A Discrete Particle Swarm Optimization for Covering Array Generation

Huayao Wu, Changhai Nie, *Member, IEEE,* Fei-Ching Kuo, *Member, IEEE,* Hareton Leung, *Member, IEEE,* and Charles J. Colbourn

***Abstract*—Software behavior depends on many factors. Combinatorial testing (CT) aims to generate small sets of test cases to uncover defects caused by those factors and their interactions. Covering array generation, a discrete optimization problem, is the most popular research area in the field of CT. Particle swarm optimization (PSO), an evolutionary search-based heuristic technique, has succeeded in generating covering arrays that are competitive in size. However, current PSO methods for covering array generation simply round the particle's position to an integer to handle the discrete search space. Moreover, no guidelines are available to effectively set PSOs parameters for this problem. In this paper, we extend the set-based PSO, an existing discrete PSO (DPSO) method, to covering array generation. Two auxiliary strategies (particle reinitialization and additional evaluation of *gbest*) are proposed to improve performance, and thus a novel DPSO for covering array generation is developed. Guidelines for parameter settings both for conventional PSO (CPSO) and for DPSO are developed systematically here. Discrete extensions of four existing PSO variants are developed, in order to further investigate the effectiveness of DPSO for covering array generation. Experiments show that CPSO can produce better results using the guidelines for parameter settings, and that DPSO can generate smaller covering arrays than CPSO and other existing evolutionary algorithms. DPSO is a promising improvement on PSO for covering array generation.**

***Index Terms*—Combinatorial testing (CT), covering array generation, particle swarm optimization (PSO).**

I. Introduction

As SOFTWARE functions and run-time environments become more complex, testing of modern software systems is becoming more expensive. Effective detection of failures at a low cost is a key issue for test case generation. Combinatorial testing (CT) is a popular testing method to detect failures triggered by various factors and their interactions [1]. By employing covering arrays as test suites, the CT method aims to sample the large combination space with few test cases to cover different interactions among a fixed number of factors. Kuhn and Reilly [2] shows that more than 70% of the failures in certain software were caused by the interactions of one or two factors, and almost all the failures could be detected by checking the interactions among six factors. Therefore, CT can be an effective method in practice.

Generating a covering array with fewest tests (minimum size) is a major challenge in CT. In general, the minimum size of a covering array is unknown; hence, methods have focused on finding covering arrays that have as few tests as possible at reasonable search cost. The many methods that have been proposed can be classified into two main groups: 1) mathematical methods and 2) computational methods [1]. Mathematical (algebraic or combinatorial) methods typically exploit some known combinatorial structure. Computational methods primarily use greedy strategies or heuristic-search techniques to generate covering arrays, due to the size of the search space.

Mathematical methods yield the best possible covering arrays in certain cases. For example, orthogonal arrays used in the design of experiments provide covering arrays with a number of tests that is provably minimum. However, all known mathematical methods can be applied only for restrictive sets of factors. This limitation has led to an emphasis on computational methods. Greedy algorithms have been quite effective in generating covering arrays, but their accuracy suffers from becoming trapped in local optima.

In recent years, search-based software engineering (SBSE) has focused on using search-based optimization algorithms to find high-quality solutions for software engineering problems. Inspired by SBSE, many artificial intelligence-based heuristic-search techniques have been applied to software testing. For example, simulated annealing (SA) [3]–[7], genetic algorithm (GA) [8]–[10], and ant colony optimization (ACO) [9], [11], [12] have all been applied to covering array generation. These techniques can generate any types of covering arrays, and the constraint solving and prioritization techniques can be easily integrated. Their applications have been shown to be effective, producing relatively small covering arrays in many cases. Particle swarm optimization (PSO), a relatively new evolutionary algorithm,

Manuscript received May 12, 2013; revised December 29, 2013 and May 18, 2014; accepted September 28, 2014. Date of publication October 9, 2014; date of current version July 28, 2015. This work was supported in part by the National Natural Science Foundation of China under Grant 61272079, in part by the Research Fund for the Doctoral Program of Higher Education of China under Grant 20130091110032, in part by the Science Fund for Creative Research Groups of the National Natural Science Foundation of China under Grant 61321491, in part by the Major Program of National Natural Science Foundation of China under Grant 91318301, and in part by the Australian Research Council Linkage under Grant LP100200208. *(Corresponding author: Changhai Nie.)*

H. Wu and C. Nie are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: hywu@outlook.com; changhainie@nju.edu.cn).

F.-C. Kuo is with the Faculty of Information and Communication Technologies, Swinburne University of Technology, Hawthorn, VIC 3122, Australia (e-mail: dkuo@swin.edu.au).

H. Leung is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong (e-mail: hareton.leung@polyu.edu.hk).

C. J. Colbourn is with Arizona State University, Tempe, AZ 85287-8809, USA (e-mail: colbourn@asu.edu).

Digital Object Identifier 10.1109/TEVC.2014.2362532

has also been used in this area [13]–[16]. It is easy to implement and has fast initial progress.

The conventional PSO (CPSO) algorithm was originally designed to find optimal or near optimal solutions in a continuous space. Nevertheless, many discrete PSO (DPSO) algorithms and frameworks have been developed to solve discrete problems [17]–[22]. For covering array generation, current discrete methods [13]–[16] simply round the particle's position to an integer while keeping the velocity as a real number. They suffer from two main shortcomings. First, the performance of PSO is significantly impacted by its parameter settings. In [23], effects of the general parameter selection and initial population of PSO have been analyzed, but no guidelines on parameter settings have been reported for covering array generation. Hence, a clear understanding of how to set these execution parameters is needed. Second, simple rounding fractional positions to integers introduces a substantial source of errors in the search. Instead, a specialized DPSO version is needed. Because current PSO methods show promise for generating covering arrays, these two main shortcomings should be addressed.

In this paper, we adapt set-based PSO (S-PSO) [18] to generate covering arrays. S-PSO utilizes set and probability theories to develop a new representation scheme for combinatorial optimization problems as well as refining the related evolution operators. In our adaptation of S-PSO, two auxiliary strategies are proposed to enhance performance, and a novel DPSO algorithm is thus proposed. DPSO has the same conceptual basis and exhibits similar search behavior to CPSO, with parameters playing similar roles. Then, we explore the optimal parameter settings for both CPSO and DPSO to improve their performance, and identify recommended settings. Furthermore, because many CPSO variants [24]–[27] can be easily extended to discrete versions based on our DPSO, the performance of these discrete versions is also compared with their original ones. Finally, we compare CPSO and DPSO with existing GA and ACO [9], [11] algorithms to generate covering arrays.

The main contributions of this paper are as follows.

1) Based on the set-based representation, we design a version of S-PSO [18] for covering array generation.
2) We propose two auxiliary strategies, particle reinitialization, and additional evaluation of *gbest*, to enhance the performance of PSO. A novel DPSO for covering array generation is proposed.
3) We design experiments to explore the optimal parameter settings for CPSO and DPSO for covering array generation.
4) We implement original and discrete versions of four representative PSO variants (TVAC [24], CLPSO [25], APSO [26], and DMS-PSO [27]) to compare their efficacy for covering array generation.

The rest of this paper is organized as follows. Section II gives background on CT, covering array generation, and the CPSO algorithm. Section III summarizes related work. Section IV presents our DPSO algorithm, including the representation scheme, related operators, and two auxiliary strategies. Section V evaluates the performance of CPSO and DPSO, and explores optimal parameter settings. Section VI gives a comparison among CPSO, DPSO, and original and discrete variants. Section VII compares CPSO and DPSO with GA and ACO. Section VIII concludes this paper and outlines future work.

TABLE I
E-COMMERCE SOFTWARE SYSTEM

Payment Server	Smart Phone	Web Server	User Browser	Business Database
Master	iPhone	iPlanet	Chrome	SQL
VISA	Blackberry	Apache	Explore	Oracle
			Firefox	Access

II. BACKGROUND

A. CT

Suppose that the behavior of the software under test (SUT) is controlled by n independent factors, which may represent configuration parameters, internal or external events, user inputs, and the like. The ith factor has ℓ_i discrete levels, chosen from the finite set V_i of size ℓ_i. An n-tuple $(x_1, x_2, \ldots, x_n)$ forms a test case, where $x_i \in V_i$ for $1 \leq i \leq n$.

Consider a simple e-commerce software system [15]. This system consists of five different components. Each of these five components can be regarded as a factor, and its configurations can be regarded as different levels. Table I shows these five factors and their corresponding levels. In this example, $n = 5$, $\ell_1 = \ell_2 = \ell_3 = 2$, $\ell_4 = \ell_5 = 3$.

System failures are often triggered by interactions among some factors, which can be represented by the combinations of factor levels. In order to detect these failures, their combinations should be covered at least once by the test suite. A t-way schema can be used to represent them.

Definition 1 (t-way schema): The n-tuple $(-, y_1, \ldots, y_t, \ldots)$ is a *t-way schema* $(t > 0)$ when some t factors have fixed levels and the others can take any valid levels, represented as "$-$."

For example, suppose that when factor Payment Server takes the level Master and factor Web Server takes the level Apache, a system failure occurs. To detect this failure, the 2-way schema (Master, –, Apache, –, –) must be covered at least once by the test suite. To simplify later discussion, we use the index in the level set of each factor to present a schema. For example, (0, –, 1, –, –) is used to represent (Master, –, Apache, –, –).

Exhaustive testing covers all n-way schemas of the system; in our example, it uses $2 \times 2 \times 2 \times 3 \times 3 = 72$ test cases. Testing cost becomes prohibitive as the numbers of factors and levels increase. Moreover, because only interactions among few factors are likely to trigger failures [2], testing high-way schemas can lead to many uninformative test cases. At the other extreme, if we only guarantee to cover each 1-way schema once, only three test cases are needed (a single test case can cover five 1-way schemas at most). But we may fail to detect some interaction triggered failures involving two factors.

TABLE II
COVERING ARRAY $CA(9; 2, 2^3 3^2)$

#	Payment Server	Smart Phone	Web Server	User Browser	Business Database
1	Master	Blackberry	Apache	Firefox	Oracle
2	VISA	iPhone	iPlanet	Explorer	Oracle
3	VISA	iPhone	Apache	Chrome	SQL
4	Master	Blackberry	iPlanet	Chrome	Access
5	Master	Blackberry	iPlanet	Explorer	SQL
6	VISA	iPhone	iPlanet	Firefox	Access
7	Master	iPhone	Apache	Explorer	Access
8	VISA	Blackberry	iPlanet	Chrome	Oracle
9	VISA	Blackberry	iPlanet	Firefox	SQL

Instead, CT covers all t-way schemas. Such a test suite is a t-way covering array, with t being the covering strength. The value of t determines the depth of coverage. It is a key setting of CT, and should be decided by the testers. We give a precise definition.

Definition 2 (Covering Array): If an $N \times n$ array, where N is the number of test cases, has the following properties: 1) each column i ($1 \leq i \leq n$) contains only elements from the set V_i with $\ell_i = |V_i|$ and 2) the rows of each $N \times t$ sub array cover all $|V_{k_1}| \times |V_{k_2}| \times \ldots \times |V_{k_t}|$ combinations of the t columns at least once, where $t \leq n$ and $1 \leq k_1 < \ldots < k_t \leq n$, then it is a t-way covering array, denoted by $CA(N; t, n, (\ell_1, \ell_2, \ldots, \ell_n))$. When $\ell_1 = \ell_2 = \ldots = \ell_n = \ell$, it is denoted by $CA(N; t, n, \ell)$.

Reference [2] demonstrated that more than 70% failures can be detected by a 2-way covering array, and almost all failures can be detected by a 6-way covering array. Hence, using CT, we can detect many failures of the system by applying relatively low strength covering arrays. In other words, CT can greatly reduce the size of test suite while maintaining high fault detection ability.

In the example of Table I, if we only consider the interactions between any two factors, only nine test cases are required to construct a 2-way covering array instead of 72 for exhaustive testing. Table II shows a covering array, where each row represents one test case. This table covers all possible 2-way schemas for any two factors corresponding to the columns. For example, consider factor Payment Server and User Browser, all $2 \times 3 = 6$ schemas, (Master, –, –, Firefox, –) (Master, –, –, Explorer, –), (Master, –, –, Chrome, –), (VISA, –, –, Firefox, –), (VISA, –, –, Explorer, –), (VISA, –, –, Chrome, –), can be found in the table.

For convenience, if several groups of g_i factors ($g_i < n$) have the same number of levels a_k, $a_k^{g_i}$ can be used to represent these factors and their levels. Thus, the covering array can be denoted by $CA(N; t, a_1^{g_1}, a_2^{g_2}, \ldots, a_k^{g_k})$ where $\sum g_i = n$, or $CA(N; t, a^n)$ when $g_1 = n$ and $a_1 = a$. For example, the covering array in Table II is a $CA(9; 2, 2^3 3^2)$.

In many software systems, the impacts of the interactions among factors are not uniform. Some interactions may be more prone to trigger system failures, while other may have little or no impact on the system. To effectively detect these different interactions, variable strength (VS) covering arrays can be applied. This can offer different covering strengths to different groups of factors, and can therefore provide a practical approach to test real applications.

TABLE III
ADDING THREE TEST CASES TO CONSTRUCT $VCA(12; 2, 2^3 3^2, CA(3, 2^2 3^1))$

10	Master	Blackberry	iPlanet	Explorer	Oracle
11	Master	iPhone	Apache	Firefox	SQL
12	VISA	Blackberry	Apache	Chrome	Oracle

Definition 3 (VS Covering Array): A VS covering array, denoted by $VCA(N; t, a_1^{\ell_1} \ldots a_k^{\ell_k}, CA_1(t_1, b_1^{m_1} \ldots b_p^{m_p}), \ldots, CA_j(t_j, c_1^{n_1} \ldots c_q^{n_q}))$, is an $N \times n$ covering array of covering strength t containing one or more sub covering arrays, namely $CA_1, \ldots, CA_j$, each of which has covering strength $t_1, \ldots, t_j$ all larger than t.

Consider the e-commerce system shown in Table I. If the interactions of three factors, Payment Server, Web Server, and Business Database, have a higher probability to trigger system failures, then a VS covering array can be constructed. As in Table II, only three more test cases (Table III) are needed to construct the $VCA(12; 2, 2^3 3^2, CA(3, 2^2 3^1))$. With these 12 test cases, not only are all 2-way schemas of all five factors covered, but also all 3-way schemas of these three factors (Payment Server, Web Server, and Business Database) are covered.

B. Covering Array Generation

Covering arrays are used as test suites in CT. Covering array generation is the process of test suite construction. It is the most active area in CT with more than 50% of research papers focusing on this field [1]. Due to limited testing resources, all aim to construct a minimal covering array while still maintaining full coverage of combinations. Computational methods have been used widely for covering array generation because they can be applied to any systems. In general, these methods generate all possible combinations first. Then they generate test cases to cover these combinations one-by-one. One-test-at-a-time is the most widely used strategy among evolutionary algorithms to generate a covering array.

The one-test-at-a-time strategy was popularized by AETG [28] and was further used by Bryce and Colbourn [29]. This strategy takes the model of SUT(n, ℓ_1, …, ℓ_n) where n is the number of factors and ℓ_i is the number of valid levels of factor i, and the covering strength t as input. At first, an empty test suite TS and a set S of t-way schemas to be covered are initialized. In each iteration, a test case is generated with the highest fitness value according to some heuristic techniques. Then it is added to TS and the t-way schemas covered by it are removed. When all the t-way schemas have been covered, the final test suite TS is returned. This process is shown in Algorithm 1.

In this strategy, a fitness function must be used to evaluate the quality of a candidate test case (line 6 in Algorithm 1). It is an important part of all heuristic techniques. In covering array generation, the fitness function takes the test case as the input and then outputs a fitness value representing its "goodness." It is defined as follows.

Algorithm 1 One-Test-at-a-Time Strategy

1: Input: SUT(n, ℓ_1, …, ℓ_n) and covering strength t
2: Output: covering array TS
3: $TS = \emptyset$
4: Construct S (all the t-way schemas to be covered) based on n, ℓ_1, …, ℓ_n and t
5: **while** $S \neq \emptyset$ **do**
6: Generate a test case p with the highest fitness value according to some heuristics
7: Add p to the test suite TS
8: Remove the t-way schemas covered by p from S
9: **end while**
10: **return** TS

Definition 4 (Fitness Function): Let TS be the generated test set, and p be a test case. Then $fitness(p)$ is the number of uncovered t-way schemas in TS that are covered by p.

The fitness function can be formulated as

$$fitness(p) = |schema_t(\{p\}) - schema_t(TS)| \tag{1}$$

where $schema_t(TS)$ represents the set of all t-way schemas covered by test set TS, and $|\cdot|$ stands for cardinality. When all C_n^t t-way schemas covered by p are not covered by TS, the fitness function reaches the maximum value $fitness(p) = |schema_t(\{p\})| = C_n^t$.

For example, consider the 2-way covering array generation of the e-commerce system shown in Table II. Suppose that TS consists of test cases (0, 0, 0, 0, 0) and (0, 0, 1, 1, 1). The fitness function computes the fitness value for a candidate test case p = (1, 0, 1, 2, 2) as follows: because p covers ten 2-way schemas, namely $schema_2(\{p\})$ = {(1, 0, –, –, –), (1, –, 1, –, –,), (1, –, –, 2, –), (1, –, –, –, 2), (–, 0, 1, –, –), (–, 0, –, 2, –), (–, 0, –, –, 2), (–, –, 1, 2, –), (–, –, 1, –, 2), (–, –, –, 2, 2)} and TS only covers (–, 0, 1, –, –) in p, the function returns $fitness(p)$ = 9.

C. PSO

PSO is a swarm-based evolutionary computation technique. It was developed by Kennedy *et al*. [30], inspired by the social behavior of bird flocking and fish schooling. PSO utilizes a population of particles as a set of candidate solutions. Each of the particles represents a certain position in the problem hyperspace with a given velocity. A fitness function is used to evaluate the quality of each particle. Initially, particles are distributed in the hyperspace uniformly. Then each particle repeatedly updates its state according to the individual best position in its history (*pbest*) and the current global best position (*gbest*). Eventually, each particle possibly moves toward the direction of the individual optimum and global optimum, and finds an optimal or near optimal solution.

Suppose that the problem domain is a $D-$dimensional hyperspace. Then the position and velocity of particle i can be represented by $x_i \in R^D$ and $v_i \in R^D$ respectively. CPSO uses the following equations to update a particle's velocity and position, where $v_{i,j}(k)$ represents the jth component of the velocity of particle i at the kth iteration, and $x_{i,j}(k)$ represents its corresponding position:

$$v_{i,j}(k+1) = \omega \times v_{i,j}(k) + c_1 \times r_{1,j} \times (pbest_{i,j} - x_{i,j}(k)) + c_2 \times r_{2,j} \times (gbest_j - x_{i,j}(k)) \tag{2}$$

$$x_{i,j}(k+1) = x_{i,j}(k) + v_{i,j}(k+1). \tag{3}$$

The best position of particle i in its history is $pbest_i$, and $gbest$ is the best position among all particles. The velocity-update equation (2) captures the three basic concepts of PSO. The first is inertia, the tendency of the particle to continue in the direction it has been moving. The second is memory of the best position ever found by itself. The third is cooperation using the best position found by other particles.

The parameter ω is inertia weight. It controls the balance between exploration (global search state) and exploitation (local search state). Two positive real numbers c_1 and c_2 are acceleration constants that control the movement tendency toward the individual and global best position. Most studies set $\omega = 0.9$, and $c_1 = c_2 = 2$ to get the best balance [17], [31], [32]. In addition, $r_{1,j}$ and $r_{2,j}$ are two uniformly distributed random numbers in the range of [0.0, 1.0], used to ensure the diversity of the population.

If the problem domain (the search space of particles) has bounds, a bound handling strategy is adopted to keep the particles inside the search space. Many different strategies have been proposed [33]. In the reflecting strategy, when a particle exceeds the bound of the search space in any dimension, the particle direction is reversed in this dimension to get back to the search space. For example, in case of a dimension with a range of values from 0 to 2, if a particle moves to 3, its position is reversed to 1.

In addition, as the velocity can increase over time, a limit is set on velocity to prevent an infinite velocity or invalid position for the particle. Setting a maximum velocity, which determines the distance of movement from the current position to the possible target position, can reduce the likelihood of explosion of the swarm traveling distance [31]. Generally, the value of the maximum velocity is selected as $\ell_i/2$, where ℓ_i is the range of dimension i.

The pseudocode in Algorithm 2 presents the process of generating a test case by PSO. This algorithm can be invoked by line 6 of Algorithm 1 to generate a test case for t-way schemas. The n factors of the test case can be treated as an n-dimensional hyperspace. A particle $p_i = (x_1, x_2, \ldots, x_n)$ can be regarded as a candidate test case. The fitness function is that of Definition 4, the number of uncovered t-way schemas in the generated test suite that are covered by particle p_i. PSO employs real numbers but the valid values are integers for covering array generation, so each dimension of particle's position can be rounded to an integer while maintaining the velocity as a real number. This method is used in all prior research applying PSO to covering array generation [13]–[16].

III. Related Work

In this section, three different but related aspects are discussed. We first summarize search-based CT, especially the current applications of PSO for covering array generation. Then, we introduce prior research on discrete versions of PSO,

Algorithm 2 Generate Test Case by PSO

```
Input: SUT(n, ℓ1, ..., ℓn), covering strength t and the related parameters of PSO
Output: the best test case gbest
it = 0, gbest = NULL
for each particle pi do
    Initialize the position xi and velocity vi randomly
end for
while it < maximum iteration do
    for each particle pi do
        Compute the fitness value fitness(pi)
        if fitness(pi) > fitness(pbesti) then
            pbesti = pi
        end if
        if fitness(pi) > fitness(gbest) then
            gbest = pi
        end if
    end for
    for each particle pi do
        Update the velocity and position according to Equations 2 and 3
        Apply maximum velocity limitation and bound handling strategy
    end for
    it = it + 1
end while
return gbest
```

to improve CPSO for discrete problems. Finally, some PSO variants are introduced. These try to enhance PSO for conventional continuous problems by adopting strategies that can further improve covering array generation.

A. PSO in Search-Based CT

SBSE has grown quickly in recent years. Many problems in software engineering have been formulated into search-based optimization problems, and heuristic techniques have been used to find solutions. Software testing is a major topic in software engineering. Many heuristic techniques have also been applied to this field, including functional testing, mutation testing, stress testing, temporal testing, CT, and regression testing. Currently, many classic heuristic techniques, such as SA [3]–[7], GA [8]–[10], and ACO [9], [11], [12] have been applied to generate uniform and VS covering arrays successfully.

PSO has also been applied to software testing. Windisch *et al*. [34] applied PSO to structural testing, and compared its performance with GA. They showed that PSO outperformed GA for most cases in terms of effectiveness and efficiency. Ganjali [35] proposed a framework for automatic test partitioning based on PSO, observing that PSO performed better than other existing heuristic techniques.

PSO has been applied to covering array generation. Ahmed and Zamli [15] proposed frameworks of PSO to generate different kinds of covering arrays, including t-way (PSTG [14]) and VS-PSTG. Then they extended PSTG to support VS covering array generation [16]. Chen *et al*. [18] also applied PSO to 2-way covering array generation. They further used a test suite minimization algorithm to reduce the size of the generated covering array.

Current applications of PSO for covering array generation can yield smaller covering arrays than most greedy algorithms, but they all apply the same rounding operator to the particle's position, and they lack guidelines on the parameter settings.

B. Discrete Versions of PSO

PSO was initially developed to solve problems in continuous space, but PSO can also be applied to some discrete optimization problems including binary or general integer variables. Many discrete versions of PSO have been proposed [17]–[22]. Chen *et al*. [18] classify existing algorithms into four types.

1) Swap operator-based PSO [19] uses a permutation of numbers as position and a set of swaps as velocity.
2) Space transformation-based PSO [20] uses various techniques to transform the position, defined as a vector of real numbers, to the corresponding solution in discrete space.
3) Fuzzy matrix-based PSO [21] defines the position and velocity as a fuzzy matrix, and decode it to a feasible solution.
4) Incorporating-based PSO [22] consists of hybrid methods incorporating some heuristic or problem dependent techniques.

Chen *et al*. [18] also propose a S-PSO method based on sets with probabilities, which we later adapt to represent a particle's velocity. S-PSO uses a set-based scheme to represent the discrete search space. The velocity is defined as a set with probabilities, and the operators are all replaced by procedures defined on the set. They extend some PSO variants to discrete versions and test them on the traveling salesman problem and the multidimensional knapsack problem. They show that the discrete version of CLPSO [25] can perform better than other variants. S-PSO was found to characterize the combinatorial optimization problems very well. Gong *et al*. [36] then employed such a set-based representation scheme for solving vehicle routing problems to obtain a new method, S-PSO-VRPTW.

C. PSO Variants

The original PSO may become trapped in a local optimum. In order to improve the performance, many variants have been proposed [17], [24]–[27]. Chen *et al*. [37] classify these variants into four different groups. The first group of variants aims to adjust the control parameters during the evolution, for example by decreasing inertia weight ω linearly from 0.9 to 0.4 over the search process. The second group employs different neighborhood topologies. The basic local version of PSO makes the particle learn from the local best position $lbest_i$ found by particle i's neighborhood instead of the global best position *gbest*. RPSO and VPSO are two common versions which use a ring topology and a Von Neumann topology, respectively. The third group uses hybrid strategies with other search techniques. Many types of operators, such as genetic operators

TABLE IV
DIFFERENT GROUPS OF DPSO AND PSO VARIANTS

	Groups
Discrete PSO	Swap operator based PSO [19]
	Space transformation based PSO [20]
	Fuzzy matrix based PSO [21]
	Incorporating based PSO [22]
PSO Variants	Adjusting the parameters, TVAC [24]
	Designing different neighborhood topologies, CLPSO [25]
	Using hybrid strategies with search techniques, APSO [26]
	Using multi-swarm techniques, DMS-PSO [27]

(selection, crossover, and mutation), evolutionary computation paradigms and biology inspired operators have been used. The fourth uses multiswarm techniques. Several sets of swarms optimize different components of the solution concurrently or cooperatively.

In our experiments, four representative PSO variants are included, as follows.

1) Ratnaweera *et al*. [24] proposed a typical variant of the first group, TVAC, which uses a time varying inertia weight and acceleration constant to adjust the parameters.
2) Liang *et al*. [25] proposed the CLPSO method, a well-known variant of the second group, which allows the particle to learn from other particles' individual best positions in different dimensions.
3) Zhan *et al*. [26] proposed an adaptive PSO (APSO) that can be seen as a variant of the third group. They developed an evolutionary state estimation to adaptively control the parameter and used an elitist learning strategy.
4) DMS-PSO [27], a variant of the fourth group, was proposed by Liang and Suganthan [27]. It is characterized by a set of swarms with small sizes and these swarms are frequently regrouped.

In summary, Table IV lists the different groups of discrete versions of PSO and PSO variants.

IV. DPSO

In this section, a new DPSO for covering array generation is presented. We firstly illustrate the weakness of CPSO with a simple example. Then the representation scheme of a particle's velocity and the corresponding redefined operators are introduced. Finally, we give two auxiliary strategies to enhance the performance of DPSO.

A. Example

In CPSO, a particle's position represents a candidate test case; its velocity, a real vector, represents the movement tendency of this particle in each dimension. This scheme is meaningful in a continuous optimization problem, because an optimal solution may exist near the current best particle's position. So it is desirable to move the particle to this area for further search. This may not hold for covering array generation.

Here, we use $CA(N; 2, 3^4)$ as an example. Suppose that three test cases (0, 1, 0, 1), (2, 1, 1, 2), and (1, 2, 0, 0) have been generated and added to *TS* in Algorithm 1, and the fourth one is to be generated according to CPSO. A possible candidate particle p_i may have the position (0, 0, 0, 0) and velocity (0.5, 0.6, −0.4, 0.2) with fitness value 4, and its individual best position $pbest_i$ may be (0, 0, 1, 1) with fitness value 5. The global best position *gbest* may be (0, 2, 2, 2) with fitness value 6. According to the update (2), if we take $\omega = 0.9$ and $c_1 \times r_1 = c_2 \times r_2 \approx 0.65$, in the next iteration, p_i's velocity may be transformed to (0.45, 1.84, 1.59, 2.13). Thus, p_i moves to the new position (0, 1, 1, 1) with a limitation on the velocity (the velocity is kept in $[-\ell_i/2, \ell_i/2]$), or (0, 1, 1, 2) with no limitation. In both cases, p_i only has fitness value 2 after updating. When this occurs, p_i evolves to a worse situation, although its position is closer to the $pbest_i$ and *gbest* than its original one.

Analyzing the fitness measurement, the main contribution to the fitness value is the combinations that the test case can cover, not the concrete "position" at which it is located. For example, test case (2, 1, 1, 2) has a larger fitness value than (0, 0, 0, 0) because it covers six new schemas [(2, 1, –, –), (2, –, 1, –), (2, –, –, 2) etc.], not because of its relative distance to other particles.

B. DPSO

To overcome this weakness, the movement of particles should be modified. Inspired by the set-based representation scheme of velocity, DPSO is designed as a version of S-PSO [18] to make the particle learn from the individual and global best more effectively when generating covering arrays. Unlike S-PSO, the element of the velocity set in DPSO is designed for covering array generation, and DPSO does not classify the velocity set into different dimensions to avoid the inconsistency of different dimensions when updating velocities in S-PSO.

In DPSO, a particle's position represents a candidate test case, while its velocity is changed to a set of t-way schemas with probabilities. Other than the velocity's representation and the newly defined operators, the evolution procedure of DPSO is the same as CPSO (Algorithm 2).

Definition 5 (Velocity): The velocity is a set of pairs $(s, p(s))$, where s is a possible t-way schema of the covering array and $p(s)$ is a real number between 0 and 1 representing the probability of the selection of schema s to update the current position.

In the initialization of the swarm, the particle's position is randomly assigned. Then C_n^t possible different schemas are selected randomly and each of them is assigned a random probability $p(s) \in (0, 1)$. These C_n^t pairs form the initial velocity set of this particle; the size of this set changes dynamically during the evaluation.

We consider the same example $CA(N; 2, 3^4)$. In DPSO, when particle p_i is initialized, its position $x_i(k)$ may be (0, 0, 0, 0) representing a candidate test case as before, and its velocity $v_i(k)$ may be such a set {((1, 1, –, –), 0.7), ((0, –, 0, –), 0.3), ((–, 0, –, 1), 0.8), ((0, –, –, 2), 0.9),

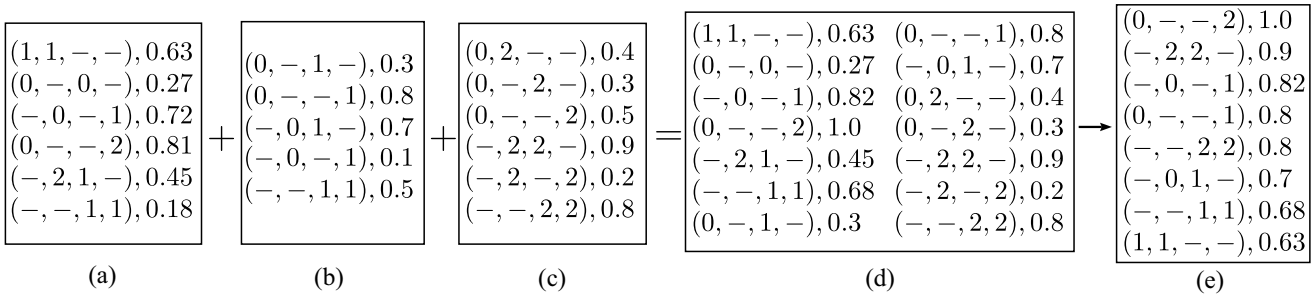

Fig. 1. Example of p_i's velocity updating. (a) $0.9 \times v_i(k)$. (b) $2 \times r_1 \times (pbest_i - x_i(k))$. (c) $2 \times r_2 \times (gbest - x_i(k))$. (d) $v_i(k+1)$. (e) Final $v_i(k+1)$ where $pro_1 = 0.5$.

((–, 2, 1, –), 0.5), ((–, –, 0, 1), 0.2)} which contains $C_4^2 = 6$ pairs.

DPSO follows the conventional evolution procedure (Algorithm 2) that uses (2) and (3) to update the velocity and position of a particle. To adapt the new scheme for velocity in Definition 5, the related operators in these equations must be redefined.

1) Coefficient × Velocity: The coefficient is a real number which may be a parameter or a random number. It modifies all the probabilities in the velocity.

Definition 6 (Coefficient × Velocity): Let a be a nonnegative real number and v be a velocity, $a \times v = \{(s, p(s) \times a) | (s, p(s)) \in v\}$. (If $p(s) \times a > 1$, $p(s) \times a = 1$.)

For example, Fig. 1(a) shows the result for $\omega \times v_i(k)$ where $\omega = 0.9$.

2) Position–Position: The difference of two positions gives the direction on which a particle moves. The results of the minus operator is a set of $(s, p(s))$ pairs, as velocity.

Definition 7 (Position–Position): Let x_1 and x_2 be two positions. Then $x_1 - x_2 = \{(s, 0.5) | s$ is a schema that exists in x_1 but not in $x_2\}$.

In the newly generated schema, probability $p(s)$ for s is set to 0.5 so that the acceleration constants take similar values in both CPSO and DPSO. As in (2), the result of position–position is multiplied by $c_i \times r_i$. In CPSO, c_i is often set to 2 and r_i is a random number between 0 and 1 (recall Section II-C). In DPSO, we want the value of final probability to have a range between 0 and 1 after multiplying by c_i. Setting $p(s)$ to 0.5 puts the result in [0, 1] when $c_i = 2$.

For example, suppose that $x_i(k) = (0, 0, 0, 0)$, $pbest_i = (0, 0, 1, 1)$, and $gbest = (0, 2, 2, 2)$ as before. We can get $pbest_i - x_i(k) = \{((0, –, 1, –, –), 0.5), ((0, –, –, 1), 0.5), ((–, 0, 1, –), 0.5), ((–, 0, –, 1), 0.5), ((–, –, 1, 1), 0.5)\}$. Fig. 1(b) and (c) shows the results for $2 \times r_1 \times (pbest_i - x_i(k))$ and $2 \times r_2 \times (gbest - x_i(k))$ respectively.

3) Velocity + Velocity: The addition of velocities gives a particle's movement path. The plus operator results in the union of two velocities.

Definition 8 (Velocity + Velocity): Let v_1 and v_2 be two velocities. Then $v_1 + v_2 = \{(s, p(s)) |$ if $(s, p_1(s)) \in v_1$ and $(s, p_2(s)) \in v_2$, $p(s) = p_1(s) + p_2(s)$; if $(s, p_i(s)) \in v_1$ and $(s, p_i(s)) \notin v_2$ or $(s, p_i(s)) \in v_2$ and $(s, p_i(s)) \notin v_1$, $p(s) = p_i(s)\}$. (If $p_1(s) + p_2(s) > 1$, $p_1(s) + p_2(s) = 1$.)

For example, Fig. 1(d) shows the results for p_i's new velocity $v_i(k+1)$ after updating velocity.

Algorithm 3 Position Updating

```
Input: position x_i(k), velocity v_i(k+1), pro_2 and pro_3
Output: new position x_i(k+1)
x_i(k+1) = (–, –, ..., –)
Sort v_i(k+1) in descending order of p(s)
for each pair (s_i,p(s_i)) in v_i(k+1) do
    Generate a random number α ∈ [0, 1]
    if α < p(s_i) then
        for each fixed level ℓ in s_i do
            Generate a random number β ∈ [0, 1]
            if β < pro_2 and the corresponding factor
            of ℓ has not been fixed in x_i(k+1) then
                Update x_i(k+1) with ℓ
            end if
        end for
    end if
end for
if x_i(k+1) has unfixed factors then
    Fill these factors by the same levels of previous
    position x_i(k)
end if
Generate a random number γ ∈ [0, 1]
if γ < pro_3 then
    randomly change the level of one factor of x_i(k+1)
end if
return x_i(k+1)
```

In the velocity-updating phase of DPSO (2), we introduce a new parameter pro_1 to control the size of the final velocity set. If $p(s_i) < pro_1$, the pair $(s_i, p(s_i))$ is removed from the final velocity. For example, if we set $pro_1 = 0.5$, $v_i(k+1)$ is removed as shown in Fig. 1(e). Here, the velocity has been sorted in descending order of $p(s)$, where if several $p(s)$ have the same value, they are in an arbitrary order. If $v_i(k+1)$ becomes empty, it stays empty until new pairs are added to it. As long as the velocity is empty, the particle's position is not updated and no better solutions can be found from this particle. In Section IV-C1, we discuss how to reinitialize this particle.

4) Position+Velocity: Position plus velocity is the position updating phase. Algorithm 3 gives the pseudo code of this procedure. Here, two new parameters, pro_2 and pro_3, numbers in the range [0, 1], are introduced. pro_2 is used to determine

the probability of selecting each fixed level in schema s and pro_3 is a mutation probability to make the particle mutate randomly.

An example helps to describe this procedure. We already have p_i's current position $x_i(k) = (0, 0, 0, 0)$, and its updated velocity $v_i(k+1)$ in descending order of $p(s)$ as shown in Fig. 1(e). We also assume that pro_2 and pro_3 are both set to 0.5. Each schema s_i here is selected to update the position with probability $p(s_i)$. For the first pair ((0, –, –, 2), 1.0), suppose that the random number α satisfies $\alpha < 1.0$. Then, for each fixed level of this pair, namely level 0 of the first factor and level 2 of the fourth factor, its corresponding factor has not been fixed in $x_i(k+1)$. Suppose that we have the first $\beta < 0.5$ but the second $\beta > 0.5$, the first factor will be selected to update the position and the second factor will not. So the new position becomes (0, –, –, –). For the second pair, we regenerate the random number α, and compare it with the probability 0.9. If $\alpha < 0.9$, the second pair is selected. If we generate $\beta < 0.5$ in two rounds, the new position becomes (0, 2, 2, –). Accordingly, if the third pair is selected, and its second factor's level 0 is chosen to update, it does not change position because this factor has been set to a fixed level 2. This procedure is repeated until all factors in the new position $x_i(k+1)$ are set to fixed levels. If all pairs in velocity have been considered, unfixed factors of $x_i(k+1)$ are filled by the same levels of previous position $x_i(k)$. For example, after finishing the For loop in line 15, if the fourth factor of $x_i(k+1)$ has not been given any level, the fourth factor of $x_i(k)$ is used to update it. Then $x_i(k+1)$ becomes (0, 2, 2, 0).

C. Auxiliary Strategies

Auxiliary strategies can be introduced to improve the performance of PSO. Two of them are added to DPSO: 1) particle reinitialization to make the search more effective and 2) additional evaluation of *gbest* to improve the selection of the global best test case.

1) Particle Reinitialization: The PSO algorithm starts with a random distribution of particles, which finally converge. Then the best position that has been found is returned. The swarm may jump out of a local optimum, but this can not be guaranteed because CPSO lacks specific strategies for this. When applying PSO for covering array generation, increasing the number of iterations does not improve the ability to escape a local optimum. Hence, particle reinitialization, a widely used method, is employed to help DPSO to jump out of the local optimum.

The main issue with particle reinitialization is when to reinitialize the particle. We may select a threshold, so that the reinitialization is done when the number of iterations exceeds the threshold. In DPSO, a better method can be applied. Using the new representation for velocity, when the current particle p_i's position equals its individual best position $pbest_i$ and global best position *gbest*, the size (norm) of p_i's velocity reduces gradually, because no pairs are generated from $(pbest_i - x_i)$ and $(gbest - x_i)$, and the original pairs in velocity are removed gradually under the influence of $\omega \times v$ (reduce the $p(s)$ of original pairs) and parameter pro_1. After a few fluctuations around *gbest*, the particle may stay at *gbest*, and its size (norm) of velocity is zero. We can use this scenario to trigger reinitialization of the particle. When the reinitialization is done, each dimension of a particle's position is randomly assigned a valid value, and its velocity is regenerated as in the initialization of the swarm.

TABLE V
TWO DIFFERENT CONSTRUCTIONS OF $CA(N; 2, 3^4)$

#	Construction 1					Construction 2			
1	0	0	0	0		0	0	0	0
2	1	1	1	1		0	1	1	1
3	2	2	2	2		0	2	2	2
4	-	-	-	-		1	0	1	2

2) Additional Evaluation of gbest: Current PSO methods to generate covering arrays generally employ the fitness function in Definition 4. This fitness function only focuses on the current candidate, and does not consider the partial test suite *TS*.

Consider the generation of $CA(N; 2, 3^4)$. Table V shows two different construction processes. Both constructions generate (0, 0, 0, 0) as the first test case. Then they choose different test cases, but each of the first three reaches the largest number of newly covered combinations, $C_4^2 = 6$. The difference between these two constructions emerges when generating the fourth test case. In Construction 1, because the combinations with the same level between any two factors have all been covered, we cannot find a new test case that can still cover six combinations. However, in Construction 2, such a new test case can be found, (1, 0, 1, 2). Because the minimum size of $CA(N; 2, 3^4)$ is 9, each test case is required to cover six new combinations. Thus, Construction 1 cannot generate the minimum test suite, but Construction 2 can.

In general, there may exist multiple test cases with the same highest fitness value, which make them equally qualified to be *gbest* in Algorithm 2. Instead of arbitrarily selecting one as *gbest*, it is better to apply additional distance metric to select one among them. As shown in Table V, if the new test case is similar to the existing tests [as (0, 1, 1, 1) is closer to (0, 0, 0, 0) than (1, 1, 1, 1)], there may be a better chance to find test cases with larger fitness subsequently.

In order to measure the "similarity" between a test case t and an existing test suite *TS*, we use the average Hamming distance. The Hamming distance d_{12} indicates the number of factors that have different levels between two test cases t_1 and t_2. Hence, the similarity between t and *TS* can be defined by the average Hamming distance

$$H(t, TS) = \frac{1}{|TS|} \sum_{k \in TS} d_{tk}. \tag{4}$$

The new *gbest* selection strategy selects the best candidate *gbest* based on two metrics. A test case that has the minimum average Hamming distance (i.e., the maximum similarity) from the set of test cases that all have the same highest fitness value is selected as *gbest* in Algorithm 2. For example, when we generate the second test case for $CA(N; 2, 3^4)$ in Table V, although (1, 1, 1, 1) and (0, 1, 1, 1) both have the highest fitness value 6, (0, 1, 1, 1) is selected as *gbest*

TABLE VI
COMPARING PERFORMANCE BETWEEN CPSO AND DPSO

CA	CPSO[1]				CPSO[2]				DPSO				t-test between
	best	worst	mean	time(ms)	best	worst	mean	time(ms)	best	worst	mean	time(ms)	CPSO[2] and DPSO
CA_1	64	68	65.93	4	62	65	63.13	41	59	62	60.97	41	**0.0000**
CA_2	227	239	233.97	4	225	232	228.8	41	224	231	227.37	41	**0.0018**
CA_3	199	208	204.37	18	195	205	200.2	129	194	201	197.3	129	**0.0000**
CA_4	43	47	45.07	2	42	48	44.32	29	40	45	42.73	29	**0.0000**
VCA	29	37	33.83	8	28	37	33.73	100	27	36	31.63	100	**0.0019**

because its average Hamming distance, 3, is smaller than that of (1, 1, 1, 1), 4. We expect that this additional evaluation can enhance the probability of generating a smaller test suite. In addition, because we still want to make the particle follow the conventional search behavior on the individual best direction, this additional distance metric will not be used in updating the *pbest* of DPSO.

V. EVALUATION AND PARAMETER TUNING

In this section, we first evaluate the effectiveness of DPSO in some representative cases, and compare the results against CPSO. Then the optimal parameter settings for both CPSO and DPSO are explored. The goal of evaluation and parameter tuning is to make the size of generated covering array as small as possible. Five representative cases of covering arrays, listed below, are selected for our experiments

$$CA_1(N; 2, 6^{10}) \quad CA_2(N; 3, 5^7) \quad CA_3(N; 4, 3^9)$$
$$CA_4(N; 2, 4^3 5^3 6^2) \quad VCA(N; 2, 3^{15}, CA(3, 3^4)).$$

We consider four independent parameters, iteration number (*iter*), population size (*size*), inertia weight (ω), and acceleration constant (c), which play similar roles in both CPSO and DPSO, and three new parameters for DPSO, pro_1, pro_2, and pro_3. We carry out a base choice experiment to study the impact of various values of these parameters on CPSO and DPSO's performance and find the recommended settings for them. First, a base setting is chosen as the basic configuration. Then the value of each parameter is changed to create a series of configurations, while leaving the other parameters unchanged. Initially, we set $iter = 50$, $size = 20$, $\omega = 0.9$, $c = 1.3$, and $pro_1 = pro_2 = pro_3 = 0.5$ as a basic configuration based on the previous studies [13], [16] and our empirical experience. To obtain statistically significant results, the generation of each case of covering array is executed 30 times.

A. Evaluation of DPSO

We compare the performance between CPSO and DPSO with the basic configuration. Five classes of covering arrays are generated by these two algorithms. The sizes obtained and average execution time per test case are shown as CPSO[1] and DPSO in Table VI. The best and mean array sizes of DPSO are all better than those of CPSO[1]. The new discrete representation scheme and auxiliary strategies improve PSO for covering array generation.

DPSO can produce smaller covering arrays than CPSO with the basic configuration. However, DPSO spends more time during the evolution, because its new operators for updating are more intricate than the conventional ones. DPSO needs to deal with many elements of the velocity set, whereas the conventional scheme only needs simple arithmetic operations. To compare the performance between CPSO and DPSO given the same execution time, for each case we let the execution time per test case for CPSO equal to that for DPSO, so that CPSO can spend more time in searching. We refer to this version of CPSO as CPSO[2]. In addition, a t-test between CPSO[2] and DPSO is conducted and the corresponding p-value is presented. A p-value smaller than 0.05 indicates that the performance of these two algorithms is statistically different with 95% confidence.

From CPSO[1], CPSO[2], and DPSO in Table VI, the performance of CPSO is improved with more search time. DPSO, which must use fewer iterations, still works better than CPSO. The effectiveness of DPSO comes from the essential improvement of its new representation scheme for velocity and the auxiliary strategies. The results of the t-test demonstrate the significance of these differences. Therefore, we can conclude that DPSO performs better than CPSO with fewer iterations for covering array generation.

B. Parameter Tuning

In the base choice experiment, because the sizes of the various covering arrays differ, the mean size of each case (s_i) obtained is normalized using

$$s_i = \frac{s_i - s_{\min}}{s_{\max} - s_{\min}}.$$

This normalization enables the graphical representation of different cases on a common scale.

Because some parameters may not significantly impact the performance, we use ANOVA (significance level = 0.05) to test whether there exist significant differences among the mean results obtained by different parameter settings. When changing the parameter settings have no significant impact on the generation results, these results will be presented as dotted lines in the corresponding figures. For example, CA_4 is presented as a dotted line in Fig. 2(b). It means that the iteration number does not significantly impact the generation of this case of covering array, and so this case will not be further considered when identifying the optimal settings.

1) Iteration Number (iter): Iteration number determines the number of updates. PSO typically requires thousands of iterations to evolve. Nevertheless, only a few iterations are required to generate covering array according to [14]–[16]. So we

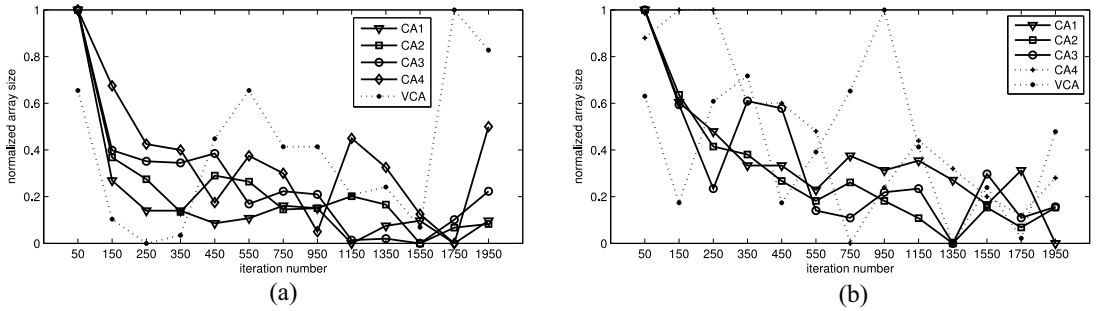

Fig. 2. Comparing array sizes under different iteration numbers. (a) CPSO. (b) DPSO.

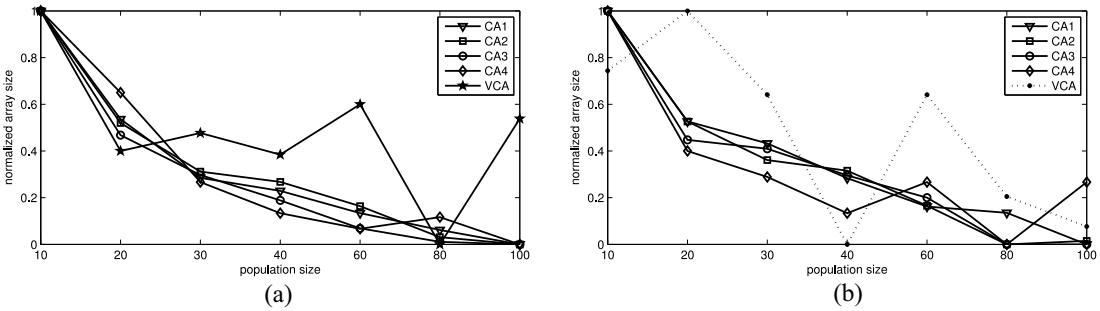

Fig. 3. Comparing array sizes under different population sizes. (a) CPSO. (b) DPSO.

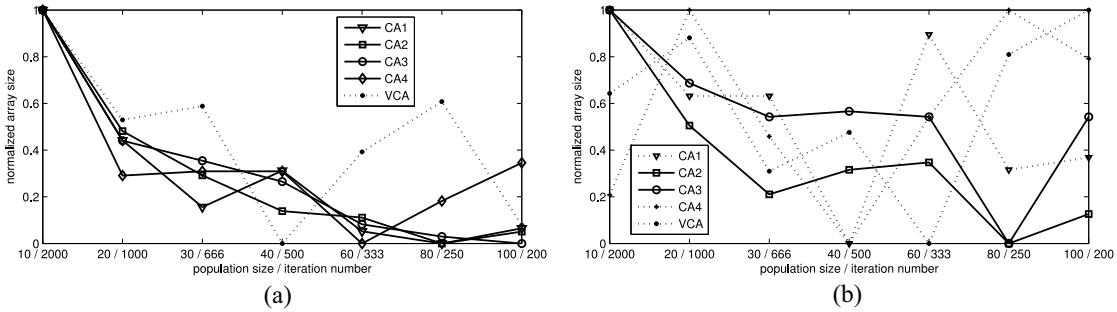

Fig. 4. Comparing array sizes under different settings of population size and iteration number. (a) CPSO. (b) DPSO.

change its value from 50, the base setting, to 1950 incrementally. Much larger values are not used because the execution time may increase markedly without a commensurate increase in the quality of the results.

Fig. 2 shows the results for different choices of iteration numbers, where each line represents a covering array. Performance is improved with increasing iterations for both CPSO and DPSO. A small number of iterations may not be appropriate due to insufficient searching. Because the optimal settings are different among different cases, and several settings can be chosen to generate the minimum covering arrays, it is open to debate which setting is the "best" one. Given time constraints, for a population size of 20, a good setting of iteration number could be approximately 1000 for both CPSO and DPSO.

2) Population Size (Size): Population size determines the initial search space. Generally a small value, e.g., 20, is satisfactory for most cases. A large population size may bring a higher diversity and find a better solution, but it also increases the evolving time. For the same reason as before, we change the value of population size from 10 to 100.

Fig. 3 shows the results for different choices of population size. There is no doubt that the mean array size obtained decreases as population size increases. A large population size can have more chances to generate smaller covering arrays, but its execution time can become prohibitive.

In addition, because iteration number and population size together determine the search effort of PSO, we explore the combinations between these two parameters. We let $iter \times size$ be a constant 20 000, and generate each case under different settings of these two parameters. Fig. 4 shows the results, where PSO prefers a relatively larger population size. In CPSO, ten particles with 2000 iterations is the worst setting, and most cases produce good results with 60 or 80 particles. In DPSO, the best choice of population size is still 80. In both CPSO and DPSO, the largest population size 100 cannot produce as good results as that of 80 due to having fewer iterations. So achieving smallest covering arrays requires a good balance between these two parameters, and a moderately large population size is necessary for both CPSO and DPSO. Thus, we can set iteration number to 250, and population size to 80, as the recommended settings for both CPSO and DPSO.

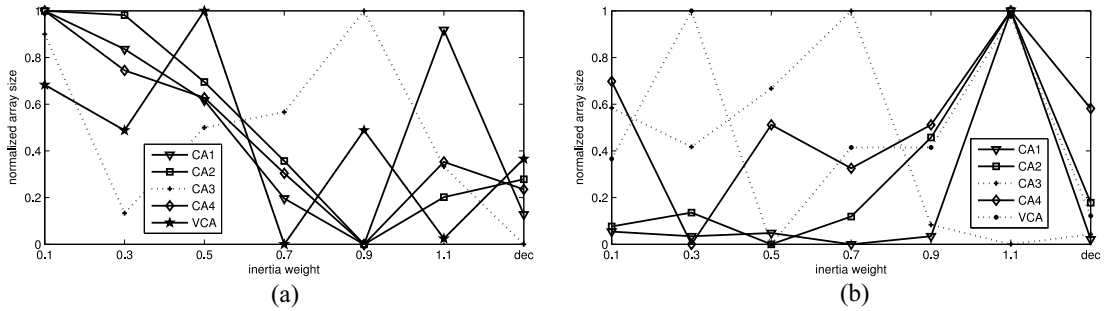

Fig. 5. Comparing array sizes under different inertia weights. (a) CPSO. (b) DPSO.

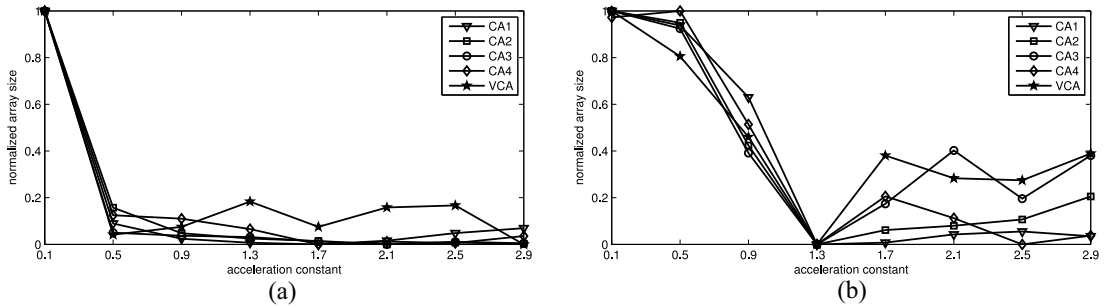

Fig. 6. Comparing array sizes under different acceleration constants. (a) CPSO. (b) DPSO.

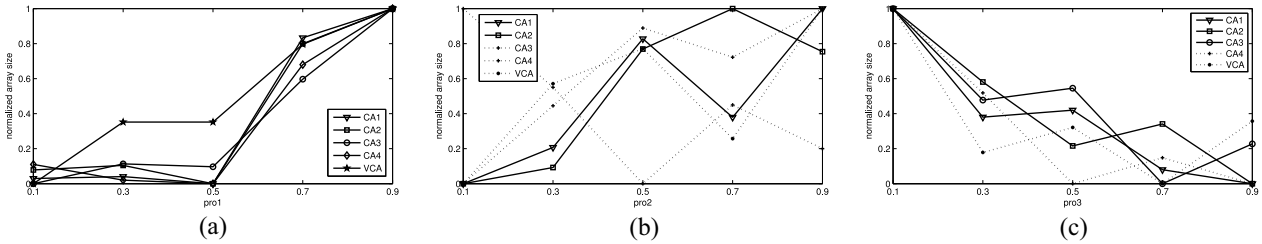

Fig. 7. Comparing array sizes under different settings of (a) pro_1, (b) pro_2, and (c) pro_3.

3) Inertia Weight (ω): Inertia weight determines the tendency of the particle to continue in the same direction. A small value help the particle move primarily toward the best position, while a large one is helpful to continue its previous movement. A linearly decreasing value is also used as it can make the swarm gradually narrow the search space. Here, we investigate both fixed values and a linearly decreasing value from 0.9 to 0.4 over the whole evolution (presented as "*dec*").

Fig. 5 shows the results for different choices of inertia weight. In CPSO, most of the smallest covering arrays are generated by large fixed inertia weights. The decreasing value does not perform as well as the large values, such as 0.9 for CA_1, CA_2, and CA_4. So 0.9 can be the recommended choice for CPSO. In DPSO, 1.1 is the worst choice because the particle may update its position based on its own velocity and thus fail to learn from individual and global best positions. The decreasing value can perform reasonably well, but a fixed value 0.5 may be a better choice in that it keeps the effort of global search moderate. Thus, we can recommend the fixed inertia weight of 0.9 for CPSO, and 0.5 for DPSO.

4) Acceleration Constant (c): Acceleration Constants c_1 and c_2 control the learning from *pbest* and *gbest*, respectively. Generally, they are set to the same value to balance the influence of these two positions. Setting c_1 and c_2 to a large value may make the particle more likely attracted to the best position ever found, while a small one may make the search far from the current optimal region. In this paper, we set $c_1 = c_2 = c$, and vary c from 0.1 to 2.9.

Fig. 6 shows the results for different choices of acceleration constant. Unlike other parameters, there is a consistent trend in all five cases. In CPSO, the values larger than 0.5 all produce good results. In DPSO, 1.3 can definitely be regarded as the optimal value. Thus, we set 1.3 as the recommended value of acceleration constant for both CPSO and DPSO.

5) pro_1, pro_2, and pro_3: These three parameters are new to DPSO. Parameter pro_1 determines the size of the final velocity. As those pairs whose probabilities are smaller than pro_1 are removed from the final velocity set, a small value may keep more pairs in the velocity for the next evolution, but it

TABLE VII
RECOMMENDED PARAMETER SETTINGS

	$iter$	$size$	ω	c	pro_1	pro_2	pro_3
CPSO	250	80	0.9	1.3	–	–	–
DPSO	250	80	0.5	1.3	0.5	0.3	0.7

TABLE VIII
SIZES(N) OF COVERING ARRAYS FOR n FACTORS, EACH HAVING THREE LEVELS, WITH COVERING STRENGTH t, $CA(N; t, n, 3)$

t	n	**Reported**[16]		**CPSO**		**DPSO**		**TVAC**		**CLPSO**		**APSO**		**DMS-PSO**	
		best	mean	best	mean	best	mean	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d
2	4	9	10.15	9	10.13	**9**	**9**	11.2	**9**	10.4	**9**	10.5	**9**	10.3	**9**
	5	12	13.81	11	13.07	**11**	**11.53**	13.8	**11.3**	13.2	**11.4**	13.3	**11.3**	13	**11.8**
	6	13	15.11	14	14.93	**14**	**14.5**	15.4	**14.6**	14.7	14.5	14.7	14.4	14.5	14.3
	7	15	16.94	15	15.47	**15**	**15.17**	17	**15.1**	15.2	15	15.6	**15.1**	15.3	15.2
	8	15	17.57	15	15.93	15	16	18.2	**15.7**	15.7	15.7	16.5	**15.8**	15.8	15.9
	9	17	19.38	16	16.63	15	16.43	19.2	**16.3**	16.7	**16.3**	17	**16.1**	16.7	**16.4**
	10	17	19.78	16	17.7	16	17.3	20.5	**17.3**	17.4	17.4	17.7	**17.2**	17.6	**17.1**
	11	17	20.16	16	17.9	17	17.7	19.7	**17.7**	17.8	18	18.4	**17.6**	18.1	**17.6**
	12	18	21.34	17	18.6	**16**	**17.93**	22.7	**17.9**	18.4	18.5	18.8	**18.1**	18.4	**17.8**
3	5	39	41.37	**38**	**40.23**	41	43.17	**40.9**	43.4	**40**	43.3	**40.7**	43.3	**40.4**	43.1
	6	45	46.76	42	45.53	**33**	**38.3**	46.7	**37.8**	45.6	**37.7**	46.5	**37.8**	45.7	**37.9**
	7	50	52.2	49	51.13	**48**	**50.43**	57.3	**50.5**	50.5	50.5	51	50.5	51	**50.4**
	8	54	56.76	53	55.27	**52**	**53.83**	60.2	**52.9**	54.2	**53.2**	57	**53.7**	54.9	**53.6**
	9	58	60.3	58	59.4	**56**	**57.77**	59.1	**57.7**	58.2	57.7	61.5	**58**	58.2	**57.7**
	10	62	63.95	61	62.67	**59**	**60.87**	75.7	**60.8**	61.8	61.93	64.4	**61**	61.5	**61**
	11	64	65.68	63	65.6	**63**	**63.97**	75.4	**63.6**	**65**	65.7	68	**64.1**	64.1	**63.4**
	12	67	68.23	68	68.97	**65**	**66.83**	86.1	**66.2**	**67.7**	69.5	71.3	**66.9**	67.1	**66.1**
4	6	133	135.31	132	135.33	131	134.77	135.8	**134.6**	135.2	134.5	135.7	**134.3**	134.7	134.9
	7	155	158.12	153	157.47	**150**	**155.23**	171.9	**154.6**	157.6	**154.2**	157.5	**153.8**	157.2	**155**
	8	175	176.94	174	177.77	**171**	**175.6**	201.7	**174.3**	176.3	**174.9**	182.2	**175.3**	176.8	**174.8**
	9	195	198.72	191	195.47	**187**	**192.27**	204.8	**188.6**	193.9	**189.6**	200.1	**192.1**	194	**185.4**
	10	210	212.71	211	213.37	**206**	**219.07**	233.9	**208**	210.3	210.4	221.8	**1209.7**	209.6	**208.4**
	11	222	226.59	226	229.3	**221**	**224.27**	246.6	**223.5**	**226.4**	227.8	235.2	**224.5**	224.9	**223.5**
	12	244	248.97	242	244.23	**237**	**239.83**	267.2	**238**	**242.3**	246.8	249.9	**239.4**	239.4	**237**

also requires more computation time. Parameter pro_2 determines the probability of selecting each fixed level from each schema when updating positions. A larger value may lead to a quick construction of new position, but it may also lead to fast convergence toward a local optimum. Parameter pro_3 determines the mutation probability when updating positions. A larger value may enhance the randomness, but it also lowers the convergence speed. In this paper, values from 0.1 to 1.0 for these three parameters are investigated.

Fig. 7 shows the results. For pro_1, a large value is not appropriate because it removes nearly all pairs from the final velocity set. A medium value 0.5, which appears to lead to the best result, may be the best choice. For pro_2, the smallest value 0.1 yields the best results. Slower construction of the new position may slow the convergence toward a local optimum, but it also takes longer. So, in order to balance performance and execution time, we take 0.3 as the recommended value for pro_2. For pro_3, a larger value may be a good choice. The frequent mutation of new position may bring better results, but also takes longer to converge. So, we take 0.7 as the recommended value for pro_3.

In summary, the recommended parameter settings for PSO for covering array generation are different from previously suggested ones [13], [16]. Some parameters may significantly impact the performance in some cases, and parameter tuning is necessary to enhance heuristic techniques for particular applications. Naturally, the optimal settings vary for different cases of covering arrays. There may not exist a common setting that can always lead to the best results. Because it is impractical to tune parameters for each case in real life, we suggest two general settings for CPSO and DPSO, as shown in Table VII, which can typically lead to better performance within reasonable execution time.

VI. COMPARING AMONG PSOS

In this section, we compare the best reported array sizes generated by PSO in [16] with our findings for CPSO, DPSO, and four representative variants. Because the research in [16] demonstrated that their generation results typically outperform greedy algorithms, in this paper, we do not compare CPSO and DPSO with greedy algorithms.

We implement both the original and discrete versions of four variants (TVAC [24], CLPSO [25], APSO [26], and DMS-PSO [27]) to generate covering arrays. Their discrete versions are extended based on new representation scheme of velocity and auxiliary strategies. We name them D-TVAC, D-CLPSO, D-APSO, and D-DMS-PSO, respectively. In CLPSO, a particle can learn from different particles' *pbest* in each dimension whereas we do not distinguish dimensions strictly in DPSO. So in D-CLPSO, a particle can fully learn from different particles' *pbest* in all dimensions. That may weaken the search ability of CLPSO. For the other three variants, they can be directly extended based on DPSO.

All algorithms are compared using the same number of fitness function evaluations. CPSO and DPSO use the recommended settings shown in Table VII. For the variants, *iter*, *size* and pro_1, pro_2 and pro_3 for discrete versions are set to

TABLE IX
SIZES(N) OF COVERING ARRAYS FOR SEVEN FACTORS, EACH HAVING ℓ LEVELS, WITH COVERING STRENGTH t, $CA(N; t, 7, \ell)$

t	ℓ	**Reported**[16]		**CPSO**		**DPSO**		**TVAC**		**CLPSO**		**APSO**		**DMS-PSO**	
		best	mean	best	mean	best	mean	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d
2	2	6	6.82	7	7.5	**7**	**7**	7.5	**7**	7.5	**7**	7.7	**7**	7.6	**7**
	3	15	15.23	15	15.27	**14**	**15**	17	**15.1**	15.3	15.3	15.5	15.1	15.2	15.2
	4	26	27.22	25	25.77	24	25.33	30.5	**25.8**	25.9	25.6	26.2	**25.5**	26.2	25.4
	5	37	38.14	36	37.97	**34**	**35.47**	47.8	**35.6**	37.5	36.9	39.9	**36.1**	37.4	**35.5**
	6	–	–	51	52.83	**47**	**49.23**	70.2	**48.8**	52	52	56.2	**49.6**	52	**49**
	7	–	–	68	69.97	**64**	**66.37**	95.9	**65.7**	69.9	70.6	74.9	**65.7**	69.2	**65.6**
3	2	13	13.61	12	14.63	15	15.06	15.1	15.1	14.9	15	14.5	15.2	**14.8**	15.2
	3	50	51.75	50	51.2	**49**	**50.6**	57.4	**50.6**	50.6	50.7	51.8	50.9	50.4	50.9
	4	116	118.13	115	117.97	**112**	**115.27**	141.2	**115**	116.4	115.6	123.9	**115.6**	116.9	**115.2**
	5	225	227.21	221	224.77	**216**	**219.2**	280	**217.8**	221.7	**221**	225.1	**219.6**	222.1	**217.5**
	6	–	–	376	380.93	**365**	**370.57**	488	**368.1**	376.5	375.7	411	**370.9**	375.8	**368.3**
	7	–	–	591	595.4	**574**	**577.67**	782.3	**574.2**	590	591.5	629.3	**579.6**	590.4	**574.3**
4	2	29	31.49	**24**	**31.2**	34	34	**30.7**	34	**31.3**	34	**30.9**	34	**31.1**	34
	3	155	157.77	155	158.3	**150**	**154.73**	159.50	**154.1**	157	155.1	158	**154.1**	157.4	**154**
	4	487	489.91	487	491.83	**472**	**481.53**	563.6	**480.8**	489.1	**480.9**	499.8	**482.8**	487.7	**479.4**
	5	1176	1180.63	1167	1175.17	**1148**	**1155.63**	1201.1	**1153.7**	1168.7	**1155.8**	1219	**1155.1**	1168.7	**1154.5**
	6	–	–	2386	2394.6	**2341**	**2357.73**	2946.5	**2352**	2379.8	**2358.2**	2584.9	**2361**	2380.8	**2351.1**
	7	–	–	4350	4366.2	**4290**	**4309.6**	5501.1	**4286.6**	4340.9	**4307**	4773.3	**4307.4**	4345.1	**4287.4**

their recommended values. ω and c are also set to recommended values unless they are adaptively adjusted during the evolution, in which case their range is set to $[0.4, 0.9]$ and $[0.8, 1.8]$, respectively. The new control parameters for these variants follow their suggested settings.

Tables VIII–XIII give the results. Because of the execution time, we only consider covering strengths from 2 to 4, and the generation of each covering array is repeated 30 times. A t-test (significance level = 0.05) is also conducted to test whether there exists a significant difference between the mean sizes produced by the two algorithms. In the first three columns, we report the best and mean array sizes obtained from previous results, CPSO and DPSO, where boldface numbers indicate that the difference between CPSO and DPSO is significant based on the t-test. In the last four columns, we report the mean array sizes from the original and discrete versions of each PSO variant (presented as mean^c and mean^d respectively), where boldface numbers indicate that the difference between mean^c and mean^d of each variant is significant.

A. Uniform Covering Arrays

Tables VIII–X present the results for uniform covering arrays. We extend the cases considered in [16], where "–" indicates the not available cases. In Tables VIII, we report array sizes for n factors, each having three levels. In Tables IX and X, we report array sizes for 7 and 10 factors, each having ℓ levels. Their covering strengths all range from 2 to 4.

Typically, CPSO can produce smaller sizes than those reported in [16], demonstrating the effectiveness of parameter tuning. Furthermore, DPSO can produce the smallest best and mean sizes in almost all cases, and usually the performance of DPSO is significantly better than CPSO. When the covering strength $t = 2$ is adopted (Table VIII), DPSO does not generally outperform CPSO. But when covering strength increases, DPSO performs better. Sometimes the *mean* sizes for DPSO are smaller than the *best* sizes for CPSO. Because generating a covering array with higher covering strength is more complicated, DPSO may be more appropriate for generating such covering arrays. In Tables IX and X, we also find that DPSO is superior for covering arrays with high covering strength. Overall, these results show the effectiveness of DPSO for uniform covering array generation.

Surprisingly, DPSO does not beat previous results for CPSO when covering arrays have two levels for each factor (Table IX). This appears to be a weakness of DPSO. Although DPSO generates smaller covering arrays than previous results and CPSO, other techniques may still yield better results than DPSO (some best known sizes can be found in [38]).

B. VS Covering Array

Tables XI–XIII give the results for VS covering arrays. Based on $CA(N; 2, 3^{15})$, $CA(N; 3, 3^{15})$, and $CA(N; 2, 4^2 5^2 6^3)$, some different cases of sub covering arrays conducted in [16] are examined. Their covering strengths are at most 4.

Generally, we can draw similar conclusions as for uniform covering arrays. CPSO with the suggested parameter setting can produce better results than reported sizes in some cases. DPSO also usually beats them on the best and mean sizes. In Table XI, often the difference between CPSO and DPSO is not significant. In part this is because for the $CA(3, 3^3)$, $CA(3, 3^3)^2$, $CA(3, 3^3)^3$, and $CA(4, 3^4)$, the same best results are provably the minimum (e.g., the minimum size of the $CA(3, 3^3)$ is $3 \times 3 \times 3 = 27$). For the other cases, although sometimes the difference is not significant, DPSO can still generate smaller covering arrays. DPSO remains a good choice for generating VS covering arrays. In Tables XII and XIII, similar results can be found.

TABLE X
SIZES(N) OF COVERING ARRAYS FOR TEN FACTORS, EACH HAVING ℓ LEVELS, WITH COVERING STRENGTH t, $CA(N; t, 10, \ell)$

t	ℓ	**Reported**[16]		**CPSO**		**DPSO**		**TVAC**		**CLPSO**		**APSO**		**DMS-PSO**	
		best	mean	best	mean	best	mean	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d
2	4	–	–	29	29.63	**28**	**29.2**	37.6	**29**	**29.2**	30.1	30.8	**29.1**	29.2	29.2
	5	45	48.31	43	44.33	**42**	**43.67**	59.3	**43.5**	**44.3**	46.4	46.6	**43.3**	44	43.5
	6	–	–	60	61.87	**58**	**59.23**	86	**59.3**	**62.6**	65.9	65.6	**59.5**	61.4	**59**
3	4	–	–	146	147.83	**141**	**143.7**	188	**142.6**	**144.4**	149.9	154.9	**144.1**	144.8	**142.6**
	5	287	298	281	283.9	**273**	**276.2**	376.6	**274.6**	**282.1**	295.6	302.2	**276.7**	280.3	**273.1**
	6	–	–	478	483.87	**467**	**470.5**	659.7	**467.8**	**489**	512.4	522	**470.1**	488.8	**464.8**
4	4	–	–	676	678.7	**664**	**667**	845	**660.4**	**665.4**	680.2	715.8	**667.2**	667.8	**660**
	5	1716	1726.72	1643	1646.8	**1618**	**1620.8**	2125.2	**1608.4**	**1636.6**	1682	1775.2	**1620**	1640.2	**1601.2**
	6	–	–	3392	3403	**3339**	**3342.5**	4475.8	**3318.5**	**3410.8**	3518.2	3666.6	**3345**	3452.4	**3295**

TABLE XI
SIZES(N) OF VS COVERING ARRAYS $VCA(N; 2, 3^{15}, CA)$

CA	**Reported**[16]		**CPSO**		**DPSO**		**TVAC**		**CLPSO**		**APSO**		**DMS-PSO**	
	best	mean	best	mean	best	mean	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d
$\emptyset$	19	20.92	19	20.07	**18**	**18.63**	24.6	**18.7**	20.1	20.2	20.8	**18.9**	19.7	**18.7**
$CA(3, 3^3)$	27	27.5	27	27.47	27	27.27	30.5	**27.3**	**27.4**	27.9	27.8	27.5	27.5	27.6
$CA(3, 3^3)^2$	27	27.94	27	27.93	27	27.83	32.9	**28**	**27.8**	28.3	28.9	**27.7**	27.9	28
$CA(3, 3^3)^3$	27	28.13	27	28.13	27	28	35.1	**28.2**	28.2	28.5	29.7	**28**	27.9	28.1
$CA(3, 3^4)$	30	31.47	29	33.13	**27**	**31.43**	37	**31.7**	32.7	32.3	32.9	**31.5**	32	30.7
$CA(3, 3^5)$	38	39.83	39	41.23	38	40.93	44.4	**40.9**	41.6	41	41.6	41.2	41.4	41.8
$CA(3, 3^6)$	45	46.42	44	46.27	43	45.7	51.1	**45.5**	46.4	45.8	47	**45.7**	45.4	45.6
$CA(3, 3^7)$	49	51.68	49	50.97	**47**	**49.87**	58	**49.4**	50.2	50.7	53	**50.2**	50.2	49.8
$CA(4, 3^4)$	81	82.21	81	81.07	81	81.03	82.4	**81**	81	81	82.1	**81.03**	81	81
$CA(4, 3^5)$	97	99.31	97	101.03	**85**	**94.5**	104.2	**95.5**	100.7	100.7	100.7	**95.2**	99.7	**96.2**
$CA(4, 3^7)$	158	160.31	154	157.93	152	156.83	171.9	**155.9**	156.7	157.4	161.7	**156.5**	156.6	156.6

TABLE XII
SIZES(N) OF VS COVERING ARRAYS $VCA(N; 3, 3^{15}, CA)$

CA	**Reported**[16]		**CPSO**		**DPSO**		**TVAC**		**CLPSO**		**APSO**		**DMS-PSO**	
	best	mean	best	mean	best	mean	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d
$\emptyset$	75	78.69	75	76.73	**72**	**73.97**	93.3	**73.5**	**77.4**	78	79.7	**74.4**	74.7	**73.4**
$CA(4, 3^4)$	91	91.8	86	91.45	**86**	**89.83**	110.5	**90.3**	**91.8**	93.7	92.5	**89**	89.2	88.2
$CA(4, 3^4)^2$	91	92.21	90	92.93	**88**	**90.77**	118.1	**91.2**	**93.4**	96.3	97.2	**90.3**	91	90.3
$CA(4, 3^5)$	114	117.3	108	114.03	**107**	**111.17**	128.8	**110.5**	113.7	114.9	114.4	110.7	112.6	111.6
$CA(4, 3^7)$	159	162.23	156	159.4	152	158.57	177.3	**158.2**	159	159.1	165.1	**159.4**	159.5	**157.6**
$CA(4, 3^9)$	195	199.28	193	196.57	193	196	231.3	**193.5**	195.1	194.8	204.2	**196.3**	194.1	193.3
$CA(4, 3^{11})$	226	230.64	227	229.3	**225**	**227.5**	279	**224.4**	**227.4**	229.8	240.3	**228.1**	225.1	223.7

For both uniform and variable cases of covering arrays, parameter tuning can enhance CPSO to generate smaller covering arrays than previous work. DPSO can produce the smallest covering arrays in nearly all cases, and in general the difference of performance between CPSO and DPSO is significant. DPSO is an effective discrete version of PSO for covering array generation.

C. PSO Variants

In order to further investigate the effectiveness of DPSO for covering array generation, we implement the original versions of four representative variants of PSO and extend them to their discrete versions based on DPSO. We apply them to generate the same cases for covering arrays. The mean sizes obtained are shown in the last four columns in Tables VIII–XIII.

We first compare the mean sizes of original PSO variants with those of CPSO and DPSO. In our experiments, TVAC's mean sizes are always larger than CPSO's. The linear adjustment of inertia weight and acceleration constant is not helpful for CPSO for covering array generation. Typically, APSO's mean sizes are also larger than CPSO's. Because APSO uses a fuzzy system to classify different evolutionary states, its ineffectiveness may result from inappropriate parameter settings. CLPSO and DMS-PSO can outperform CPSO typically. Although on occasion they achieve comparable performance with DPSO, they cannot perform as well as DPSO in most

TABLE XIII
SIZES(N) OF VS COVERING ARRAYS $VCA(N; 2, 4^3 5^3 6^2, CA)$

CA	**Reported**[16]		**CPSO**		**DPSO**		**TVAC**		**CLPSO**		**APSO**		**DMS-PSO**	
	best	mean	best	mean	best	mean	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d	mean^c	mean^d
$\emptyset$	42	43.6	41	43.3	**40**	**42.3**	53.3	**42.9**	43.6	43.4	45	**42.5**	43.5	**42.4**
$CA(3, 4^3)$	64	65.5	64	64.3	**64**	**64**	70.1	**64.1**	64.3	64.4	64.8	**64.1**	64.2	64.1
$CA(3, 4^3 5^2)$	124	126.6	123	127.4	**119**	**124.7**	141.5	**123.8**	127.1	126.6	129.4	**125.5**	127.3	**125.3**
$CA(3, 4^3)$, $CA(3, 5^3)$	125	127.9	125	125.1	125	125	128.1	**125**	125.1	125.1	127.3	**125**	125.1	125
$CA(3, 4^3 5^3 6^1)$	206	210.2	207	212.7	**203**	**207.5**	250.6	**208.1**	209.8	209.5	220.6	**208.8**	209.6	**207.6**
$CA(3, 4^3)$, $CA(4, 5^3 6^1)$	750	755.7	**750**	**750.1**	750	750.8	774.3	**750.1**	750	750	755.4	**751.1**	750	750.1
$CA(4, 4^3 5^2)$	472	478.1	467	475.4	**440**	**450.6**	495.9	**450.2**	475	**463.5**	483	**451.5**	473	450.6

TABLE XIV
COMPARING CPSO AND DPSO WITH GA AND ACO

CA	**CPSO**†			**DPSO**†			**GA [9]**	**ACO [9], [11]**
	best	mean	time(s)	best	mean	time(s)	best	best
$CA_1(N; 2, 3^{13})$	18	19.13	2.67	**17**	19.03	23.07	**17**	**17**
$CA_2(N; 2, 10^{10})$	156	158.1	14.23	**146**	148.77	126.87	157	159
$CA_3(N; 2, 10^{20})$	205	206.9	69.77	**191**	192.83	548.5	227	225
$CA_4(N; 3, 3^6)$	42	45.37	2.57	**33**	37.36	16.9	**33**	**33**
$CA_5(N; 3, 6^6)$	335	339.23	18.6	**321**	326.47	199.57	331	330
$CA_6(N; 3, 10^6)$	1482	1489.6	112.4	**1452**	1456.3	1091.67	1501	1496
$CA_7(N; 3, 5^7)$	223	225.26	21.17	**215**	219.37	201.83	218	218
$CA_8(N; 3, 5^2 4^2 3^2)$	117	123.63	7.3	**100**	103.5	63.1	108	106
$VCA_9(N; 2, 3^{15}, CA(3, 3^5))$	**38**	41.44	1.1	**38**	41.18	9.52	–	**38**
$VCA_{10}(N; 2, 3^{15}, CA(3, 3^6))$	45	47.34	1.48	**37**	45.52	13.94	–	45
$VCA_{11}(N; 2, 3^{15}, CA(3, 3^7))$	49	52.08	2.4	**48**	50.3	11.48	–	**48**
$VCA_{12}(N; 2, 3^{15}, CA(3, 3^9))$	58	60.36	3.94	**56**	58.18	18.4	–	57

† C++, Windows 8, Intel Core 2 Duo 2.66GHz, 2GB RAM

cases. That further demonstrates the effectiveness of DPSO for covering array generation. Because these four algorithms are representative PSO variants (as shown in Table IV), designing different neighborhood topologies and using multiswarm techniques may have potential to improve CPSO for covering array generation.

We next compare the original and discrete versions of each variant. Except for CLPSO, the other three variants can be improved using their discrete versions, and the improvement is also significant. TVAC cannot outperform CPSO, but it is enhanced by DPSO so that D-TVAC can produce smaller mean sizes than CPSO in most cases. The linear adjustment is helpful for the discrete version. For CLPSO, only in a few cases is it improved by DPSO. Sometimes D-CLPSO even leads to worse results (see Table X), due primarily to the weakened search ability of its discrete version as explained in Section VI. For APSO, DPSO can enhance its original version, but D-APSO is still worse than DPSO. That may result from inappropriate settings as explained before. For DMS-PSO, sometimes DPSO does not enhance it (see Table XI). However, in most cases D-DMS-PSO can outperform DMS-PSO and has comparable performance with D-TVAC. The multiswarm strategy is also helpful for the discrete version.

In summary, the comparison study reveals that our suggested parameter settings are more suitable for covering array generation. DPSO is an effective discrete version of PSO. It can significantly outperform previous results and CPSO in nearly all cases for uniform and VS covering arrays. Furthermore, DPSO's representation scheme of a particle's velocity and auxiliary strategies not only enhance CPSO, but also typically enhance PSO variants. DPSO is a promising improvement on PSO for covering array generation.

VII. COMPARING DPSO WITH GA AND ACO

Because GAs and ACO [8]–[12] have also been successfully used for covering array generation, we compare CPSO and DPSO with the reported array sizes in [9] and [11]. There are no widely accepted benchmarks for the comparison of search-based covering array generation, so we only consider these two representative and competitive works.

Shiba *et al.* [9] applied both GA and ACO to generate uniform covering arrays (CA_1 to CA_8 in Table XIV), and Chen *et al.* [11] applied ACO to generate VS covering arrays (VCA_9 to VCA_{12} in Table XIV). They both set algorithm parameters according to recommendations in related research fields without parameter tuning, while our CPSO and DPSO

use the recommended settings shown in Table VII. To compare these algorithms under the same number of fitness evaluations, for each case of covering array, the population size and the number of iterations of CPSO and DPSO are modified accordingly to satisfy the settings in [9] and [11]. Moreover, Shiba *et al*. [9] and Chen *et al*. [11] both applied test minimization algorithms to further reduce the size of generated covering arrays, while our CPSO and DPSO do not apply any minimization algorithms.

Table XIV shows the comparison results, where boldface numbers indicate the best array sizes obtained, and "–" represents that the corresponding data is not available. The generation of each case of covering array of CPSO and DPSO is executed 30 times and the best and average results are presented. Because we do not implement GA and ACO for covering array generation, no statistical tests can be conducted here. In addition, because the platforms used for collecting the results differ, the comparison of computational time would not be informative. We nevertheless present the execution times of our CPSO and DPSO, which can serve as references for practitioners.

From Table XIV, DPSO can outperform existing GA and ACO for covering array generation, despite the latter two applying test minimization algorithms. Because our DPSO is a version of PSO designed and tuned for covering array generation, it suggests that tuned versions of GAs and ACOs may be a promising area for further study. In addition, CPSO performs worse than GA and ACO in seven of 12 cases. That may result from the improvement by minimization algorithms in [9] and [11]. But for CA_2, CA_3, and CA_6, CPSO can still achieve smaller covering arrays.

In summary, the results further demonstrate that DPSO is an effective discrete version of PSO for covering array generation. Further investigations of tuned versions of GA and ACO should be considered.

VIII. Conclusion

Covering array generation is a key issue in CT. We developed a new DPSO for covering array generation, by adapting S-PSO to generate covering arrays and incorporating two auxiliary strategies to improve the performance. Parameter tuning was applied to both CPSO and DPSO to identify their best parameter settings. The original and discrete versions of four representative PSO variants were implemented and their efficacy for covering array generation was compared. The performance of DPSO was also compared with other existing evolutionary algorithms, GA and ACO.

DPSO can perform better than CPSO with fewer iterations, and the performance of CPSO and DPSO is significantly impacted by their parameter settings. Different cases require different parameter settings; there may not exist a single choice that leads to the best results. After parameter tuning, we identified two recommended settings, which lead to relatively good performance for CPSO and DPSO. Indeed, CPSO with our recommended parameter settings improves on previously reported results, and our DPSO with recommended parameter settings usually beats CPSO. Typically, DPSO also significantly enhances the performance of PSO variants. In addition, DPSO often outperforms GA and ACO to generate covering arrays. Consequently, DPSO is a promising improvement of PSO for covering array generation.

Improvements on the methods here may be possible in a number of ways. One would be to investigate further evolution procedures and strategies proposed in PSO, such as hybridizing with penalty approaches to handle discrete unknowns [39], and compare the results with some exact schemes like branch and bound method. A second would be to examine one-column-at-a-time approaches or methods that construct the entire array, rather than the one-row-at-a-time approach adopted here. A third would be to incorporate DPSO with other methods, in particular with test minimization methods.

References

[1] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11.1–11.29, 2011.

[2] D. Kuhn and M. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proc. 27th Annu. NASA Goddard/IEEE Softw. Eng. Workshop*, Greenbelt, MD, USA, 2002, pp. 91–95.

[3] M. Cohen, P. Gibbons, W. Mugridge, and C. Colbourn, "Constructing test suites for interaction testing," in *Proc. 25th Int. Conf. Softw. Eng.*, Portland, OR, USA, 2003, pp. 38–48.

[4] M. B. Cohen, C. J. Colbourn, and A. C. Ling, "Constructing strength three covering arrays with augmented annealing," *Discrete Math.*, vol. 308, no. 13, pp. 2709–2722, 2008.

[5] J. Torres-Jimenez and E. Rodriguez-Tello, "Simulated annealing for constructing binary covering arrays of variable strength," in *Proc. Congr. Evol. Comput.*, Barcelona, Spain, Jul. 2010, pp. 1–8.

[6] B. Garvin, M. Cohen, and M. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empir. Softw. Eng.*, vol. 16, no. 1, pp. 61–102, 2011.

[7] J. Torres-Jimenez and E. Rodriguez-Tello, "New bounds for binary covering arrays using simulated annealing," *Inf. Sci.*, vol. 185, no. 1, pp. 137–152, 2012.

[8] S. Ghazi and M. Ahmed, "Pair-wise test coverage using genetic algorithms," in *Proc. Congr. Evol. Comput.*, vol. 2. Canberra, ACT, Australia, 2003, pp. 1420–1424.

[9] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf.*, vol. 1. Hong Kong, 2004, pp. 72–77.

[10] J. McCaffrey, "An empirical study of pairwise test set generation using a genetic algorithm," in *Proc. 7th Int. Conf. Inf. Technol. New Gener.*, Las Vegas, NV, USA, 2010, pp. 992–997.

[11] X. Chen, Q. Gu, A. Li, and D. Chen, "Variable strength interaction testing with an ant colony system approach," in *Proc. Asia-Pacific Softw. Eng. Conf.*, Penang, Malaysia, 2009, pp. 160–167.

[12] X. Chen, Q. Gu, X. Zhang, and D. Chen, "Building prioritized pairwise interaction test suites with ant colony optimization," in *Proc. 9th Int. Conf. Qual. Softw.*, Jeju-do, Korea, 2009, pp. 347–352.

[13] X. Chen, Q. Gu, J. Qi, and D. Chen, "Applying particle swarm optimization to pairwise testing," in *Proc. 34th Annu. Comput. Softw. Appl. Conf.*, Seoul, Korea, 2010, pp. 107–116.

[14] B. S. Ahmed and K. Z. Zamli, "PSTG: A T-way strategy adopting particle swarm optimization," in *Proc. 4th Asia Int. Conf. Math. Anal. Model. Comput. Simulat.*, Kota Kinabalu, Malaysia, 2010, pp. 1–5.

[15] B. S. Ahmed and K. Z. Zamli, "A variable strength interaction test suites generation strategy using particle swarm optimization," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2171–2185, 2011.

[16] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, "Application of particle swarm optimization to uniform and variable strength covering array construction," *Appl. Soft Comput.*, vol. 12, no. 4, pp. 1330–1347, 2012.

[17] Y. del Valle, G. Venayagamoorthy, S. Mohagheghi, J.-C. Hernandez, and R. Harley, "Particle swarm optimization: Basic concepts, variants and applications in power systems," *IEEE Trans. Evol. Comput.*, vol. 12, no. 2, pp. 171–195, Apr. 2008.

[18] W.-N. Chen *et al.*, "A novel set-based particle swarm optimization method for discrete optimization problems," *IEEE Trans. Evol. Comput.*, vol. 14, no. 2, pp. 278–300, Apr. 2010.
[19] M. Clerc, *Discrete Particle Swarm Optimization* (New Optimization Techniques in Engineering). New York, NY, USA: Springer, 2004.
[20] W. Pang *et al.*, "Modified particle swarm optimization based on space transformation for solving traveling salesman problem," in *Proc. Int. Conf. Mach. Learn. Cybern.*, vol. 4. Shanghai, China, Aug. 2004, pp. 2342–2346.
[21] W. Pang, K.-P. Wang, C.-G. Zhou, and L.-J. Dong, "Fuzzy discrete particle swarm optimization for solving traveling salesman problem," in *Proc. 4th Int. Conf. Comput. Inf. Technol.*, Wuhan, China, 2004, pp. 796–800.
[22] Y. Wang *et al.*, "A novel quantum swarm evolutionary algorithm and its applications," *Neurocomputing*, vol. 70, nos. 4–6, pp. 633–640, 2007.
[23] E. Campana, G. Fasano, and A. Pinto, "Dynamic analysis for the selection of parameters and initial population, in particle swarm optimization," *J. Global Optim.*, vol. 48, no. 3, pp. 347–397, 2010.
[24] A. Ratnaweera, S. Halgamuge, and H. Watson, "Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients," *IEEE Trans. Evol. Comput.*, vol. 8, no. 3, pp. 240–255, Jun. 2004.
[25] J. Liang, A. Qin, P. Suganthan, and S. Baskar, "Comprehensive learning particle swarm optimizer for global optimization of multimodal functions," *IEEE Trans. Evol. Comput.*, vol. 10, no. 3, pp. 281–295, Jun. 2006.
[26] Z.-H. Zhan, J. Zhang, Y. Li, and H.-H. Chung, "Adaptive particle swarm optimization," *IEEE Trans. Syst., Man, Cybern., B, Cybern.*, vol. 39, no. 6, pp. 1362–1381, Dec. 2009.
[27] J. Liang and P. Suganthan, "Dynamic multi-swarm particle swarm optimizer," in *Proc. IEEE Swarm Intell. Symp.*, Pasadena, CA, USA, 2005, pp. 124–129.
[28] D. Cohen, S. Dalal, M. Fredman, and G. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, Jul. 1997.
[29] R. C. Bryce and C. J. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *Proc. 9th Annu. Conf. Genet. Evol. Comput.*, London, U.K., 2007, pp. 1082–1089.
[30] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. Int. Conf. Neural Netw.*, vol. 4. Perth, WA, Australia, 1995, pp. 1942–1948.
[31] R. Eberhart and Y. Shi, "Particle swarm optimization: Developments, applications and resources," in *Proc. Congr. Evol. Comput.*, vol. 1. Seoul, Korea, 2001, pp. 81–86.
[32] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm Intell.*, vol. 1, no. 1, pp. 33–57, 2007.
[33] S. Helwig, J. Branke, and S. Mostaghim, "Experimental analysis of bound handling techniques in particle swarm optimization," *IEEE Trans. Evol. Comput.*, vol. 17, no. 2, pp. 259–271, Apr. 2013.
[34] A. Windisch, S. Wappler, and J. Wegener, "Applying particle swarm optimization to software testing," in *Proc. 9th Annu. Conf. Genet. Evol. Comput.*, London, U.K., 2007, pp. 1121–1128.
[35] A. Ganjali, "A requirements-based partition testing framework using particle swarm optimization technique," M.S. thesis, Dept. Electr. Comput. Eng., Univ. Waterloo, Waterloo, ON, Canada, 2008.
[36] Y.-J. Gong *et al.*, "Optimizing the vehicle routing problem with time windows: A discrete particle swarm optimization approach," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 2, pp. 254–267, Mar. 2012.
[37] W.-N. Chen *et al.*, "Particle swarm optimization with an aging leader and challengers," *IEEE Trans. Evol. Comput.*, vol. 17, no. 2, pp. 241–258, Apr. 2013.
[38] C. J. Colbourn. (Apr. 2013). *Covering Array Tables for t=2, 3, 4, 5, 6*. [Online]. Available: http://www.public.asu.edu/ ccolbou/src/tabby/catable.html
[39] M. Corazza, G. Fasano, and R. Gusso, "Particle swarm optimization with non-smooth penalty reformulation, for a complex portfolio selection problem," *Appl. Math. Comput.*, vol. 224, pp. 611–624, Nov. 2013.

Huayao Wu received the B.S degree from Southeast University, Nanjing, China and the M.S degree from Nanjing University, Nanjing, China, where he is currently working toward the Ph.D. degree from Nanjing University, Nanjing.

His research interests include software testing, especially on combinatorial testing and search-based software testing.

Changhai Nie (M'12) received the B.S. and M.S. degrees in mathematics from Harbin Institute of Technology, Harbin, China, and the Ph.D. degree in computer science from Southeast University, Nanjing, China.

He is a Professor of Software Engineering with State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. His research interests include software analysis, testing and debugging.

Fei-Ching Kuo (M'06) received the B.Sc. (Hons.) degree in computer science and the Ph.D. degree in software engineering from Swinburne University of Technology, Hawthorn, VIC, Australia.

She was a Lecturer with University of Wollongong, Wollongong, NSW, Australia. She is currently a Senior Lecturer with the Swinburne University of Technology. Her research interests include software analysis, testing, and debugging.

Hareton Leung (M'90) received the Ph.D. degree in computer science from University of Alberta, Edmonton, AB, Canada.

He is an Associate Professor and a Director of the Laboratory for Software Development and Management, Department of Computing, Hong Kong Polytechnic University, Hong Kong. His research interests include software testing, project management, risk management, quality and process improvement, and software metrics.

Charles J. Colbourn received the Ph.D. degree from University of Toronto, Toronto, ON, Canada, in 1980.

He is a Professor of Computer Science and Engineering with Arizona State University, Tempe, AZ, USA. He has authored the books *The Combinatorics of Network Reliability* (Oxford) and *Triple Systems* (Oxford), and also 320 refereed journal papers focussing on combinatorial designs and graphs with applications in networking, computing, and communications.

Prof. Colbourn received the Euler Medal for Lifetime Research Achievement by the Institute for Combinatorics and its Applications in 2004.