3571	3.7857	1.6429	1.8333
Friedman value	59.9451					
Critical value in $\chi^2(\alpha = 0.05)$	11.07					
CD $(\alpha = 0.05)$	1.484					

The details of calculating the Friedman value and CD can be found in [47].

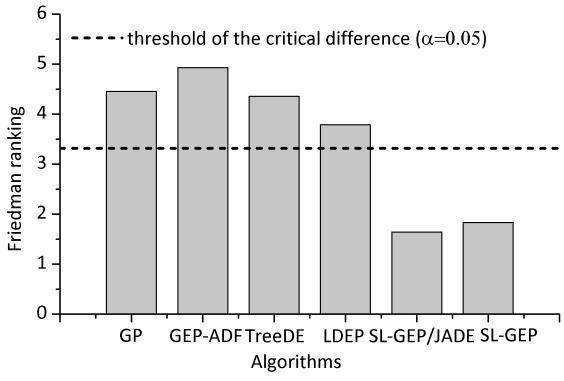

Fig. 7. Bonferroni-Dunn graphic for testing accuracies of GP, GEP-ADF, TreeDE, LDEP, SL-GEP/JADE, and SL-GEP.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

TABLE IX
COMPARE AND CONTRAST THE RESULTS OF GP, GEP-ADF, TREEDE, LDEP, SL-GEP/JADE AND SL-GEP.

Problem	GP		GEP-ADF		TreeDE		LDEP		SL-GEP/JADE		SL-GEP	
	Suc	$RMSE$	Suc	$RMSE$	Suc	$RMSE$	Suc	$RMSE$	Suc	$RMSE$	Suc	$RMSE$
F_1	47	0.0086 $-$	63	0.0154 $-$	63	0.0057 $-$	78	0.0054 $-$	100	0 $\approx$	100	0
F_2	15	0.0050 $-$	1	0.0088 $-$	5	0.0036 $-$	37	0.0021 $-$	89	0.0002 $+$	71	0.0011
F_3	4	0.0044 $-$	98	0.0002 $\approx$	2	0.0034 $-$	69	0.0008$-$	100	0 $\approx$	99	9.32E-6
F_4	95	0.0040 $-$	97	0.0006 $\approx$	100	0 $\approx$	93	0.0007 $-$	100	0 $\approx$	100	0
F_5	31	0.0125 $-$	4	0.0405 $-$	14	0.0237 $-$	19	0.0279 $-$	100	0 $\approx$	100	0
F_6	8	0.0210 $-$	0	0.0401 $-$	0	0.0292 $-$	16	0.0300 $-$	94	0.0010 $\approx$	91	0.0024
F_7	14	0.0052 $-$	11	0.0029 $-$	25	0.0034 $-$	22	0.0033 $-$	79	0.0008 $\approx$	85	0.0005
F_8	70	0.0036 $-$	72	0.0026 $-$	75	0.0030 $-$	83	0.0029 $-$	100	0 $\approx$	100	0
F_9	8	0.0049 $-$	12	0.0190 $-$	12	0.0123 $-$	24	0.0088 $-$	23	0.0064 $\approx$	27	0.0069
F_{10}	20	0.0162 $-$	77	0.0090 $-$	13	0.0064 $-$	79	0.0020 $\approx$	96	0.0007 $\approx$	93	0.0016
F_{11}	45	0.0118 $-$	84	0.0034 $-$	91	0.0011 $-$	97	0.0003 $\approx$	100	0 $\approx$	100	0
F_{12}	58	0.0055 $-$	99	0.00025 $\approx$	62	0.0046 $-$	99	0.0001 $\approx$	100	0 $\approx$	100	0
F_{13}	0	0.1885 $-$	0	0.1714 $-$	0	0.1830 $-$	0	0.1623 $-$	0	0.1587 $\approx$	0	0.1591
F_{14}	0	0.1777 $\approx$	0	0.6003 $-$	0	0.1185 $+$	0	0.1691 $\approx$	0	0.1394 $+$	0	0.1583
F_{15}	0	58.8639 $-$	0	59.3920 $-$	0	56.0740 $-$	0	52.8176 $\approx$	0	53.8910 $-$	0	52.5662
3-parity	100	0 $\approx$	100	0 $\approx$	100	0 $\approx$	100	0 $\approx$	100	0 $\approx$	100	0
4-parity	88	0.03 $-$	1	0.4406 $-$	1	0.4192 $-$	14	0.3002 $-$	100	0 $\approx$	100	0
5-parity	51	0.1289 $-$	0	0.5804 $-$	0	0.5645 $-$	0	0.5257 $-$	100	0 $\approx$	100	0
6-parity	3	0.3201 $-$	0	0.6467 $-$	0	0.6433 $-$	0	0.6234 $-$	100	0 $\approx$	100	0
7-parity	0	0.4272 $-$	0	0.6755 $-$	0	0.6815 $-$	0	0.6726 $-$	100	0 $\approx$	100	0
8-parity	0	0.5143 $-$	0	0.6929 $-$	0	0.6981 $-$	0	0.6947 $-$	100	0 $\approx$	100	0
$-$		19		17		18		15		1		
$\approx$		2		4		2		6		18		
$+$		0		0		1		0		2		

Symbols $-$, $\approx$ and $+$ represent that the competitor is respectively significantly worse than, similar to, and better than SL-GEP according to the Wilcoxon signed-rank test at $\alpha = 0.05$.

TABLE XI
RESULTS OF THE MULTIPLE-PROBLEM WILCOXON'S TEST FOR GP, GEP-ADF, TREEDE, LDEP, SL-GEP/JADE AND SL-GEP.

Algorithms	GP vs SL-GEP	GEP-ADF vs SL-GEP	TreeDE vs SL-GEP	LDEP vs SL-GEP	SL-GEP vs SL-GEP/JADE
p-Value	5.167E-5	4.429E-5	0.0004837	4.429E-5	0.08654

p-Value < 0.05 indicates that the second competitor performed significantly better than the first competitor according to the multiple-problem Wilcoxon signed-rank test at $\alpha = 0.05$, in terms of the $RMSE$.

D. Comparisons with Other GP Variants

This subsection evaluates the results of SL-GEP against the GP, GEP-ADF, TreeDE, LDEP, and SL-GEP/JADE algorithms. Table IX reports the Suc and $RMSE$ of the algorithms considered for comparison. To assess the performances of multiple algorithms on multiple problems, we first applied the Friedman test to detect whether significant differences exist among all the mean $RMSE$ values obtained by the algorithms, as considered in [47]. Table X presents the results of the Friedman tests for $\alpha = 0.05$. The statistical Friedman value is 59.9451, which is greater than the critical value of 11.07. This indicates a significant difference among the observed results obtained by the algorithms with a probability error of $p < 0.05$. Next, we also adopted the Bonferroni-Dunn's test as a post-hoc test to detect the significant differences for the control algorithm SL-GEP. The critical difference value (CD) of the Bonferroni-Dunn's test at $\alpha = 0.05$ is 1.484. Fig. 7 plots the ranking obtained via the Friedman test and the threshold of the critical differences of Bonferroni-Dunn's procedure. The threshold value is equal to the CD value plus the ranking of SL-GEP (i.e., 1.484 + 1.8333 = 3.3173). An algorithm with a ranking larger than the threshold value is considered significantly worse than SL-GEP. It can be observed that SL-GEP performed significantly better than GP, GEP-ADF, TreeDE, and LDEP, in terms of the $RMSE$. Meanwhile, as the ranking of SL-GEP is smaller than the ranking of SL-GEP/JADE plus the CD, the performances of SL-GEP and SL-GEP/JADE are deemed as competitive.

The last three rows of Table IX summarize the results of the Wilcoxon's sign rank test on each problem. It is worth noting that the proposed SL-GEP exhibited significantly better performance than GP, GEP-ADF, TreeDE, and LDEP on most of the problems, while the performances of SL-GEP and SL-GEP/JADE are competitive. Further, a multiple-problem Wilcoxon's test is conducted to check the behaviors of the six algorithms on the whole test suite. The statistical results in Table XI indicate that the SL-GEP exhibited significantly better performance than GP, GEP-ADF, TreeDE, and LDEP, while the performances of SL-GEP and SL-GEP/JADE are not significantly different at $\alpha = 0.05$, in terms of the $RMSE$.

In addition, with respect to the Suc values, SL-GEP outperformed GP, GEP-ADF, TreeDE and LDEP on most of the problems. Particularly, for the six even parity problems, even though the GP, GEP-ADF, TreeDE and LDEP algorithms

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

achieved a 100% success rate on the 3-parity problem, their Suc values is observed to decrease dramatically as the dimensionality of the problem increases. The GEP-ADF, TreeDE, and LDEP are noted to have failed in solving the 5, 6, 7, 8-parity problems. Meanwhile, SL-GEP consistently achieved a 100% success rate on the even parity problems. In summary, SL-GEP exhibits improved performance over GP, GEP-ADF, TreeDE and LDEP in terms of the Suc metric.

E. Analysis of Convergence Speed

This subsection investigates the convergence speeds of the algorithms under consideration. First of all, we analyze the RT values of all algorithms under-studied to evaluate their efficiencies. As listed in Table XII, the proposed SL-GEP is observed to have outperformed all algorithms except SL-GEP/JADE on most of the problems, in terms of the RT values. As GEP-ADF2 did not use the proposed DE-based search mechanism, it was outperformed by SL-GEP on all 18 problems. This indicates the better efficiency of the proposed DE-based operators versus the original genetic-based operators. Meanwhile, the SL-GEP/JADE outperformed SL-GEP on most of the problems in terms of the RT. This demonstrates that the proposed SL-GEP can be integrated with other forms of state-of-the-art DE to further improve the search performance.

Further, we compare the RT results of the proposed SL-GEP in Table XII with the published results of ECGP [48]. In [48], the ECGP was applied to five even parity problems, i.e., 4-parity to 8-parity problems. The RT values of ECGP (SL-GEP) on the five problems are 65296 (36001), 181920 (64224), 287764 (94542), 311940 (158146), and 540224 (254855) respectively. It is clear that the proposed SL-GEP has a much faster convergence speed than ECGP, because the RT values of ECGP are almost two times as large as those of the proposed SL-GEP.

Next, we investigate the impacts of problem dimension on the performances of the algorithms. Fig. 9 shows the RT versus the dimensionality of the even parity problem. The published results of ECGP are also plotted for comparison analysis. It can be observed that the RT of GP, GEP, GEP-ADF, TreeDE, and LDEP increased dramatically as the dimensionality of the even parity problem increases, while our proposed GEP-ADF2, SL-GEP and SL-GEP/JADE exhibited a much lower rate of increase. The proposed algorithms showcased better RT values than the ECGP as the dimensionality of the even parity problem increases. The results thus indicate that the proposed algorithms exhibited higher scalabilities than the other counterpart algorithms.

Fig. 8 depicts the search convergence trends of the best fitness values attained by the algorithms on six representative problems: F_1, F_5, F_{10}, F_{15}, 3-parity problem and 8-parity problem. In the plots, note that a search trace terminates when the global optima is reached. It can be observed that the proposed SL-GEP displayed a much faster convergence rate to the global optima or high quality solutions than the GEP, GEP-ADF, TreeDE, and LDEP. The GP also showcased very fast convergence speed (e.g., F_{10}), but it often quickly gets trapped in local optima.

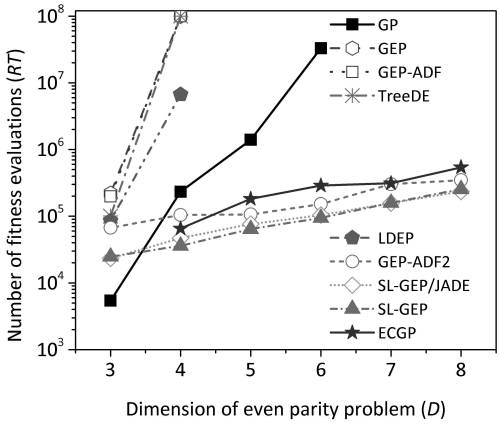

Fig. 9. RT versus the dimension of even parity problem. (For GP, GEP, GEP-ADF, TreeDE, and LDEP, their RT values on large scale problems are missing because they failed to locate a global optimal solution in the 100 runs with E_{max} = 1,000,000; The results of ECGP are cited from [48]).

F. Examples of Converged Solutions

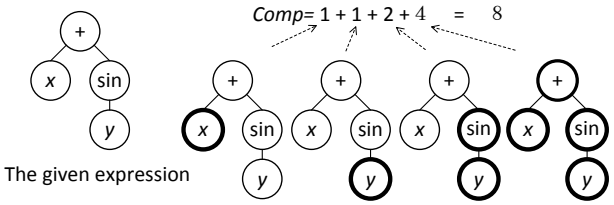

Fig. 10. An example of calculating the structure complexity of an expression.

This section showcases some of the converged solutions found by the proposed SL-GEP algorithm. Our objective is to analyze the conciseness and readability of the converged solutions. Table VI reports the structural complexity of the converged solutions on F_5, F_{10}, the 3-parity problem, and the 8-parity problem. Each of these solutions achieved a perfect hit. If an algorithm failed to locate a solution with a perfect hit across the 100 independent runs, the corresponding value is marked by "N/A" in the table. To quantify the complexities of the converged solutions, the complexity metric proposed in [49] is adopted. This metric determines the structural complexity of an expression based on the total number of nodes in all subtrees. Fig. 10 illustrates an example to determine the complexity of an expression tree. This measure favors trees with fewer layers and nodes, and is shown as effective for defining the structural complexity of an expression [40], [49]. In our experimental studies, the structural complexity of a solution is equal to the sum of the structural complexity of the "main program" and those of the ADFs. Since the solutions found by LDEP comprise of a number of low level instructions, its solutions are omitted from Table XIII.

It can be observed that the solutions provided by the SL-GEP were generally more readable, because of their lower structure complexities in most cases. Specifically, on the first symbolic regression problem, the SL-GEP outperformed the GP, GEP, GEP-ADF and TreeDE. As for the second symbolic problem, the GP, GEP, and GEP-ADF performed slightly better

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

TABLE XII
THE RT RESULTS OF GP, GEP, GEP-ADF, TREEDE, LDEP, GEP-ADF2, SL-GEP/JADE AND SL-GEP.

	GP	GEP	GEP-ADF	TreeDE	LDEP	GEP-ADF2	SL-GEP/JADE	SL-GEP
F_1	1,159,360	204,111	587,302	830,440	442,793	288,519	**13,906**	14,945
F_2	5,670,058	11,882,400	99,000,000	19,328,800	2,080,800	3,372,760	**384,067**	678,365
F_3	24,002,680	8,405,980	**20,408**	49,385,200	754,178	1,428,040	84,233	101,686
F_4	58,526	35,148	30,928	74,187	152,667	71,647	**4,952**	5,984
F_5	2,238,734	1,312,930	24,000,000	6,342,840	4,730,990	546,913	**21,100**	24,247
F_6	11,512,150	13,677,300	N/A	N/A	5,605,110	1,163,360	**220,633**	265,811
F_7	6,145,858	836,884	8,090,910	3,381,980	3,957,770	3,051,210	622,914	**554,090**
F_8	441,528	67,800	388,889	569,604	414,804	67,338	**11,919**	13,940
F_9	11,508,536	16,061,100	7,333,330	7,691,560	3,600,750	24,220,500	3,643,110	**3,045,980**
F_{10}	4,020,667	1,666,250	298,701	7,017,500	479,232	4,734,480	**173,634**	209,584
F_{11}	1,236,924	55,158	190,476	339,392	236,894	69,868	**16,579**	19,561
F_{12}	754,502	824,071	**10,101**	833,282	90,083	448,095	57,667	72,845
3-parity	**5,418**	219,858	197,969	100,643	85,524	67,153	23,290	24,776
4-parity	232,556	99,722,400	99,351,700	99,925,500	6,698,860	104,453	46,997	**36,001**
5-parity	1,396,190	N/A	N/A	N/A	N/A	106,193	76,265	**64,224**
6-parity	32,997,902	N/A	N/A	N/A	N/A	151,861	103,974	**94,542**
7-parity	N/A	N/A	N/A	N/A	N/A	302,288	**157,125**	158,146
8-parity	N/A	N/A	N/A	N/A	N/A	347,335	**234,982**	254,855
$worse$	17	18	16	18	18	18	5	
$better$	1	0	2	0	0	0	13	

$better$ and $worse$ respectively represent the number of problems where the competitor is better and worse than the proposed SL-GEP, in terms of RT. The results of $F_{13} - F_{15}$ are omitted because all algorithms failed to find a perfect hit on these three problems.

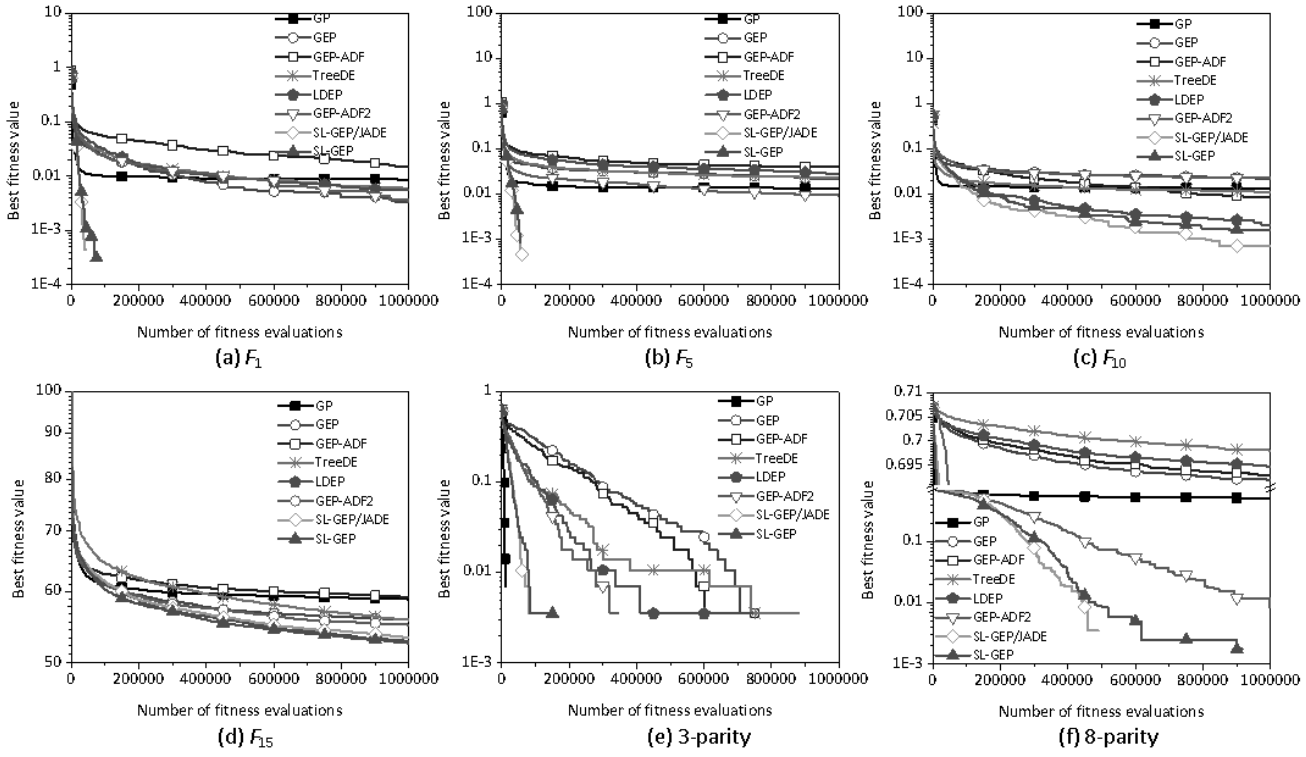

Fig. 8. Evolution of the best fitness values derived from all compared algorithms on F_1, F_5, F_{10}, F_{15}, 3-parity problem and 8-parity problem.

than the SL-GEP, because the expression of this problem can be expressed concisely without using ADFs. On the 3-parity problem, SL-GEP significantly outperformed GP, GEP, GEP-ADF and TreeDE, because the structural complexity of its solution was much smaller than those of the other four algorithms. On the 8-parity problem which is much more complex than the 3-parity problem, the GP, GEP, GEP-ADF and TreeDE all failed to find a global optimal solution in their search. Nevertheless, the SL-DEP was capable of converging to a global optimal solution on the 8-parity problem. Besides, the solution provided by SL-GEP on the 8-parity problem had a relatively small structural complexity, which is similar to that of the solution found by GEP-ADF on the 3-parity problem.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

TABLE XIII
EXAMPLES OF THE BEST SOLUTIONS FOUND BY GP, GEP, GEP-ADF, TREEDE AND SL-GEP.

Problem	Algorithm	Example of the best solutions	*Comp*
F_5	GP	$S=(((x*x)+x)*(x+(x*(x*x))))+x$	59
	GEP	$S=x+((x+((x*x)*x))$ $*((x*x)+x))$	59
	GEP-ADF	$S=(((G_1+((G_1*G_1)*G_1))*((G_1*$ $G_1)+G_1))+G_1)$ $G_1=((x*x)/x)$	70
	TreeDE	$S=((x+(x*(x*x)))*(x*((x/x)+$ $x)))+x$	71
	SL-GEP	$S=G_1(G_1(x*x,G_1(x,x)),x)$ $G_1(t_1,t_2)=t_2+(t_2*t_1)$	**38**
F_{10}	GP	$S=exp((x*log(x))/(x+x))$	30
	GEP	$S=exp(((log(x)*x)/(x+x)))$	30
	GEP-ADF	$S=exp(G_1)$ $G_1=(log(x)*x)/(x+x)$	**24**
	TreeDE	$S=x/exp((log(x)*(x/(x+x))))$	43
	SL-GEP	$S=exp((log(x)/((x+x)/x)))$	31
3-parity	GP	$S=((((x_3*x_2)[!*]x_2)+((x_1[!*]x_2)*$ $(x_3+x_1)))*(((x_2+x_3)+(x_1+$ $x_3))[!*]((x_3+x_2)[!*]((x_1+x_2)*(x_1+$ $x_3)))))$	143
	GEP	$S=(((x_1*(x_3*x_2))[!+](x_1+x_2)[!*]$ $((x_2*x_1)+x_3)))+(x_3[!+](x_1+x_2)))$	83
	GEP-ADF	$S=(G_1[!+]G_1)$ $G_1=((((x_3+(x_2+x_2))[!*]((x_2+x_2)$ $[!*]x_3))*x_1)+((x_2+x_3)*(x_1[!+]$ $(x_3*x_2))))$	100
	TreeDE	$S=(((x_3+(x_1[!+](x_2[!*]x_2)))+((x_3$ $[!+](x_1+x_3))[!+]x_2))[!*]((x_1[!*]x_2)[!*]$ $((x_1+x_2)*x_3)))$	103
	SL-GEP	$S=G_1((x_2[!*]x_2),G_1(x_3,x_1))$ $G_1(t_1,t_2)=(t_2+t_1)[!*](t_1[!*]t_2)$	**34**
8-parity	GP	N/A	N/A
	GEP	N/A	N/A
	GEP-ADF	N/A	N/A
	TreeDE	N/A	N/A
	SL-GEP	$S=G_1(G_1(x_7,G_1(G_2(x_5,x_2),x_4)),$ $G_1(G_1(G_1(x_8,x_3),(x_1[!+](x_1*x_1))),$ $G_1(x_2,x_6)))$ $G_1(t_1,t_2)=(t_2[!+]t_1)+(t_1*t_2)$ $G_2(t_1,t_2)=t_1[!*]t_1$	**105**

G. Analysis of Algorithm Parameters

The SL-GEP involves four important parameters which are NP, h, h', and K. In this subsection, we study their impacts by studying SL-GEP with different parameter settings on three representative symbolic regression problems, i.e., F_1, F_6 and F_{11}. F_1 and F_6 respectively represent a kind of simple and a kind of complex symbolic regression problems whose solutions can be compressed effectively with ADFs, while F_{11} represents another kind of symbolic regression problems whose solutions can be expressed concisely without using ADFs.

1) Impact of NP: First, we study the impact of NP. We varied the value of NP to be 20, 50, 100, 400, and 1000 respectively, while the other parameters remained the same as in Table V. For each parameter setting, 100 independent runs were performed. The results presented in Table XIV show that NP should not be set too small or too large. If NP is set too small, the algorithm may easily get trapped in local optima due to the loss of population diversity. For example, the Suc of SL-GEP on F_6 decreased when NP decreased to 20. Meanwhile, a large NP will slowdown the search speed. For the three test instances, SL-GEP generally required larger computational cost as NP increased from 50 to 1000. It appears that $NP = 50$ and $NP = 100$ are promising settings.

TABLE XIV
IMPACT OF NP.

NP	F_1		F_6		F_{11}	
	Suc	RT	Suc	RT	Suc	RT
20	100	15,755	71	606,060	100	20,382
50	100	14,945	91	265,811	100	19,561
100	100	22,741	97	156,532	100	28,467
400	100	41,357	100	181,992	100	65,672
1,000	100	66,565	100	273,782	100	99,520

TABLE XV
IMPACT OF h.

h	F_1		F_6		F_{11}	
	Suc	RT	Suc	RT	Suc	RT
3	100	26656	0	N/A	98	68,556
5	100	26,638	29	2,642,840	100	11,897
10	100	14,945	91	265,811	100	19,561
20	100	22,554	97	181,561	100	33,475
40	100	30,534	95	247,879	100	40,734

2) Impact of h: Generally speaking, a larger h enables the main program to represent more complex tree structures, but this will also result in a larger search space. To assess the impact of h, we varied the value of h to be 3, 5, 10, 20, and 40 respectively, while other parameters were set the same as in Table V. The results presented in Table XV show that h should not be set too small. If h is too small, there will be no solution in the search space. For example, with $h = 3$, the SL-GEP failed to find the optimal solution to F_6 on all 100 runs. On the other hand, the search space will enlarge as h increases. Hence, SL-GEP generally requires a larger computational cost as h increases. The results in Table XV demonstrate that the performance of SL-GEP gradually degrades when h increases from 20 to 40.

3) Impact of h': h' determines the $Head$ length of ADFs. Generally, an ADF with a larger h' can accomplish more complex subtasks, but this will also lead to a larger search space. Here we set h' to be 1, 2, 3, 5, and 10 respectively to investigate its impact. The results in Table XVI show that the performance of the algorithm degraded significantly on F_1 and F_6 as h' decreased to 1. However, as for F_{11}, the value of h' seems to have a small impact on the performance of the algorithm. This is because the expressions of F_1 and F_6 can be effectively compressed by using ADFs, while that of F_{11} is already concise enough without using ADFs. If $h' = 1$, the ADFs in each chromosome are too simple to improve the search efficiently. Meanwhile, as h' increases, the ADFs in each chromosome become less general (i.e., less likely to be a frequent substructure). Thus, the performance of the algorithm degraded on these two problems when h' became too large. In general, $h' = 2$ and $h' = 3$ lead to better performance.

4) Impact of K: Parameter K is the number of ADFs in each chromosome. To assess its impact, we varied its value to be 1, 2, 5, 10 and 20 respectively. The results in Table XVII

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2015.2424410, IEEE Transactions on Evolutionary Computation

TABLE XVI
IMPACT OF h'.

h'	F_1		F_6		F_{11}	
	Suc	RT	Suc	RT	Suc	RT
1	100	22376	28	2943900	100	17709
2	100	15,243	92	267,495	100	21,583
3	100	14,945	91	265,811	100	19,561
5	100	23,067	71	671,602	100	19,806
10	100	21,727	64	810,332	100	21,546

TABLE XVII
IMPACT OF K.

K	F_1		F_6		F_{11}	
	Suc	RT	Suc	RT	Suc	RT
1	100	18,372	90	289,774	100	19,162
2	100	14,945	91	265,811	100	19,561
5	100	18,318	82	417,230	100	21,297
10	100	19,513	82	474,927	100	22,987
20	100	20,360	79	575,705	100	24,969

show that K had a small impact on the performance of the algorithm for solving F_{11}, because the expressions of solutions to F_{11} can be concisely represented without using ADFs. However, the performance of the algorithm on F_1 and F_6 degraded significantly when K became too small (e.g., $K = 1$) or too large (e.g., larger than 5). It appears that the value of K should be set between 2 and 5.

V. CONCLUSION

In this paper, we have proposed a self-learning GEP (SL-GEP) methodology for automatic generation of computer programs. The proposed SL-GEP features a novel chromosome representation that facilitates the formation of C-ADFs that encompass ADFs and/or any subtree of the main program as input arguments. This feature makes the algorithm more flexible to arrive at sophisticated and constructive C-ADFs that improve the accuracy and efficiency of the search. Further, a novel DE-based search mechanism is proposed to efficiently evolve the chromosomes in the SL-GEP. Experiments on 15 symbolic regression problems and six even-parity problems show that the proposed SL-GEP generally performs better than several state-of-the-art GP variants, in terms of accuracy and search efficiency. Besides, the SL-GEP is able to provide more readable solutions that have smaller structural complexities.

There are several interesting future research directions. One direction is to extend the proposed SL-GEP by considering the structural complexity as another objective during the evolution process. By incorporating multi-objective optimization technique, the extended algorithm can provide multiple alternative solutions which have trade-offs among the accuracy and the readability. The second direction is to further enhance SL-GEP by adaptively controlling the parameters of SL-GEP, such as the length of chromosome and the number of ADFs in each chromosome. Another promising research topic is to apply the proposed SL-GEP to complex practical applications.

ACKNOWLEDGMENT

The research reported in this paper is financially supported by the Tier 1 Academic Research Fund (AcRF) under project Number RG23/14.

REFERENCES

[1] J. R. Koza, *Genetic Programming: vol. 1, On the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.
[2] P. G. Espejo, S. Ventura, and F. Herrera, "A survey on the application of genetic programming to classification," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 40, no. 2, pp. 121–144, 2010.
[3] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Trans. Evol. Comput.*, vol. 5, no. 4, pp. 349–358, 2001.
[4] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Genetic Programming*. Springer, 2000, pp. 121–132.
[5] C. Ferreira, "Gene expression programming: a new adaptive algorithm for solving problems," *arXiv preprint cs/0102027*, 2001.
[6] M. F. Brameier and W. Banzhaf, *Linear genetic programming*. Springer, 2007.
[7] C. Zhou, W. Xiao, T. M. Tirpak, and P. C. Nelson, "Evolving accurate and compact classification rules with gene expression programming," *IEEE Trans. Evol. Comput.*, vol. 7, no. 6, pp. 519–531, 2003.
[8] C. Ferreira, *Gene expression programming*. Springer Berlin, 2006.
[9] N. Sabar, M. Ayob, G. Kendall, and R. Qu, "The automatic design of hyper-heuristic framework with gene expression programming for combinatorial optimization problems," *IEEE Trans. Evol. Comput.(in pressing)*, vol. PP, no. 99, pp. 1–1, 2014.
[10] ——, "A dynamic multiarmed bandit-gene expression programming hyper-heuristic for combinatorial optimization problems," *IEEE Transactions on Cybernetics (in pressing)*, vol. PP, no. 99, pp. 1–1, 2014.
[11] J. Zhong, L. Luo, W. Cai, and M. Lees, "Automatic rule identification for agent-based crowd models through gene expression programming," in *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2014, pp. 1125–1132.
[12] R. Meuth, M.-H. Lim, Y.-S. Ong, and D. C. Wunsch II, "A proposition on memes and meta-memes in computing for higher-order learning," *Memetic Computing*, vol. 1, no. 2, pp. 85–100, 2009.
[13] X. Chen, Y.-S. Ong, M.-H. Lim, and K. C. Tan, "A multi-facet survey on memetic computation," *IEEE Trans. Evol. Comput.*, vol. 15, no. 5, pp. 591–607, 2011.
[14] Y.-S. Ong, M. Lim, and X. Chen, "Memetic computationpast, present & future [research frontier]," *IEEE Comput. Intell. M.*, vol. 5, no. 2, pp. 24–31, 2010.
[15] J. R. Koza, *Genetic programming II: automatic discovery of reusable programs*. MIT press, 1994.
[16] Y. Kameya, J. Kumagai, and Y. Kurata, "Accelerating genetic programming by frequent subtree mining," in *Proc. Genetic Evol. Comput. Conf.* ACM, 2008, pp. 1203–1210.
[17] M. Iqbal, W. Browne, and M. Zhang, "Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems," *IEEE Trans. Evol. Comput.*, vol. 18, no. 4, pp. 465–480, Aug 2014.
[18] L. Feng, Y.-S. Ong, I. Tsang, and A.-H. Tan, "An evolutionary search paradigm that learns with past experiences," in *2012 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2012, pp. 1–8.
[19] P. J. Angeline and J. Pollack, "Evolutionary module acquisition," in *Proceedings of the second annual conference on evolutionary programming*. Citeseer, 1993, pp. 154–163.
[20] K. E. Kinnear Jr, "Alternatives in automatic function definition: A comparison of performance," *Advances in Genetic Programming*, pp. 119–141, 1994.
[21] T. Van Belle and D. H. Ackley, "Code factoring and the evolution of evolvability." in *Proc. Genetic Evol. Comput. Conf.*, vol. 2, 2002, pp. 1383–1390.
[22] J. A. Walker and J. F. Miller, "The automatic acquisition, evolution and reuse of modules in cartesian genetic programming," *IEEE Trans. Evol. Comput.*, vol. 12, no. 4, pp. 397–417, 2008.
[23] C. Ferreira, "Automatically defined functions in gene expression programming," in *Genetic Systems Programming*. Springer, 2006, pp. 21–56.
[24] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[25] M. Schmidt and H. Lipson, "Distilling free-form natural laws from experimental data," *science*, vol. 324, no. 5923, pp. 81–85, 2009.

[26] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong *et al.*, "Genetic programming needs better benchmarks," in *Proc. Genetic Evol. Comput. Conf.* ACM, 2012, pp. 791–798.

[27] J. Y.-C. Liu, J.-H. A. Chen, C.-T. Chiu, and J.-C. Hsieh, "An extension of gene expression programming with hybrid selection," in *Proceedings of the 2nd International Conference on Intelligent Technologies and Engineering Systems (ICITES2013)*. Springer, 2014, pp. 635–641.

[28] Y. Zhang and J. Xiao, "A new strategy for gene expression programming and its applications in function mining," *Universal Journal of Computer Science and Engineering Technology, 1 (2), 122-126.*, 2010.

[29] Y. Peng, C. Yuan, X. Qin, J. Huang, and Y. Shi, "An improved gene expression programming approach for symbolic regression problems," *Neurocomputing*, vol. 137, pp. 293–301, 2014.

[30] E. Bautu, A. Bautu, and H. Luchian, "Adagep-an adaptive gene expression programming algorithm," in *Symbolic and Numeric Algorithms for Scientific Computing, 2007. SYNASC. International Symposium on*. IEEE, 2007, pp. 403–406.

[31] J. Zhang and A. C. Sanderson, "Jade: adaptive differential evolution with optional external archive," *IEEE Trans. Evol. Comput.*, vol. 13, no. 5, pp. 945–958, 2009.

[32] Y. Wang, Z. Cai, and Q. Zhang, "Differential evolution with composite trial vector generation strategies and control parameters," *IEEE Trans. Evol. Comput.*, vol. 15, no. 1, pp. 55–66, 2011.

[33] A. K. Qin and P. N. Suganthan, "Self-adaptive differential evolution algorithm for numerical optimization," in *The 2005 IEEE Congress on Evolutionary Computation*, vol. 2. IEEE, 2005, pp. 1785–1791.

[34] L. Tang, Y. Dong, and J. Liu, "Differential evolution with an individual-dependent mechanism," *IEEE Trans. Evol. Comput. (in pressing)*.

[35] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, 2012.

[36] C. Ferreira, "Function finding and the creation of numerical constants in gene expression programming," in *Advances in Soft Computing*. Springer, 2003, pp. 257–265.

[37] M. Zhang and W. Smart, "Genetic programming with gradient descent search for multiclass object classification," in *Genetic Programming*. Springer, 2004, pp. 399–408.

[38] M. Kommenda, G. Kronberger, S. Winkler, M. Affenzeller, and S. Wagner, "Effects of constant optimization by nonlinear least squares minimization in symbolic regression," in *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*. ACM, 2013, pp. 1121–1128.

[39] S. Mukherjee and M. J. Eppstein, "Differential evolution of constants in genetic programming improves efficacy and bloat," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*. ACM, 2012, pp. 625–626.

[40] E. J. Vladislavleva, G. F. Smits, and D. Den Hertog, "Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming," *IEEE Trans. Evol. Comput.*, vol. 13, no. 2, pp. 333–349, 2009.

[41] S. Luke, "The ecj owners manual," *San Francisco, California, A user manual for the ECJ Evolutionary Computation Library*, 2010.

[42] C. Fonlupt, D. Robilliard, and V. Marion-Poty, "Linear imperative programming with differential evolution," in *IEEE Symposium on Differential Evolution*. IEEE, 2011, pp. 1–8.

[43] C. B. Veenhuis, "Tree based differential evolution," in *Genetic Programming*. Springer, 2009, pp. 208–219.

[44] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "A computational study of representations in genetic programming to evolve dispatching rules for the job shop scheduling problem," *IEEE Trans. Evol. Comput.*, vol. 17, no. 5, pp. 621–639, 2013.

[45] D. F. Barrero, "Relibility of performance measures in tree-based genetic programming: a study on koza's computational effort," Ph.D. dissertation, School of Computing of the University of Alcala, 2011.

[46] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Pošík, "Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009," in *Proc. Genetic Evol. Comput. Conf.* ACM, 2010, pp. 1689–1696.

[47] S. García, A. Fernández, J. Luengo, and F. Herrera, "A study of statistical techniques and performance measures for genetics-based machine learning: accuracy and interpretability," *Soft Computing*, vol. 13, no. 10, pp. 959–977, 2009.

[48] J. A. Walker and J. F. Miller, "Evolution and acquisition of modules in cartesian genetic programming," in *Genetic Programming*. Springer, 2004, pp. 187–197.

[49] G. F. Smits and M. Kotanchek, "Pareto-front exploitation in symbolic regression," in *Genetic programming theory and practice II*. Springer, 2005, pp. 283–299.

Jinghui Zhong received his B.Eng. degree, M.Eng degree, and Ph.D. degree from the School of Information Science and Technology, Sun YAT-SEN University, China, in 2005, 2007, and 2012 respectively. Currently, he is a Research Fellow in the School of Computer Engineering, Nanyang Technological University (NTU), Singapore. His current research interests include genetic programming, differential evolution, ant colony optimization, and the applications of evolutionary computations.

Yew-Soon Ong received the B.S. and M.Eng. degrees in electrical and electronics engineering from Nanyang Technological University (NTU), Singapore, in 1998 and 1999, respectively, and the Ph.D. degree on artificial intelligence in complex design from the Computational Engineering and Design Center, University of Southampton, Southampton, United Kingdom in 2003. He is currently an Associate Professor and the Director of the A*Star SIMTECH-NTU Joint Lab on Complex Systems and Programme at the School of Computer Engineering, NTU. He is also a Principal Investigator of the Rolls-Royce@NTU Corporate Lab on Large Scale Data Analytics. His current research interest in computational intelligence spans across memetic computation, evolutionary design, machine learning and Big data. Dr. Ong is the Founding Technical Editor-in-Chief of the Memetic Computing Journal, Founding Chief Editor of the Springer book series on Studies in Adaptation, Learning, and Optimization, and an Associate Editor of the IEEE Transactions on Evolutionary Computation, IEEE Transactions on Neural Network and Learning Systems, IEEE Computational Intelligence Magazine, IEEE Transactions on Cybernetics, IEEE Transactions on Big data and others. His research work on Memetic Algorithm was featured in the Emerging Research Fronts of the Essential Science Indicators in August 2007. He also received the 2015 IEEE Computational Intelligence Magazine Outstanding Paper Award and the 2012 IEEE Transactions on Evolutionary Computation Outstanding Paper Award for his published works on Memetic Computation.

Wentong Cai is a Professor in the School of Computer Engineering at Nanyang Technological University. He is also the Director of the Parallel and Distributed Computing Centre. He received his Ph.D. in Computer Science from University of Exeter (UK) in 1991. His expertise is mainly in the areas of Modeling and Simulation (particularly, modeling and simulation of large-scale complex systems, and system support for distributed simulation and virtual environments) and Parallel and Distributed Computing (particularly, Cloud, Grid and Cluster computing). He is an associate editor of the ACM Transactions on Modeling and Computer Simulation (TOMACS) and an editor of the Future Generation Computer Systems (FGCS).