Search-Based Algorithm With Scatter Search Strategy for Automated Test Case Generation of NLP Toolkit

Fangqing Liu, Han Huang, *Member, IEEE*, Zhongming Yang, Zhifeng Hao, and Jiangping Wang

***Abstract*—Natural language processing (NLP), as a theory-motivated computational technique, has extensive applications. Automated test case generation based on path coverage, which is a popular structural testing activity, can automatically reveal logic defects that exist in NLP programs and can save testing consumption. NLP programs have many paths that can only be covered by specific input variables. This feature makes conventional search-based algorithm very difficult covering all possible paths in NLP programs. A strategy is required for improving the search ability of search-based algorithms. In this paper, we propose a scatter search strategy to automatically generate test cases for covering all possible paths of NLP programs. The scatter search strategy empowers search-based algorithms to explore all input variables and cover the paths that require specific input variables within a small amount of test cases. The experiment results show that the proposed scatter search strategy can quickly cover the paths, which requires specific input variables. Many test cases and running time consumptions will be saved when search-based algorithms combine with scatter search strategy.**

***Index Terms*—Automated test case generation, path coverage, natural language processing, search-based algorithm, scatter search.**

I. Introduction

NATURAL language processing (NLP) is a theory-motivated computational technique which reflects the understanding of human language representation and analysis [1]. NLP researchers focus on decoding human languages and designing intelligent systems to understand input speech or text content [2]. As a popular research topic, NLP has several applications such as automated recommending web [3], video dynamics detection [4], social emotion classification [5] and travel chat robot [6].

The Stanford CoreNLP toolkit [7] provides a framework for analyzing natural language and has a wide scope of applications. The CoreNLP toolkit was used for preprocessing and tokenization for a Microsoft COCO Cation data set and evaluation server [8]. It was also applied to an open-domain question answering system [9]. Many NLP techniques and toolkits are designed based on the CoreNLP, as some logical defects exist in those programs. Automated test case generation (ATCG) [10] is one category of structural testing activity. Compared to functional testing [11] which focuses on satisfying requirements of programs, structural testing reveals the logical defects on tested programs. ATCG usually selects a coverage criterion and aims to cover all possible coverage targets in the program. Statement coverage [12], branch coverage [10], mutation coverage [13] and path coverage [14] are the common criteria in structural testing. Among those coverage criteria, path coverage is the most critical and challenging one [15], [16]. Most defects in NLP programs can be easily found if the test cases which satisfy the path coverage criterion can be automatically generated. In this paper, we have selected automated test case generation based on path coverage (ATCG-PC) for NLP programs as our research topic.

Two issues can be found with ATCG-PC. First, ATCG-PC is one of the unit testing activities [17] which needs to generate test cases for covering paths in tested functions. Given a tested NLP program, the number of paths is finite and discrete. Therefore, this feature makes it difficult for ATCG-PC to be derivable. Second, the relationship between test cases and paths is unknown. The generated test cases may cover the path which is already found. This feature makes conventional optimization methods generate too many redundant test cases.

The objective of ATCG-PC is to find a set of test cases which cover all possible paths in the NLP program. Some researchers consider ATCG-PC as a single-objective optimization problem [14], [16] while others consider ATCG-PC as a multi-objective optimization problem [10], [18]–[20]. Fraser [16] proposed a model to maximize fitness value of its generated test case set, wherein only the set of test cases that cover all possible paths have maximum fitness value. Huang [14] proposed a model to minimize the number of generated test cases for covering all

Manuscript received October 3, 2018; revised February 8, 2019; accepted April 14, 2019. This work was supported in part by the National Natural Science Foundation of China under Grant 61876207, in part by Guangdong Natural Science Funds for Distinguished Young Scholar under Grant 2014A030306050, in part by Guangdong High-level personnel of special support Program under Grant 2014TQ01X664, in part by the International Cooperation Project of Guangzhou under Grant 201807010047, and in part by Guangzhou Science and Technology Project under Grants 201802010007, 201804010276, and in part by the Guangdong Province Key Area R&D Program under Grant 2018B010109003. *(Corresponding author: Han Huang.)*

F. Liu and H. Huang are with the School of Software Engineering, South China University of Technology, Guangzhou 510006, China, and also with the Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun 130012, China (e-mail: 564376030@qq.com; hhan@scut.edu.cn).

Z. Yang is with the College of Computer Engineering Technical, Guangdong Institute of Science and Technology, Zhuhai 510640, China (e-mail: yzm8008@126.com).

Z. Hao is with the School of Computer, Guangdong University of Technology, Guangzhou 510090, China, and also with the School of Mathematics and Big Date, Foshan University, Foshan 528000, China (e-mail: zfhao@fosu.edu.cn).

J. Wang is with the BeMing Software Company, Ltd., Guangzhou 510663, China (e-mail: wangjingping@bmsoft.com.cn).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors. The material consists of a PDF, viewable with Windows Adobe Acrobat. The size of the pdf is 24 kB.

Digital Object Identifier 10.1109/TETCI.2019.2914280

paths in the tested program. The above references [14], [16] consider ATCG-PC as a single-objective optimization problem and both used single-objective methods to optimize it. Conversely, other researchers have built ATCG-PC as a multi-objective optimization problem [10], [18]–[20]. Each path in the tested program has a separated single fitness function, and only the test case covering the corresponding path has the highest fitness value. With multi-objective optimization, every test case needs to be evaluated several times; thus, making it expensive to evaluate test cases in multi-objective models. Therefore, considering ATCG-PC as a single-objective optimization problem is a better choice.

Static analysis and search-based algorithm are two common methods for solving ATCG-PC. The static method analyzes the inequality constraints of each path in the system [21], meaning the target test case set is calculated based on the constraints of input variables [22]. However, the computational complexity of static analysis increases exponentially if the number of constraints increases [23]. Correspondingly, search-based algorithm [24] can avoid the limitations of the static analysis method in ATCG-PC.

Among all search-based algorithms, three categories are proposed for ATCG-PC. For example, Bouchachia [25] provided an immune strategy to empower the global search ability of genetic algorithm, while Wang [26] and Suresh [27] both used genetic algorithm to generate offspring for test cases. Zhang [28] introduced a multi-population genetic algorithm for ATCG-PC. Multi-population genetic algorithm [28] increases the diversity of the whole population to solve the ATCG-PC. In short, above researchers [25]–[28] used genetic algorithm or its variants to solve ATCG-PC problem. The second category of search-based algorithms consists of swarm intelligence algorithms. Mala [29] proposed an artificial bee colony algorithm to generate a test suite for path coverage. In addition, Girgis [30] and Nayak [31] both used a particle swarm optimization algorithm to generate test cases for path coverage. The third category of search-based algorithm includes evolutionary algorithm. Huang [14] proposed a self-adaptive differential evolution to generate test cases for covering all paths in tested programs. However, NLP programs usually use character strings as their input variables, whereas some paths can only be covered with specific strings, making the consumption very expensive for finding specific character strings. Therefore, it is difficult for genetic algorithms [25]–[28], swarm intelligence algorithms [29]–[31] and evolutionary algorithms [14] to generate test cases for all possible paths of NLP programs.

We summarize our main contributions as follows:

1) A scatter search strategy is presented to empower the search ability of search-based algorithms such as differential evolution. The scatter search strategy aims to explore all variables and cover the objective path. Many test case and time consumptions can be saved when search-based algorithms are combined with the scatter search strategy.
2) We have comprehensively evaluated the proposed scatter search strategy with differential evolution (DE-SS) and other state-of-the-art search-based algorithms. A mathematical proof proves that the proposed scatter search strategy can work the best when the parameter $s = 2$. The experiment results also show that DE-SS is an efficient algorithm for automatically generating path coverage test cases of NLP programs.

The structure of this paper is as follows: Section II will introduce the ATCG-PC for NLP programs and the mathematical model of ATCG-PC for NLP programs. Section III will introduce the proposed scatter search strategy. The experimental results of the proposed DE-SS and other state-of-the-art algorithms will be presented in Section IV. Finally, the conclusion of this paper will be summarized in Section V.

II. Background

In this section, we will provide necessary background information to present our motivations. Some concepts regarding the model of ATCG-PC will also be introduced.

A. ATCG-PC for NLP Programs

Some basic definitions about test case, path and path coverage will be introduced in this subsection, as the proposed algorithm and the mathematical model are designed based on those three concepts.

Definition 1 (test case): A test case is usually an integer vector $x_i = (x_{i1}, x_{i2}, ..., x_{in})$ with n elements, where each dimension is an integer number in its domain.

Definition 2 (path): A path is a sequence of running direction of vertexes, starting at some initial vertexes and ending at some final vertexes, where each vertex is the branch condition of the tested NLP program.

Definition 3 (path coverage): A test case x_i covers path p_j, when x_i generates the same branch conditions with path p_j in each vertex of the tested NLP program.

Path coverage is more challenging and efficient than other coverage criteria such as statement coverage and branch coverage. If the found test case set satisfies the criterion of path coverage, the test case set will have a proper subset which satisfies the criterion of statement coverage and branch coverage. We assume that more logical software defects can be revealed by path coverage test cases, compared to statement coverage or branch coverage test cases.

NLP programs usually consist of test cases with character strings and integer variables instead of pure integer variables. This feature poses a problem in ATCG-PC for NLP software. Each character string is composed of a string of integer variables, which correspond to their ASCII codes. The size of the solution space will explode when the number of character strings increases. In addition, some paths require specific character strings, meaning that the input variables, which consist of the character strings, should be their exact values. As shown in Fig. 1, the path "Yes, Yes" can only be covered if the input string equals to "Anewstring". It is very difficult for conventional search-based algorithms, such as differential evolution, to cover this path. In our previous work [14], we proposed a self-adaptive differential evolution for ATCG-PC in some simple programs. These programs only used integer

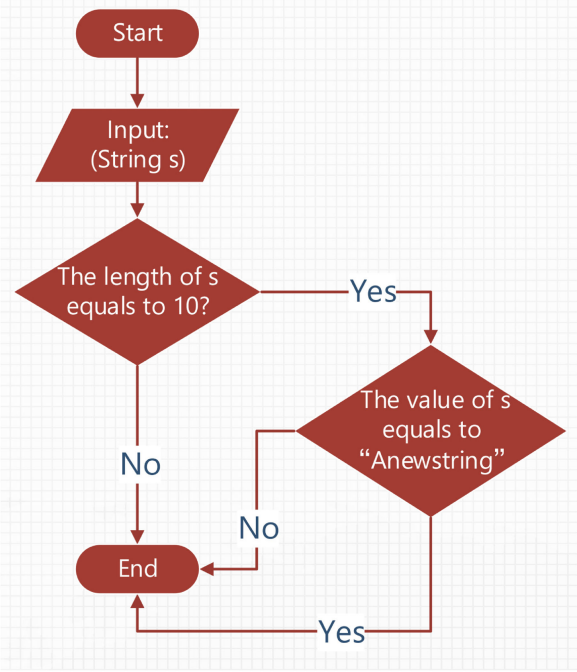


Fig. 1. An example of NLP programs.

Name	Description
$X = \{x_1, x_2, \ldots, x_N\}$	The candidate test case set.
N	The number of candidate test cases in solution space.
$x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,n})$	A test case which is consisted by an integer vector.
n	The number of dimensions in each test case.
$P = \{p_1, p_2, \ldots, p_L\}$	The set of paths in the SUT.
L	The number of paths in the SUT.
m	The number of test cases generated by optimization algorithm.
M	The acceptable maximum number of test cases.
p_j	The jth path in the SUT.
n	The dimension size in each test case.
μ	The number of variables to present character strings.

Fig. 2. Notation of variables in ATCG-PC model of NLP programs.

variables, while their paths were covered easily. In this paper, a new strategy will be proposed to empower differential evolution. The proposed algorithm can cover paths which require specific input variables within a small number of test cases.

B. Mathematical Model of ATCG-PC for NLP Programs

The mathematical model of ATCG-PC for NLP programs is a variant of model [14]. NLP programs contain many variables which are character strings. A character string cannot be represented by single real or integer number. However, each character has an exclusive integer value with an ASCII code. Common characters are usually in the range of [0, 255]. Therefore, a character string can be formulated by an integer vector, while the value of each dimension is in the range of [0, 255].

Some basic variables in this model are defined in Fig. 2. Test case consumption m is usually been used to evaluate the performance of the compared algorithm [14], [32]. The objective of ATCG-PC can be reformulated as follows.

$$\text{Minimize } m$$

s.t.

$$\sum_{j=1}^{L} \min\left\{1, \sum_{i=1}^{m} \theta_{ij}\right\} = L \tag{1}$$

$$\sum_{j=1}^{L} \theta_{ij} = 1 \tag{2}$$

$$\theta_{ij} = \begin{cases} 1, & x_{n_i} \text{ covers } p_j \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$m \leq M \tag{4}$$

$$x_{n_i} \in S_\theta; S_\theta \in X; i = 1, 2, ..., m; m = |S_\theta|;$$
$$j = 1, 2, ..., L; 1 \leq n_1 < n_2 < ... < n_m \leq N; \tag{5}$$
$$x_i = (x_{i,1}, ..., x_{i,\mu}, x_{i,\mu+1}, ..., x_{i,n})$$
$$x_{i,k} \in Z, 0 \leq x_{i,k} \leq 255, 1 \leq k \leq \mu$$

$$x_{i,k'} \in Z, lb_{k'} \leq x_{i,k'} \leq ub_{k'}, \mu + 1 \leq k^{'} \leq n \tag{6}$$

ATCG-PC aims to use minimal test cases to cover all paths in the system under test (SUT). Constraint (1) indicates that the generated m test cases cover all paths in the SUT. Constraints (2) and (3) require that each test case covers exactly one path. Besides, x_{n_i} is the n_ith solution among all N possible solutions. Constraint (4) shows that m should not exceed the acceptable maximum number of test cases M. In this model, Constraint (5) presents the relationship between each generated test case x_{n_i} and candidate test case set X. All generated test cases belong to X, while the ith generated test cases is the n_ith element in X. Constraint (6) defines the range of variables in test cases. The first μ variables of x_i present the character strings in the SUT. Most common characters used in NLP programs can be presented by ASCII codes in range of [0, 255]. The last $(n - \mu)$ elements of x_i represent other integer variables in the SUT. Their upper bound and lower bound is set according to their references. Some paths in NLP programs can only be covered when the first μ variables equal a specific value.

Two issues are presented in this mathematical model. First, Constraint (1) shows that different test cases may cover the same path. However, if the generated test cases cover the paths which have been covered before, many test cases will be wasted. Second, NLP programs usually have character strings as their input variables, while some paths require specific input variables. It is very difficult to generate offspring test cases to cover those paths, given a SUT with an input string consisting of ten ASCII

Name	Description
$\bar{X} = \{x_1, x_2, \ldots, x_{pop}\}$	The population to be optimized by scatter search strategy.
pop	The population size of DE-SS.
x_i	The optimized test case.
p^{target}	The selected target path which needs to be covered first.
p^{x_i}	The path covered by x_i.
$p^{x_i}_k$	The kth character in the path encoding string of p^{x_i}.
P^r	The set of remaining uncovered paths.
$p^{r,j}$	The jth path in P^r.
τ	The number of elements in P^r.
ζ	The number of vertexes in tested NLP problems.
s	The integer control parameter in scatter search strategy.
$step$	The scatter search distance in DE-SS.
$temp$	A template variable to record original value of $x_{i,k}$.
ub_k/lb_k	The upper/lower bound of solution space in kth dimension.
$fitness(x_i)$	The fitness value of test case x_i.
fit	A template variable to record original fitness value of x_i.
fit'	A template variable to record the revised fitness value of x_i.

Fig. 3. Notation of variables in scatter search strategy.

Algorithm 1: SA-SS.

Input : tested NLP program,
Output: A test suite which contains the test cases of all possible paths

1 Initialize the test suite.
2 Initialize the population $\bar{X}$ randomly and evaluate all individuals in $\bar{X}$.
3 Record the covered paths and update the test suite.
4 **while** *Constraint (1) not satisfied and* $m \leq M$ **do**
5 Update the population $\bar{X}$ by search-based algorithms and evaluate the offspring test cases.
6 Record the covered paths and update the test suite.
7 Optimize the population $\bar{X}$ by scatter search strategy (Algorithm 3).
8 **end**

characters. If all characters are in the range of [0, 255], the probability for a randomly generated test case covering a path which requires specific character strings, is less than $(1/256)^{10}$ = 8.28E-25. Therefore, it is essential to design a suitable algorithm to cover remaining uncovered paths instead of the already covered paths. In addition, the designed algorithm should also consider generating test cases to cover the paths which require specific character strings.

III. The Proposed Algorithm With Scatter Search Strategy

There are three essential contents in our proposed scatter search strategy. The first one is the path encoding of the tested NLP programs. The second is the fitness function, and the third is the scatter search strategy (SS) for empowering search-based algorithms (SA-SS).

The variables which are used in the proposed strategy are shown in Fig. 3. Most variables, except for *pop* and s, are intermediate variables or default variables based on the test program. The population size *pop* will be set based on the references, while the influence of s will be discussed in Sections III-F and IV-B.

A. The Overview of SA-SS

The flowchart of the proposed SA-SS is depicted in Fig. 4, while the process of SA-SS is also shown as Algorithm 1. There are three steps for the proposed strategy: 1) initialization of population; 2) updating processes of search-based algorithms; and 3) scatter search strategy to cover remaining uncovered paths.

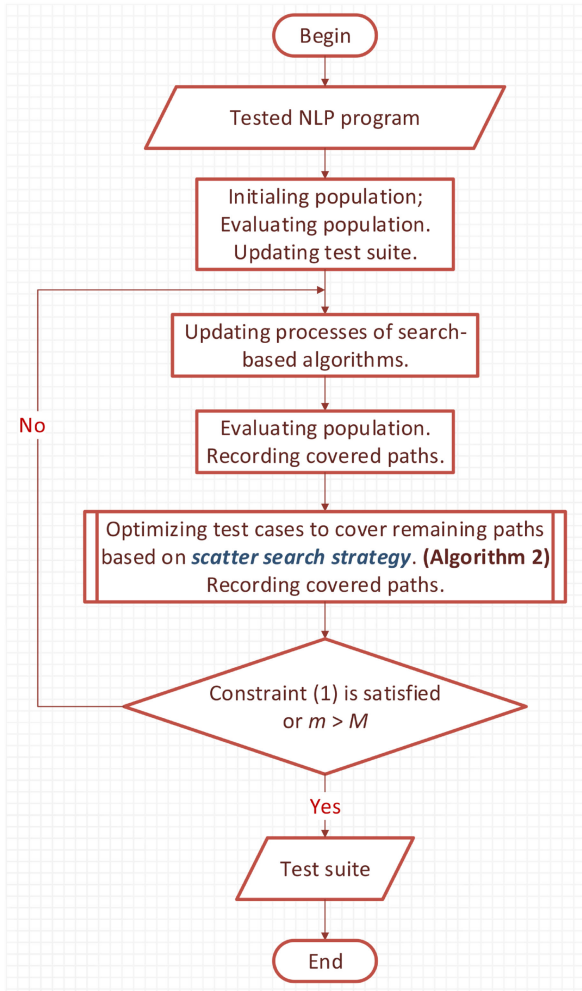


Fig. 4. The flowchart of SA-SS.

In the first step of SA-SS (line 1-3), all individuals will be initialized randomly. The value of $x_{i,k}$ will be set to $\lfloor lb_k + rand \times (ub_k - lb_k) \rfloor$, where $i = 1, 2, ..., pop$, $k = 1, 2, ..., n$ and $rand$ is a random real number in the range of [0, 1]. All individuals will be evaluated, while the covered paths and their corresponding test cases will be recorded by the test suite. Then, the process of proposed SA-SS will enter an iteration of second and third steps until Constraint (1) is satisfied or m is larger than M (line 4-8). The individuals will be updated based on the update processes of different search-based algorithms such as differential evolution [33]–[36], particle swarm optimization [37], [38] and competitive swarm optimizer [39] in the second step (line 5-6). In the third step, all individuals will be optimized by the scatter search strategy which is shown in Algorithm 3 (line 7). The test suite of all possible paths will be found at the end of SA-SS.

B. Path Encoding

The path encoding in this paper is based on the definition of the path in Section II-A. Path encoding is essential for ATCG-PC because Constraint (1) shows that the generated m test cases cover all paths in the SUT. The path encoding of the SUT can be represented by a string of characters.

Assume that the tested NLP program contains ζ vertexes, where each path is a sequence of running direction of ζ vertexes based on the Definition 2. Given a vertex in the tested NLP program, the paths will enter or skip the vertex. If the path enters the vertex, the running direction of the vertex can be represented by a unique nonempty character. The maximum number of running direction in each vertex is usually less than ten. The characters from 0 to 9 can express ten different running directions in all vertexes. If the path skips the vertex, an empty character can be used to represent that this path skipped this vertex. Therefore, each path in a SUT can be represented by a character string which has ζ characters.

If 0 represents the "Yes" direction and 1 indicates the "No" direction, the paths in Fig. 1 can be represented by a binary string with two digits. The paths "Yes, No", "Yes, Yes" and "No, No" can be represented by "01", "00" and "1 ", respectively.

Some essential tasks can be completed after proposing the path encoding of ATCG-PC for NLP programs. For example, the similarity between any two paths can be calculated based on path encoding. If we choose path p_j as our optimized target path, the offspring test case which covers the path with more similar characters with p_j will be allocated to the higher fitness value. This means that this test case has a higher-probability of being selected and optimized in the next generation. The detail calculation of the fitness value is shown in Section III-B.

C. The Fitness Function of SA-SS

Two fitness function strategies are applied in the proposed SA-SS algorithm.

The first fitness function strategy is the same as [14]. This strategy helps the algorithm search globally as no objective path is selected in this strategy. The individuals will be evaluated based on the fitness function in [14], after they are initialized or updated based on the processes of search-based algorithms.

The second fitness function strategy will be used in the scatter search strategy. The fitness value of x_i is the sum of all branch distances [40] between its p^{target} and x_i in each vertex. Specifically, the fitness value of test case x_i is calculated as:

$$\text{fitness}(x_i) = \sum_{j=1}^{\zeta} f(p^{x_i}, j) \tag{7}$$

where ζ is the number of vertex in the tested program, and $f(p^{x_i}, j)$ is the evaluation value of the jth vertex for x_i. The value of $f(p^{x_i}, j)$ will be calculated by Equation (8).

$$f(p^{x_i}, j) = \begin{cases} \frac{1}{BD(v_{x_i}^j)+\varepsilon}, & p^{x_i} \text{ covers } j\text{th vertex of } p^{target} \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

where $BD(v_{x_i}^j)$ is the branch distance between x_i and p^{target} in jth vertex of the tested program, ε is a small constant to avoid a zero denominator. The value of $BD(v)$ [25] in each judgement vertex is calculated based on Fig. 5.

If the offspring test case x_i has the same running direction of p^{target} in the jth vertex, the value of $BD(v_{x_i}^j)$ will be zero and $f(p^{x_i}, j)$ will have the highest value $1/\varepsilon$. Only the test case x_i, which covers the p^{target}, has the highest fitness value $fitness(x_i)$.

Predicated of statement	Branch Distance $BD(v)$
Boolean	If true then 0 else K
A>B	If (B−A)<0 then 0 else (B−A)+K
A≥B	If (B−A)≤0 then 0 else (B−A)+K
A<B	If (A−B)<0 then 0 else (A−B)+K
A≤B	If (A−B)≤0 then 0 else (A−B)+K
A=B	If |A−B| =0 then 0 else |A−B|+K
A≠B	If |A−B| ≠0 then 0 else K
A∧B	$BD(A) + BD(B)$
A∨B	Min$(BD(A), BD(B))$

Fig. 5. The Calculation of branch distance $BD(v)$.

All remaining uncovered paths are selected as the target paths which need to be covered in the first fitness function strategy. This strategy helps SA-SS to search globally. The second fitness strategy selects the single remaining uncovered path p^{target} as the objective. Only the test case which is the closest to p^{target} has the highest fitness value. If the branch distance value $BD(v_{x_i}^j)$ of test case x_i is large, the value of $f(p^{x_i}, j)$ will be small based on Equation (8) and Fig. 5. The total fitness value of x_i will be a small value after the calculation with Equation (7).

D. Scatter Search Strategy

As introduced in Section II-B, NLP programs usually have character strings as their input variables, while some paths need specific test case to cover. It is difficult for conventional search-based algorithms to generate test cases to cover those paths. In this subsection, a scatter search strategy is proposed to empower search-based algorithms in solving ATCG-PC for NLP programs.

Scatter search strategy is designed for covering the paths which requires specific values. Those paths usually can be found in NLP and programs in other research areas. As a result, the proposed scatter search strategy is suitable to empower the performance of search-based algorithms for ATCG-PC, while SS will have good performance if the test functions contains the paths which can only be covered if some variables should be their specific values.

There are two steps in the scatter search strategy. First, a target path p^{target} should be selected for optimized test cases x_i. This process is shown in Algorithm 2. The target path p^{target} is selected based on the path encoding string p^{x_i} and other remaining uncovered paths in P^r. For a path $p^{r,j}$, the more the same characters exist between p^{target} and $p^{r,j}$, the higher the probability that $p^{r,j}$ will be selected. The weight vector R is initialized and updated in Line 1 to 8 of Algorithm 2. The process in Line 9 shows that the target path p^{target} is selected by roulette-wheel selection based on weight vector R.

The second step is to generate an offspring test case x_i to cover p^{target}. This process is shown as Algorithm 3. As indicated from Line 1 to Line 2, the target path $p^{targety}$ will be selected for each optimized test case x_i. Then, all dimensions of x_i will be explored from Line 3 to Line 26 of Algorithm 3.

At first, the scatter search distance $step$ will be initialized based on the upper or lower bound in each solution space dimension. Then, the iteration will not stop until p^{target} has been

Algorithm 2: Target path selecting.

Input : x_i, P^r and τ
Output: p^{target}

Initialize the vector $R = (r_1, r_2, ..., r_\tau)$ to all zero vector.
for *each* $j \in \{1, 2, ..., \tau\}$ **do**
  for *each* $k \in \{1, 2, .., \zeta\}$ **do**
    if $p_k^{x_i}$ *equals to* $p_k^{r_j}$ **then**
      $r_j \leftarrow r_j + 1$
    end
  end
end
Generate a target path p^{target} by roulette-wheel selection based on vector $R = (r_1, r_2, ..., r_\tau)$.

Algorithm 3: Scatter search strategy.

Input : The population $\bar{X}$
Output: A set of test cases which cover the remaining uncovered paths

for *each* $i \in \{1, 2, ..., pop\}$ **do**
  Select a target path p^{target} for x_i based on Algorithm 2.
  for *each* $k \in \{1, 2, ..., n\}$ **do**
    Initialize the scatter search distance: $step \leftarrow (ub_k - lb_k)/s$.
    while p^{target} *is not be covered and* $step > 0$ **do**
      for *each* $\alpha \in \{1, 2, ..., s\}$ **do**
        $temp \leftarrow x_{i,k}$.
        $fit \leftarrow fitness(x_i)$.
        $x_{i,k} \leftarrow x_{i,k} + step \times (\alpha - s/2)$.
        Evaluate test case x_i based on Equation (7).
        $fit^{'} \leftarrow$ the fitness value of x_i.
        if $fit^{'} < fit$ **then**
          $x_{i,k} \leftarrow temp$.
          $fitness(x_i) \leftarrow fit$.
        end
        else
          $fitness(x_i) \leftarrow fit^{'}$.
        end
        Evaluate the test case x_i based on Equation (7) and find the path p^{x_i} which is covered by x_i.
        if p^{x_i} *equals to* p^{target} **then**
          Go back to Line 2.
        end
      end
      $step \leftarrow step/s$.
    end
  end
end

covered or $step \leq 0$. s offspring test cases will be generated based on the formulation in Line 9. These s test cases will be evaluated and compared with the original test case x_i. If the offspring test case has a higher fitness value than x_i, x_i will be replaced with this test case. If p^{target} has been covered by the generated test cases, the search process will go back to Line 2 and will cover remaining uncovered paths.

An example of using scatter search strategy is exhibited in Fig. 6. Given a SUT with three input variables, we assume that test case $x_i = (1, 2, 3)$ needs to be optimized and covers the target path p^{target}. p^{target} can only been covered when the character string is "ABC". The ASCII code string which corresponds to p^{target} is (65, 66, 67). This means that x_i covers p^{target} only when x_i equals to (65, 66, 67). The value of control parameter s for SA-SS is set to be two.

As indicated from Line 3 to Line 4 of Algorithm 3, if k equals to one, $step$ will be initialized to 127. Then, two offspring test cases (128, 2, 3) and (255, 2, 3) will be generated based on the formulation in Line 9. The fitness function among x_i, (128, 2, 3) and (255, 2, 3) will be compared. The value of x_i will be replaced by (128, 2, 3) because it has the highest fitness value. The value of $step$ will be updated to be 63. In the next iteration of Line 6 to Line 23, two offspring test cases (65, 2, 3) and (191, 2, 3) will be generated. After comparing the fitness value among x_i, (65, 2, 3) and (191, 2, 3), x_i will be set to (65, 2, 3). In the next several iteration of Line 6 to Line 23, $step$ will be updated to 31, 15, 7, 3, 1 and 0 progressively based on the process of Line 24. Ten test cases will be generated based on x_i and its corresponding $step$ value. However, both offspring test cases have a lower fitness value than (65,2,3). Fourteen test cases will be generated for exploring the first dimension of x_i.

The second and third dimension of x_i will also be explored when k equals to two and three. The number of offspring test cases for exploring the second and third dimension of x_i is 26 (two offspring test cases are out of range). With the same operation in Line 2 to Line 26 of Algorithm 3, x_i will be updated to (65,66,67) and covers target path p^{target}, whose corresponding character string is "ABC". The number of offspring test cases for scatter search strategy optimizing x_i to cover p^{target} is just 40. This example shows that scatter search strategy can easily cover one selected path with litte test case consumption. Many test cases will be saved with the proposed scatter search strategy.

E. Computational Complexity Analysis

The computational complexity of SA-SS for ATCG-PC mainly contains the following two parts: 1) the conventional update processes of search-based algorithms and 2) searching based on scatter search strategy.

For the first part, it needs $O(pop \cdot n)$ time to initialize the population. The computational time of different search-based algorithms are different. For example, the computational time of differential evolution is $O(pop \cdot n)$. For the second part, the variables of all dimensions will be explored. The computational complexity of scatter search strategy is $O(pop \cdot n \cdot s \cdot log_s B)$, where B equals to $(ub_k - lb_k)$. Considering most search-based algorithms have less computational time than scatter search strategy, the totally computational complexity of SA-SS is $O(pop \cdot n \cdot s \cdot log_s B)$.

F. Parameter Analysis

From the computational complexity of SA-SS, we find that parameter s will influence the computational time of SA-SS in ATCG-PC. If the value of s is very large or close to the value of $(ub_k - lb_k)$, too many test cases will be generated when using

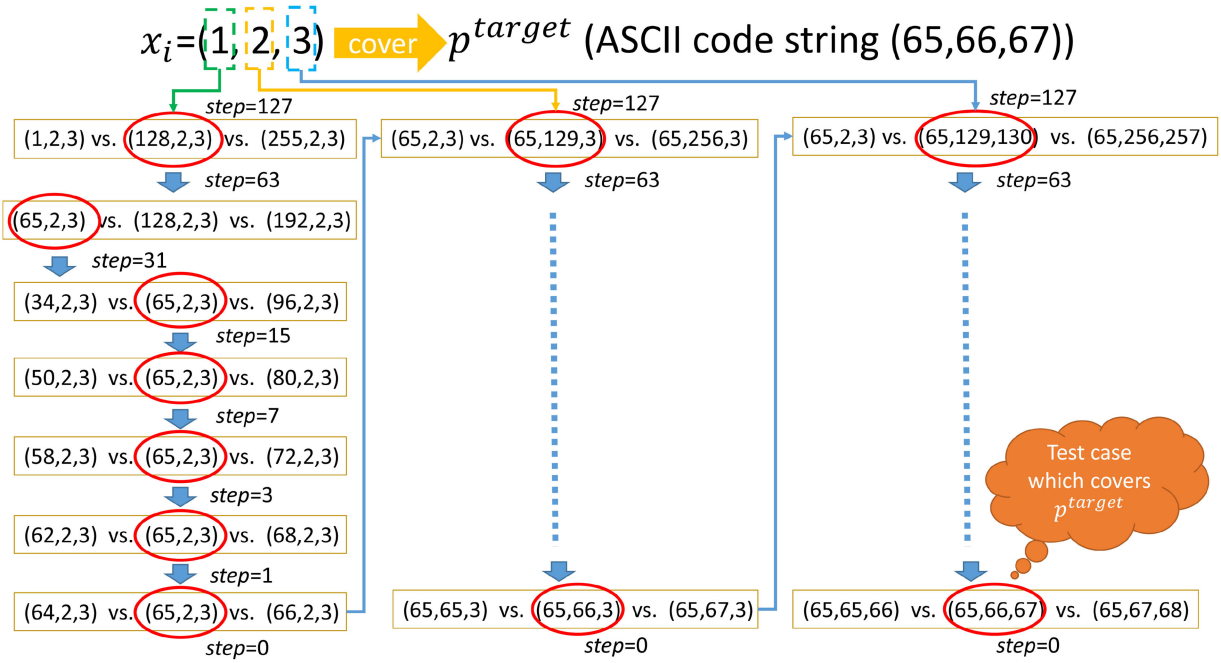


Fig. 6. An example of optimization based on scatter search strategy.

scatter search strategy. However, the most suitable value of s needs to be obtained by mathematical proof.

Let B equals to $(ub_k - lb_k)$ and t equals to $\lceil log_s B \rceil$. The number of times for evaluating test cases in the iteration from Line 6 to Line 23 of Algorithm 3 is computed as:

$$E_a = \frac{s^{a+1}}{B} \tag{9}$$

where $a = 1, 2, ..., t$. The expectation of total evaluation time is:

$$\sum_{a=1}^{t} E_a = \frac{s^2}{B} \times \frac{1 - s^t}{1 - s} \tag{10}$$

The value of t is very close to $log_s B$. If t equals to $log_s B$, the value of $\sum E_a$ can be calculated as:

$$\sum_{a=1}^{t} E_a = \frac{B-1}{B} \times \left[(s-1) + \frac{1}{s-1} + 2 \right] \tag{11}$$

Therefore, $\sum_{a=1}^{t} E_a$ can have its minimal value when $s = 2$. We also find that the computational complexity of SA-SS will have the smallest value when s equals to two.

IV. Simulation Results

In this section, three experiments will be conducted. First, the efficient of scatter search strategy will be evaluated with different s values. We will use DE-SS as our comparing algorithm. Then, DE-SS and DE will be compared based on NLP programs. Finally, DE-SS will be compared with other state-of-the-art algorithms and their variants based on the scatter search strategy.

A. Experiment Setting

All benchmark programs were selected from a professional NLP toolkit–the Standford CoreNLP [7]. The CoreNLP simulates the application about natural language processing. More detailed information about these benchmark programs is presented in Table I and II. Those programs are some of the common processes in NLP technique. The maximum number of paths in these benchmarks is 48, while some paths need specifical input variables to cover them. As shown in Table II, each benchmark program has very low minimum path probability. Besides, the solution space is much larger than the maximum test case number M. It is difficult for some simple search-based algorithms such as conventional DE to cover all paths within the maximum test case number M. Only the search-based algorithms with strong searching ability can use few test cases to cover all possible paths in these benchmark functions.

There are three experiments in this section.

1) The parameter sensitive analysis about s in DE-SS will be discussed.
2) DE-SS algorithm will be compared with DE [14] based on the tested NLP programs.
3) DE-SS will be compared with immune genetic algorithm (IGA) [25], artificial bee colony (ABC) [41], particle swarm optimization (PSO) [30], competitive swarm optimizer (CSO) [39] and their variants based on the scatter search strategy (IGA-SS, ABC-SS, PSO-SS, CSO-SS) and based on the tested NLP programs. Besides, DE-SS will also be compared with the differential evolution based on relationship matrix (RP-DE) [32].

TABLE I
BENCHMARK PROGRAMS

No.	Program	Loc	Dim	Description	Path
1	initFactory	86	7	to return the right type of token based on the options in the properties file and the type	48
2	cleanXmlAnnotator	21	6	to create a new object of cleanXmlAnnotator class	3
3	wordsToSentenceAnnotator	108	11	to transform a text of natural language to an annotator type	12
4	annotate	30	4	to turn the annotation into a sentence	3
5	nerClassifierCombiner	30	11	to create a new object of nerClassifierCombiner class	4
6	setTrueCaseText	37	6	to set the attribute of class trueCaseText	10

* Loc, Dim and Path represent the number of lines, input dimensions and paths in each tested program.
* Description introduces the task of test program in CoreNLP toolkit.

TABLE II
COMPLEXITY OF BENCHMARK PROGRAMS

Function ID	Solution space (N)	Minimum path probability
1	6.04E+23	1.65E-24
2	1.84E+19	3.55E-15
3	1.89E+22	2.33E-10
4	3.60E+16	8.33E-17
5	1.44E+17	3.55E-15
6	2.20E+12	9.07E-13

* Solution space represents the number of possible solutions in the benchmark programs. Minimum path probability denotes the probability for a randomly generated test case to cover the path with the smallest possible solutions.

TABLE III
PARAMETER SETTING

Parameter	value
Population size pop	50
Maximum test case number M	3.00E+05
Number of trials	30
Wilcoxon rank-sum testing α	0.05
DE factor parameter F	0.5
DE crossover probability P_c	0.2
IGA crossover probability	0.8
IGA mutation probability	0.1
ABC limit	2
ABC bee number	50
c1 and c2 in PSO	1.5, 2
weight value in PSO	0.4
phi in CSO	0.8

To investigate the influence of parameter s of SA-SS, the first experiment will discuss the performance of SA-SS with different s values in the DE-SS algorithm. The second experiment will show that the scatter search strategy empowers DE in automatically generating test cases based on path coverage. The third experiment will show the efficiency of the proposed scatter search strategy for other state-of-the-art algorithms.

We established compared algorithms on the same computational environment. All our evaluations were performed on a 64-bit Windows Education 10 OS PC with an Intel i5-4590 3.30 GHz processor and 16 GB RAM. All the parameter values are listed in Table III. The population size was set to 50 [14], [32]. The maximum number of test cases in our proposed mathematical model was attributed to 3.00E+05 [42]. Each experiment will be independently executed 30 times. The method of significance testing was based on Wilcoxon rank-sum testing [43] with $\alpha = 0.05$. The parameters of compared algorithms were set per the parameter settings in their references [14], [25], [30], [39], [41].

TABLE IV
EXPERIMENTAL RESULTS OF DE-SS WITH DIFFERENT s VALUE

Function ID	DE-SS-2		DE-SS-3	
	Ave.m(Std.m)	Rate	Ave.m(Std.m)	Rate
1	**7.42E+03(1.32E+03)**	**100%**	6.23E+04(6.21E+03)	100%
2	**2.81E+02(2.59E+00)**	**100%**	4.22E+02(1.30E+02)	100%
3	**2.24E+03(4.42E+02)**	**100%**	2.73E+03(6.39E+02)	100%
4	**4.30E+02(1.42E+02)**	**100%**	2.05E+04(7.01E+03)	100%
5	**6.48E+02(1.92E+02)**	**100%**	7.14E+02(2.39E+03)	100%
6	**1.51E+03(2.35E+02)**	**100%**	4.85E+03(4.23E+02)	100%
+/=/-			6/0/0	
Function ID	DE-SS-5		DE-SS-10	
	Ave.m(Std.m)	Rate	Ave.m(Std.m)	Rate
1	2.26E+05(8.73E+04)	46.7%	2.83E+05(3.62E+04)	20%
2	1.16E+04(7.83E+03)	100%	1.38E+04(9.59E+03)	100%
3	2.64E+04(1.98E+04)	100%	3.10E+04(1.44E+04)	100%
4	2.43E+04(5.78E+03)	100%	2.40E+04(6.69E+03)	100%
5	1.27E+05(8.19E+04)	93.3%	1.36E+05(7.39E+04)	90%
6	9.99E+04(4.68E+04)	100%	8.99E+04(2.80E+04)	100%
+/=/-	6/0/0		6/0/0	

B. Parameter Sensitive Analysis in DE-SS

In this subsection, we test the performance of DE-SS algorithm with different s values. More specifically, s equals to 2, 3, 5 and 10, respectively. As introduced in Section III-E, SA-SS has the smallest computational complexity when parameter $s = 2$. The objective of this experiment is to prove this conclusion by the experimental result. We use DE-SS as our selected search-based algorithm.

The performance of DE-SS with different s values is presented in Table IV. DE-SS-k (k = 2, 3, 5 and 10) represents that DE-SS algorithm sets its s value to k. The best values among these compared algorithms are noted in bold font. Measures used in Table IV are introduced as follows.

1) Ave.m (Std.m): the average number (standard deviation) of test case consumption in test benchmark programs.
2) Rate: the proportion for the compared algorithm covering all possible paths in 30 trials. The value of Rate equals to $(c/30) * 100\%$, while c is the number of times for the compared algorithm covering all possible paths in 30 trials.
3) $+/=/-$: the numbers of '$+$', '$=$' and '$-$' represent that DE-SS-2 uses significantly less, equal and more test cases than DE-SS-3, DE-SS-5 and DE-SS-10 by the Wilcoxon rand-sum test with $\alpha = 0.05$.

As indicated in Table IV, DE-SS-2 performs the best and uses the least average test case consumption when it is compared with DE-SS-5, DE-SS-10 and DE-SS-100 under 30 trials. DE-SS-2

TABLE V
EXPERIMENTAL RESULTS OF DE-SS WITH DIFFERENTIAL EVOLUTION

Function ID	DE-SS		DE	
	Ave.m(Std.m)	Rate	Ave.m(Std.m)	Rate
1	**7.33E+03(1.35E+03)**	**100%**	3.00E+05(0)	0%
2	**2.80E+02(3.62E+00)**	**100%**	2.67E+05(5.93E+04)	13.3%
3	**2.27E+03(5.26E+02)**	**100%**	3.00E+05(0)	0%
4	**4.14E+02(1.15E+02)**	**100%**	3.00E+05(0)	0%
5	**6.47E+02(1.72E+02)**	**100%**	3.70E+04(1.99E+04)	100%
6	**1.54E+03(2.53E+02)**	**100%**	3.00E+05(0)	0%
+/=/-			6/0/0	

uses significantly less test cases than compared algorithms in all benchmark programs, while its standard deviation is also the smallest. Even in program No. 1 which has 48 paths, the Ave.m of DE-SS-3 is eight times more the Ave.m of DE-SS-2. DE-SS-5, DE-SS-10 also uses several orders of magnitude more Ave.m than DE-SS-2.

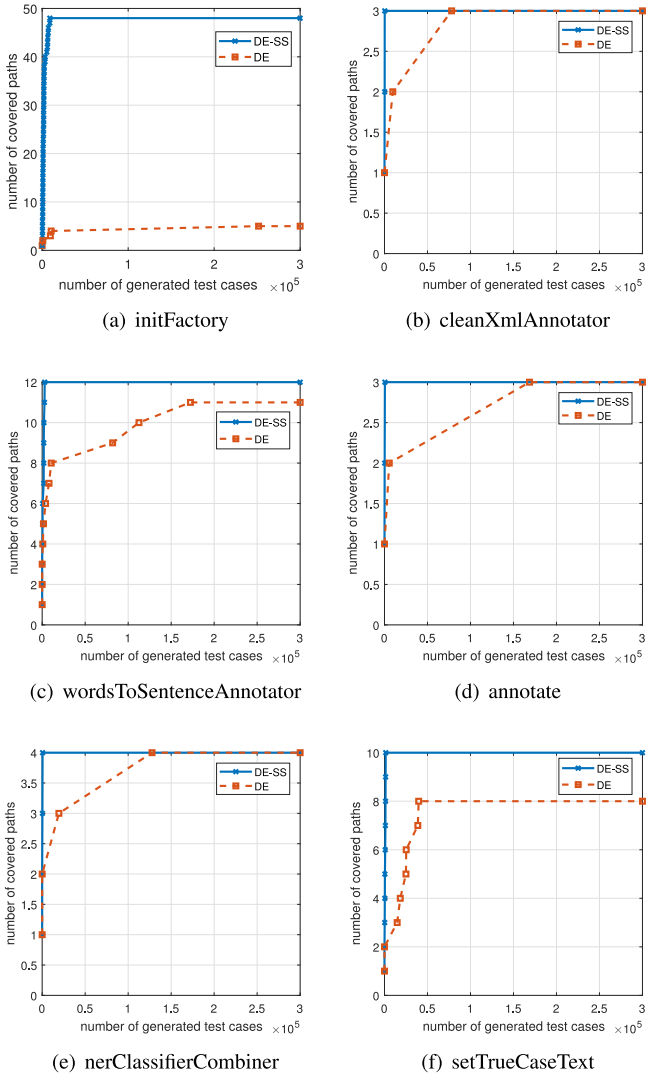
If SA-SS is used to solve ATCG-PC for NLP programs, the s value should be set to 2. The significant testing analysis in Table IV shows that DE-SS-2 uses significantly less test cases than DE-SS-3, DE-SS-5 and DE-SS-10 in all benchmark programs. It also seems that the average test case consumption of SA-SS, such as DE-SS, will increase when the value of s increases. The Rate value may also decrease comparing with s increasing. The experimental result corresponds to our proof in Section III-F that SA-SS has the smallest computational complexity when parameter s equals to two.

C. Comparing DE-SS With DE

In this subsection, DE-SS will be compared with DE based on the benchmark functions. The value of s is set to two, according to our explanation in Section III-E and the experimental result in Section IV-B. The experimental results are shown in Table V and Fig. 7. Table V indicates the Ave.m (Std.m) and Rate value when DE-SS and DE are compared on the benchmark functions. Fig. 7 is presented to show the convergence speed of DE-SS and DE on the benchmark functions. Most measures are the same in Section IV-B, while "+/=/-" represents that DE-SS uses significantly less, equal or more test cases than differential evolution.

As shown in Table V, DE-SS outperforms DE in all benchmark functions. DE cannot cover all paths once, it is within 30 trials in tested programs No. 1, No. 3, No. 4 and No. 6. However, DE-SS can cover all paths in all 30 trials, while the maximum average test case consumption m is less than 1.00E+04. In tested program No. 5, DE covers all path within 30 trials, while DE-SS uses significantly less test cases than DE. The proposed scatter search strategy not only saves many test case consumptions, but also greatly increases the probability of DE covering all possible paths.

The scatter search strategy quickly explodes all test case dimensions and significantly increases the convergence speed of DE. After exploding all dimensions, SS helps to cover all paths with a small number of test cases. Fig. 7 shows an example of the covered paths achieved over the generated test cases by DE-SS and DE in NLP programs. From Fig. 7, we can observe that



(a) initFactory (b) cleanXmlAnnotator

(c) wordsToSentenceAnnotator (d) annotate

(e) nerClassifierCombiner (f) setTrueCaseText

Fig. 7. The number of covered path over generated test cases in NLP programs.

within 5.00E+04 test cases, DE-SS is particularly efficient compared to DE. DE cannot cover all paths within 5.00E+04 test cases, while DE-SS can cover all paths very quickly, especially in test programs of Fig. 7(a), (c) and (f).

In summary, we can conclude that the scatter search strategy empowers the DE and increases its convergence speed in solving ATCG-PC for NLP programs. In other words, the scatter search strategy helps DE cover the remaining uncovered paths with less test cases.

D. Comparing DE-SS With Other State-of-the-Art Algorithms

In this subsection, the performance of our proposed DE-SS algorithm will be compared with the performance of other state-of-the-art algorithms and their variants based on the scatter search strategy. Besides, DE-SS will also be compared with RP-DE. The parameter s of SS is set to 2. Table VI shows the Ave.m (Std.m), Rate values and running time of DE-SS and other state-of-the-art algorithms on the benchmark functions. Time represents the average running time of the compared algorithm in 30

TABLE VI
EXPERIMENTAL RESULTS OF DE-SS WITH OTHER STATE-OF-THE-ART ALGORITHMS

Function ID	DE-SS			IGA			IGA-SS		
	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time
1	**7.14E+03(9.75E+02)**	**100%**	**7.40E00**	3.00E+05(0)	0	3.64E+02	7.80E+03(1.60E+03)	100%	8.63E00
2	2.79E+02(3.79E+00)	100%	7.33E-01	2.90E+05(1.91E+04)	3.3%	3.11E+02	**2.04E+02(2.55E+00)**	**100%**	**7.67E-01**
3	**2.21E+03(5.20E+02)**	**100%**	**3.33E00**	2.91E+05(1.78E+04)	3.3%	3.91E+02	1.94E+04(1.45E+04)	100%	1.51E+01
4	**4.26E+02(1.70E+02)**	**100%**	**9.33E-01**	3.00E+05(0)	0	2.13E+02	6.23E+02(5.67E+02)	100%	1.27E00
5	**5.82E+02(1.66E+02)**	**100%**	**1.63E00**	1.26E+05(7.34E+04)	86.7%	1.02E+02	2.24E+03(2.94E+03)	100%	2.33E00
6	**1.40E+03(1.65E+02)**	**100%**	**2.07E00**	3.00E+05(0)	0	3.42E+02	1.13E+04(4.57E+03)	100%	9.53E00
+/=/-				6/0/0			2/3/1		

(a) DE-SS compares with IGA, its variant based on scatter search strategy.

Function ID	DE-SS			ABC			ABC-SS		
	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time
1	**7.14E+03(9.75E+02)**	**100%**	**7.40E00**	3.00E+05(0)	0	1.69E+02	7.78E+03(1.14E+03)	100%	6.13E00
2	**2.79E+02(3.79E+00)**	**100%**	**7.33E-01**	3.00E+05(0)	0	1.33E+02	2.88E+02(7.52E+00)	100%	1.2E00
3	**2.21E+03(5.20E+02)**	**100%**	**3.33E00**	3.00E+05(0)	0	1.21E+02	2.44E+03(6.03E+02)	100%	3.3E00
4	**4.26E+02(1.70E+02)**	**100%**	**9.33E-01**	3.00E+05(0)	0	9.14E+01	4.73E+02(3.01E+02)	100%	1.03E00
5	**5.82E+02(1.66E+02)**	**100%**	**1.63E00**	3.00E+05(0)	0	1.50E+02	8.86E+02(3.99E+02)	100%	1.3E00
6	**1.40E+03(1.65E+02)**	**100%**	**2.07E00**	3.00E+05(0)	0	1.36E+02	1.85E+03(4.57E+02)	100%	2.87E00
+/=/-				6/0/0			3/3/0		

(b) DE-SS compares with ABC and its variant based on scatter search strategy.

Function ID	DE-SS			PSO			PSO-SS		
	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time
1	**7.14E+03(9.75E+02)**	**100%**	**7.40E00**	3.00E+05(0)	0	2.60E+02	8.54E+03(2.27E+03)	100%	6.27E00
2	**2.79E+02(3.79E+00)**	**100%**	**7.33E-01**	3.00E+05(0)	0	2.27E+02	2.30E+02(3.79E+00)	100%	1.07E00
3	**2.21E+03(5.20E+02)**	**100%**	**3.33E00**	3.00E+05(0)	0	3.33E+02	3.34E+03(1.30E+03)	100%	5.43E00
4	**4.26E+02(1.70E+02)**	**100%**	**9.33E-01**	3.00E+05(0)	0	1.48E+02	4.99E+02(2.56E+02)	100%	1.03E00
5	**5.82E+02(1.66E+02)**	**100%**	**1.63E00**	3.00E+05(0)	0	2.45E+02	7.67E+02(3.83E+02)	100%	1.43E00
6	**1.40E+03(1.65E+02)**	**100%**	**2.07E00**	3.00E+05(0)	0	2.14E+02	2.01E+03(5.31E+02)	100%	2.27E00
+/=/-				6/0/0			4/1/1		

(c) DE-SS compares with PSO and its variants based on scatter search strategy.

Function ID	DE-SS			CSO			CSO-SS		
	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time
1	**7.14E+03(9.75E+02)**	**100%**	**7.40E00**	3.00E+05(0)	0	2.36E+02	7.97E+03(1.68E+03)	100%	8.33E00
2	2.79E+02(3.79E+00)	100%	7.33E-01	2.70E+05(5.56E+04)	10%	1.82E+02	**2.05E+02(3.41E+00)**	**100%**	**1.17E00**
3	**2.21E+03(5.20E+02)**	**100%**	**3.33E00**	3.00E+05(0)	0	3.28E+02	2.52E+03(7.27E+02)	100%	5.07E00
4	4.26E+02(1.70E+02)	100%	9.33E-01	3.00E+05(0)	0	1.51E+02	**4.10E+02(2.12E+02)**	**100%**	**9.00E-01**
5	5.83E+02(1.66E+02)	100%	1.63E00	3.00E+05(0)	0	2.56E+02	**5.62E+02(2.13E+02)**	**100%**	**1.17E00**
6	**1.40E+03(1.65E+02)**	**100%**	**2.07E00**	3.00E+05(0)	0	2.14E+02	1.50E+03(3.63E+02)	100%	2.23E00
+/=/-				6/0/0			1/4/1		

(d) DE-SS compares with CSO and its variants based on scatter search strategy.
* Time: the average running time (ms) of the compared algorithm in 30 trials.

trials. Fig. 8 presents the box plots of test case consumption between DE-SS and the compared state-of-the-art algorithms. The best values among these compared algorithms are noted in bold font. Most measures are the same in Section IV-B, while symbols "+", "=" and "−" represent that DE-SS's performance is statistically better than, equivalent to, or worse than the compared algorithms.

As shown in Table VI, DE-SS empowers other state-of-the-art algorithms such as IGA [25], ABC [41], PSO [30] and CSO [39] in automatically generating test cases based on path coverage. DE-SS uses significantly less test cases and running time to cover all possible paths in benchmark functions than the compared state-of-the-art-algorithms.

The scatter search strategy helps these state-of-the-art algorithms achieve better performance in ATCG-PC for NLP programs. IGA-SS uses several orders of magnitude less test cases than IGA in all benchmark functions. Furthermore, the running time of IGA-SS is significantly less than IGA. The result shows that SS will not increase too much the running time of IGA. Similarly, ABC-SS performs statistically better than ABC in all benchmark functions. PSO-SS uses significantly less test cases than PSO in all benchmark functions, while CSO-SS achieves better performance than CSO in all benchmark functions.

The proposed scatter search strategy helps the compared state-of-the-art algorithms to cover all possible paths. Full path coverage can significantly strengthen the process of identification software bugs [14]. As shown in Table VI, the compared state-of-the-art algorithms usually has very low Rate value. The compared algorithms even achieved a 0-rate at some benchmark functions. The reason is that the compared algorithms cannot find test cases for all paths within the limited test case consumption. According to the definition of Rate, the value of Rate will be zero if the compared algorithm cannot achieve 100% path coverage at once in 30 trials. Even if the compared algorithm can cover 99% paths in each trial, its Rate value is still zero.

When we compare DE-SS with the variants of the state-of-the-art algorithms (GA-SS, ABC-SS and PSO-SS) based on the scatter search strategy, we find that DE-SS performs the best

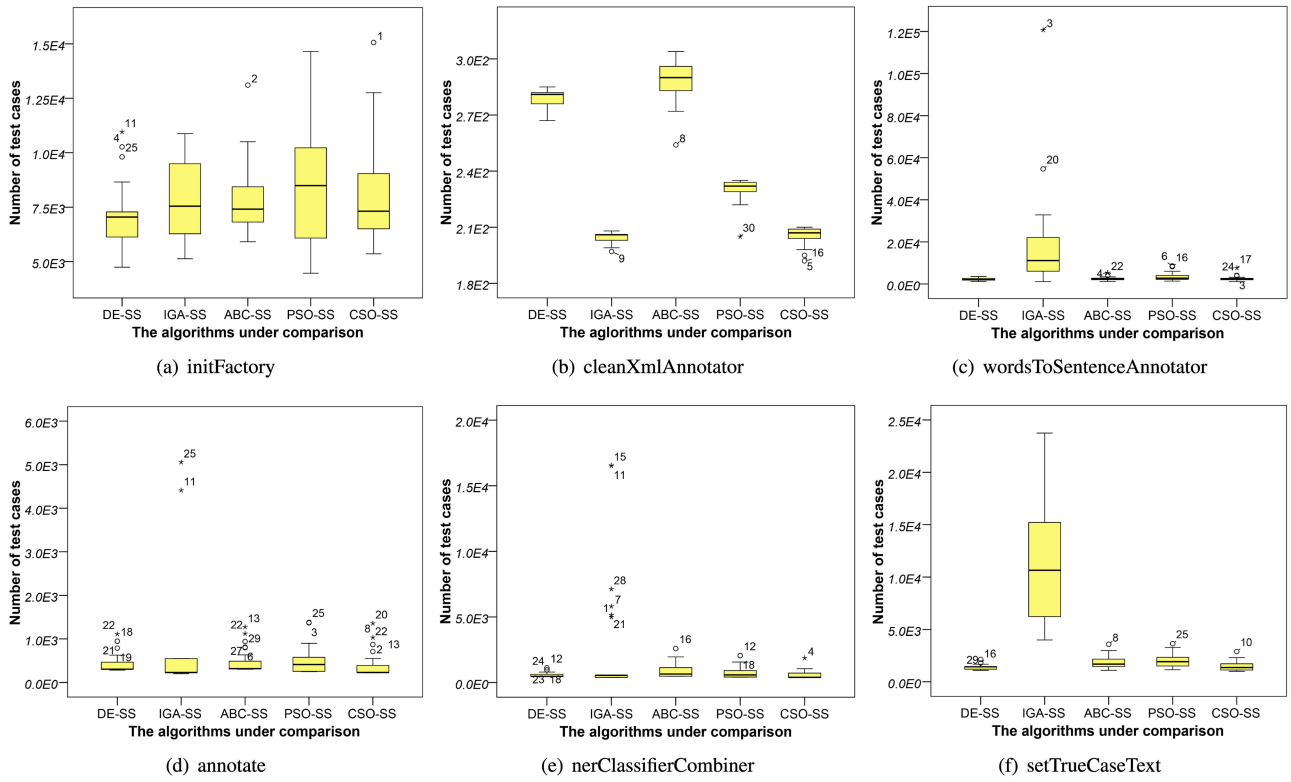


Fig. 8. The box plots on benchmark functions for algorithms based on scatter search strategy.

when the number of paths in the tested functions are larger than ten, while CSO-SS has the best performance in the other condition. To depict the performance of the compared algorithm, the box plots of DE-SS, IGA-SS, ABC-SS, PSO-SS and CSO-SS on benchmark programs are presented in Fig. 8. DE-SS uses the least number of test cases in program No. 1, while IGA-SS and CSO-SS generates very few test cases in program No.2. The performance of DE-SS, ABS-SS, PSO-SS and CSO-SS cannot be compared in programs No. 3, 4,5 and 6. If we focus on the value of "+/=/−" between DE-SS and CSO-SS in Table VI (b), we find that these two algorithms indicate similar performance in benchmark functions. CSO-SS uses less test cases than DE-SS in program No. 2, 4 and 5, while DE-SS outperforms CSO-SS in programs No. 1, 3 and 6. Programs No. 1, No. 3 and No. 6 have more than ten paths while the programs No. 2, No. 4 and No. 5 only have three or four paths. We consider that DE-SS is more suitable for tested NLP programs which contain more than ten paths, while CSO-SS is more suitable in tested NLP programs with less than ten paths.

The experiment results for comparing DE-SS and RP-DE is shown in Table VII. RP-DE records the dependent relationship between test cases and paths by a test-case-path relationship matrix. The offspring test cases will be generated based on the collected matrix. Table VII shows that DE-SS uses significantly less test cases for covering all possible paths in all benchmark programs. RP-DE achieves the 100% Rate value in all benchmark programs, however, DE-SS uses less test case consumption and achieves the same result. Furthermore, the running time of

TABLE VII
EXPERIMENTAL RESULTS OF DE-SS WITH RP-DE

Function ID	DE-SS			RP-DE		
	Ave.m(Std.m)	Rate	Time	Ave.m(Std.m)	Rate	Time
1	**7.66E+03(1.21E+03)**	**100%**	**7.40E00**	4.74E+04(9.23E+02)	100%	1.01E+02
2	**2.31E+02(2.38E+00)**	**100%**	**7.30E-01**	7.04E+03(8.29E+02)	100%	1.19E+01
3	**2.50E+03(6.40E+02)**	**100%**	**3.33E00**	7.70E+02(1.52E+02)	100%	2.27E+01
4	**4.50E+02(2.05E+02)**	**100%**	**9.33E-01**	5.21E+02(8.08E+02)	100%	1.55E+01
5	**5.79E+02(1.82E+02)**	**100%**	**1.63E00**	8.57E+03(7.71E+02)	100%	1.59E+01
6	**1.41E+03(2.24E+02)**	**100%**	**2.07E00**	1.89E+04(4.66E+03)	100%	4.80E+01
+/=/-				6/0/0		

RP-DE is much larger than DE-SS in all benchmark programs. The results show that scatter search strategy for ATCG-PC of NLP programs is better than the updating strategy based on test-case-path relationship matrix.

From the experimental results in Table VI and Fig. 8, we can conclude that the proposed scatter search strategy empowers other search-based algorithms such as IGA [25], ABC [41], PSO [30] and CSO [39] in solving ATCG-PC for NLP programs. Many other search-based algorithms can be applied to the proposed framework. In addition, DE-SS and CSO-SS are the two most competitive algorithms among all compared search-based algorithms.

V. CONCLUSION

In this paper, a framework based on the scatter search strategy is proposed for ATCG-PC of NLP programs. Some paths can only be covered with some specific input variables, while the proposed scatter search strategy can explore each input variable dimension with small test cases and cover those paths. In

addition, we have proven that the scatter search strategy will have the best performance when parameter s equals to two. The experimental results indicate that the scatter search strategy increases the convergence speed of search-based algorithms and empower their performance in ATCG-PC. Among all compared algorithms, DE-SS and CSO-SS perform the best. Specifically, DE-SS is more suitable for ATCG-PC of NLP programs when tested programs have more than ten paths, while CSO-SS performs better in the tested programs with less than ten paths.

In our future work, we will further improve the performance of search-based algorithm in solving ATCG-PC for more real-world software. We will extend our algorithm in the tested programs with large number of variables or paths by reducing dimensions or using a grouping strategy.

REFERENCES

[1] E. Cambria and B. White, "Jumping NLP curves: A review of natural language processing research [review article]," *IEEE Comput. Intell. Mag.*, vol. 9, no. 2, pp. 48–57, May 2014.

[2] G. G. Chowdhury, "Natural language processing," *Annu. Rev. Inf. Sci. Technol.*, vol. 37, no. 1, pp. 51–89, 2003.

[3] F. Thung, R. J. Oentaryo, D. Lo, and Y. Tian, "WebAPIRec: Recommending web APIs to software projects via personalized ranking," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 1, no. 3, pp. 145–156, Jun. 2017.

[4] K. Zheng, W. Q. Yan, and P. Nand, "Video dynamics detection using deep neural networks," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 2, no. 3, pp. 224–234, Jun. 2018.

[5] X. Li, Y. Rao, H. Xie, X. Liu, T.-L. Wong, and F. L. Wang, "Social emotion classification based on noise-aware training," *Data Knowl. Eng.*, 2017, doi: 10.1016/j.datak.2017.07.008.

[6] A. Argal, S. Gupta, A. Modi, P. Pandey, S. Shim, and C. Choo, "Intelligent travel chatbot for predictive recommendation in echo platform," in *Proc. IEEE 8th Annu. Comput. Commun. Workshop Conf.*, 2018, pp. 176–183.

[7] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. Mcclosky, "The Stanford Corenlp natural language processing toolkit," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics, Syst. Demonstrations*, 2014, pp. 55–60.

[8] X. Chen *et al.*, "Microsoft coco captions: Data collection and evaluation server," 2015, arXiv preprint arXiv:1504.00325.

[9] D. Chen, A. Fisch, J. Weston, and A. Bordes, "Reading wikipedia to answer open-domain questions," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, Vancouver, Canada, Jul. 2017, vol. 1, pp. 1870–1879, doi: 10.18653/v1/P17-1171.

[10] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, Feb. 2018.

[11] P. S. Douglas *et al.*, "Outcomes of anatomical versus functional testing for coronary artery disease," *New Engl. J. Med.*, vol. 372, no. 14, pp. 1291–1300, 2015.

[12] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007.

[13] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 597–608.

[14] H. Huang, F. Liu, X. Zhuo, and Z. Hao, "Differential evolution based on self-adaptive fitness function for automated test case generation," *IEEE Comput. Intell. Mag.*, vol. 12, no. 2, pp. 46–55, May 2017.

[15] J. R. Horgan, S. London, and M. R. Lyu, "Achieving software quality with testing coverage measures," *Comput.*, vol. 27, no. 9, pp. 60–69, 1994.

[16] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[17] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3440. Berlin, Germany: Springer, 2005, pp. 365–381.

[18] X. Yao, D. Gong, and W. Wang, "Test data generation for multiple paths based on local evolution," *Chin. J. Electron.*, vol. 24, no. 1, pp. 46–51, 2015.

[19] D. J. Mala, V. Mohan, and M. Kamalapriya, "Automated software test optimisation framework—an artificial bee colony optimisation-based approach," *Softw. IET*, vol. 4, no. 5, pp. 334–348, 2010.

[20] Y. Xiang, Y. Zhou, Z. Zheng, and M. Li, "Configuring software product lines by combining many-objective optimization and sat solvers," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 4, 2018, Art. no. 14.

[21] S. Law and I. Bate, "Achieving appropriate test coverage for reliable measurement-based timing analysis," in *Proc. 28th Eur. Conf. Real-Time Syst.*, 2016, pp. 189–199.

[22] S. Anand *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[23] J. C. King, "A new approach to program testing," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 228–233, 1975.

[24] M. Harman, "Software engineering meets evolutionary computation," *Computer*, vol. 44, no. 10, pp. 31–39, 2011.

[25] A. Bouchachia, "An immune genetic algorithm for software test data generation," in *Proc. Int. Conf. Hybrid Intell. Syst.*, 2007, pp. 84–89.

[26] L. Wang, Z. Yue, and H. Hou, "Genetic algorithms and its application in software test data generation," *J. Beijing Univ. Aeronaut. Astronaut.*, vol. 2, pp. 617–620, 1998.

[27] Y. Suresh and S. K. Rath, "A genetic algorithm based approach for test data generation in basis path testing," *Int. J. Soft Comput. Softw. Eng.*, vol. 3, no. 3, 2014, doi: 10.7321/jscse.v3.n3.49.

[28] N. Zhang, B. Wu, and X. Bao, "Automatic generation of test cases based on multi-population genetic algorithm," *Int. J. Multimedia Ubiquitous Eng.*, vol. 10, no. 6, pp. 113–122, 2015.

[29] D. J. Mala, M. Kamalapriya, R. Shobana, and V. Mohan, "A non-pheromone based intelligent swarm optimization technique in software test suite optimization," in *Proc. Int. Conf. Intell. Agent Multi-Agent Syst.*, 2009, pp. 1–5.

[30] M. R. Girgis, A. S. Ghiduk, and E. H. Abdelkawy, "Automatic data flow test paths generation using the genetical swarm optimization technique," *Int. J. Comput. Appl.*, vol. 116, no. 22, pp. 25–33, 2015.

[31] N. Nayak and D. P. Mohapatra, "Automatic test data generation for data flow testing using particle swarm optimization," in *Proc. Int. Conf. Contemporary Comput.*, 2010, pp. 1–12.

[32] H. Han, F. Liu, Z. Yang, and Z. Hao, "Automated test case generation based on differential evolution with relationship matrix for IFOGSIM toolkit," *IEEE Trans. Ind. Inform.*, vol. 14, no. 11, pp. 5005–5016, Nov. 2018.

[33] J. J. Liang, B.-Y. Qu, X. Mao, B. Niu, and D. Wang, "Differential evolution based on fitness Euclidean-distance ratio for multimodal optimization," *Neurocomputing*, vol. 137, pp. 252–260, 2014.

[34] W. Gong, Z. Cai, and D. Liang, "Adaptive ranking mutation operator based differential evolution for constrained optimization," *IEEE Trans. Cybern.*, vol. 45, no. 4, pp. 716–727, Apr. 2015.

[35] K. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*. Berlin, Germany: Springer, 2006.

[36] J. Brest, S. Greiner, B. Boskovic, M. Mernik, and V. Zumer, "Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems," *IEEE Trans. Evol. Comput.*, vol. 10, no. 6, pp. 646–657, Dec. 2006.

[37] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of Machine Learning*. Berlin, Germany: Springer, 2011, pp. 760–766.

[38] C. Yue, B. Qu, and J. Liang, "A multi-objective particle swarm optimizer using ring topology for solving multimodal multi-objective problems," *IEEE Trans. Evol. Comput.*, vol. 22, no. 5, pp. 805–817, Oct. 2018.

[39] R. Cheng and Y. Jin, "A competitive swarm optimizer for large scale optimization," *IEEE Trans. Cybern.*, vol. 45, no. 2, pp. 191–204, Feb. 2015.

[40] J. C. Lin and P. L. Yeh, "Automatic test data generation for path testing using gas," *Inf. Sci.*, vol. 131, no. 14, pp. 47–64, 2001.

[41] D. J. Mala, V. Mohan, and M. Kamalapriya, "Automated software test optimisation framework—An artificial bee colony optimisation-based approach," *Softw. IET*, vol. 4, no. 5, pp. 334–348, 2010.

[42] N. Mansour and M. Salame, "Data generation for path testing," *Softw. Qual. J.*, vol. 12, no. 2, pp. 121–136, 2004.

[43] R. G. Steel and J. H. Torrie, *Principle and Procedures of Statistic: A Biometrical Approach*. New York, NY, USA: McGraw-Hill, 1980.

Fangqing Liu received the B.E. degree in software engineering, in 2016, from South China University of Technology, Guangzhou, China, where he is currently working toward the Ph.D. degree in software engineering with the School of Software Engineering. His current research interests include automated software test case generation and evolutionary computation for software testing.

Han Huang received the B.Man. degree in information management and information system from the School of Mathematics, South China University of Technology (SCUT), Guangzhou, China, in 2003, and the Ph.D. degree in computer science from SCUT, Guangzhou, China, in 2008. He is currently a Professor with the School of Software Engineering, SCUT. His research interests include theoretical foundation and application of evolutionary computation and stochastic heuristics. He is a Senior Member of CCF.

Zhongming Yang received the Information Engineering degree from the Guangdong University of Technology, Guangzhou, China, in 1999, and the master's degree in software engineering from the Huazhong University of Science Technology, Wuhan, China, in 2008. He is currently an Associate Professor with the College of Computer Engineering Technical, Guangdong Institute of Science and Technology, Guangzhou, China. His research interests include intelligent algorithm, software engineering, and computer network.

Zhifeng Hao received the B.Sc. degree in mathematics from Sun Yatsen University, Guangzhou, China, in 1990, and the Ph.D. degree in mathematics from Nanjing University, Nanjing, China, in 1995. He is currently a Professor with the School of Computer, Guangdong University of Technology, Guangzhou, China, and the School of Mathematics and Big Date, Foshan University, Foshan, China. His current research interests include various aspects of algebra, machine learning, data mining, and evolutionary algorithms.

Jiangping Wang received the master's degree in system engineering from Xiamen University, Xiamen, China, in 1995, and the MBA degree from Zhongshan University, Guangzhou, China, in 2002. He is currently with Beiming Software Company Ltd., which is one of top 100 Software Enterprises in China. He is in charge of R&D of Smart City Solutions. He is an expert of software development, and cloud, and dig data.