Hindawi
Mathematical Problems in Engineering
Volume 2018, Article ID 1860673, 14 pages
https://doi.org/10.1155/2018/1860673

Research Article

A Simulated Annealing Algorithm to Construct Covering Perfect Hash Families

Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez

CINVESTAV-Tamaulipas, Cd. Victoria 87130, Tamaulipas, Mexico

Correspondence should be addressed to Jose Torres-Jimenez; jtj@cinvestav.mx

Received 27 January 2018; Accepted 29 May 2018; Published 26 July 2018

Academic Editor: Fiorenzo A. Fazzolari

Covering perfect hash families (CPHFs) are combinatorial designs that represent certain covering arrays in a compact way. In previous works, CPHFs have been constructed using backtracking, tabu search, and greedy algorithms. Backtracking is convenient for small CPHFs, greedy algorithms are appropriate for large CPHFs, and metaheuristic algorithms provide a balance between execution time and quality of solution for small and medium-size CPHFs. This work explores the construction of CPHFs by means of a simulated annealing algorithm. The neighborhood function of this algorithm is composed of three perturbation operators which together provide exploration and exploitation capabilities to the algorithm. As main computational results we have the generation of 64 CPHFs whose derived covering arrays improve the best-known ones. In addition, we use the simulated annealing algorithm to construct quasi-CPHFs from which quasi covering arrays are derived that are then completed and postoptimized; in this case the number of new covering arrays is 183. Together, the 247 new covering arrays improved the upper bound of 683 covering array numbers.

1. Introduction

A *covering array* $\mathrm{CA}(N; t, k, v)$ of strength t is an $N \times k$ array over the symbol set $\mathbb{Z}_v = \{0, 1, \ldots, v-1\}$ such that in every $N \times t$ subarray all t-tuples over $\mathbb{Z}_v$ occur at least once. If each t-tuple over $\mathbb{Z}_v$ occurs exactly once in every $N \times t$ subarray, then the array is an *orthogonal array* of strength t, denoted by $\mathrm{OA}(N; t, k, v)$. Every orthogonal array is a covering array, but not every covering array is an orthogonal array.

The main applications of covering arrays are in software and hardware testing. In the testing strategy called combinatorial testing, covering arrays are viewed as test-suites where the strength t is the size of the interactions to be checked in a software component. For example, a covering array of strength $t = 3$ can be used to test all possible combinations of values among any three of the k parameters; see [1] for an introduction to the use of covering arrays in combinatorial testing. In the area of hardware testing, covering arrays have been used to check the presence of hardware Trojans [2].

Given the values of the strength t, the number of columns k, and the order v, the problem of constructing covering arrays is the problem of finding the minimum number of rows N such that a $\mathrm{CA}(N; t, k, v)$ exists. Currently, there is no polynomial-time algorithm to solve optimally this problem for general values of t, k, and v; only some particular cases have been solved optimally [3–6].

The smallest N, for which a covering array exists with strength t, k columns, and order v, is called the *covering array number* of t, k, v, and it is denoted by $\mathrm{CAN}(t, k, v)$. A trivial lower bound for the covering array number is $\mathrm{CAN}(t, k, v) \geq v^t$, since every subarray of t columns must cover every one of the v^t possible t-tuples over the v symbols. In addition to the cases with optimal solution, a number of improvements in the lower bounds for specific cases have been reported in [7–9].

With regard to upper bounds, much work has been done to constantly improve them, and several algorithms have been developed to construct increasingly better covering arrays. To keep track of the advances in the construction of covering arrays, the Covering Array Tables [10] contain the current best-known upper bounds for strengths $2 \leq t \leq 6$ and orders $2 \leq v \leq 25$.

The techniques used to construct covering arrays can be classified as exact, greedy, metaheuristic, algebraic, and

recursive; see [11–14] for an overview of covering array construction methods. Metaheuristic algorithms to construct covering arrays include tabu search [15], genetic algorithms [16], simulated annealing [17], ant colony [18], particle swarm optimization [19], harmony search [20], hill climbing [21], bird swarm [22], a combination of bee colony and harmony search [23], Cuckoo search [24], differential evolution [25], tabu search as a hyperheuristic [26], and a combination of simulated annealing with a greedy algorithm [27].

One way to construct covering arrays is to use a high level representation of them called *covering perfect hash family (CPHF)*. The elements of a CPHF are $(t-1)$-tuples or t-tuples over $\mathbb{Z}_v$ that represent a column vector of length v^t. CPHFs are a compact representation of covering arrays because a CPHF with n rows and k columns represents a covering array with nv^t rows and k columns. Commonly CPHFs have few rows (say $n \leq 20$), but their corresponding covering arrays may have a large number of rows depending on the values of v and t.

In this work we develop a simulated annealing (SA) algorithm to construct CPHFs. The neighborhood function of the SA algorithm is composed of three perturbation operators which together provide exploration and exploitation capabilities to the SA algorithm. The probability of using one of the three operators in an application of the neighborhood function is tuned by experimenting with a number of probability distributions. Relevant results obtained with the SA algorithm are 64 new CPHFs whose respective covering arrays improved the best-known ones and other 183 new covering arrays obtained by means of a procedure based on constructing quasi-CPHFs. In total, these $64+183 = 247$ new covering arrays improve the upper bound of 683 covering array numbers CAN(t, k, v).

This paper is organized as follows: Section 2 provides a background on CPHFs; Section 3 reviews some related works; Section 4 describes the components of the SA algorithm and shows the experimentation done to set the utilization rate of the operators in the neighborhood function; Section 5 presents the computational results; and Section 6 gives the conclusions of the work.

2. Covering Perfect Hash Families

CPHFs were introduced by Sherwood et al. [28]. Let v be a prime power and let $\mathbb{F}_v$ denote the finite fields with v elements. For a given strength t, let $(\beta_0^{(i)}, \beta_1^{(i)}, \ldots, \beta_{t-1}^{(i)})$ be the base v representation of $i \in \{0, 1, \ldots, v^t - 1\}$; that is, $i = \sum_{j=0}^{t-1} v^j \cdot \beta_j^{(i)}$. As defined in [28], a *permutation vector* $\overrightarrow{(h_1, h_2, \ldots, h_{t-1})}$ is the vector of length v^t that has the symbol $\beta_0^{(i)} + (h_1 \times \beta_1^{(i)}) + (h_2 \times \beta_2^{(i)}) + \cdots + (h_{t-1} \times \beta_{t-1}^{(i)})$ in position i for $0 \leq i \leq v^t - 1$.

Permutation vectors are vectors with v^t elements from $\mathbb{F}_v$, but they are represented by $(t-1)$-tuples over $\mathbb{F}_v$. For fixed v and t the number of distinct permutation vectors is v^{t-1}. The term "permutation vector" is because the vectors of length v^t are formed by concatenating v^{t-1} permutations of $(0, 1, \ldots, v-1)$. For example, Table 1 shows the $v^{t-1} = 3^2 = 9$ distinct permutation vectors for $v = 3$ and $t = 3$.

Table 1: Permutation vectors for $v = 3$ and $t = 3$. Starting from the first element, every group of $v = 3$ elements is a permutation of $(0, 1, 2)$.

Perm Vec	Elements
$\overrightarrow{(0,0)}$	(0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2)
$\overrightarrow{(0,1)}$	(0 1 2 0 1 2 0 1 2 1 2 0 1 2 0 1 2 0 2 0 1 2 0 1 2 0 1)
$\overrightarrow{(0,2)}$	(0 1 2 0 1 2 0 1 2 2 0 1 2 0 1 2 0 1 1 2 0 1 2 0 1 2 0)
$\overrightarrow{(1,0)}$	(0 1 2 1 2 0 2 0 1 0 1 2 1 2 0 2 0 1 0 1 2 1 2 0 2 0 1)
$\overrightarrow{(1,1)}$	(0 1 2 1 2 0 2 0 1 1 2 0 2 0 1 0 1 2 2 0 1 0 1 2 1 2 0)
$\overrightarrow{(1,2)}$	(0 1 2 1 2 0 2 0 1 2 0 1 0 1 2 1 2 0 1 2 0 2 0 1 0 1 2)
$\overrightarrow{(2,0)}$	(0 1 2 2 0 1 1 2 0 0 1 2 2 0 1 1 2 0 0 1 2 2 0 1 1 2 0)
$\overrightarrow{(2,1)}$	(0 1 2 2 0 1 1 2 0 1 2 0 0 1 2 2 0 1 2 0 1 1 2 0 0 1 2)
$\overrightarrow{(2,2)}$	(0 1 2 2 0 1 1 2 0 2 0 1 1 2 0 0 1 2 1 2 0 0 1 2 2 0 1)

Another characteristic of permutation vectors is that their first v elements are $0, 1, \ldots, v-1$ in that order.

Given a tuple of t permutation vectors, let A be the array of size $v^t \times t$ whose columns are the t permutation vectors of the tuple. If A is an OA$(v^t; t, t, v)$ then the t-tuple of permutation vectors is *covering*; otherwise the t-tuple is *noncovering*. For example, the 3-tuple of permutation vectors $(\overrightarrow{(0,2)}, \overrightarrow{(1,0)}, \overrightarrow{(1,1)})$ is covering, because we can verify in Table 1 that the array of size 27×3 with columns $\overrightarrow{(0,2)}$, $\overrightarrow{(1,0)}$, and $\overrightarrow{(1,1)}$ is an OA$(27; 3, 3, 3)$. On the other hand, the tuple $(\overrightarrow{(0,0)}, \overrightarrow{(0,1)}, \overrightarrow{(0,2)})$ is noncovering since we can see in Table 1 that the array formed by the permutation vectors of the tuple is not an orthogonal array.

A CPHF$(n; k, v^{t-1}, t)$ is an $n \times k$ array with elements from $\mathbb{F}_v^{t-1}$ such that every subarray of t columns has at least one row that is a covering tuple of t permutation vectors [28]. A CPHF$(n; k, v^{t-1}, t)$ generates a CA$(n(v^t); t, k, v)$ replacing the elements of the CPHF by the corresponding permutation vectors of length v^t. However, since the first v elements of every permutation vector are $(0, 1, \ldots, v-1)$, it is possible to delete the first v elements from every permutation vector given by the last $n-1$ rows of the CPHF; the result is a CA$(n(v^t - v) + v; t, k, v)$. Equation (1) shows a CPHF$(3; 15, 5^2, 3)$; the permutation vectors $\overrightarrow{(h_1, h_2)}$ are written as $h_1 h_2$. Every one of the $\binom{15}{3}$ subarrays of $t = 3$ columns contains as a row a covering tuple of permutation vectors. This CPHF generates a CA$(365; 3, 15, 5)$.

Example of a CPHF$(3; 15, 5^2, 3)$

$$\begin{pmatrix} 11 & 00 & 02 & 01 & 20 & 03 & 34 & 23 & 24 & 13 & 31 & 44 & 40 & 41 & 33 \\ 23 & 01 & 21 & 04 & 12 & 40 & 44 & 01 & 14 & 22 & 44 & 02 & 31 & 41 & 30 \\ 21 & 04 & 30 & 30 & 13 & 01 & 40 & 02 & 31 & 41 & 21 & 12 & 55 & 43 & 03 \end{pmatrix} \quad (1)$$

In addition to permutation vectors generated by a $(t-1)$-tuple over $\mathbb{F}_v$, we have that a t-tuple $(h_0, h_1, \ldots, h_{t-1})$ over $\mathbb{F}_v$ also generates a vector of v^t elements when evaluated as $(h_0 \times \beta_0^{(i)}) + (h_1 \times \beta_1^{(i)}) + \cdots + (h_{t-1} \times \beta_{t-1}^{(i)})$ in the base v representation $(\beta_0^{(i)}, \beta_1^{(i)}, \ldots, \beta_{t-1}^{(i)})$ of every $0 \leq i \leq v^t - 1$. However, some of these t-tuples $(h_0, h_1, \ldots, h_{t-1})$ generate

Table 2: Example of extended permutation vectors for $v = 3$ and $t = 3$.

E. Perm Vec	Elements
$\overrightarrow{(0,0,1)}$	(0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2)
$\overrightarrow{(0,2,1)}$	(0 0 0 2 2 2 1 1 1 1 1 1 0 0 0 2 2 2 2 2 2 1 1 1 0 0 0)
$\overrightarrow{(1,0,1)}$	(0 1 2 0 1 2 0 1 2 1 2 0 1 2 0 1 2 0 2 0 1 2 0 1 2 0 1)
$\overrightarrow{(1,1,2)}$	(0 1 2 1 2 0 2 0 1 2 0 1 0 1 2 1 2 0 1 2 0 2 0 1 0 1 2)
$\overrightarrow{(2,0,1)}$	(0 2 1 0 2 1 0 2 1 1 0 2 1 0 2 1 0 2 2 1 0 2 1 0 2 1 0)
$\overrightarrow{(2,1,2)}$	(0 2 1 1 0 2 2 1 0 2 1 0 0 2 1 1 0 2 1 0 2 2 1 0 0 2 1)

vectors that are not permutation vectors; specifically, the t-tuples with $h_0 = 0$ generate vectors of length v^t, where every group of v elements is formed by v occurrences of the same symbol.

An *extended permutation vector* $\overrightarrow{(h_0, h_1, \ldots, h_{t-1})}$ is defined as the vector of length v^t that has the symbol $(h_0 \times \beta_0^{(i)}) + (h_1 \times \beta_1^{(i)}) + \cdots + (h_{t-1} \times \beta_{t-1}^{(i)})$ in position i for $0 \leq i \leq v^t - 1$ [29]. For given v and t, the total number of extended permutation vectors is v^t. As examples of extended permutation vectors, Table 2 shows six of the $v^t = 3^3 = 27$ extended permutation vectors for $v = 3$ and $t = 3$.

Extended permutation vectors enlarge the available options to construct CPHFs, because the number of possible symbols is now v^t. As for permutation vectors, a CPHF$(n; k, v^t, t)$ generates a covering array replacing the elements of the CPHF by their corresponding extended permutation vectors of length v^t. However, in this case it is not possible to delete the first v elements of the extended permutation vectors given by the last $n-1$ rows of the CPHF; only the first element can be deleted because it is zero in any extended permutation vector. Thus, a CPHF$(n; k, v^t, t)$ generates a CA$(n(v^t - 1) + 1; t, k, v)$ [29].

To make a distinction between CPHFs with elements from $\mathbb{F}_v^{t-1}$ and CPHFs with elements from $\mathbb{F}_v^t$, we follow the convention of Colbourn et al. [30], where the first kinds of CPHFs are called Sherwood-CPHFs or SCPHFs, and the second ones are called simply CPHFs.

For the same number of columns k and the same number of rows $n > 1$, a SCPHF$(n; k, v^{t-1}, t)$ is better than a CPHF$(n; k, v^t, t)$, because the SCPHF produces a covering array with less rows. However, CPHFs may have more columns than SCPHFs since there are more extended permutation vectors than permutation vectors.

3. Related Works

This section reviews some methods to construct SCPHFs and CPHFs, and some SA algorithms to construct covering arrays.

3.1. Methods to Construct CPHFs. CPHFs were introduced by Sherwood et al. [28]. In that work SCPHFs were constructed by a backtracking algorithm. The first step of the algorithm is to initialize a candidate array of size $n \times k$; in this candidate array each entry takes its first valid element from $\mathbb{F}_v^{t-1}$, but not all entries of the candidate array have the same set of valid elements due to the following constraints:

(1) If $k < v^{t-1}$ the elements of the first row are in ascending order, and all elements are distinct. The other $n-1$ rows contain any permutation of $\mathbb{F}_v^{t-1}$.

(2) If $k \geq v^{t-1}$ the array is divided into subarrays with at most v^{t-1} columns; in each subarray every row has distinct elements from $\mathbb{F}_v^{t-1}$, and the first row of each subarray has the ith element of $\mathbb{F}_v^{t-1}$ in its ith column.

Once a candidate array is formed, i.e., all its entries are assigned with an element from $\mathbb{F}_v^{t-1}$, the array is tested to see if it is a SCPHF. The test consists in verifying that every subarray of t columns contains a covering tuple of permutation vectors. The following candidate arrays are generated using backtracking on the elements assigned to each entry of the array.

This backtracking technique is only effective for a small number of columns, say $k \leq 50$, since the number of candidate arrays grows considerably with each new column. The authors report SCPHFs with up to $k = 44$ columns for strength $t = 3$, and with $k = 16$ columns for strength four.

Walker II and Colbourn [31] constructed SCPHFs using tabu search. The initial solution is an array of size $n \times k$ generated randomly. The algorithm employs a tabu list to store the last 50000 moves performed by the algorithm. A move replaces the value of one entry of the current solution by another value. Moves are only done in columns that are part of subarrays of t columns with no covering tuple; such subarrays are called "uncovered subarrays". The reason to restrict moves is that moves in columns that are not part of uncovered subarrays cannot improve the "score" of the current array. The score of the current solution is the number of subarrays with no covering tuple; when the score becomes zero a SCPHF has been obtained.

The neighborhood of the current solution is all possible moves in all columns that are part of an uncovered subarray. At the beginning of the execution, the size of the neighborhood can be very large, and for this reason the neighborhood is restricted to the column that appears in more uncovered subarrays; such column is the worst column of the current solution. As the algorithm executes, the number of columns that belong to an uncovered subarray decreases and so the restriction of only considering the worst column is removed.

The tabu search algorithm was able to construct SCPHFs with up to 255 columns for strength three, 62 columns for strength four, 19 columns for strength five, 14 columns for strength six, and 13 columns for strength seven.

Recently Colbourn et al. [30] constructed SCPHFs and CPHFs by combining randomized and greedy algorithms. Firstly, a column resampling algorithm generates an initial CPHF (or SCPHF) and then a greedy algorithm adds new columns to the CPHF. The column resampling algorithm works as follows: the initial array of size $n \times k$ is generated by assigning its entries randomly; then, all subarrays of t columns are checked to see if they are "covered", that is if they have a covering tuple as a row. From each uncovered subarray one column is selected and its entries are reassigned with

random elements to try to decrease the number of uncovered subarrays. Columns are resampled until all subarrays are covered.

After generating the initial CPHF, a greedy algorithm adds new columns to the CPHF. For each new column the algorithm generates randomly a number of candidates columns; if one of these candidates does not introduce uncovered subarrays then it is appended to the current CPHF, which now has one more column. If none candidate column can be appended to the CPHF without introducing uncovered subarrays, then one of the candidate columns replaces one of the current columns if there is one current column that is part of all uncovered subarrays introduced by the candidate column. This is a greedy strategy to replace the column most likely to participate in uncovered subarrays.

By means of the combination of the column resampling algorithm and the greedy random extension algorithm, the authors constructed a great number of new SCPHFs and CPHFs whose corresponding covering arrays established a new upper bound of CAN(t, k, v). Sizes of the SCPHFs and CPHFs constructed are $k = 10000$ columns for strength three, $k = 2500$ for strength four, $k = 600$ for strength five, and $k = 200$ for strength six.

It is very difficult for a metaheuristic algorithm to construct CPHFs of these sizes in a reasonable amount of time. Then, our simulated annealing algorithm concentrates in small and medium-size CPHFs ($k \leq 50$ and $50 < k \leq 350$, respectively); we consider CPHFs with $k > 350$ columns as large CPHFs.

3.2. SA Algorithms to Construct Covering Arrays. The SA technique has been used many times to find new upper bounds of CAN(t, k, v). One of the most successful algorithms is the augmented annealing algorithm developed by Cohen et al. [32]. In that work the target covering array is divided into smaller arrays or "ingredients" using a combinatorial construction; after that, each ingredient is constructed either by a combinatorial technique or by SA if there is no combinatorial construction to generate the ingredient.

Other work that uses SA is the one by Torres-Jimenez and Rodriguez-Tello [17]. In this SA algorithm only binary ($v = 2$) covering arrays are constructed. The initial solution is created randomly, but in such a way the symbols are balanced in each column; that is, there are $\lfloor N/2 \rfloor$ zeros and $N - \lfloor N/2 \rfloor$ ones in every column. To generate a new solution from the current one the algorithm uses one of two relatively simple neighborhood functions N_1 and N_2. The function N_1 employs a procedure called *switch* (A, i, j) that changes an entry of the current solution from 0 to 1, or from 1 to 0. The function N_2 uses a procedure called *swap* (A, i, j, l) to interchange the elements in the rows i and l of the column j.

Avila-George et al. [33] developed three parallel SA algorithms called, respectively, *independent search*, *semi-independent search*, and *cooperative search*. In the independent search each processes runs an instance of the SA algorithm independently of the others; at the end, the best solution among all processes is taken as the final solution of the parallel SA algorithm. On the other hand, in the semi-independent search the processes exchange intermediate solutions in a synchronous way. After a predefined number of applications of the neighborhood function the processes share their best solution to select a global winner which is communicated to all processes, in order for all of them to know the current best global solution. In the cooperative search the processes share their current best solutions asynchronously.

The SA algorithms developed so far to construct covering arrays work directly in the domain of covering arrays; that is, to construct a CA$(N; t, k, v)$ they work with matrices of size $N \times k$ over $\mathbb{Z}_v$, or with parts of the matrix as in the augmented annealing technique [32]. On the other hand, the SA algorithm developed in this work constructs the CAs in the domain of CPHFs, which is a high level representation of some covering arrays. Covering arrays and CPHFs are two distinct combinatorial objects; in covering arrays the requirement is that every subarray of t columns contains at least once t-tuple over $\mathbb{Z}_v$, while in CPHFs the requirement is that every subarray of t columns contains at least one covering tuple of permutation vectors. Then, although the structure of the SA algorithms proposed in this work may be similar to the structure of the SA algorithms developed to construct covering arrays, the neighborhood functions are different.

4. Simulated Annealing Algorithm

This section describes the SA algorithm to construct SCPHFs and CPHFs. Section 4.1 shows the general structure of the SA algorithm. Section 4.2 presents the operators that form the neighborhood function and describes the experimentation done to find the more convenient values for the utilization rate of each operator. Section 4.3 reviews the noncovering test of [28, 31] to determine efficiently if a tuple of permutation vectors is noncovering; also this subsection describes how the noncovering test is adapted to extended permutation vectors.

4.1. Structure of the SA Algorithm. Simulated annealing is a metaheuristic inspired in the process of heating and cooling metals to obtain a strong crystalline structure. A metal is heated until it melts and then it is cooled in a controlled way. At the end of the process, the particles of the metal are arranged in such a way the energy of the system is minimal [34]. The evolution of the metal from melted to ground state is simulated as follows: given a current solution x with energy E_x; from a perturbation of x a new solution y with energy E_y is created. If $E_y < E_x$ then y becomes the new current solution; otherwise y becomes the new current solution with probability $e^{-(E_y - E_x)/T}$, where T is the current temperature. When the temperature is high, the probability of accepting a solution with more energy than the current one is high; but as the temperature decreases the probability of accepting such solutions also decreases. The cooling process requires three parameters: the initial temperature T_i, the final temperature T_f, and the cooling rate $0 < \gamma < 1$.

Perturbations of the current solution x are done by means of a neighborhood function that takes x as argument, makes some changes to x, and returns a new solution y. In simulated annealing, the neighborhood function is applied a certain

```
(1) T_i = 4
(2) T_f ⟵ 1 × 10^−10
(3) γ = 0.99
(4) L_i = nkv
(5) L_f ⟵ (L_i)^2
(6) m ⟵ log_γ(T_f/T_i)
(7) δ ⟵ (L_f/L_i)^1/m
(8) T ⟵ T_i
(9) L ⟵ L_i
(10) G ⟵ M ⟵ random_initial_solution()
(11) if c(G) = 0 then
(12)     return G
(13) end if
(14) while T > T_f do
(15)     best_global_improved ⟵ false
(16)     for i = 0, 1, ..., L − 1 do
(17)         M′ ⟵ neighborhood_function(M)
(18)         if c(M′) < c(M) or random(0, 1) < e^−(c(M′)−c(M))/T then
(19)             M ⟵ M′
(20)             if c(M) < c(G) then
(21)                 G ⟵ M
(22)                 best_global_improved ⟵ true
(23)                 if c(G) = 0 then
(24)                     return G
(25)                 end if
(26)             end if
(27)         end if
(28)     end for
(29)     if best_global_improved = false then
(30)         T ⟵ γT
(31)         L ⟵ δL
(32)     end if
(33) end while
(34) return NULL
```

Algorithm 1: SA(n, k, v, t).

number of times for each temperature value. In this work, the number of perturbations of the current solution (the length of the Markov chain) done at each temperature value is not fixed. The number of perturbations is incremented as the temperature drops. For this reason, the SA algorithm requires another three parameters for increasing the length of the Markov chain: the initial length L_i, the final length L_f, and the increment factor $\delta > 1$.

The parameter δ can be expressed in terms of the other five parameters if we establish that the Markov chain reaches its final value L_f when the temperature reaches its final value T_f. Let m be the number of iterations to decrease the temperature from T_i to T_f; then $T_f = \gamma^m T_i$. From this last expression we get $\gamma^m = T_f/T_i$, and so $m = \log_\gamma(T_f/T_i)$. Now, from the equation $L_f = \delta^m L_i$ we obtain $\delta^m = L_f/L_i$, and so $\delta = (L_f/L_i)^{1/m}$.

For the cooling schedule we take the values of the successful SA algorithm developed in [17] to construct covering arrays: $T_i = 4$, $T_f = 1 \times 10^{-10}$, and $\gamma = 0.99$. In addition, we set the initial length of the Markov chain to $L_i = nkv$ and the final length to $L_f = (L_i)^2$. Algorithm 1 shows the SA algorithm to construct either a SCPHF$(n; k, v^{t-1}, t)$ or a CPHF$(n; k, v^t, t)$.

The current solution is stored in matrix M, and matrix G stores the best global solution found. At the end of the *while* loop the current temperature T and the current length of the Markov chain L are updated only if the global best solution was not improved in the last Markov chain. The neighborhood function is called in every iteration of the *for* loop to generate a new solution M' based on the current solution M. The cost of a solution is the number of subarrays of t columns that do not have a covering tuple as a row. When the cost of the current solution is zero a CPHF has been constructed. Function c is used to compute the cost of a solution. If the cost of the solution M' generated by the neighborhood function is better than the cost of the current solution M, then M' is accepted as the new current solution; otherwise M' has a probability of $e^{-(c(M')-c(M))/T}$ of being accepted as the new current solution.

4.2. Neighborhood Function. Let M be the $n \times k$ matrix that stores the current solution. As said before, the cost of M is the

number of subarrays with no row containing a covering tuple. We will refer to such subarrays as *uncovered combinations*, because a subarray is associated with a combination of t columns from M.

The neighborhood function changes M by applying one of three operators R_1, R_2, and R_3, with the objective of reducing the number of uncovered combinations. Since Algorithm 1 can be used to construct SCPHFs and CPHFs we denote by $\mathscr{W}$ the symbol set of the current solution M; then, $\mathscr{W} = \mathbb{F}_v^{t-1}$ for SCPHFs and $\mathscr{W} = \mathbb{F}_v^t$ for CPHFs; the number of elements in $\mathscr{W}$ is $w = |\mathscr{W}|$.

The most basic operator is R_1, which only changes the content of a random cell $M_{i,j}$ by a random value from $\mathscr{W}$. The purpose of this operator is to provide exploration capabilities to the SA algorithm, given that a random change can direct the algorithm to another region of the search space.

Operator R_2 selects an uncovered combination denoted by $X = (x_0, x_1, \ldots, x_{t-1})$ and makes changes in every cell of the submatrix of t columns given by X, which is $S_{i,j} = M_{i,X_j}$, in order to transform one row of S in a covering tuple. Let π be a permutation of $\mathscr{W}$. For $r = 0, 1, \ldots, w-1$ the value of $S_{i,j}$ is replaced by π_r until finding p elements $f_0, f_1, \ldots, f_{p-1}$ of π such that when f_l ($l \in \{0, 1, \ldots, p-1\}$) is assigned to $S_{i,j}$ the row i of S is a covering tuple. Then, for each cell $S_{i,j}$ the operator tests at most p distinct covering tuples in row i, for a total of at most $(n)(t)(p)$ distinct covering tuples in S, and the one that minimizes the number of uncovered combinations is selected as a result of R_2. Sometimes no element of $\mathscr{W}$ makes covering the row i of S; in these cases all elements of $\mathscr{W}$ are assigned to $S_{i,j}$. The changes in the cells of S are done independently, so that when a cell is being changed the other cells in its row have their original values. The value p limits the number of changes done in a cell; the reason to set a limit is that when w is large the process of assigning every symbol of $\mathscr{W}$ to $S_{i,j}$ can be time consuming. By experimenting with several values of w and p, we set $p = 4$ in the operator R_2.

The last operator R_3 is a specialized version of R_2 that randomly selects a cell $S_{i,j}$ of the submatrix S given by an uncovered combination X and assigns independently to $S_{i,j}$ all elements of $\mathscr{W}$. There may be several elements of $\mathscr{W}$ that cover the row i of S, but the symbol selected for $S_{i,j}$ is the one that covers X and minimizes the number of uncovered combinations in M. In case of ties, the symbol for $S_{i,j}$ is selected randomly among the best symbols. If no symbol of $\mathscr{W}$ covers the row i of S then $S_{i,j}$ is assigned randomly. The reason for this operator is to increase the probability of finding the best symbol for $S_{i,j}$ that at the same time covers X and minimizes the number of uncovered combinations in the current solution M. Thus, this operator is intended for exploitation of the current neighborhood.

For better performance of the neighborhood function, the probabilities of using one of the three operators R_1, R_2, and R_3 in an application of the neighborhood function were tuned by experimentation. Assuming a granularity of 0.1 for the probability of using each operator, we consider the distinct sixty-six triples (a, b, c) where a, b, and c are nonnegative integers such that $a + b + c = 10$. From a triple (a, b, c) the probabilities for the three operators are given by $P(R_1) = a/10$, $P(R_2) = b/10$, and $P(R_3) = c/10$. The experimentation is based on executing sixty-six instances of the SA algorithm, one for each distinct triple (a, b, c), with the objective of identifying the utilization rate of the operators for which the SA algorithm performs better.

Table 3: Best four results for each one of the six CPHFs used in the experimentation.

Instance	a	b	c	Avg uncovered
SCPHF$(3; 14, 3^4, 5)$	2	6	2	9.1613
SCPHF$(3; 14, 3^4, 5)$	0	7	3	9.1935
SCPHF$(3; 14, 3^4, 5)$	2	7	1	9.4516
SCPHF$(3; 14, 3^4, 5)$	1	7	2	9.5806
SCPHF$(5; 34, 4^3, 4)$	0	9	1	64.0000
SCPHF$(5; 34, 4^3, 4)$	1	9	0	64.6452
SCPHF$(5; 34, 4^3, 4)$	0	8	2	65.4194
SCPHF$(5; 34, 4^3, 4)$	2	7	1	65.7419
SCPHF$(3; 21, 4^3, 4)$	0	10	0	13.2581
SCPHF$(3; 21, 4^3, 4)$	1	6	3	13.3871
SCPHF$(3; 21, 4^3, 4)$	1	7	2	13.3871
SCPHF$(3; 21, 4^3, 4)$	2	8	0	13.4194
CPHF$(4; 32, 4^4, 4)$	1	7	2	25.2258
CPHF$(4; 32, 4^4, 4)$	0	9	1	25.6129
CPHF$(4; 32, 4^4, 4)$	2	7	1	25.8710
CPHF$(4; 32, 4^4, 4)$	2	6	2	25.9677
CPHF$(3; 53, 5^3, 3)$	0	5	5	33.4286
CPHF$(3; 53, 5^3, 3)$	0	6	4	34.0000
CPHF$(3; 53, 5^3, 3)$	2	6	2	34.7143
CPHF$(3; 53, 5^3, 3)$	1	4	5	34.8571
CPHF$(6; 44, 3^4, 4)$	0	9	1	51.2581
CPHF$(6; 44, 3^4, 4)$	0	8	2	52.8065
CPHF$(6; 44, 3^4, 4)$	1	9	0	53.2903
CPHF$(6; 44, 3^4, 4)$	2	8	0	53.7097

The CPHFs used in the experimentation were SCPHF$(3; 14, 3^4, 5)$, SCPHF$(5; 34, 4^3, 4)$, SCPHF$(3; 21, 4^3, 4)$, CPHF$(4; 32, 4^4, 4)$, CPHF$(3; 53, 5^3, 3)$, and CPHF$(6; 44, 3^4, 4)$. However, given the nondeterministic nature of the SA algorithm, we repeated each case 31 times. Thus, the SA algorithm was executed $(66)(6)(31) = 12276$ times to determine the utilization rate of the operators R_1, R_2, and R_3.

Table 3 shows the best four results obtained for each CPHF. The first column contains the CPHF instance. Columns 2-4 contain the triple (a, b, c) that gives the probabilities of using the three operators of the neighborhood function, where a corresponds to R_1, b corresponds R_2, and c corresponds to R_3. The last column contains the average over the 31 runs of the number of uncovered combinations reached by the SA algorithm after 7000 executions of the neighborhood function.

The results of Table 3 do not show an absolute winner triple (a, b, c), but they show that operator R_2 should have a higher probability of being used. To obtain the winner triple we averaged the values in columns 2, 3, and 4:

$$a = \frac{(2 + 0 + 2 + 1 + 0 + \cdots + 1 + 2)}{23} = \frac{23}{24} \approx 1$$

$$b = \frac{(6+7+7+7+9+\cdots+9+8)}{24} = \frac{175}{24} \approx 7$$
$$c = \frac{(2+3+1+2+1+\cdots+0+0)}{24} = \frac{42}{24} \approx 2 \tag{2}$$

Therefore, the probability that each operator has been used in an application of the neighborhood function is

$$P(R_1) = \frac{a}{10} = \frac{1}{10} = 0.1$$
$$P(R_2) = \frac{b}{10} = \frac{7}{10} = 0.7 \tag{3}$$
$$P(R_3) = \frac{c}{10} = \frac{2}{10} = 0.2$$

From this result we concluded that the three operators are required by the neighborhood function, but each one with a different utilization rate.

4.3. Noncovering Test. This section describes the test of [28, 31] to check efficiently if a t-tuple of permutation vectors is noncovering. The noncovering test is executed several times in every application of the neighborhood function, and therefore it is very important to execute this test efficiently.

A t-tuple of permutation vectors is noncovering if the array of size $v^t \times t$ on v symbols generated by the t permutation vectors has two distinct rows $i, j \in \{0, 1, \ldots, v^t - 1\}$ that contain the same t-tuple on v symbols. If the array generated by the t permutation vectors has two identical rows, then it cannot be an orthogonal array $\mathrm{OA}(v^t; t, t, v)$, and therefore the t-tuple is noncovering.

For $0 \le r \le t-1$ let $\overrightarrow{h^{(r)}} = \overrightarrow{(h_1^{(r)}, h_2^{(r)}, \ldots, h_{t-1}^{(r)})}$ be a permutation vector. Then, the t-tuple $(\overrightarrow{h^{(0)}}, \overrightarrow{h^{(1)}}, \ldots, \overrightarrow{h^{(t-1)}})$ is noncovering if and only if there exist distinct $i, j \in \{0, 1, \ldots, v^t - 1\}$ such that

$$\left[\beta_0^{(i)} + \left(h_1^{(r)} \times \beta_1^{(i)}\right) + \cdots + \left(h_{t-1}^{(r)} \times \beta_{t-1}^{(i)}\right)\right] = \left[\beta_0^{(j)} + \left(h_1^{(r)} \times \beta_1^{(j)}\right) + \cdots + \left(h_{t-1}^{(r)} \times \beta_{t-1}^{(j)}\right)\right] \tag{4}$$

for $0 \le r \le t-1$. Now, let $\alpha_r = \beta_r^{(i)} - \beta_r^{(j)}$ for $0 \le r \le t-1$; then $(\overrightarrow{h^{(0)}}, \overrightarrow{h^{(1)}}, \ldots, \overrightarrow{h^{(t-1)}})$ is noncovering if and only if the following linear system with unknowns $\alpha_0, \alpha_1, \ldots, \alpha_{t-1}$ has a nonzero solution over $\mathbb{F}_v$:

$$\begin{aligned}
&\alpha_0 + \left(h_1^{(0)} \times \alpha_1\right) + \left(h_2^{(0)} \times \alpha_2\right) + \cdots + \left(h_{t-1}^{(0)} \times \alpha_{t-1}\right) = 0\\
&\alpha_0 + \left(h_1^{(1)} \times \alpha_1\right) + \left(h_2^{(1)} \times \alpha_2\right) + \cdots + \left(h_{t-1}^{(1)} \times \alpha_{t-1}\right) = 0\\
&\qquad\qquad\vdots\\
&\alpha_0 + \left(h_1^{(t-1)} \times \alpha_1\right) + \left(h_2^{(t-1)} \times \alpha_2\right) + \cdots\\
&\quad + \left(h_{t-1}^{(t-1)} \times \alpha_{t-1}\right) = 0
\end{aligned} \tag{5}$$

Solving this linear system for each t-tuple of permutation vectors is time consuming. Thus, we follow the method of [28] to find the set of permutation vectors for which a nonzero tuple $(\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ solves the system (5). For each t-tuple $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ over $\mathbb{F}_v$, where at least one α_i is distinct of zero for some $1 \le i \le t-1$, consider the following equation with unknowns $h_1, h_2, \ldots, h_{t-1}$:

$$\alpha_0 + \left(h_1 \times \alpha_1\right) + \left(h_2 \times \alpha_2\right) + \cdots + \left(h_{t-1} \times \alpha_{t-1}\right) = 0 \tag{6}$$

If α_i is nonzero for some $1 \le i \le t-1$ then the corresponding h_i is obtained by assigning arbitrarily the other unknowns h_j, $j \ne i$, and solving for h_i. Therefore, there are v^{t-2} solutions in $\mathbb{F}_v$ for (6). Let A be the set of t-tuples $(\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ over $\mathbb{F}_v$ with at least one nonzero α_i for some $1 \le i \le t-1$; the cardinality of A is $v(v^{t-1} - 1)$.

To store the set of vectors that are solved by each $\alpha \in A$ we use a binary matrix H of size $|A| \times v^{t-1}$. This matrix has a row for each $\alpha \in A$ and has a column for each permutation vector; recall that the total number of permutation vectors is v^{t-1} and each one can be represented by an integer $j \in \{0, 1, \ldots, v^{t-1} - 1\}$. For each row i of H the entry at column $0 \le j \le v^{t-1}$ is 1 if and only if the permutation vector j is solved by the t-tuple $(\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ associated with row i of H; otherwise the entry at column j is 0. A t-tuple of permutation vectors $(j_0, j_1, \ldots, j_{t-1})$ is noncovering if there exists a row i of H such that $H_{i,j_l} = 1$ for $0 \le l \le t-1$; in this case the t-tuple $(\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ associated with row i of H is a nonzero solution of system (5) generated by the permutation vectors $j_0, j_1, \ldots, j_{t-1}$. On the other hand, if in every row i of H there is at least one l such that $H_{i,j_l} = 0$ then the t-tuple $(j_0, j_1, \ldots, j_{t-1})$ is covering.

This noncovering test requires in the worst case $v(v^{t-1} - 1)(t)$ operations. The strategy of [31] stores for each $\alpha \in A$ only the v^{t-2} permutation vectors solved by α and uses binary search to check if a permutation vector j_l is in the set solved by α. In that method, the worst case is $v(v^{t-1}-1)(t)(\log(v^{t-2}))$, because for each α are required t binary searches in a set of v^{t-2} elements.

For extended permutation vectors consider (7) with unknowns $h_0, h_1, \ldots, h_{t-1}$ for each nonzero t-tuple $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ over $\mathbb{F}_v$:

$$\left(h_0 \times \alpha_0\right) + \left(h_1 \times \alpha_1\right) + \left(h_2 \times \alpha_2\right) + \cdots + \left(h_{t-1} \times \alpha_{-1}\right) = 0 \tag{7}$$

In this case the nonzero element of α can be at any position, and therefore there are $v^t - 1$ nonzero t-tuples α. In addition, there are v^{t-1} solutions in $\mathbb{F}_v$ for (7) because $t-1$ of the t unknowns can be assigned arbitrarily in order to solve the equation for the remaining unknown (that must be associated with a nonzero α_i). Since there are v^t extended permutation vectors, the matrix H has v^t columns, and so its dimensions are $(v^t - 1) \times v^t$. The worst case of the noncovering test for extended permutation vectors is $(v^t - 1)(t)$.

5. Computational Results

This section shows the main computational results obtained with the SA algorithm. In Section 5.1, the results for complete

Table 4: New upper bounds of CAN(t, k, v) obtained from SCPHFs. The CPHF$(n; k, v^{t-1}, t)$ of the first column generates the CA$(n(v^t - v) + v; t, k, v)$ of the second column. The third column shows the number of rows of the previous best-known covering array, and the fourth column contains the number of upper bounds of CAN(t, k, v) improved by the covering array constructed.

SCPHF$(n; k, v^{t-1}, t)$	CA$(N; t, k, v)$	Previous N	New UBs
SCPHF$(3; 17, 3^3, 4)$	CA$(237; 4, 17, 3)$	267	1
SCPHF$(2; 12, 3^4, 5)$	CA$(483; 5, 12, 3)$	485	1
SCPHF$(4; 17, 3^4, 5)$	CA$(963; 5, 17, 3)$	1083	1
SCPHF$(5; 21, 3^4, 5)$	CA$(1203; 5, 21, 3)$	1276	2
SCPHF$(2; 12, 3^5, 6)$	CA$(1455; 6, 12, 3)$	2181	1
SCPHF$(3; 13, 3^5, 6)$	CA$(2181; 6, 13, 3)$	2667	1
SCPHF$(5; 17, 3^5, 6)$	CA$(3633; 6, 17, 3)$	3653	1
SCPHF$(4; 71, 4^2, 3)$	CA$(244; 3, 71, 4)$	246	1
SCPHF$(2; 16, 4^3, 4)$	CA$(508; 4, 16, 4)$	511	3
SCPHF$(5; 48, 4^3, 4)$	CA$(1264; 4, 48, 4)$	1396	3
SCPHF$(6; 64, 4^3, 4)$	CA$(1516; 4, 64, 4)$	1588	1
SCPHF$(4; 20, 4^4, 5)$	CA$(4084; 5, 20, 4)$	4093	1
SCPHF$(5; 28, 4^4, 5)$	CA$(5104; 5, 28, 4)$	5872	2
SCPHF$(6; 34, 4^4, 5)$	CA$(6124; 5, 34, 4)$	6640	1
SCPHF$(7; 42, 4^4, 5)$	CA$(7144; 5, 42, 4)$	7408	1
SCPHF$(8; 56, 4^4, 5)$	CA$(8164; 5, 56, 4)$	8680	2
SCPHF$(9; 69, 4^4, 5)$	CA$(9184; 5, 69, 4)$	9448	4
SCPHF$(2; 12, 4^5, 6)$	CA$(8188; 6, 12, 4)$	11260	1
SCPHF$(4; 17, 4^5, 6)$	CA$(16372; 6, 17, 4)$	18424	1
SCPHF$(5; 20, 4^5, 6)$	CA$(20464; 6, 20, 4)$	20476	1
SCPHF$(6; 24, 4^5, 6)$	CA$(24556; 6, 24, 4)$	26608	1
SCPHF$(7; 29, 4^5, 6)$	CA$(28648; 6, 29, 4)$	30700	1
SCPHF$(8; 34, 4^5, 6)$	CA$(32740; 6, 34, 4)$	33772	1
SCPHF$(9; 41, 4^5, 6)$	CA$(36832; 6, 41, 4)$	37864	1
SCPHF$(10; 50, 4^5, 6)$	CA$(40924; 6, 50, 4)$	41956	1
SCPHF$(3; 52, 5^2, 3)$	CA$(365; 3, 52, 5)$	373	2
SCPHF$(4; 115, 5^2, 3)$	CA$(485; 3, 115, 5)$	497	4
SCPHF$(5; 237, 5^2, 3)$	CA$(605; 3, 237, 5)$	621	8
SCPHF$(4; 41, 5^3, 4)$	CA$(2485; 4, 41, 5)$	2497	1
SCPHF$(2; 13, 5^4, 5)$	CA$(6245; 5, 13, 5)$	8745	1
SCPHF$(3; 18, 5^4, 5)$	CA$(9365; 5, 18, 5)$	11245	1
SCPHF$(4; 25, 5^4, 5)$	CA$(12485; 5, 25, 5)$	14365	1
SCPHF$(5; 33, 5^4, 5)$	CA$(15605; 5, 33, 5)$	16865	2
SCPHF$(6; 45, 5^4, 5)$	CA$(18725; 5, 45, 5)$	20605	2
SCPHF$(2; 12, 5^5, 6)$	CA$(31245; 6, 12, 5)$	43745	1
SCPHF$(5; 24, 5^5, 6)$	CA$(78105; 6, 24, 5)$	84365	1
SCPHF$(6; 31, 5^5, 6)$	CA$(93725; 6, 31, 5)$	103105	1
SCPHF$(3; 90, 7^2, 3)$	CA$(1015; 3, 90, 7)$	1027	2

CPHFs are presented; complete CPHFS are CPHFs where all t-combinations of columns have at least one covering tuple as a row. On the other hand, in Section 5.2, the results obtained by a three-stage procedure based on constructing quasi-CPHFs are presented; a quasi-CPHF is a CPHF with a relatively few number of uncovered combinations.

5.1. Construction of Complete CPHFs. The SA algorithm produced in total 64 new complete CPHFs whose derived covering arrays are the best-known ones. Of these results, 38 covering arrays were derived from SCPHFs and 26 were derived from CPHFs.

Table 4 shows the 38 new SCPHFs. The first column of the table shows the SCPHF$(n; k, v^{t-1}, t)$ constructed; the second column shows the CA$(n(v^t - v) + v; t, k, v)$ generated by the SCPHF in the first column; the third column shows the number of rows of the previous best-known covering array with the same t, k, v of the covering array in the second column; and the fourth column contains the number of covering array numbers improved by the covering array in the

Table 5: New upper bounds obtained from CPHFs. The CPHF$(n; k, v^t, t)$ of the first column generates the CA$(n(v^t - 1) + 1; t, k, v)$ of the second column. The third column shows the number of rows of the previous best-known covering array, and the fourth column contains the number of upper bounds of CAN(t, k, v) improved by the covering array constructed.

CPHF$(n; k, v^t, t)$	CA$(N; t, k, v)$	Previous N	New UBs
CPHF$(5; 49, 4^4, 4)$	CA$(1276; 4, 49, 4)$	1396	4
CPHF$(6; 71, 4^4, 4)$	CA$(1531; 4, 71, 4)$	1708	8
CPHF$(7; 91, 4^4, 4)$	CA$(1786; 4, 91, 4)$	1840	3
CPHF$(2; 12, 4^5, 5)$	CA$(2047; 5, 12, 4)$	2812	1
CPHF$(6; 35, 4^5, 5)$	CA$(6139; 5, 35, 4)$	6640	2
CPHF$(7; 43, 4^5, 5)$	CA$(7162; 5, 43, 4)$	7660	2
CPHF$(8; 58, 4^5, 5)$	CA$(8185; 5, 58, 4)$	8932	4
CPHF$(9; 72, 4^5, 5)$	CA$(9208; 5, 72, 4)$	9700	4
CPHF$(10; 90, 4^5, 5)$	CA$(10231; 5, 90, 4)$	10720	3
CPHF$(11; 113, 4^5, 5)$	CA$(11254; 5, 113, 4)$	11992	7
CPHF$(12; 140, 4^5, 5)$	CA$(12277; 5, 140, 4)$	12508	7
CPHF$(3; 14, 4^6, 6)$	CA$(12286; 6, 14, 4)$	14332	1
CPHF$(4; 18, 4^6, 6)$	CA$(16381; 6, 18, 4)$	19444	2
CPHF$(6; 25, 4^6, 6)$	CA$(24571; 6, 25, 4)$	27628	2
CPHF$(7; 30, 4^6, 6)$	CA$(28666; 6, 30, 4)$	30700	2
CPHF$(8; 36, 4^6, 6)$	CA$(32761; 6, 36, 4)$	34792	3
CPHF$(9; 43, 4^6, 6)$	CA$(36856; 6, 43, 4)$	37864	3
CPHF$(10; 51, 4^6, 6)$	CA$(40951; 6, 51, 4)$	42976	2
CPHF$(3; 27, 5^4, 4)$	CA$(1873; 4, 27, 5)$	2245	1
CPHF$(4; 42, 5^4, 4)$	CA$(2497; 4, 42, 5)$	2865	1
CPHF$(5; 63, 5^4, 4)$	CA$(3121; 4, 63, 5)$	3365	1
CPHF$(6; 47, 5^5, 5)$	CA$(18745; 5, 47, 5)$	20605	4
CPHF$(7; 61, 5^5, 5)$	CA$(21869; 5, 61, 5)$	23105	3
CPHF$(5; 25, 5^6, 6)$	CA$(78121; 6, 25, 5)$	87485	2
CPHF$(6; 32, 5^6, 6)$	CA$(93745; 6, 32, 5)$	103105	2
CPHF$(7; 37, 5^6, 6)$	CA$(109369; 6, 37, 5)$	112485	1

second column. Let r be the value in the last column; then, the number of rows N of the covering array CA$(N; t, k, v)$ in the second column is the new upper bound of the covering array numbers CAN(t, k, v), CAN$(t, k-1, v)$, ..., CAN$(t, k-r+1, v)$. Sizes of the best-known covering arrays were taken from the Covering Array Tables [10].

To obtain the SCPHFs shown in Table 4, the SA algorithm was launched with parameters n, k, t, v such that if constructed the SCPHF gives a covering array that improves a current upper bound. For example, a SCPHF$(5; k, 4^3, 4)$ produces a covering array CA$(1264; 4, k, 4)$ with $n(v^t - v) + v = 5(4^4 - 4) + 4 = 1264$ rows. The current upper bound of CAN$(4, 45, 4)$ is 1264, and the current upper bound of CAN$(4, 46, 4)$ is 1396 (see [10]); then, to improve a current upper bound the SCPHF must have $k \geq 46$ columns. Thus, for this particular case the SA algorithm was launched with parameters $n = 5$, $k = 46$, $t = 4$, and $v = 4$.

If the SA algorithm was able to construct a SCPHF$(n; k, v^{t-1}, t)$, then the algorithm searches for a SCPHF with one more column SCPHF$(n; k + 1, v^{t-1}, t)$. To do this, a quasi-SCPHF with $k + 1$ columns is created by appending a column generated randomly to the SCPHF with k columns. In general, this quasi-SCPHF has considerably less uncovered combinations than a quasi-SCPHF of the same size generated randomly, and therefore the SA algorithm has more possibilities of constructing a SCPHF with $k + 1$ columns. For the above-mentioned instance $n = 5$, $k = 46$, $t = 4$, $v = 4$, Table 4 shows that the SA algorithm was able to reach $k = 48$ columns.

Table 5 shows the 26 new CPHFs constructed by the SA algorithm. The best SCPHF$(n; k, v^{t-1}, t)$ constructed was used to initialize a quasi-CPHF with $k+1$ columns (where the extra column was generated randomly). From this quasi-CPHF the SA algorithm started the search of CPHF$(n; k + 1, v^t, t)$.

All SCPHFs and CPHFs constructed by the SA algorithm are small and medium-size CPHFs, according to our classification of CPHFs based on the number of columns. For large CPHFs ($k > 350$) the SA algorithm takes too much execution time, because for each change in an entry of the current solution we need to verify the $\binom{k-1}{t-1}$ subarrays of t columns affected by the change. The verification consists in determining if the subarray is covered or uncovered. However, for small CPHFs the SA algorithm produced good results. The SCPHFs and CPHFs listed in Tables 4 and 5 are the best-known ones since the covering arrays derived from them improved their respective best-known covering arrays.

Table 6: Results for strength $t = 4$.

CA constructed	Previous N	New UBs	CA constructed	Previous N	New UBs
CA(159; 4, 11, 3)	161	1	CA(471; 4, 48, 3)	487	1
CA(271; 4, 20, 3)	291	3	CA(484; 4, 49, 3)	495	1
CA(278; 4, 21, 3)	305	1	CA(488; 4, 51, 3)	499	2
CA(288; 4, 22, 3)	307	1	CA(506; 4, 53, 3)	507	2
CA(296; 4, 23, 3)	315	1	CA(509; 4, 54, 3)	512	1
CA(337; 4, 27, 3)	345	4	CA(516; 4, 57, 3)	518	2
CA(353; 4, 28, 3)	360	1	CA(526; 4, 60, 3)	531	2
CA(361; 4, 30, 3)	363	2	CA(546; 4, 66, 3)	548	3
CA(378; 4, 33, 3)	387	3	CA(559; 4, 70, 3)	561	3
CA(380; 4, 34, 3)	410	1	CA(566; 4, 71, 3)	567	1
CA(389; 4, 35, 3)	411	1	CA(573; 4, 74, 3)	579	2
CA(396; 4, 36, 3)	423	1	CA(577; 4, 76, 3)	585	2
CA(403; 4, 37, 3)	433	1	CA(579; 4, 78, 3)	591	2
CA(420; 4, 40, 3)	453	3	CA(596; 4, 81, 3)	597	3
CA(436; 4, 41, 3)	455	1	CA(602; 4, 82, 3)	603	1
CA(441; 4, 42, 3)	460	1	CA(605; 4, 85, 3)	609	3
CA(450; 4, 43, 3)	471	1	CA(614; 4, 88, 3)	615	3
CA(453; 4, 44, 3)	472	1	CA(624; 4, 92, 3)	627	4
CA(454; 4, 45, 3)	474	1	CA(630; 4, 94, 3)	633	2
CA(463; 4, 46, 3)	475	1	CA(637; 4, 97, 3)	639	3
CA(465; 4, 47, 3)	481	1	CA(639; 4, 99, 3)	645	2

The total number of upper bounds of covering array numbers CAN(t, k, v) improved by the covering arrays in Tables 4 and 5 is 62 + 75 = 137.

5.2. Construction of Quasi-CPHFs. In this section, we employ the SA algorithm given in Algorithm 1 to construct quasi-CPHFs with a relatively few number of uncovered combinations, say less than or equal to 5% of the total number of t-combinations of columns. The advantage of constructing quasi-CPHFs instead of complete CPHFs is that quasi-CPHFs are constructed in less time, and also quasi-CPHFs may be constructed for values of n, t, k, v for which a complete CPHF$(n; k, v^t, t)$ may not even exist. To construct quasi-CPHFs, Algorithm 1 was modified to finalize when the number of uncovered combinations in the best global solution is less than or equal to $0.05\binom{k}{t}$.

The array derived from a quasi-CPHF is a quasi-CA (quasi covering array) with some missing tuples. To be a covering array with strength t, a matrix of size $N \times k$ over $\mathbb{Z}_v$ must satisfy the property that every submatrix of t columns contains every t-tuple over $\mathbb{Z}_v$ at least once. The t-tuples not covered in a subarray of t columns are the missing tuples in that subarray.

Let A be the array of size $(n(v^t - 1) + 1) \times k$ derived from a quasi-CPHF$(n; k, v^t, t)$ or the array of size $(n(v^t - v) + v) \times k$ derived from a quasi-SCPHF$(n; k, v^{t-1}, t)$. To cover the missing tuples in the submatrices S of t columns from A we use a greedy algorithm that works in two steps. The first step of the algorithm is to determine which t-tuples over $\mathbb{Z}_v$ are missing in S; these missing tuples are stored in a list π. The second step covers the tuples in π by adding new rows to A or by overwriting unassigned elements in the rows previously added to cover a tuple in π. For each tuple of π, the algorithm first searches if there is a candidate row to cover the tuple by overwriting some unassigned elements of the row. Suppose S is formed by columns $j_0, j_1, \ldots, j_{t-1}$, then a row $r = (r_0, r_1, \ldots, r_{k-1})$ is a candidate row to cover a tuple $x = (x_0, x_1, \ldots, x_{t-1})$ if for $0 \le l \le t - 1$ either $r_{j_l} = x_l$ or r_{j_l} is unassigned. If there is no candidate row to cover a tuple x, then a new row is added to A. The final result of the greedy algorithm is a complete covering array that has a number of unassigned or redundant elements.

The redundant elements are elements that can be freely modified without affecting the coverage properties of a covering array; so these elements are not needed to satisfy the covering conditions. To reduce redundancy in the covering array, we use the postoptimization method of [35]. This postoptimization method deletes some rows from a covering array by copying their nonredundant elements to the redundant elements of other rows.

Because of time constraints we only apply the three-stage procedure (construction of a quasi-CPHF, coverage of missing tuples, and postoptimization) to order $v = 3$ and strength $t \in \{4, 5, 6\}$. A similar three-stage procedure was developed in [36].

For $t = 4$ we constructed CAs with up to $k = 99$ columns, and the relevant results obtained are shown in Table 6. The first column of the table contains the covering array generated by the three-stage procedure; the second column of the table contains the number of rows of the previous best-known covering array; and the third column contains the number of upper bounds of covering array numbers improved. For $t = 5$

Table 7: Results for strength $t = 5$.

CA constructed	Previous N	New UBs	CA constructed	Previous N	New UBs
CA(687; 5, 13, 3)	723	1	CA(2157; 5, 59, 3)	2211	1
CA(805; 5, 14, 3)	885	1	CA(2200; 5, 60, 3)	2229	1
CA(842; 5, 15, 3)	939	1	CA(2258; 5, 61, 3)	2370	1
CA(920; 5, 16, 3)	963	1	CA(2266; 5, 62, 3)	2387	1
CA(1034; 5, 18, 3)	1143	2	CA(2284; 5, 63, 3)	2397	1
CA(1064; 5, 19, 3)	1190	1	CA(2288; 5, 64, 3)	2413	1
CA(1108; 5, 20, 3)	1239	1	CA(2297; 5, 65, 3)	2439	1
CA(1200; 5, 22, 3)	1311	2	CA(2304; 5, 66, 3)	2447	1
CA(1258; 5, 23, 3)	1360	1	CA(2308; 5, 67, 3)	2459	1
CA(1302; 5, 24, 3)	1394	1	CA(2318; 5, 68, 3)	2477	1
CA(1350; 5, 25, 3)	1435	1	CA(2327; 5, 70, 3)	2504	2
CA(1371; 5, 26, 3)	1474	1	CA(2354; 5, 71, 3)	2518	1
CA(1413; 5, 27, 3)	1511	1	CA(2361; 5, 72, 3)	2531	1
CA(1435; 5, 28, 3)	1542	1	CA(2364; 5, 73, 3)	2546	1
CA(1480; 5, 29, 3)	1575	1	CA(2379; 5, 74, 3)	2564	1
CA(1520; 5, 30, 3)	1600	1	CA(2391; 5, 75, 3)	2588	1
CA(1541; 5, 31, 3)	1635	1	CA(2403; 5, 76, 3)	2593	1
CA(1580; 5, 32, 3)	1659	1	CA(2419; 5, 77, 3)	2608	1
CA(1600; 5, 33, 3)	1695	1	CA(2427; 5, 78, 3)	2615	1
CA(1630; 5, 34, 3)	1707	1	CA(2438; 5, 79, 3)	2625	1
CA(1661; 5, 35, 3)	1743	1	CA(2450; 5, 80, 3)	2635	1
CA(1681; 5, 36, 3)	1767	1	CA(2457; 5, 81, 3)	2651	1
CA(1693; 5, 37, 3)	1785	1	CA(2488; 5, 82, 3)	2668	1
CA(1730; 5, 38, 3)	1821	1	CA(2494; 5, 83, 3)	2677	1
CA(1760; 5, 39, 3)	1839	1	CA(2497; 5, 84, 3)	2684	1
CA(1780; 5, 40, 3)	1863	1	CA(2505; 5, 85, 3)	2700	1
CA(1811; 5, 41, 3)	1887	1	CA(2519; 5, 87, 3)	2728	2
CA(1838; 5, 42, 3)	1899	1	CA(2544; 5, 89, 3)	2747	2
CA(1855; 5, 44, 3)	1941	2	CA(2548; 5, 90, 3)	2761	1
CA(1870; 5, 45, 3)	1965	1	CA(2617; 5, 97, 3)	2834	7
CA(1903; 5, 46, 3)	1989	1	CA(2626; 5, 98, 3)	2838	1
CA(1931; 5, 47, 3)	2007	1	CA(2646; 5, 99, 3)	2854	1
CA(1947; 5, 48, 3)	2025	1	CA(2649; 5, 100, 3)	2862	1
CA(1975; 5, 49, 3)	2043	1	CA(2660; 5, 101, 3)	2871	1
CA(2041; 5, 50, 3)	2067	1	CA(2675; 5, 102, 3)	2878	1
CA(2054; 5, 51, 3)	2085	1	CA(2681; 5, 103, 3)	2898	1
CA(2068; 5, 52, 3)	2103	1	CA(2690; 5, 104, 3)	2901	1
CA(2076; 5, 53, 3)	2115	1	CA(2699; 5, 105, 3)	2908	1
CA(2098; 5, 54, 3)	2133	1	CA(2880; 5, 108, 3)	2934	3
CA(2109; 5, 55, 3)	2157	1	CA(2907; 5, 110, 3)	2955	2
CA(2125; 5, 56, 3)	2169	1	CA(2942; 5, 114, 3)	2993	4
CA(2131; 5, 57, 3)	2187	1	CA(3140; 5, 148, 3)	3156	32
CA(2150; 5, 58, 3)	2205	1			

we generate CAs with maximum number of columns $k = 148$, and the more important results are shown in Table 7. Finally, Table 8 displays the main results for $t = 6$.

The accumulated sum of the results in Tables 6, 7, and 8 is 183 new covering arrays and 546 new upper bounds of covering array numbers CAN(t, k, v).

6. Conclusions

In this work, we developed a simulated annealing algorithm to construct covering perfect hash families (CPHFs) formed by either permutation vectors or extended permutation vectors. CPHFs formed by permutation vectors are denominated

Table 8: Results for strength $t = 6$.

CA constructed	Previous N	New UBs	CA constructed	Previous N	New UBs
CA(1431; 6, 11, 3)	1449	1	CA(7348; 6, 45, 3)	7702	1
CA(2701; 6, 14, 3)	2907	1	CA(7482; 6, 46, 3)	7766	1
CA(2901; 6, 15, 3)	3216	1	CA(7567; 6, 47, 3)	7856	1
CA(3126; 6, 16, 3)	3433	1	CA(7576; 6, 50, 3)	8108	3
CA(3961; 6, 19, 3)	4050	1	CA(7635; 6, 51, 3)	8179	1
CA(4006; 6, 20, 3)	4359	1	CA(7656; 6, 52, 3)	10179	1
CA(4200; 6, 21, 3)	4534	1	CA(7746; 6, 54, 3)	10419	2
CA(4400; 6, 22, 3)	4699	1	CA(7797; 6, 55, 3)	10659	1
CA(4600; 6, 23, 3)	4854	1	CA(7999; 6, 58, 3)	10665	3
CA(4661; 6, 24, 3)	5032	1	CA(8687; 6, 61, 3)	10911	3
CA(4880; 6, 25, 3)	5178	1	CA(8758; 6, 62, 3)	11145	1
CA(5020; 6, 26, 3)	5351	1	CA(8955; 6, 63, 3)	11151	1
CA(5200; 6, 27, 3)	5478	1	CA(9025; 6, 64, 3)	11385	1
CA(5359; 6, 28, 3)	5631	1	CA(9033; 6, 66, 3)	11619	2
CA(5410; 6, 29, 3)	5757	1	CA(9154; 6, 68, 3)	11631	2
CA(5572; 6, 30, 3)	5883	1	CA(9249; 6, 73, 3)	12111	5
CA(5691; 6, 31, 3)	6245	1	CA(9997; 6, 82, 3)	12831	9
CA(5786; 6, 32, 3)	6348	1	CA(10406; 6, 86, 3)	12993	4
CA(5825; 6, 35, 3)	6715	3	CA(10490; 6, 88, 3)	13089	2
CA(6248; 6, 36, 3)	6832	1	CA(10673; 6, 90, 3)	13323	2
CA(6316; 6, 37, 3)	6932	1	CA(10927; 6, 96, 3)	13563	6
CA(6454; 6, 38, 3)	7036	1	CA(10942; 6, 98, 3)	13791	2
CA(6583; 6, 39, 3)	7131	1	CA(10974; 6, 120, 3)	15249	22
CA(6663; 6, 40, 3)	7233	1	CA(12297; 6, 136, 3)	15975	16
CA(6755; 6, 41, 3)	7315	1	CA(14592; 6, 186, 3)	17427	50
CA(6870; 6, 42, 3)	7411	1	CA(15310; 6, 219, 3)	41141	33
CA(7153; 6, 43, 3)	7506	1	CA(19657; 6, 300, 3)	48965	81
CA(7264; 6, 44, 3)	7600	1	CA(14467; 6, 350, 3)	49598	50

Sherwood-CPHFs or SCPHFs. For the same number of columns k and for the same $n > 1$, a SCPHF$(n; k, v^{t-1}, t)$ is better than a CPHF$(n; k, v^t, t)$ because the first generates a CA$(n(v^t - v) + v; t, k, v)$, while the second generates a CA$(n(v^t - 1) + 1; t, k, v)$. However, sometimes CPHFs can be constructed with more columns than SCPHFs.

The simulated annealing algorithm developed in this work has a neighborhood function composed of three perturbation operators whose probabilities of being used in an application of the neighborhood function were tuned by experimentation. The use of a compound neighborhood function allows combining exploration and exploitation operators in the neighborhood function, and this is very important for the success of the algorithm. The results obtained from the simulated annealing algorithm were 64 new CPHFs whose derived covering arrays improved the best-known ones. In addition, the simulated annealing algorithm was used to construct quasi-CPHFs whose derived arrays are then completed and postprocessed; in this case the number of new covering arrays constructed was 183. Then, in total we constructed $64 + 183 = 247$ new covering arrays; and these 247 covering arrays improved in total the upper bound of 683 covering array numbers.

The construction of CPHFs using metaheuristic algorithms can be further improved by developing more sophisticated neighborhood functions or by employing parallel computing to handle larger instances.

Data Availability

All the input data needed to produce the output data is given in the paper; in case output data is needed write to the corresponding author.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors acknowledge ABACUS-CINVESTAV, CONACYT Grant, EDOMEX-2011-COI-165873 for providing access of high performance computing and General Coordination of Information and Communications Technologies (CGSTIC) at CINVESTAV for providing HPC resources on the Hybrid Cluster Supercomputer "Xiuhcoatl". The CONACyT

Métodos Exactos para Construir Covering Arrays Óptimos Project 238469 has funded partially the research reported in this paper.

References

[1] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," Tech. Rep., United States,, National Institute of Standards Technology, Gaithersburg, MD, 2010.

[2] P. Kitsos, D. E. Simos, J. Torres-Jimenez, and A. G. Voyiatzis, "Exciting FPGA cryptographic Trojans using combinatorial testing," in *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015*, pp. 69–76, usa, November 2015.

[3] K. A. Bush, "Orthogonal arrays of index unity," *Annals of Mathematical Statistics*, vol. 23, pp. 426–434, 1952.

[4] G. O. Katona, "Two applications (for search theory and truth functions) of Sperner type theorems," *Periodica Mathematica Hungarica*, vol. 3, pp. 19–26, 1973.

[5] D. J. Kleitman and J. Spencer, "Families of k-independent sets," *Discrete Mathematics*, vol. 6, pp. 255–262, 1973.

[6] K. A. Johnson and R. Entringer, "Largest induced subgraphs of the n-cube that contain no 4-cycles," *Journal of Combinatorial Theory, Series B*, vol. 46, no. 3, pp. 346–355, 1989.

[7] C. J. Colbourn, G. Kéri, P. P. Soriano, and J.-C. Schlage-Puchta, "Covering and radius-covering arrays: constructions and classification," *Discrete Applied Mathematics: The Journal of Combinatorial Algorithms, Informatics and Computational Sciences*, vol. 158, no. 11, pp. 1158–1180, 2010.

[8] J. Lawrence, R. N. Kacker, Y. Lei, D. Kuhn, and M. Forbes, "A survey of binary covering arrays," *Electronic Journal of Combinatorics*, vol. 18, no. 1, Paper 84, 30 pages, 2011.

[9] S. Choi, H. K. Kim, and D. Y. Oh, "Structures and lower bounds for binary covering arrays," *Discrete Mathematics*, vol. 312, no. 19, pp. 2958–2968, 2012.

[10] C. J. Colbourn, "Covering array tables for t = 2, 3, 4, 5, 6," Accessed on May 14, 2018, http://www.public.asu.edu/~ccolbou/src/tabby/catable.html.

[11] C. J. Colbourn, "Combinatorial aspects of covering arrays," *Le Matematiche*, vol. 59, no. 1-2, pp. 125–172 (2006), 2004.

[12] V. V. Kuliamin and A. A. Petukhov, "A survey of methods for constructing covering arrays," *Programming and Computer Software*, vol. 37, no. 3, pp. 121–146, 2011.

[13] J. Torres-Jimenez and I. Izquierdo-Marquez, "Survey of covering arrays," in *Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 20–27, rou, 2013.

[14] J. Zhang, Z. Zhang, and F. Ma, *Automatic Generation of Combinatorial Test Data*, Incorporated, Springer Publishing Company, 2014.

[15] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics: The Journal of Combinatorial Algorithms, Informatics and Computational Sciences*, vol. 138, no. 1-2, pp. 143–152, 2004.

[16] S. Esfandyari and V. Rafe, "A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy," *Information and Software Technology*, vol. 94, pp. 165–185, 2018.

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

[18] X. Chen, Q. Gu, A. Li, and D. Chen, "Variable Strength Interaction Testing with an Ant Colony System Approach," in *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 160–167, Batu Ferringhi, Penang, Malaysia, December 2009.

[19] T. Mahmoud and B. S. Ahmed, "An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use," *Expert Systems with Applications*, vol. 42, no. 22, pp. 8753–8765, 2015.

[20] X. Bao, S. Liu, N. Zhang, and M. Dong, "Combinatorial Test Generation using Improved Harmony Search Algorithm," *International Journal of Hybrid Information Technology*, vol. 8, no. 9, pp. 121–130, 2015.

[21] M. Bazargani, J. H. Drake, and E. K. Burke, "Late Acceptance Hill Climbing for Constrained Covering Arrays," in *Applications of Evolutionary Computation*, vol. 10784 of *Lecture Notes in Computer Science*, pp. 778–793, Springer International Publishing, Cham, 2018.

[22] Y. Zhang, L. Cai, and W. Ji, "Combinatorial testing data generation based on bird swarm algorithm," in *Proceedings of the 2017 2nd International Conference on System Reliability and Safety (ICSRS)*, pp. 491–499, Milan, December 2017.

[23] P. Bansal, S. Sabharwal, and N. Mittal, "A hybrid artificial bee colony and harmony search algorithm to generate covering arrays for pair-wise testing," *International Journal of Intelligent Systems and Applications*, vol. 9, no. 8, pp. 59–70, 2017.

[24] B. S. Ahmed, T. S. Abdulsamad, and M. Y. Potrus, "Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the Cuckoo Search algorithm," *Information and Software Technology*, vol. 66, pp. 13–29, 2015.

[25] Y. Wang, M. Zhou, X. Song, M. Gu, and J. Sun, "Constructing Cost-Aware Functional Test-Suites Using Nested Differential Evolution Algorithm," *IEEE Transactions on Evolutionary Computation*, 2017.

[26] K. Z. Zamli, B. Y. Alkazemi, and G. Kendall, "A Tabu Search hyper-heuristic strategy for t-way test suite generation," *Applied Soft Computing*, vol. 44, pp. 57–74, 2016.

[27] J. Torres-Jimenez, H. Avila-George, and I. Izquierdo-Marquez, "A two-stage algorithm for combinatorial testing," *Optimization Letters*, vol. 11, no. 3, pp. 457–469, 2017.

[28] G. B. Sherwood, S. S. Martirosyan, and C. J. Colbourn, "Covering arrays of higher strength from permutation vectors," *Journal of Combinatorial Designs*, vol. 14, no. 3, pp. 202–213, 2006.

[29] J. Torres-Jimenez and I. Izquierdo-Marquez, "Covering arrays of strength three from extended permutation vectors," *Designs, Codes and Cryptography*, 2018.

[30] C. J. Colbourn, E. Lanus, and K. Sarkar, "Asymptotic and constructive methods for covering perfect hash families and covering arrays," *Designs, Codes and Cryptography. An International Journal*, vol. 86, no. 4, pp. 907–937, 2018.

[31] I. Walker and C. J. Colbourn, "Tabu search for covering arrays using permutation vectors," *Journal of Statistical Planning and Inference*, vol. 139, no. 1, pp. 69–80, 2009.

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

[33] H. Avila-George, J. Torres-Jimenez, and V. Hernández, "New Bounds for Ternary Covering Arrays Using a Parallel Simulated Annealing," *Mathematical Problems in Engineering*, vol. 2012, pp. 1–19, 2012.

[34] E. Aarts and J. K. Lenstra, Eds., *Local search in combinatorial optimization*, John Wiley & Sons, Ltd., New York, NY, USA, 1st edition, 1997.

[35] J. Torres-Jimenez and A. Rodriguez-Cristerna, "Metaheuristic post-optimization of the NIST repository of covering arrays," *CAAI Transactions on Intelligence Technology*, vol. 2, no. 1, pp. 31–38, 2017.

[36] I. Izquierdo-Marquez, J. Torres-Jimenez, B. Acevedo-Juárez, and H. Avila-George, "A greedy-metaheuristic 3-stage approach to construct covering arrays," *Information Sciences*, vol. 460-461, pp. 172–189, 2018.