Global Optimization for Combination Test Suite by Cluster Searching Algorithm

Hao Chen[1] Xiaoying Pan[1] Jiaze Sun[1]

Abstract The test suite generation is a key task for combinatorial testing of software. In order to generate high-quality testing data, a cluster searching driven global optimization mechanism is proposed. In this approach, a binary encoding mechanism is used to transform the combination test suite generating problem into a gene sequence optimization problem. Meanwhile, a novel global optimization algorithm, cluster searching algorithm (CSA), is developed to solve it. In this paper, the validity and rationality of problem transformation mechanism is verified, and the details of CSA are shown. The simulations illustrate the proposed mechanism is feasible. Moreover, it is a simpler and more efficient test suite generation approach for small-size combinatorial testing problems.

Key words Cluster searching algorithm (CSA), combination test, global optimization, test suite optimization

Citation Hao Chen, Xiaoying Pan, Jiaze Sun. Global optimization for combination test suite by cluster searching algorithm. *Acta Automatica Sinica*, 2017, **43**(9): 1625−1635

DOI 10.16383/j.aas.2017.e160214

1 Introduction

In order to test a software system completely, we should make the functional detection for all kinds of system element combinations. If a software system under test (SUT) has k elements and each element has v_i $(i = 1, 2, \ldots, k)$ different values, then the number of whole test cases, which could covered all kinds of system element combinations for this SUT, is $\sum_{i=1}^{k} v_i$. For example, Table I is a SUT with 3 testing elements, and the number of whole test cases for completely testing this SUT is $3^3 = 27$. Since $\sum_{i=1}^{k} v_i$ might be a huge number, the cost of entire operations for testing such SUT is likely to be tremendous. So, it is infeasible to make a complete testing for the SUT with large input space.

TABLE I
A SUT With Three Test Parameters

Value	A	B	C
0	a0	b0	c0
1	a1	b1	c1
2	a2	b2	c2

Fortunately, some researches show about 70 % software bugs are caused by the combinations between two elements, meanwhile, the correlations among three elements generated about 90 % software bugs [1]. That means, we can generate a smaller test case set, which just identifies the interactions among several system elements, such as pairwise or triple combinations, to capture most of bugs in a SUT [2], [3]. Therefore, the combinatorial testing could be a more effective technique for discovering interaction faults of SUT, whose key task is to generate a test suite as small as possible to cover all given system element combinations.

Manuscript received January 5, 2017; accepted March 9, 2017.
This work was supported by the National Natural Science Foundation of China (61203311, 61105064) and the Scientific Research Program of Shaanxi Provincial Education Department of China (2015JK 1672).
Recommended by Associate Editor Jun Fu.
1. School of Computer Science and Technology, Xi'an University of Post and Telecommunications, Xi'an 710121, China

In this study, we propose a novel test suite generation and global optimization approach. In the first step, the test suite generation problem has been transformed into a binary sequence optimization problem by a binary encoding mechanism. Based on it, a novel evolutionary algorithm, cluster searching algorithm (CSA), is presented to optimize the binary sequence in solution space so as to construct the optimal test suite. In Section 2, we analyze the related research works. Section 3 introduces the problem transformation. Section 4 shows the CSA. Section 5 provides the simulations. Section 6 gives the conclusion.

2 Related Work

Generally, a test case subset, which can satisfy all covering requirements, is known as a representative set. Assuming that the cost of generating and managing each test case is the same, a representative set with a minimum number of test cases is desirable and is called the optimal testing suite. The optimal combinatorial test suite generation problem is defined as, given a SUT and a set of combinatorial covering criterion, find a minimal representative set from the complete test case set. For example, Table II is an optimal pairwise combinations representative set, which can cover all combinations between any two system elements of the SUT in Table I. In general, this problem can be expressed as a set-covering problem [4], and it is a NP-complete problem also [5].

TABLE II
An Optimal Test Suite Covering All Pairwise Combinations

Number	A	B	C
1	a0	b0	c0
2	a0	b2	c2
3	a0	b1	c1
4	a1	b0	c2
5	a1	b1	c0
6	a1	b2	c1
7	a2	b0	c1
8	a2	b2	c0
9	a2	b1	c2

In traditional studies, people make use of some mathematical methods to construct the representative set, such as orthogonal array [6]. But, there are some unsolved problems existing in the orthogonal array generation method, such as we cannot generate a necessary orthogonal array for any kind of SUT. Another mathematical method is based on the matrix recursive construction process [7]. For some special instances, this method can give a wonderful result. But, it cannot be applied to all kinds of SUT either. Moreover, the theoretical researches of combinatorial testing are still unable to give an explicit optimal covering number for most problems. The major conclusions of such studies just can give some logical relations for the optimal covering number between different problems [8].

In recent years, the evolutionary algorithm (EA) has developed into a powerful global optimization algorithm to solve many complex engineering optimization problems [9] −[12]. In combinatorial testing researches, the EA is usually coupled with one-test-at-a-time mechanism [13]. In such studies, an EA has been used to search a best test case t_i in the test case complete set, which can cover the maximum combinations in uncovering combination set (CS), in one computation. Then, let t_i join test suite (TS) and delete the covered combinations by t_i in CS. After that, the above operations will repeat until all combinations in CS have been covered. Based on this iterative construction process, the one-test-at-a-time mechanism can solve most large-size SUT very well [14], [15]. However, to a small-size SUT, it does not show a desired performance. Firstly, it is likely to generate an approximate representative set. Secondly, it always takes a much longer time and more matrix transformations to generate whole representative set. These characteristics make one-test-at-a-time mechanism not suitable to solve small-size SUT. For example, if we use this mechanism to generate the TS for the SUT in Table I, the test cases a0b0c0, a1b1c1 and a2b2c2 are likely to be preferentially selected from complete set to join TS one by one, because both of them contain 3 uncovered pairwise combinations in CS, which conform to the optimal condition for the test case selection. However, if it does, no matter what test case in the left 24 cases has been chosen to join TS in the next computation cycle, the generated representative set would be an approximate solution, because there is at least one pair-wise combination repetitive with the 9 covered pair-wise combinations, which have been generated by the above 3 test cases.

Recently, efforts have been focused on the use of meta-heuristic algorithms as part of the computational approach for test suite generation. Generally, meta-heuristic algorithms start with a random set of test cases, such as using a simple one-test-at-a-time greedy procedure to construct it [16]. Then, the initial test case set undergoes a series of transformations in an attempt to improve itself. Each of them could be an independent meta-heuristic algorithm, such as teaching learning based optimization, global neighborhood algorithm, particle swarm optimization, and cuckoo search algorithm [17]. In this process, one best candidate is selected at each iteration until all the required interactions are covered. Such researches show a good performance for large-size constrained CA problems as well. But, these algorithms always contain several searching algorithms and try to balance them in the iteration computation process. So, these approaches often have a relatively heavy and complicated structure and are more suitable to solve large-size CA problems.

By using one-test-at-a-time mechanism or meta-heuristic algorithms, it is possible to perform large combinatorial testing that was not possible before. But, for small-size CA problem, such approaches still imply a high cost and often get an approximate result. In the practical applications, a portion of software testing works belongs to the small-size problem. So, finding a simple method to improve the probability of generating the optimal test suite for small-size combinatorial testing is a worthy goal. In order to optimize the test suite more effectively, some researchers try to translate small-size CA problem into another kind of problem to solve it, such as satisfiability problem (SAT) [18] and integer program problem [19] etc. However, these researches also meet some difficult challenges. For example, even translating a small-size SUT into a SAT problem, we will get a very large clause set. Besides, the normal planning algorithm cannot deal with a big optimization problem efficiently. Above researches enlighten us that the combination test suite generation problem should be translated into a simple and concise data structure for global optimization, meanwhile, it is necessary to equip an effective global optimization algorithm to improve the solution quality.

3 Proposed Method

Fig. 1 is the flowchart of combinatorial test suite global optimization mechanism. As it shows, a one-to-one correspondence is created between a test case in its complete set and a gene of a binary code string. Based on this, we can create a mapping relation between a test case subset of complete set and a vertex in the binary code space. This means we can translate a combination test suite generation problem into a binary code based global optimization problem to solve. The binary string is a simpler and more compact data structure. Moreover, it is more suitable for global optimization computation. In this section, we will introduce the encoding and decoding procedure firstly.

3.1 Binary Code Based Problem Transformation

Generally, we can use a covering array (CA) or a mixed level covering array (MCA) to describe a SUT [4]. The difference between CA and MCA is that each test parameter of CA has the same value range, but in MCA it can be different. Actually the CA can be looked upon as a special case of MCA, and the processing methods for them have no difference. To facilitate the computation process, we make this study for CA problem only. The following is the definition of covering array given by Cohen [20].

Definition 1 (Covering array): Let N, k, t, and v be positive integers. A covering array, $CA(N; t, k, v)$, is a $N \times k$ array on v symbols, and every $N \times t$ sub-array contains all ordered subsets from v symbols of size t at least once. In such an array, t is called strength, k is called the degree, v is called the order.

By this definition, we can get that the number of test cases in the complete set of $CA(N; t, k, v)$ is v^k, and the number of whole combinations with covering strength t is

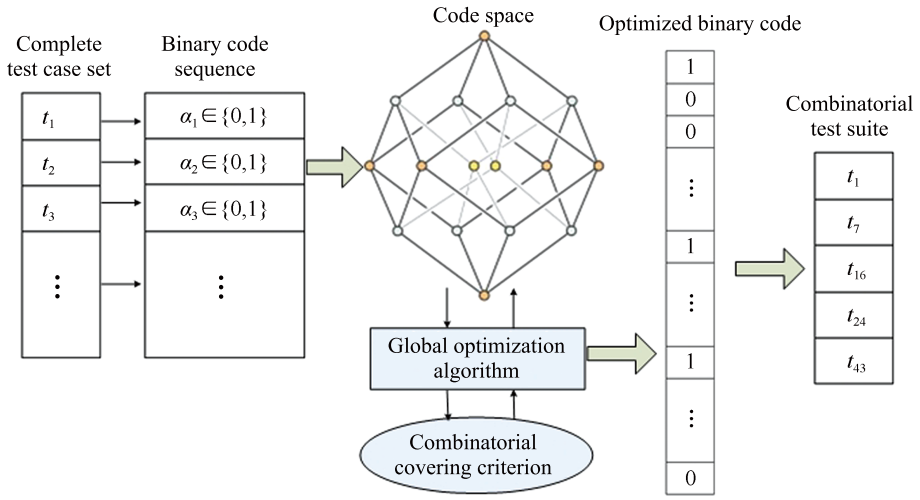


Fig. 1. Combinatorial test data global optimization mechanism.

$v^t C_k^t$. For describing the problem transformation mechanism, the other definitions are given in the following.

Definition 2: Let Φ be the complete test case set of $CA(N; t, k, v)$, then $|\Phi| = v^k$. Sort Φ in ascending order by the value of each test parameter. The id j of test case t_j is labeled from 0 to $|\Phi| - 1$.

By this definition, the complete test case set Φ will show an orderly structure. Furthermore, we can use the id j to visit a definite test case t_j in Φ. For example, Table III is the complete test case set of the SUT in Table I.

TABLE III
An Orderly Complete Test Cases Set

A	B	C	t_j	id j
a0	b0	c0	t_0	0
a0	b0	c1	t_1	1
a0	b0	c2	t_2	2
a0	b1	c0	t_3	3
⋮	⋮	⋮	⋮	⋮
a2	b2	c2	t_{26}	26

Definition 3: The binary code sequence I is a string of length L. Its genes $\alpha_j\{0, 1\}$, $j = 0, 1, \ldots L - 1$.

As Fig. 1 shows, the binary code space is an L-dimensional hypercube. The vertex in this hypercube is a unique binary coding string.

Definition 4: Let γ_i be a test cases subset of Φ and $\Gamma = \{\gamma_i | \gamma_i \in \Phi\}$. Then, Γ is the complete set of Φ's subset, and for $\forall\gamma_i$ and $\forall\gamma_j$ ($\gamma_i, \gamma_j \in \Gamma$ and $i \neq j$), $\gamma_i \neq \gamma_j$. Let λ_i be a vertex in binary code space and $\Lambda = \{\lambda_0, \lambda_1, \ldots\}$. Then, Γ is the complete set of vertex in binary code space.

Based on above definitions we can get 2 characteristic theorems of set Γ and set Λ firstly.

Theorem 1: Let $L = |\Phi|$, then the element number of set Γ is equal to set Λ's. That is $|\Gamma| = |\Lambda| = 2^L$.

Proof: By Definition 3, we can get there are 2^L unique vertexes in the L-dimensional hypercube. By Definition 1, we know there are $|\Phi| = v^k$ distinct test cases in Φ. Meanwhile, for a test case, there are also two statuses, it belong to the test case subset or not. That means the element number of Γ is $|\Gamma| = 2^{|\Phi|}$. So, if $L = |\Phi|$, we can get $|\Gamma| = |\Lambda| = 2^L$. ■

Theorem 2: Make test case t_j join the TS only when the gene $\alpha_j = 1$ ($j = 0, 1, \ldots, L - 1$). Then, both $\Gamma \to \Lambda$ and $\Lambda \to \Gamma$ are bijections.

Proof: Firstly, we can prove both $\Gamma \to \Lambda$ and $\Lambda \to \Gamma$ are surjections based on above assumption. By Theorem 1 we know the set Γ and set Λ have same number of elements when $L = |\Phi|$. Besides, both the test case in Φ and the gene in a binary string have two statuses, which is yes or no and 1 or 0. Meanwhile, the t_j has corresponded to one and only α_j by the same id j. So, under the given condition, each element in set Γ would correspond to one and only object in set Λ, and each element in set Λ is mapped by a unique object in set Γ, and vice versa. Based on this, we can get both $\Gamma \to \Lambda$ and $\Lambda \to \Gamma$ are surjections. On the other hand, by the Theorem 1, we can get all elements in set Γ and set Λ is a unique object. That means each element in set Γ and set Λ is different from others. Therefore, both $\Gamma \to \Lambda$ and $\Lambda \to \Gamma$ are injections. For $\Gamma \to \Lambda$ and $\Lambda \to \Gamma$ are both surjection and injection, we can get that both $\Gamma \to \Lambda$ and $\Lambda \to \Gamma$ are bijections. ■

The bijection relationships between set Γ and set Λ show there is a one-to-one correspondence between the test case subset and the vertex in binary code space, which is a necessary condition for the problem transformation.

3.2 Decoding Mechanism

The decoding mechanism is used to parse the binary code sequence so as to construct the corresponding combinatorial test case subset. To facilitate this process, we use a positive integer, whose value is set from 0 to $v - 1$, to mark the symbol of each parameter of SUT. Then, we can use a mod-v number to present the details of a test parameter.

Definition 5: For a covering array $CA(N; t, k, v)$, a test case can be expressed as a k figures integer sequence, and each figure is a positive mod-v integer.

Let x_i ($x_i \in \{0, 1, \ldots, v - 1\}$) express the value of ith parameter ($i \in \{0, 1, \ldots, k - 1\}$) of test case t_j. Then, an equation can be created to show the details of test case t_j after parsing the number id j. The formula is

$$j = x_{k-1} * v^{k-1} + \cdots + x_1 * v^1 + x_0 * v^0 \quad (1)$$

For example, to the CA in Table I, if the gene value of α_3 in I_i is 1, that means $j = 3$. Then, we can create an equation $3 = 0 \times 3^2 + 1 \times 3^1 + 0 \times 3^0$. Based on this equation, we can get an integer sequence, which is 010. After that, this integer sequence can be translated into a test case t_3 = a0b1c0. Repeating this process for each valuable gene in I_i, whose value is 1, we can generate a whole test suite. This procedure is shown as follows.

Algorithm 1 Decoding

Input: positive integer k and v, binary code of I_i

Output: test suite TS

```
for j from 0 to L − 1 do
  if α_j equal to 1 then
    dnum = j;
    for m from 0 to k − 1 do
      x = dnum %v; dnum/ = v;
      t_j[m] = x;
    end for
    put t_j into TS;
  end if
end for
```

In above procedure, TS $= \{t_j | j = 0, 1, \ldots, L-1\}$, $t_j = x_{k-1}, \ldots, x_1, x_0$ where x_i is a mod-v integer value. Firstly, we set TS $= \phi$.

3.3 Fitness Calculation and Constraint Handling

In the proposed approach, the CA problem can be regarded as a sort of optimization problem. So, we can formulate λ_i's fitness as

$$\max f(\lambda_i), \quad \lambda_i \in \Lambda. \quad (2)$$

By Theorem 2, we know $\lambda_i \rightarrow \gamma_i$ is a one to one mapping. So, we can give the following definition

Definition 6: the fitness of λ_i and γ_i is

$$f(\lambda_i) = f(\gamma_i) = f_i. \quad (3)$$

Since we expect to use as few test cases as possible to cover whole combinations with strength-t, the fitness of λ_i can be evaluated from two aspects. The first one is the covering degree for strength-t combination in CS. The second one is the number of test cases in λ_i. In order to balance the two sides, we propose to use following formula to calculate f_i

$$f_i = 10 * \frac{\omega}{|CS|} + 1 - \frac{|TS|}{|\Phi|} \quad (4)$$

where ω is the number of covered combinations in CS and $|CS|$ is the number of all strength-t combinations. Obviously, $0 < \omega \leq |CS|$ and $0 < |TS| \leq |\Phi|$. Based on our experience, $10 * \omega/|CS|$, which is between 0 and 10, is used to judge the coverage of combinations set. Meanwhile, $1 - |TS|/|\Phi|$, which is between 0 and 1, is used to show the test suite size relationship.

Besides, for handing constraints, a specific calculation mechanism is used to filter out all invalid genes from λ_i before fitness evaluating.

Definition 7: The constraint set $CO = \{co_i | i = 1, 2, \ldots\}$. If the constraint co_i is available, we can generate a corresponding binary string $I_i = \{\alpha_0 \alpha_1 \cdots \alpha_{L-1}\}$, in which the $\alpha_j = 0$ if its corresponding test case t_j is infeasible to co_i; otherwise, $\alpha_j = 1$. Based on $I_1, I_2, \ldots$, we can generate an invalid genes filtering string I_{co}

$$I_{co} = I_1 \wedge I_2 \wedge \cdots \quad (5)$$

where $\wedge$ is the logic and operation.

Using I_{co} we can filter out all invalid genes in λ_i easily by following formula

$$\lambda_i = \lambda_i \wedge I_{co}. \quad (6)$$

After this calculation process, the fitness of λ_i would be calculated by (4).

Based on fitness function, the value of λ_i and γ_i can be evaluated. Moreover, both of them have the same fitness value. Then, if we sort the set Γ and set Λ based on its individuals fitness, we can see the corresponding individuals between two sets also have the same position in each fitness sequence. So, we can get Theorem 3.

Theorem 3: The γ_i and λ_i have the same relative position in its fitness ordering sequence of set Γ and set Λ.

This theorem gives a sufficient condition for the problem transformation. Above all, we can get a conclusion that a combination test suite generation problem can be translated into a binary code based global optimization problem to solve.

4 Cluster Searching Algorithm

As we know, the solution of EA is much likely to be affected by the phenomena of premature convergence and searching stagnation. The adaptive mechanism is the most commonly used self-adjusting method in EA. Nevertheless, in a practical application, it is difficult to make a reliable and accurate self-adjusting strategy for EA. So, the adaptive mechanism just can make a limited impact on the performance improvement of EA. Recently, a new research trend is aimed at creating a hybrid algorithm model by merging various searching mechanisms and operators in order to improve the performance of optimization system [21], [22]. Meanwhile, a new problem has emerged. That is how to balance the diversity and the complexity in a hybrid evolution system and harmonize the specificity and the coordination among multiple operators. We find that an organization with sort of cluster form structure in a complicated and huge evolutionary group has shown a particularly important role in the occurrence and development process of such group. Inspired by this, we propose to create a certain cluster organization in the population, and use it to control and adjust the searching process of population. Fig. 2 is the model of cluster searching algorithm (CSA).

In CSA, the cluster C is a connection relationship among individuals, which can be generated by a clustering process. Such structure makes the population searching procedure divided into two parts, which are the searching

among group process and the searching inside group process. By the interactions among multiple clusters, the searching among group process is developed to explore the code space. In contrast with it, the searching inside group process is aimed at refining the individuals in each group. Such job-division mechanism is not only helpful keeping low coupling between the global and local searching computation in evolution system, but also provides an environment and foundation for merging various searching mechanisms and operators, and adjusting the interrelation between the global and local searching process. For describing CSA, we give some basic definitions firstly.

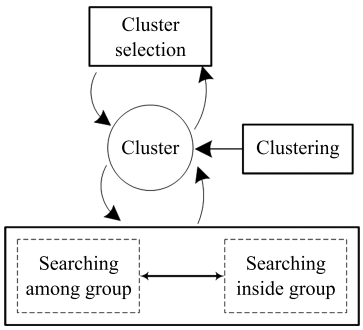


Fig. 2. Model of cluster searching algorithm.

Definition 8: A population *Pop* of CSA consists of an n-tuple of strings I_i $(i = 0, 1, \ldots, n-1)$ of length L, where the genes $\alpha_j \in \{0, 1\}$, $j = 0, 1, \ldots, L-1$. I_i is called an individual. The fitness of I_i is $f_i > 0$.

Definition 9: The cluster C is a virtual group organization and the A_i^j is a member of group C_i, which maps an individual I_k in *Pop*. We can express it as

$$C = \{C_1, C_2, \ldots\}, \quad C_i = \{A_i^1, A_i^2, \ldots\}$$
$$A_i^j = k, \quad i, j, k = 1, 2, \ldots \tag{7}$$
$$\emptyset = \{C_i \cap C_j | i \neq j\}, \quad \emptyset = \{A_i^r \cap A_j^s | i \neq j\}$$
$$Pop = \cup C_i, \quad i, j, r, s = 1, 2, \ldots. \tag{8}$$

Definition 10: In cluster C, the members of the same group have more similarity, and the members belonging to different groups have more diversity. That is

$$Distance(A_i^r, A_j^q) > Distance(A_i^r, A_i^s),$$
$$i \neq j, \quad i, j, r, s, q = 1, 2, \ldots. \tag{9}$$

Definition 11: The core member A_i^c of C_i is a member who has the best fitness in group C_i. That is

$$f_{A_i^r} \leq f_{A_i^c} \quad \forall A_i^r \in C_i, \ i, r = 1, 2, \ldots. \tag{10}$$

In the iteration process of CSA, the cluster, created in kth generation, is written as C^k, and the children, generated by the searching process of C^k, are Ch^k. The following are the main steps of CSA.

1. $C^0 = initializing(Pop^0)$;
2. **While** (the termination criteria are not reached) **do**
3. $Ch^k = searchingamonggroup(Pop^{k-1}, C^{k-1})$;
4. $C^k = clustering(Pop^{k-1}, Ch^k)$;
5. $Ch^k = searchinginsidegroup(Pop^{k-1}, Ch^k, C^k)$;
6. $C^0 = initializing(Pop^0)$;
7. **End while**.

In the first step, CSA initializes the population Pop^0 in binary code space randomly. After that each I_i in Pop^0 is set to be a group C_i $(i = 1, 2, \ldots, |n|)$. And it is the only member A_i^1 and center member A_i^c of C_i also. Then, the cluster C^0 is created. The following is the iteration searching process of CSA.

4.1 Searching Among Group Process

In the searching among group process, we choose an individual from different groups and use multi-point crossover operation $\oplus$ and mutation operation $\odot$ to explore the code space. This step will generate n offspring individuals.The number of crossover point ξ in $\oplus$ operation is randomly generated between a and b. The default parameters a and b satisfy $1 < a < b < L/2$. In $\odot$ operation, the gene mutation probability is $\rho \in (0, 0.05)$.

Algorithm 2 Searching among group

Input: population Pop^{k-1} and cluster C^{k-1}

Output: n offspring individuals in Ch^k

1: Randomly select two different groups C_i and C_j $(i \neq j)$ from current cluster C^{k-1}, then randomly select member A_i^r and A_j^s from C_i and C_j respectively.

2: Perform $\oplus$ operation between A_i^r and A_j^s to generate binary string ch_1 and ch_2. Then, perform $\odot$ operation on ch_1 and ch_2, respectively. That is

$$ch_1, ch_2 = \odot(A_i^r \oplus A_j^s). \tag{11}$$

3: The number of offspring individuals $p = p + 2$. If $p < n$, return to step 1. Otherwise, exit.

4.2 Clustering Process

After the searching among group process, $2n$ individuals will be dispersed in code space widely. Then, we use a clustering process to assign them into different groups so that they have a high degree of similarity within the group, and that the cluster is to be distinct. This process helps CSA to analyze the spatial distribution structure of population in code space. This clustering process also consists of two parts: a technique for calculating the distance between binary strings, and a grouping technique to minimize the distance between individuals in same group while maximizing the distance among groups.

4.2.1 Distance Calculation by PAD

Based on the bound researches of optimal test case number, such as symbol-fusing and the lower and upper bound [23], [8], we can see that the valuable genes, whose value is 1, just have occupied a small proportion in binary code. In order to emphasize the importance of these valuable genes in clustering process, we propose to use positive attribute distance (PAD) [24] instead of traditional hamming distance to calculate the individual distance.

The PAD of two binary sequences is as follows

$$PAD(I_i, I_j) = 0 \leq \frac{2\Psi_{ij}}{\Psi_i + \Psi_j} \tag{12}$$

where Ψ_i is the number of 1's in ith binary sequence, Ψ_j is the number of 1's in jth binary sequences, and Ψ_{ij} is the number of 1's common to both ith and jth binary sequences. Obviously, the result of PAD is in the interval between 0 and 1, where 1 expresses absolute similarity and 0 expresses absolute diversity.

Furthermore, we use the average PAD of population as the indicator of population diversity $\gamma \in (0,1)$. It can be calculated by following formula

$$\gamma = \frac{2}{|n|(|n|-1)} \sum_{i=2}^{|n|} \sum_{j=1}^{i-1} PAD(I_i, I_j). \tag{13}$$

4.2.2 Clustering Based on PAD

Algorithm 3 is the main steps of a hierarchical clustering process. In this process, we set two matrixes, IDM and GDM, to store the individual distance $idm_{ij} \in IDM$ and group distance $gdm_{ij} \in GDM$. That is

$$idm_{ij} = \begin{cases} 0, & \text{if } i \le j, \ \ i,j = 1,2,\ldots \\ PAD(I_i, I_j), & \text{else} \end{cases} \tag{14}$$

$$gdm_{ij} = \begin{cases} 0, & \text{if } i \le j, \\ & \quad i,j,r,s = 1,2,\ldots \\ \dfrac{\sum\limits_{A_i^r \in C_i} \sum\limits_{A_j^s \in C_j} PAD(A_i^r, A_j^s)}{|C_i||C_j|}, & \text{else.} \end{cases} \tag{15}$$

Meanwhile, the current population Pop^{k-1} and its offspring individuals Ch^k have been mixed together to join clustering process.

Algorithm 3 Searching among group

Input: population Pop^{k-1} and offspring individuals Ch^k

Output: new cluster C^k

1: Set each I_i in population to be a temporary group C_i. Initialize the IDM by (14). After that initialize the GDM by (15) based on IDM. Then, calculate the population diversity γ by (13), and set the average group distance $\xi = \gamma$.

2: Find the minimum group distance gdm_{pq} in GDM and merge the group C_p and C_q into a new group C_{pq}. Then, update the GDM.

3: Calculate new average group distance ξ'

$$\xi' = \frac{2}{|C|(|C|-1)} \sum_{i=2}^{|C|} \sum_{j=1}^{i-1} gdm_{ij} \tag{16}$$

where $|C|$ is the number of groups after step 2. If $\xi' < \xi$, go to step 4; otherwise, $\xi = \xi'$ and return step 2.

4: Restore the group C_p, C_q and GDM. After that, update the center member for each group and output C.

4.3 Searching Inside Group Process

The searching inside group process consists of two operations, SIG1 and SIG2, which are used to reduce the valuable genes in binary string while maintaining it can satisfy all covering requirements. In this process, a parameter m is used to limit the execution times of SIG1 and SIG2 for each group. In this paper, the m is adjusted by the following formula

$$m = \text{round}\left(\frac{m_2 - m_1}{G-1} g + \frac{m_1 G - m_2}{G-1}\right) \tag{17}$$

where $0 \le m1 < m2$, $g = 1,2,\ldots,G$ is the generation number, G is its ceiling number and round$(\cdot)$ is the rounding operation. During the searching process, (17) makes m increase from m_1 to m_2 gradually. Besides, parameter $\lambda \in (0,1)$ is used to adjust the operation probability between SIG1 and SIG2.

Algorithm 4 Searching inside group

Input: Pop^{k-1}, Ch^k, C^k and λ

Output: $m|C|$ offspring individuals in Ch^k

1: Run this operation for C_i, $i = 1,2,\ldots,|C|$, and set $t = 0$.

2: $UR(0,1)$ is a uniformly distributed random number between 0 and 1. If $UR(0,1) < \lambda$, goto step 2.1 to execute SIG1. Otherwise, goto step 2.2 to execute SIG2.

2.1: SIG1: select two members A_i^r and A_i^s from C_i randomly, and execute logic and operation $\wedge$ between them. After that, perform multi-point crossover operation $\oplus$ between the output string by $A_i^r \wedge A_i^s$ and the center members A_i^c. This process can be expressed as

$$ch_1, ch_2 = A_i^c \oplus (A_i^r \wedge A_i^s). \tag{18}$$

After that, put the generating individuals ch_1 and ch_2 into C_i, $t = t+2$ and goto step 3.

2.2: SIG2: Randomly select three members A_i^r, A_i^s and A_i^t from C_i. Then, perform logic and operation $\wedge$ among them and output a binary string ch

$$ch = A_i^r \wedge A_i^s \wedge A_i^t. \tag{19}$$

If ch can cover all strength-t combinations in CS, goto step 2.2.1. Otherwise, goto step 2.2.2.

2.2.1: Randomly select a valuable gene in ch, whose value is 1, and change it into 0. If the updated ch still can cover all strength-t combinations in CS, repeat this step. Otherwise, recover the value of last chosen gene to 1 and goto step 2.2.3.

2.2.2: Randomly select a gene in ch, whose value is 0, and change its value into 1. Then repeat this step until ch has covered all strength-t combination in CS and goto step 2.2.3.

2.2.3: Make the generating individuals ch join C_i, set $t = t+1$ and goto step 3.

3: If $t < m$, update the A_i^c of C_i and return to step 2. Otherwise, set $i = i+1$ and return to step 1.

Fig. 3 shows the practical calculation process of

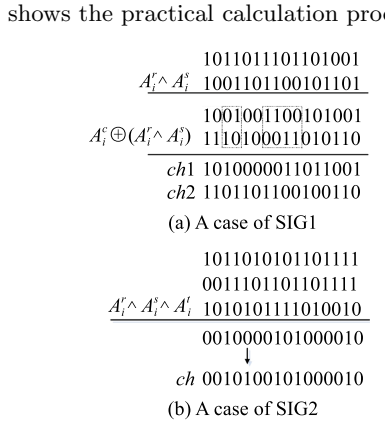


Fig. 3. Calculation cases of searching inside group process.

searching inside group. In Fig. 3 (a), two binary strings, ch_1 and ch_2, are generated by (18). In Fig. 3 (b), a temporary individual ch is generated by (19). Then, one gene of ch is selected to change its value from 0 to 1, and a solution is gotten.

4.4 Cluster Selection Process

After the cluster searching process, the scale of current population increases to $2n + m|C|$. In order to satisfy the computation requirement for next generation, we need to select n individuals from current population to form next population. In cluster selection process, n individuals will be selected from each group respectively.

Algorithm 5 Cluster selection

Input: Pop^{k-1}, Ch^k, C^k

Output: Pop^k

1: For each C_i, sort its members in descending order by their fitness. After that, set $i = 1$, $k = 1$, $j = 1$.

2: Take member A_i^k from C_i to be a surviving individual I_j and join next population. It can be expressed as

$$I_j = A_i^k, \quad j = 1, 2, \ldots, n$$
$$i = j\%|C| + 1, \quad k = \text{round}\left(\frac{j}{|C|}\right) + 1. \tag{20}$$

3: If $j < n$, $j = j + 1$ and return to step 2; otherwise, exit.

5 Simulation Experiments

In this section, we implemented 3 algorithms, a real code genetic algorithm with one-test-at-a-time mechanism GA/OTAT, a binary code GA with global optimization approach GA/BCGO, and CSA, in C++. Meanwhile, 32 CA problems are chosen to test above 3 algorithms and TCA [16], whose source code is implemented in C++, and available online[1], on a 2.1 GHz AMD Phenom PC with 2 GB memory.

5.1 Experimental Settings

An individual $x_k, \ldots, x_2, x_1$ in GA/OTAT is a k-dimensional real vector. The gene value of each dimension is initialized between 0 and 1 randomly. In decoding process, the gene value of each dimension in an individual will be translated into an integer by the formula round $(x_i \cdot v)$. According to this integer, we can get the corresponding parameter symbol in the SUT. For example, in an individual, if the gene $x_1 = 0.6$, we can get $1 = \text{round}(0.6 \cdot 3)$. For the CA in Table I, the integer 1 means $x_1 \to a_1$. Besides, the population size n of GA/OTAT is 100. The searching operations of GA/OTAT include algebraic crossover, non-uniform mutation and linear sort selection. The following is mainly the computation procedure of GA/OTAT.

1. Initialize test suite TS = ϕ;
2. Initialize combination set CS;
3. **While** (CS $\neq \phi$) **do**
4. Initialize population of GA/OTAT randomly;
5. **While** (the termination criteria are not reached) **do**
6. search a best test case t_i;
7. **End while**;
8. Make t_i join TS;
9. Delete the covered combinations in CS;
10. **End while**;
11. Output TS.

Both GA/BCGO and CSA use (4) to verify the quality of test suite. Besides, based on our experience, we set population size $n = 60$ in CSA. Furthermore, the GA/BCGO and CSA use the same maximum running iterations in the following trials and the CSA will costs more operation number than GA/BCGO in each searching iteration process. To be fair, we set a big population size for GA/BCGO, which is 100, to make its searching operations number is not less than CSA in each iteration process. Besides, GA/BCGO and CSA use same crossover and mutation operation. In multi-point crossover operation, the parameter a and b, which are used to set the lower and upper limitation of crossover point, are set to $a = \text{round}(L/10)$ and $b = \text{round}(L/4)$. The gene mutation probability is $\rho = 0.01$. But, GA/BCGO uses the linear sort selection operation to generate next population. Besides, the parameter m_1 and m_2 of CSA, which are used to limit the computation times m in searching inside process, are set to $m_1 = \text{round}(n/10)$ and $m_2 = \text{round}(n/4)$. The parameter λ, which is used to control the execute probability of SIG1 and SIG2, is set to $\lambda = 0.4$. The computation procedure of CSA and GA/BCGO is

1. Encode the binary code space;
2. Initialize population of CSA or GA/BCGO;
3. **While** (the termination criteria are not reached) **do**
4. search the best genetic string I_i in code space;
5. **End while**;
6. Decode I_i and output TS.

In the beginning of TCA, the initialization step is called to construct a CA set to cover all valid t-tuples, which works with a simple one-test-at-a-time greedy strategy. After that, TCA executes the search steps to adjust the CA set until the time budget is reached. During the search process, TCA switches between two modes, that is, the greedy mode and the random mode. With a probability $p = 0.001$, TCA works in the random mode; otherwise (with a probability 1-p), TCA works in the greedy mode. In each run of testing, we will set a cutoff time to TCA in advance.

5.2 Results and Discussions

We ran GA/OTAT, GA/BCGO, CSA and TCA for 32 CA problems over 30 independent trials. The experiment results of above 4 algorithms are shown in Table IV and Table V with the experiment results of HHH [17] and integer program method [19] for part of testing problems.

In first round of experiments, 16 CA problems with strength-2 have been tested. In the first 8 tests, we set $v = 2$ and $k = 3, 4, 5, \ldots, 12$, respectively. The maximum running iterations of GA/OTAT, GA/BCGO and CSA are set to 200, 500 and 1000 when k is less than or equal to 6, 9 and 12, respectively. In the next 5 tests, $v = 3$ and $k =$ 4, 5, ..., 8, respectively. The maximum running iterations

[1] https://github.com/leiatpku/TCA

TABLE IV
Experiment Statistical Results of 3 Algorithms for 16 CA Problems With Strength-2

v	k	$\|\Phi\|$	N	GA/OTAT $\|TS\|$ Best	GA/OTAT $\|TS\|$ Ave.	GA/OTAT Time (s)	GA/BCGO $\|TS\|$ Best	GA/BCGO $\|TS\|$ Ave.	GA/BCGO Time (s)	CSA $\|TS\|$ Best	CSA $\|TS\|$ Ave.	CSA Time (s)	TCA $\|TS\|$ Best	TCA $\|TS\|$ Ave.	HHH $\|TS\|$ Best	HHH $\|TS\|$ Ave.	Integer program $\|TS\|$ Best	Integer program $\|TS\|$ Ave.	Integer program Time (s)
2	5	32	6	7	7.7	5.97	6	6	0.27	6	6	0.36	6	6.5	-	-	6	-	0.70
	6	64	-	7	8.1	6.54	6	6.2	0.68	6	6	1.10	6	6.7	-	-	6	-	16.57
	7	128	-	7	8.4	10.89	6	9.4	1.43	6	6	1.99	6	7.3	-	-	6	-	441.2
	8	256	-	8	10.7	12.71	6	11.3	3.91	6	6	4.70	6	7.0	-	-	-	-	-
	9	512	-	8	11.2	14.5	40	51.6	7.49	6	6	9.60	7	7.9	-	-	-	-	-
	10	1024	-	8	12.9	25.3	60	70.8	19.5	6	8.4	23.1	7	8.2	-	-	-	-	-
	11	2048	7	9	14.2	28.8	141	167	40.3	7	9.5	49.0	8	8.7	-	-	-	-	-
	12	4096	-	9	16.7	32.9	196	228	112.4	20	33.9	145	8	8.8	-	-	-	-	-
3	4	81	9	9	10.6	8.8	9	9	1.29	9	9	1.70	9	9.6	9	9	9	-	0.08
	5	243	11	11	12.8	17.2	11	17.2	3.73	11	11	5.31	11	11.3	11	11.35	13	-	$*$
	6	729	12	16	18.1	19.6	44	57.8	11.5	12	12	14.9	13	14.1	13	14.2	-	-	-
	7	2187	-	19	21.7	36.9	157	165	39.1	12	16.9	50.2	13	14.5	14	15	-	-	-
	8	6561	13	20	27.9	40.3	224	277	133	88	97.5	148	14	14.7	15	15.6	-	-	-
4	5	1024	16	21	29.5	68.1	56	71.5	22.4	16	16.9	25.6	18	19.8	-	-	-	-	-
	6	4096	19	25	36.7	74.4	197	227	194	22	34.2	223	21	22.7	-	-	-	-	-
	7	16384	21	28	40.9	93	320	342	417	116	135	508	24	25.2	-	-	-	-	-

TABLE V
Experiment Statistical Results of Test Case Number for 16 CA Problems With Strength-3

v	k	$\|\Phi\|$	N	GA/OTAT $\|TS\|$ Best	GA/OTAT $\|TS\|$ Ave.	GA/OTAT Time (s)	GA/BCGO $\|TS\|$ Best	GA/BCGO $\|TS\|$ Ave.	GA/BCGO Time (s)	CSA $\|TS\|$ Best	CSA $\|TS\|$ Ave.	CSA Time (s)	TCA $\|TS\|$ Best	TCA $\|TS\|$ Ave.	HHH $\|TS\|$ Best	HHH $\|TS\|$ Ave.
2	5	32	10	11	13.6	11.1	10	11.2	0.41	10	10	0.42	10	10.6	-	-
	6	64	-	13	14.2	17.6	10	12.6	1.29	10	10	1.34	10	10.7	-	-
	7	128	-	14	15.6	24.4	12	15.1	2.09	10	10	2.15	10	11.2	-	-
	8	256	-	17	19.8	31.8	15	19.6	5.22	10	10	5.62	11	12.0	-	-
	9	512	-	19	21.9	41.0	17	23.9	9.85	10	10	11.0	11	11.9	-	-
	10	1024	-	22	25.2	52.3	41	54.7	25.9	10	11.8	25.3	13	15.8	-	-
	11	2048	12	24	27.7	69.9	80	92.8	56.8	12	19.4	55.5	14	15.5	-	-
	12	4096	15	28	30.9	82.1	239	257	131	69	85.7	133	17	18.1	-	-
3	4	81	27	29	34.2	68.1	27	27	1.62	27	27	1.91	27	28.9	27	29.45
	5	243	-	30	36.1	71.3	27	34.2	3.93	27	27	5.53	29	31.6	39	41.25
	6	729	33	36	42.5	80.5	55	62.9	15.5	33	35.5	15.4	34	36.1	33	39
	7	2187	39	48	54.4	89.9	195	217	87.9	49	54.9	93.2	46	47.7	49	50.8
	8	6561	42	54	58.9	99.6	251	275	145	156	170	157	51	52.2	52	53.65
4	5	1024	64	72	81	148	60	78.2	22.7	64	67.9	27.6	66	68.7	-	-
	6	4096	88	93	105	198	229	249	212	194	216	231	94	96.6	-	-
	7	16384	-	99	117	153	370	379	499	233	247	527	96	98.4	-	-

of 3 algorithms are set to 200, 500 and 1000 when k is less than or equal to 4, 6 and 8, respectively. In the last 3 tests, $v = 4$ and $k = 5$, 6, 7, respectively. The maximum running iterations of 3 algorithms are set to 1000 and 2000 when k is less than or equal to 6 and 7, respectively. The experiment statistical results of above 16 tests are shown in Table IV. In second round of experiments, the covering criterion of above 16 CA problems is changed to strength-3. Meanwhile, the parameters of v, k and maximum running iterations of above 16 CA problems have the same value as the first round experiments. Table V is the experiment statistical results of these 16 CA problems. For a fair comparison, the cutoff time of TCA for each CA problem is set to an integer number, which is gotten by rounding up CSAs running time. HHH does not provide its running time and just shows the best and average number of $|TS|$ for $CA(N; t, 3^k)$ where $4 \leq k \leq 8$ with strength-2 and strength-3. The integer program method just provides the results of 8 small-size CA problems with strength-2. The $*$ in Table IV indicates the best possible solution found at the

point when the process was stopped after running for about 6.5 hours. Besides, the number N is the known minimal number of $|TS|$ for 36 CA problems, which can be found in Colbourns website[2].

In Table IV, we can see the CSA has gotten 11 best solutions of 16 CA problems and GA/BCGO has gotten 6 best solutions, but GA/OTAT just gets an approximate solution for each 16 problems. In Table V, the experiment results show a similar formation as Table IV. With a holistic view, GA/BCGO can get the optimal solution with a high probability when the problems scale $|\Phi|$ is less than 250. It is worth noting that CSA has improved the performance of global optimization mechanism immensely. It can find the optimal solution with a high probability when the problems scale $|\Phi|$ is less than 2000. Instead, GA/OTAT just can get the approximate solution even the $|\Phi|$ of CA problem is very small. However, when the $|\Phi|$ of CA problem is more than 4000, the solution qualities and average CPU times of GA/OTAT begin to transcend CSA distinctly. Moreover, even the $|\Phi|$ is more than 16 000, the solutions and average CPU times of GA/OTOA still keep an acceptable approximate result.

Comparing the experiment results between GA/OTAT, TCA and HHH, we can see the meta-heuristic algorithms improve the average quality of $|TS|$ remarkably relative to the traditional one-test-at-a-time algorithm for its search step can adjust the generated CA set dynamically. However, the heuristic algorithms should cost lots of array transformation operations to adjust the CA set and combinations set in its greedy and random search processes, which make its running time increase clearly when the cover strength has been augmented. In [16], TCA sets its cutoff time up to 1000 second so as to get a good performance. For above experiments, we can see an extended cutoff time is very helpful for improving the stability of solutions in TCA. It is worth noting that the global optimization ability of TCA is usually limited, notwithstanding it can adjust the generated CA set constantly. So, for most of experiment trials, even the problems scale $|\Phi|$ is not very huge, TCA is likely to generate a high quality approximate solution. Under the same runtime, CSA shows a better average quality of $|TS|$ than TCA when the problems $|\Phi|$ is less than 2000. Meanwhile, the Integer Problems results shows that it is difficult for an integer program method to solve the problem which has more than about 100 integer variables. Instead, CSA shows a good performance when the $|\Phi|$ is less than 2000. That means the proposed approach can acquire more powerful global optimization ability through CSA than others. Moreover, the problem translation mechanism makes CSA less sensitive to strength-t than TCA and HHH, which is a very helpful character for CSA to get a higher performance when the covering strength is augmented.

On the other hand, comparing the experiment data between CSA and GA/BCGO in Table IV and Table V, we also find the best solution of GA/BCGO is very close to CSA when the $|\Phi|$ is less than 250. But, with the increasing of $|\Phi|$, the solution qualities of GA/BCGO begin to decline clearly. We believe the traditional searching operations, such as crossover operation and mutation operation are great at rearranging genetic structure and finding new code pattern, but not so good at refining the existing genetic structure, especially when the gene sequence is too long. In CSA, the searching among group process could be used to prospect the potential gene code, and the searching inside group process could be used to reduce multiple gene patterns in different groups. Based on the collaboration between the searching inside group process and the searching among group process, CSA can improve the performance of proposed approach remarkably.

Above all, we believe the proposed global mechanism is a more efficient calculation approach for small-size SUT with multiple covering strength.

5.3 Parameter Analysis

In this section, we will discuss how to adjust and control the coordination process between the searching among group process and the searching inside group process.

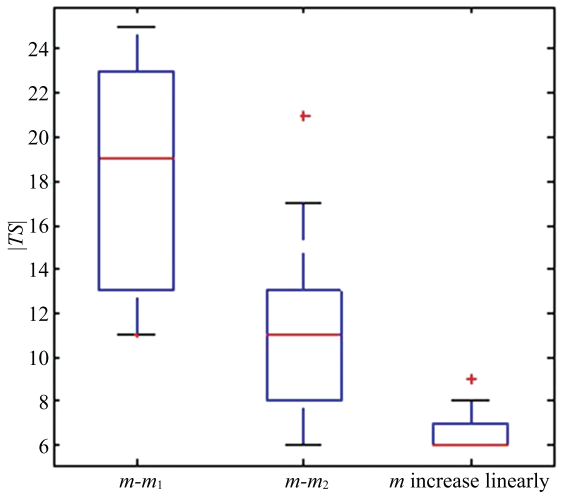
In CSA, the parameter m is used to control the computing number of searching inside group process in each group, which affects the proportion between global searching and local searching. For discussing the proper range and adjusting rule of m, the CA problem ($v = 2$, $k = 10$ and with strength-2) is used to do a testing. In this experiment, the m has been assigned 3 different values, which is $m = m_1$ ($m_1 = n/10$), $m = m_2$ ($m_2 = n/4$), and m is increased from m_1 to m_2 linearly by (17). Besides, the population size n is 60 and the other parameters in CSA are same as the testing in above section.

Fig. 4 is the statistical data of this experiment after 30 independent trials. The boxplot shows the dispersion of 30 experiment results of $|TS|$. We can see that too smaller m is likely to affect the solution quality, and too bigger m makes the distribution of solution much scattered. From the mean value curve of γ, which shows the changing trend of population diversity, we can find the bigger m makes its curve fall rapidly in early stages of searching process and the smaller m makes its curve fall slowly. The quick diversity loss in early searching stages is likely to make the probability of premature increasing. Conversely, too slow convergence will affect the solution accuracy. The formula (17) makes m increase gradually during the searching process. This test shows it is a feasible balancing mechanism, which makes the population keep a higher diversity against premature convergence in early searching stage while making population accelerate convergence to improve the accuracy of solution in later searching stage.

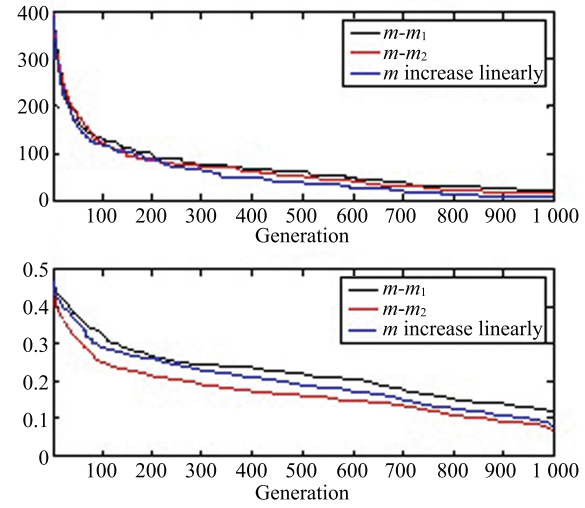
The parameter λ is another important controlling parameter, which is used to adjust the running probability between SIG1 and SIG2. For discussing the proper range of λ, the CA problem ($v = 3$, $k = 6$ and with strength-3) is used to do the test. In this experiment, the λ has been set 3 different values 0, 0.4 and 1. The values of other parameter in CSA are same as in the above section.

Fig. 5 is the statistical data of this experiment after 30 independent trials. Obviously, this parameter has a great influence on the gene sequence reducing process. If $\lambda = 0$, only SIG2 operation would be executed in searching inside group process. The boxplot shows it makes the distribution of solution much scattered and the quality of solution more

[2]http://www.public.asu.edu/ ccolbou/src/tabby/catable.html

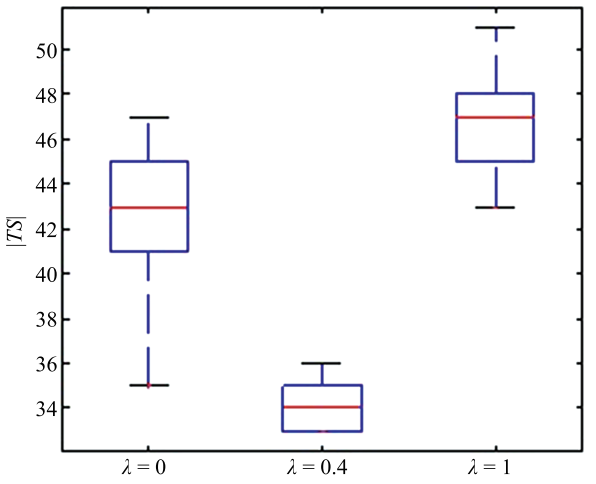


(a) Boxplot of experiment results for $|TS|$

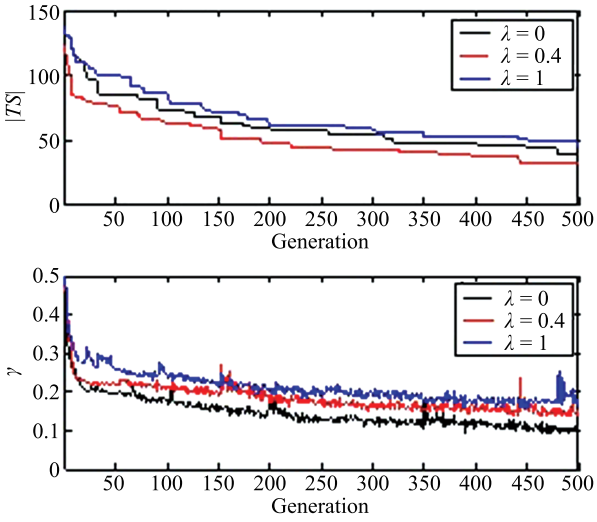


(b) Mean value curve of $|TS|$ and γ

Fig. 4. Experiment statistical data for $CA(6;2,10,2)$ with different parameter m.



(a) Boxplot of experiment results for $|TS|$ with different λ



(b) Mean value curve of $|TS|$ and γ

Fig. 5. Experiment statistical data for $CA(33;3,6,3)$ with different parameter λ.

unstable. Oppositely, if $\lambda = 1$, it would take only SIG1 operation to be executed in searching inside group process and makes the gene sequence unable to reduce effectively. The mean value curve of γ has shown the same changing trends. When $\lambda = 0$, this curve has fallen more rapidly. Meanwhile, when $\lambda = 1$, the falling speed of γ is too slow to ensure the quality of population convergence. Therefore, the efficiency of searching inside group process depends on the reasonable coordination between SIG1 and SIG2. Obviously, 0.4 is a feasible experience value for this parameter.

6 Conclusion

In this paper, we propose a cluster searching driven combinatorial test data global optimization and generation method. A program based on the proposed method that can be executed on compatible PC has been implemented. The experimental results show the proposed method can get a good performance for small-size CA problems. Within a reasonable time, the optimal test suite can be obtained with higher probability when the scale of complete test case set is less than 2000. But, the quality of its solution declines clearly when the count of test case is more than 4000. So, the proposed method is not ideal for solving the large-size CA problems. Furthermore, we have discussed 2 main control parameters of CSA and given a feasible adjusting approach for them. In future work, we will try to make this approach applicable to MCA problems.

References

1 D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, Greenbelt, MD, USA, 2002, pp. 91−95.

2 D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418−421, Jun. 2004.

3 J. F. Wang, C. A. Wei, and Y. L. Sheng, "Locating errors in combinatorial testing using set of possible faulty interactions," *Acta Electron. Sinica*, vol. 42, no. 6, pp. 1173−1178, Jun. 2014.

4 J. Yan and J. Zhang, "Combinatorial testing: Principles and methods," *Chin. J. Softw.*, vol. 20, no. 6, pp. 1393−1405, Jun. 2009.

5 G. Seroussi and N. H. Bshouty, "Vector sets for exhaustive testing of logic circuits," *IEEE Trans. Inform. Theory*, vol. 34, no. 3, pp. 513−522, May 1988.

6 J. Yan and J. Zhang, "A backtracking search tool for constructing combinatorial test suites," *J. Syst. Softw.*, vol. 81, no. 10, pp. 1681−1693, Oct. 2008.

7 C. J. Colbourn, S. S. Martirosyan, G. L. Mullen, D. Shasha, G. B. Sherwood, and J. L. Yucas, "Products of mixed covering arrays of strength two," *J. Combinat. Des.*, vol. 14, no. 2, pp. 124−138, Mar. 2006.

8 M. Sosina and T. van Tran, "On t-covering arrays," *Des. Codes Cryptogr.*, vol. 32, no. 1−3, pp. 323−339, May 2004.

9 F. Kang, S. X. Han, S. Rodrigo, and J. J. Li, "System probabilistic stability analysis of soil slopes using Gaussian process regression with Latin hypercube sampling," *Comput. Geotechn.*, vol. 63, pp. 13−25, Jan. 2015.

10 F. Kang, J. S. Li, and J. J. Li, "System reliability analysis of slopes using least squares support vector machines with particle swarm optimization," *Neurocomputing*, vol. 209, pp. 46 −56, Oct. 2016.

11 F. Kang, Q. Xu, and J. J. Li, "Slope reliability analysis using surrogate models via new support vector machines with swarm intelligence," *Appl. Math. Model.*, vol. 40, no. 11−12, pp. 6105−6120, Jun. 2016.

12 F. Kang and J. J. Li, "Artificial bee colony algorithm optimized support vector regression for system reliability analysis of slopes," *J. Comput. Civil Eng.*, vol. 30, no. 3, pp. 04015040, May 2016.

13 R. J. Zha, D. P. Zhang, C. H. Nie, and B. W. Xu, "Test data generation algorithms of combinatorial testing and comparison based on cross-entropy and particle swarm optimization method," *Chin. J. Comput.*, vol. 33, no. 10, pp. 1896−1908, Oct. 2010.

14 Y. L. Liang and C. H. Nie, "The optimization of configurable genetic algorithm for covering arrays generation," *Chin. J. Comput.*, vol. 35, no. 7, pp. 1522−1538, Jul. 2012.

15 C. H. Nie and J. Jiang, "Optimization of configurable greedy algorithm for covering arrays generation," *Chin. J. Softw.*, vol. 24, no. 7, pp. 1469−1483, Jul. 2013.

16 J. K. Lin, C. Luo, S. W. Cai, K. L. Su, D. Hao, and L. Zhang, "TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, Lincoln, NE, USA, 2015, pp. 494−505.

17 K. Z. Zamli, B. Y. Alkazemi, and G. Kendall, "A tabu search hyper-heuristic strategy for t-way test suite generation," *Appl. Soft Comput.*, vol. 44, pp. 57−74, Jul. 2016.

18 J. Yan and J. Zhang, "Backtracking algorithms and search heuristics to generate test suites for combinatorial testing," in *Proc. 30th IEEE Annual Int. Computer Software and Applications Conf.*, Chicago, IL, USA, 2006, pp. 385−394.

19 A. W. Williams and R. L. Probert, "Formulation of the interaction test coverage problem as an integer program," in *Testing of Communicating Systems XIV*, US: Springer, 2002, pp. 283−298.

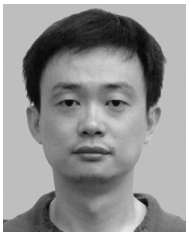
20 M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proc. 25th Int. Conf. Software Engineering*, Portland, OR, USA, 2003, pp. 38−48.

21 L. Wang and L. P. Li, "A coevolutionary differential evolution with harmony search for reliability redundancy optimization," *Expert Syst. Appl.*, vol. 39, no. 5, pp. 5271−5278, Apr. 2012.

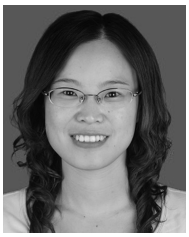
22 I. Ciornei and E. Kyriakides, "Hybrid ant colony-genetic algorithm (GAAPI) for global continuous optimization," *IEEE Trans. Syst. Man Cybern.-Part B*, vol. 42, no. 1, pp. 234−245, Feb. 2012.

23 M. Chateauneuf and D. L. Kreher, "On the state of Strength-Three Covering Arrays," *J. Combin. Des.*, vol. 10, no. 4, pp. 217−238, May 2002.

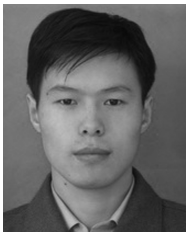
24 R. Gelbard, O. Goldman, and I. Spiegler, "Investigating diversity of clustering methods: An empirical comparison," *Data Knowl. Eng.*, vol. 63, no. 1, pp. 155−166, Oct. 2007.



Hao Chen is a Ph.D., and a member of China Computer Federation. He is currently an Associate Professor at the School of Computer Science and Technology, Xi'an University of Post and Telecommunications (XUPT). His research interests include engineering optimization, soft testing, and data mining. Corresponding author of this paper.
E-mail: chenhao@xupt.edu.cn



Xiaoying Pan is a Ph.D., and a member of China Computer Federation. She is currently an Associate Professor at the School of Computer Science and Technology, Xi'an University of Post and Telecommunications (XUPT). Her research interests include multi-agent system and numerical optimization.
E-mail: panxiaoying@xiyou.edu.cn



Jiaze Sun is a Ph.D., and a member of China Computer Federation. He is currently an Associate Professor at the School of Computer Science and Technology, Xi'an University of Post and Telecommunications (XUPT). His research interests include swarm intelligence optimization algorithm, software testing, and data mining. E-mail: sunjiaze@xupt.edu.cn