New bounds for binary covering arrays using simulated annealing[☆]

J. Torres-Jimenez, E. Rodriguez-Tello*

CINVESTAV-Tamaulipas, Information Technology Laboratory
Km. 5.5 Carretera Victoria-Soto La Marina, 87130 Victoria Tamps., MEXICO

Abstract

Covering arrays (CAs) are combinatorial structures specified as a matrix of N rows and k columns over an alphabet on v symbols such that for each set of t columns (called the strength of the array) every t-tuple of symbols is covered. Recently they have been used to represent interaction test suites for software testing given that they provide economical sized test cases while still preserving important fault detection capabilities.

This paper introduces an improved implementation of a Simulated Annealing algorithm, called ISA, for constructing CAs of strengths three through six over a binary alphabet (i.e., binary CAs). Extensive experimentation is carried out, using 127 well-known benchmark instances, for assessing its performance with respect to an existing simulated annealing implementation, a greedy method, and five state-of-the-art algorithms. The results show that our algorithm attains 104 new bounds and equals the best-known solutions for the other 23 instances consuming reasonable computational time. Furthermore, the implications of using these results as ingredients to recursive constructions are also analyzed.

Key words: covering arrays, best-known bounds, simulated annealing, software interaction testing

1. Introduction

A covering array, CA$(N; t, k, v)$, of size N, strength t, degree k, and order v is an $N \times k$ array on v symbols such that every $N \times t$ sub-array includes, at least once, all the ordered subsets from v symbols of size t (t-tuples) [17]. The minimum N for which a CA$(N; t, k, v)$ exists is the *covering array number* and it is defined according to (1).

$$\text{CAN}(t, k, v) = min\{N : \exists\, \text{CA}(N; t, k, v)\} \tag{1}$$

Finding the covering array number is also known in the literature as the Covering Array Construction (CAC) problem. It is equivalent to the problem of maximizing the degree k of a covering array given the values N, t, and v [54].

There exist only some special cases where it is possible to find the covering array number using polynomial order algorithms. For instance, the case $t = v = 2$ was solved (for k even) by Rényi [47] and independently (for all k) by Kleitman and Spencer [32]. Additionally, in [39, 46] the case $N = v^t$, $t = 2$, $k = v + 1$ was completely solved when $v = p^\alpha$ is a prime or a power of prime and $v > t$. This case was later generalized by Bush for $t > 2$ [8]. According to Lawrence et al. [35] it remains still open the problem of determining the NP-completeness of the CAC problem in the general case. However, there exist certain closely related problems which are NP-complete [52, 17, 11], suggesting that the CAC problem is a hard combinatorial optimization problem.

A major application of covering arrays (CAs) arises in *software interaction testing*, where a covering array can be used to represent an interaction test suite as follows. In a software test we have k components or factors. Each of these

[☆]This research work was partially funded by the following projects: CONACyT 58554, Cálculo de Covering Arrays; CONACyT 99276, Algoritmos para la Canonización de Covering Arrays; 51623, Fondo Mixto CONACyT - Gobierno del Estado de Tamaulipas.

*Corresponding author.

Email addresses: jtj@cinvestav.mx (J. Torres-Jimenez), ertello@tamps.cinvestav.mx (E. Rodriguez-Tello)

has v values or levels. A test suite is an $N \times k$ array where each row is a test case. Each column represents a component and a value in the column is the particular configuration. By mapping a software test problem to a covering array of strength t we can guarantee that we have tested, at least once, all t-way combinations of component values [59]. Thus, software testing costs can be substantially reduced by minimizing the number of test cases N in the covering array [13]. Please observe that software interaction testing is a black-box testing technique, thus it exhibits weaknesses that should be addresses by employing white-box testing techniques. For a detailed example of the use of CAs in software interaction testing the reader is referred to [27].

Other applications related to the CAC problem arise in diverse fields like: data compression, regulation of gene expression, authentication, intersecting codes, universal hashing and some experimental design applications like drug screening. The reader is referred to [10] for a detailed survey.

The binary case of the CAC problem has been extensively studied in the literature because of its theoretical and practical importance [35]. In this paper, we aim to develop an improved implementation of a Simulated Annealing algorithm (hereafter called ISA) for constructing binary CAs[1] of strengths three through six for their use in software interaction testing. This is because previously published empirical studies demonstrate that testing all the 6-way combinations of components can provide high confidence that nearly all faults, triggered by a combination of 6 or fewer factors, in a software system can be discovered [33, 34].

Contrary to existing simulated annealing implementations [15, 55], our algorithm has the merit of improving two key features that have a great impact on its performance: an efficient heuristic to generate good quality initial solutions and a compound neighborhood function which combines two carefully designed neighborhood relations.

The performance of ISA is assessed through extensive experimentation over a set of well known benchmark instances [54, 10, 42, 23], which includes 127 binary CAs of strengths three through six, and compared with several state-of-the-art algorithms. Computational results show that our ISA algorithm has the advantage of producing smaller CAs than the other compared methods, at a moderate computational cost, and without having important fluctuations on its average performance. Indeed, ISA attains 104 new bounds and equals the best-known solutions for the other 23 selected instances. In practical terms, ISA offers a good trade-off between the needs for small test suites, fast test suite construction, and maximum number of errors detected in the context of software testing.

The remainder of this paper is organized as follows. In Section 2, a brief review is given to present the most representative solution procedures for the CAC problem. Then, the components of our ISA algorithm are discussed in detail in Section 3. Section 4 is dedicated to computational experiments aiming at comparing the ISA algorithm with several previously published methods for constructing CAs. Some implications of the results achieved from these computational experiments, when used as ingredients to recursive constructions, are also analyzed. Section 5 presents experiments carried out to study the influence of some key features in the global performance of the proposed ISA algorithm. Finally, the advantages of using our ISA algorithm for generating software interaction test cases are discussed in Section 6, which also presents some possible directions for future research.

2. Relevant related work

Addressing the problem of finding the covering array number in reasonable time has been the focus of much research. Among the approximate methods that have been developed for constructing CAs are: a) recursive methods [28, 41], b) algebraic methods [29, 10, 27], c) greedy methods [12, 6] and d) metaheuristics such as simulated annealing [16, 15], genetic algorithms [55], memetic algorithms [50] and tabu search [42]. Most of these algorithms are specially designed for constructing strength two and three CAs. Next, we give a brief review of some representative procedures which were used in our experimental comparisons.

In 1952, a generalization of the concept of a set of orthogonal Latin squares, called *orthogonal arrays of index unity*, was presented by Bush [8]. In his paper Bush introduced a very ingenious procedure for constructing these arrays. It employs a special class of polynomials which have coefficients in the finite Galois field GF(v), where $v = p^{\alpha}$ is a prime or a power of prime and $v > t$. The resulting orthogonal arrays of index unity are equivalent to CAs of size $N = v^t$, strength $t > 2$ and degree $k = v + 1$.

[1] i.e., covering arrays of order $v = 2$

Besides Bush's work, Sloane published in [54] a procedure which improved some elements of the work reported in Roux's PhD dissertation [51]. This procedure constructs a CAN$(3, 2k, 2)$ by combining two CAs with the following characteristics: CA$(N_2; 2, k, 2)$ and CA$(N_3; 3, k, 2)$. It starts by appending CA$(N_2; 2, k, 2)$ to a CA$(N_3; 3, k, 2)$, which results into a $k \times (N_2 + N_3)$ array. Then this array is copied below itself, producing a $2k \times (N_2 + N_3)$ array. Finally, the copied strength two array is replaced by its bit-complement array (i.e., switch 0 to 1 and 1 to 0).

Following these ideas, Chateauneuf and Kreher presented later in [10] an algebraic procedure designed for constructing CAN$(3, 2k, v)$. Let A =CA$(N_3; 3, k, v)$ and B =CA$(N_2; 2, k, v)$ be two CAs on the symbol set $V = \{1, 2, \ldots, v\}$ and π^i the i-th cyclic permutation on the symbols of V $(1 \leq i \leq v - 1)$. A CAN$(3, 2k, v)$ can be constructed as follows:

$$\text{CAN}(3, 2k, v) = \begin{pmatrix} A & B & B & \ldots & B \\ A & B^{\pi^1} & B^{\pi^2} & \ldots & B^{\pi^{v-1}} \end{pmatrix} \tag{2}$$

where B^{π^i} is the array obtained by applying π^i, to each symbol of B. This polynomial time algorithm has permitted to attain some of the best-known bounds for binary CAs of strength three.

Later, Martirosyan proposed in her PhD thesis [40] recursive methods which generalize some Roux type constructions [51, 54] to produce a CAN$(t, 2k, v)$ for any $t \geq 4$ and $v \geq 2$. For instance, a CAN$(4, 2k, v)$ can be constructed by combining three other CAs following the procedure described in [41]. Let A be a CA$(N_4; 4, k, v)$, B a CA$(N_3; 3, k, v)$, C a CA$(N_2; 2, k, v)$, and D a CA$(N_1; 2, v, v)$, all on the symbol set $V = \{1, 2, \ldots, v\}$. This procedure starts by copying A below itself which results in E_1, a $2k \times N_4$ array. Then, a $2k \times N_3$ array, called E_2, is constructed from B as follows:

$$E_2 = \begin{pmatrix} B & B & \ldots & B \\ B^{\pi^1} & B^{\pi^2} & \ldots & B^{\pi^{v-1}} \end{pmatrix} \tag{3}$$

where B^{π^i} is the array resulting from applying the cyclic permutations π^i on the symbol set V of B $(1 \leq i \leq v - 1)$. Next, a third $2k \times N_2$ array E_3 is created based on C and D as indicated in (4),

$$E_3 = \begin{pmatrix} C & C & \ldots & C \\ C^{f_1} & C^{f_2} & \ldots & C^{f_{N_1}} \end{pmatrix} \tag{4}$$

where C^{f_i} is the array obtained by applying a function $f_i : V \to V$, which maps the vector $(1, 2, \ldots, v)$ to the i-th column of D, i.e., $f_i(j) = d_{j,i}$. Using the same principle employed to build E_3, a last array, E_4, is composed in the following form:

$$E_4 = \begin{pmatrix} C^{f_1} & C^{f_2} & \ldots & C^{f_{N_1}} \\ C & C & \ldots & C \end{pmatrix}. \tag{5}$$

Finally the arrays E_1, E_2, E_3 and E_4 are all combined as shown in (6) to form the resulting covering array of size $N = N_4 + (v - 1)N_3 + 2N_2N_1$.

$$\text{CAN}(4, 2k, v) = \begin{pmatrix} E_1 & E_2 & E_3 & E_4 \end{pmatrix}. \tag{6}$$

Some improvements to this procedure were presented later by Colbourn et al. in [20]. The improved procedure permitted the authors to attain some of the best-known bounds for binary CAs of strength four. Although, this procedure has the merit to be a polynomial time algorithm it does not return the optimal solution in all the cases.

In [19] Colbourn and Kéri proposed a method for the explicit construction of binary CAs of strengths four and five by using existentially closed graphs and Hadamard matrices. This method is based on the observation that the adjacency matrix of a t-existentially closed graph on v vertices is a CA$(n; t, n, 2)$. In particular, the authors explored connections arising from the Paley graphs and tournaments and demonstrated that the proposed approach allowed them to obtain substantial improvements on the best-known bounds on CAN$(4, k, 2)$ and CAN$(5, k, 2)$ for $k \leq 10000$.

A Tabu Search (TS) algorithm for solving this problem was developed by Nurmela [42]. This algorithm starts with an $N \times k$ randomly generated matrix that represents a potential covering array. The number of uncovered t-tuples is used to evaluate the cost of a candidate solution (matrix)[2]. Next an uncovered t-tuple is selected at random and the rows of the matrix are searched to find those that require only the change of a single element in order to cover

[2] i.e., the number of ordered subsets from v symbols of size t not present in the candidate solution

the selected t-tuple. These changes, called *moves*, correspond to the neighboring solutions of the current candidate solution. The variation of cost corresponding to each such move is calculated and the move having the smallest cost is selected, provided that the move is not tabu. If there are several equally good non-tabu moves, one of them is randomly chosen. Then another uncovered t-tuple is selected and the process is repeated until a matrix with zero cost (a covering array) is found or a predefined maximum number of moves is reached. The tabu condition prevents changing an element of the matrix, if it has been changed during the last T moves. This feature prevents looping and increases the exploration capacity of the algorithm.

The results produced by Nurmela's TS implementation have demonstrated that it is able to slightly improve some previous best-known solutions, specially the instance CA$(15; 3, 12, 2)$. However, an important drawback of this algorithm is that it consumes considerably much more computational time than any of the previously presented algorithms.

In [58], another TS algorithm was presented by Walker and Colbourn. It employs a compact representation of CAs based on permutation vectors and covering perfect hash families [53] in order to reduce the size of the search space. Using this algorithm, improved CAs of strengths three to five have been found, as well as the first arrays of strength six and seven found by computational search.

A Simulated Annealing metaheuristic (hereafter called SAC) has been applied by Cohen et al. in [16] for solving the CAC problem. Their implementation starts with a randomly generated initial solution A with cost $c(A)$ measured as the number of uncovered t-tuples. A series of iterations is then carried out to visit the search space according to a neighborhood. At each iteration, a neighboring solution A' is generated by changing the value of the element $a_{i,j}$ by a different legal member of the alphabet in the current solution A. The cost of this iteration is evaluated as $\Delta_c = c(A') - c(A)$. If Δ_c is negative or equal to zero then the neighboring solution A' is accepted. Otherwise, it is accepted with probability $P(\Delta_c) = e^{-\Delta_c/T_n}$, where T_n is determined by a cooling schedule. In their implementation, Cohen et al. use a simple linear function $T_n = 0.9998T_{n-1}$ with an initial temperature fixed at $T_i = 0.20$. At each temperature, 2000 neighboring solutions are generated. The algorithm stops either if a valid covering array is found, or if no change in the cost of the current solution is observed after 500 trials. The authors justify their choice of these parameter values based on some experimental tuning. They conclude that their algorithm is able to produce smaller CAs than other computational methods, sometimes improving upon algebraic constructions. However, they also indicate that SAC fails to match the algebraic constructions for larger problems, specially when $t = 3$ [16].

In [15], Cohen et al. proposed a hybrid metaheuristic called Augmented Annealing. It employs recursive and direct combinatorial constructions to produce small building blocks which are then augmented with a simulated annealing algorithm to construct a covering array. This method has been successfully used to construct CAs that are smaller than those created by using their simple SAC algorithm.

Bryce and Colbourn published in [5] a method called Deterministic Density Algorithm (DDA) which constructs strength two CAs one row at a time using a steepest ascent approach. In this algorithm the value for each column k is dynamically fixed one at a time in an order based on a quantity called density δ, which indicates the fraction of pairs of assignments to columns remaining to be tested. In DDA new rows are continually added making selections to increase the density as much as possible. This process continues until all interactions have been covered. Later the authors extended DDA to generate CAs of strength $t \geq 3$ [6]. The main advantage of DDA over other one-row-at-a-time methods is that it provides a worst-case logarithmic guarantee on the size N of the covering array. Moreover, it is able to produce CAs that are of competitive size, and expending less computational time than other published methods like TS [42] and SAC [15].

More recently Forbes et al. [23] introduced an algorithm for the efficient production of CAs of strength t up to 6, called IPOG-F (In-Parameter Order-Generalized). Contrary to many other algorithms that build CAs one row at a time, the IPOG-F strategy constructs them one column at a time. The main idea is that CAs of $k - 1$ columns can be used to efficiently build a covering array with degree k. In order to construct a covering array, IPOG-F initializes a $v^t \times t$ matrix which contains each of the possible v^t distinct rows having entries from $\{0, 1, \ldots, v - 1\}$. Then, for each additional column, the algorithm performs two steps, called *horizontal growth* and *vertical growth*. Horizontal growth adds an additional column to the matrix and fills in its values, then any remaining uncovered t-tuples are covered in the vertical growth stage. The choice of which rows will be extended with which values is made in a greedy manner: it picks an extension of the matrix that covers as many previously uncovered t-tuples as possible.

IPOG-F is currently implemented in a software package called FireEye [36], which was written in Java. Even if IPOG-F is a very fast algorithm for producing CAs it generally provides poorer quality results than other state-of-the-art algorithms. An exception is the case CA$(N; 6, k, 2)$ for $51 \leq k \leq 64$ where IPOG produces the smallest known

CAs [18].

In [21], a modern construction method which takes advantage of small near optimal CAs is presented. It employs those small CAs as input ingredients, and combines them using Perfect Hash Families (PHF). In Section 4.5 we show that several new bounds can be achieved by this construction method when it employs as input ingredients certain CAs constructed by our ISA algorithm.

The reader is referred to [17], [27] and [35] for detailed surveys on covering array construction methods.

3. An improved implementation of a simulated annealing algorithm

Simulated Annealing (SA)[3] is a general-purpose stochastic optimization technique that has proved to be an effective tool for approximating globally optimal solutions to many NP-hard optimization problems. However, it is well known that developing an effective SA algorithm requires a careful implementation of some essential components and an appropriate tuning of the parameters used [30, 31].

In this section we present an improved implementation of a Simulated Annealing algorithm, that we called ISA, for constructing binary CAs of different strengths. Contrary to existing SA implementations for the CAC problem, our algorithm has the merit of improving two key features that have a great impact on its performance: an efficient method to generate initial solutions containing a balanced number of symbols in each column and a composed neighborhood function. Next all the implementation details of the proposed ISA algorithm are presented.

3.1. Internal representation and search space

Let A be a potential solution in the search space $\mathscr{A}$, that is a covering array CA$(N; t, k, v)$ of size N, strength t, degree k, and order v. Then A is represented as an $N \times k$ array on v symbols, in which the element $a_{i,j}$ denotes the symbol assigned in the test configuration i to the parameter j. The size of the search space $\mathscr{A}$ is then given by the following expression:

$$|\mathscr{A}| = v^{Nk} \tag{7}$$

3.2. Evaluation function

The evaluation function is one of the key elements for the successful implementation of metaheuristic algorithms because it is in charge of guiding the search process toward good solutions in a combinatorial search space.

Previously reported metaheuristic methods for solving the CAC problem have commonly evaluated the quality of a potential solution (covering array), $\mathscr{F}(A)$, as the number of uncovered t-tuples [15, 55, 42]. In our ISA implementation this evaluation function was also used. Its computational complexity is equivalent to $O(N\binom{k}{t})$. However, an incremental cost evaluation of neighboring solutions, in $O(2\binom{k-1}{t-1})$ operations, is possible by using appropriate data structures.

3.3. Initial solution

The initial solution is the starting covering array used for the algorithm to begin the search of better configurations in the search space $\mathscr{A}$. In the existing SA implementations for the CAC problem [43, 15] the initial solution is randomly generated. In contrast, ISA creates the starting solution using a procedure, inspired by the work of Roux [51], that guarantees a balanced number of symbols in each column of the generated covering array CA$(N; t, k, v)$. This procedure assigns randomly $\lfloor N/2 \rfloor$ ones and the same number of zeros to each column of the covering array when its size N is even, otherwise it allocates $\lfloor N/2 \rfloor + 1$ ones and $\lfloor N/2 \rfloor$ zeros to each column.

This particular method for constructing the initial covering array was selected mainly because we have observed, from preliminary experiments (see Section 5.2), that the solutions containing a balanced number of symbols in each column lead our algorithm to reach better final solutions.

[3] In the rest of this document, we will refer to the general-purpose simulated annealing algorithm as SA.

3.4. Neighborhood function

Given that ISA is based on Local Search (LS) then a neighborhood function must be defined. The main objective of the neighborhood function is to identify the set of potential solutions which can be reached from the current solution in an LS algorithm. Formally, a neighborhood relation is a function $\mathcal{N} : \mathscr{A} \rightarrow \mathscr{A}$ that assigns to every potential solution (a covering array) $A \in \mathscr{A}$ a set of neighboring solutions $\mathcal{N}(A) \subseteq \mathscr{A}$, which is called the neighborhood of A.

The results of our preliminary experimentations (see Section 5.3) lead us to find a suitable neighborhood structure for the CAC problem. It is composed by two complementary functions which allow ISA to achieve better performance. Our compound neighborhood function is inspired by the ideas reported in [24, 49, 38], where the advantage of using this approach is well documented.

Before introducing the proposed combined neighborhood structure, we present two preliminary concepts used in its definition. Let us define a function $switch(A, i, j)$ which changes the value $a_{i,j}$ in the current solution A by a different legal member of the alphabet, and $swap(A, i, j, l)$ another function allowing us to exchange the values $a_{i,j}$ and $a_{l,j}$ ($a_{i,j} \neq a_{l,j}$) within the same column of A.

The first function $\mathcal{N}_1(A, \omega)$, in our compound neighborhood, produces a subset $W \subseteq \mathscr{A}$ containing ω different neighboring solutions of A and selects the one having the minimum number of uncovered t-tuples among them. The subset W is created by making ω successive calls to the function $switch(A, i, j)$ using different random values of i and j ($0 \leq i < N$, $0 \leq j < k$). Formally, $\mathcal{N}_1(A, \omega)$ can be defined as:

$$\mathcal{N}_1(A, \omega) = \left\{ A' \in \mathscr{A} : A' = \underset{A'' \in W}{\operatorname{argmin}}[\mathscr{F}(A'')] \right\} \tag{8}$$

The second function $\mathcal{N}_2(A, \gamma)$ constructs a subset $R \subseteq \mathscr{A}$ composed of γ different neighboring solutions of A and returns the best one, i.e., that with the minimum number of uncovered t-tuples. Each of the neighboring solutions contained in the subset R are produced by the application of the $swap(A, i, j, l)$ function with different randomly chosen values for i, j and l ($0 \leq i, l < N$, $0 \leq j < k$). Thus, we can formally define $\mathcal{N}_2(A, \gamma)$ as follows:

$$\mathcal{N}_2(A, \gamma) = \left\{ A' \in \mathscr{A} : A' = \underset{A'' \in R}{\operatorname{argmin}}[\mathscr{F}(A'')] \right\} \tag{9}$$

During the search process a combination of both $\mathcal{N}_1(A, \omega)$ and $\mathcal{N}_2(A, \gamma)$ neighborhood functions is employed by our ISA algorithm. The former is applied with probability p, while the latter is employed at a $(1 - p)$ rate. This combined neighborhood function $\mathcal{N}_3(A, x, \omega, \gamma)$ is defined in (10), where x is a random number in the interval $[0, 1]$.

$$\mathcal{N}_3(A, x, \omega, \gamma) = \begin{cases} \mathcal{N}_1(A, \omega) & \text{if } x \leq p \\ \mathcal{N}_2(A, \gamma) & \text{if } x > p \end{cases} \tag{10}$$

3.5. Cooling schedule

A cooling schedule is defined by the following parameters: an initial temperature, a final temperature or a stopping criterion, the maximum number of neighboring solutions that can be generated at each temperature (Markov chain length), and a rule for decrementing the temperature. The cooling schedule governs the convergence of the SA algorithm. At the beginning of the search, when the temperature is large, the probability of accepting solutions of worse quality than the current solution (uphill moves) is high. It allows the algorithm to escape from local minima. The probability of accepting such moves is gradually decreased as the temperature goes to zero.

Cooling schedules which rapidly decrement the temperature can lead the search process to get trapped in an early local minima. On the contrary, a very slow cooling of the temperature guides the algorithm towards non-promising searching regions, resulting often in a waste of computational time. A good selection of the cooling schedule is thus critical to the SA algorithm's performance.

The literature offers a number of different cooling schedules, see for instance [1, 26, 56, 2, 57, 48]. They can be divided into two main categories: static and dynamic. In a static cooling schedule, the parameters are fixed and cannot be changed during the execution of the algorithm. With a dynamic cooling schedule the parameters are adaptively changed during the execution.

In our ISA implementation we preferred a geometrical cooling scheme (static) mainly for its simplicity. It starts at an initial temperature T_i which is decremented at each round by a factor α using the relation $T_j = \alpha T_{j-1}$. For each

temperature, the maximum number of visited neighboring solutions is L. It depends directly on the parameters (N, k and v) of the studied covering array. This is because more moves are required for bigger CAs.

We will see later that thanks to the two main features presented previously, our ISA algorithm using this simple cooling scheme gives remarkable results.

3.6. Termination conditions

The ISA algorithm stops either if the current temperature attains T_f, when it ceases to make progress, or when a valid covering array is found. In our implementation a lack of progress exists if after ϕ (*frozen factor*) consecutive temperature decrements the best-so-far solution is not improved.

4. Computational experiments

The procedure described in the previous section was coded in C and compiled with *gcc* using the optimization flag *-O*3. It was run sequentially into a CPU Core 2 Quad at 2.4 GHz, 4 GB of RAM with Linux operating system. In all the experiments the following parameters were used for our ISA implementation:

a) Initial temperature $T_i = 4.0$
b) Final temperature T_f =1.0E-10.
c) Cooling factor $\alpha = 0.99$
d) Maximum neighboring solutions per temperature $L = (Nkv)^2$
e) Frozen factor $\phi = 11$
f) The neighborhood function $\mathcal{N}_3(A, x, \omega, \gamma)$ is applied using a probability $p = 0.6$ and parameters $\omega = 10$ and $\gamma = N/2$.

These parameter values were chosen experimentally based on a methodology reported in [45, 25], which employs CAs and Diophantine equations to determine the minimum number of experiments needed to find the combination yielding the best performance. For the reason of space limitation we did not present here all these experiments, however in Section 5 the influences of certain of those parameter values on the global performance of the ISA algorithm are analyzed.

In order to assess the performance of the ISA algorithm introduced in Section 3, a test suite composed of 127 well-known benchmark instances taken from the literature was used [54, 10, 42, 23]. It includes instances of degrees $4 \leq k \leq 257$ and strengths $3 \leq t \leq 6$.

The main criterion used for the comparison is the same as the one commonly used in the literature: the solution quality, i.e., the *best size N* found (smaller values are better) given fixed values for k, t, and v.

4.1. Comparing ISA with an existing SA implementation

For this experiment we have contacted Myra B. Cohen, who is one of the authors of an existing SA implementation for the CAC problem (SAC) [16], and asked her to construct some representative binary CAs using her algorithm. She ran once (sequentially) SAC on the following instances using a CPU Dual Opteron 250 at 2.4 GHz, 4 GB of RAM with Linux operating system [14]: CA(N; 3, 23, 2), CA(N; 3, 38, 2) and CA(N; 3, 51, 2). These are binary CAs of strength three, however they are the only benchmark instances available for us at this moment that can be used to make a fair comparison between ISA and SAC [16].

Table 1 summarizes Cohen's results as well as those obtained by one execution of our ISA algorithm over the same benchmark instances. Column 1 lists the degree k of the instance. Columns 2 and 3 show the best size N found by SAC and the CPU time in seconds required by this algorithm for reaching that solution. Column 4 indicates the time in seconds expended by ISA to attain the same solution quality as SAC, when it is ran on the computer described above in this section. The best solution quality N^* found by ISA along with its expended computational time T (in seconds) are listed in columns 6 and 7, respectively. Column 9 presents the difference between the best results produced by ISA and SAC, $\Delta_C = N^* - N$. Finally, the difference of the CPU times expended by the compared algorithms, $\Delta_T = \overline{T_{\mathrm{ISA}}} - T_{\mathrm{SAC}}$, is provided for indicative purposes in the last column.

The running times from the SAC and ISA algorithms presented in Table 1 cannot be directly compared, since the computational platforms used to run them are different. Nevertheless, we have scaled, by a factor of 2.06, our

Table 1: Comparison between ISA and SAC (an existing SA implementation [16]) over a set of selected instances of strength three. For each instance of degree k the best size N found by SAC, its CPU time in seconds (T_{SAC}) as well as the time expended by ISA (T_{ISA}) to attain the same solution as SAC are presented. N^* indicates the best solution quality found by ISA and T its computational time (in seconds). The difference between the best results produced by ISA and SAC, $\Delta_C = N^* - N$, and their expended CPU times $\Delta_T = \overline{T_{\text{ISA}}} - T_{\text{SAC}}$ are also provided.

	SAC		ISA		ISA				
k	N	T_{SAC}	T_{ISA}	$\overline{T_{\text{ISA}}}$	N^*	T	$\overline{T}$	Δ_C	Δ_T
23	22	810.84	2.28	4.70	20	38.92	80.15	-2	-806.14
38	28	9844.35	0.90	1.85	24	129.20	266.07	-4	-9842.50
51	31	18366.10	10.98	22.61	28	241.49	497.31	-3	-18343.49
Avg.	27.00	9673.76	4.72	9.72	24.00	136.54	281.18	-3.00	-9664.04

execution times according to the Standard Performance Evaluation Corporation (`http://www.spec.org`) in order to present them in a normalized form (see columns 5 and 8).

From Table 1 we can observe that SAC is the most time-consuming algorithm, since it uses an average of 9673.76 seconds for solving these three instances. On the contrary, ISA employs only 9.72 seconds on average to equal the results furnished by SAC. We can also remark that ISA can take advantage of longer executions. Indeed it is able to consistently improve the best size N found by SAC, obtaining an average amelioration of 11.11% (compare columns 2 and 6). Furthermore, ISA achieves those results by employing only a small fraction of the total time used by SAC (see columns 3 and 8).

Thus, as this experiment confirms, our ISA implementation is more effective for searching binary CAs of strength three than the existing SA algorithm reported by Cohen et al. [16]. Below, we will present more computational results obtained from a performance comparison carried out between ISA and a well-known greedy algorithm over larger strength benchmark instances.

4.2. Comparing ISA with a well-known greedy algorithm

For the second of our experiments we have obtained the IPOG-F algorithm [23], which is part of the FireEye software package (now called ACTS) and is publicly available from `http://csrc.nist.gov/groups/SNS/acts`. The objective of this experiment is to make a fair comparison between IPOG-F and our ISA algorithm. Since IPOG-F is a greedy algorithm, this comparison is mainly focused on the expended computational time needed by ISA, to attain the same solution quality produced by IPOG-F.

The experimental comparison between ISA and IPOG-F was accomplished running once each compared method over 60 benchmark instances of strengths $3 \leq t \leq 6$ and degrees $4 \leq k \leq 30$ (see Section 4). For the experiment a CPU Core 2 Quad at 2.4 GHz, 4 GB of RAM with Linux operating system was used. IPOG-F was executed with the parameter values suggested by its authors in [23].

The results from this experiment are summarized in Table 2, which presents in the first two columns the strength t and the degree k of the selected benchmark instances. The best size $N_{\text{IPOG-F}}$ found by the IPOG-F algorithm as well as the CPU time $T_{\text{IPOG-F}}$ in seconds expended for reaching these results are listed in columns 3 and 4. Column 5 shows the execution time needed by our ISA algorithm to match the same solution quality produced by IPOG-F. Finally, the difference between the running times of ISA and IPOG-F is displayed in column 6 (i.e., $\Delta = T_{\text{ISA}} - T_{\text{IPOG-F}}$).

From the data presented in Table 2 one can see that ISA is able to easily reach the same solution quality, in terms of the size N, as the IPOG-F method. Indeed, ISA is faster than IPOG-F for solving all the benchmark instances of strengths 3 through 5, since it consumes only 10.00% of the computational time employed by IPOG-F. In the case of the strength 6 benchmark instances, ISA expends more CPU time than IPOG-F for constructing the covering array $\text{CA}(375; 6, 21, 2)$, however for the rest of those instances ISA expends 39.23% less computational time than IPOG-F. Thus, we can conclude from this experiment that, for the selected benchmark instances, ISA is on average faster than IPOG-F. Next, we present a series of experiments devoted to asses the performance of ISA with respect to the best-known methods published in the literature.

4.3. Comparing ISA with the state-of-the-art procedures

The purpose of this experiment is to carry out a performance comparison of the bounds achieved by our ISA algorithm with respect to the best-known solutions reported in the literature [18], which were produced using the

Table 2: Comparison between ISA and the greedy algorithm IPOG-F [23]. For each instance of strength t and degree k, the best size ($N_{\text{IPOG-F}}$) found by the IPOG-F, its CPU time ($T_{\text{IPOG-F}}$) in seconds as well as the time expended by ISA to match these solutions are listed. The difference between both running times is displayed in last column ($\Delta = T_{\text{ISA}} - T_{\text{IPOG-F}}$).

t	k	$N_{\text{IPOG-F}}$	$T_{\text{IPOG-F}}$	T_{ISA}	Δ
	4	8	0.91	0.01	-0.90
	5	12	0.96	0.01	-0.95
	6	15	0.98	0.01	-0.97
	7	16	1.06	0.01	-1.05
	9	17	1.09	0.01	-1.08
	11	18	1.04	0.01	-1.03
	12	19	1.22	0.01	-1.21
3	13	20	1.13	0.01	-1.12
	15	21	1.20	0.01	-1.19
	16	22	1.05	0.01	-1.04
	19	24	1.25	0.01	-1.24
	21	25	1.64	0.01	-1.63
	24	26	1.14	0.03	-1.11
	26	27	1.35	0.03	-1.32
	30	28	1.46	0.08	-1.38
Avg.		19.87	1.17	0.02	

t	k	$N_{\text{IPOG-F}}$	$T_{\text{IPOG-F}}$	T_{ISA}	Δ
	5	22	1.05	0.01	-1.04
	6	26	1.23	0.01	-1.22
	7	32	2.24	0.01	-2.23
	8	34	2.60	0.01	-2.59
	9	37	2.67	0.03	-2.64
	10	41	2.76	0.02	-2.74
	11	43	2.85	0.06	-2.79
4	12	47	2.91	0.07	-2.84
	13	49	2.95	0.14	-2.81
	14	52	3.58	0.22	-3.36
	15	53	3.91	0.43	-3.48
	16	56	4.94	0.51	-4.43
	17	57	6.51	0.76	-5.75
	18	60	8.98	1.19	-7.79
	19	62	11.19	0.91	-10.28
Avg.		44.73	4.03	0.29	

t	k	$N_{\text{IPOG-F}}$	$T_{\text{IPOG-F}}$	T_{ISA}	Δ
	6	42	1.21	0.01	-1.20
	7	61	1.80	0.01	-1.79
	8	73	1.72	0.01	-1.71
	9	85	1.96	0.01	-1.95
	10	87	1.62	0.01	-1.61
	11	95	1.67	0.01	-1.66
	12	105	1.87	0.03	-1.84
5	13	111	2.13	0.06	-2.07
	14	119	2.25	0.10	-2.15
	15	127	2.90	0.25	-2.65
	16	134	3.44	0.31	-3.13
	17	140	3.93	0.40	-3.53
	18	144	4.94	1.62	-3.32
	19	148	6.50	2.62	-3.88
	20	155	8.61	2.35	-6.26
Avg.		108.40	3.10	0.52	

t	k	$N_{\text{IPOG-F}}$	$T_{\text{IPOG-F}}$	T_{ISA}	Δ
	7	64	1.05	0.01	-1.04
	8	124	1.05	0.01	-1.04
	9	154	1.56	0.01	-1.55
	10	165	1.78	0.06	-1.72
	11	192	2.28	0.08	-2.20
	12	215	2.34	0.27	-2.07
	13	237	3.08	0.63	-2.45
6	14	256	4.31	1.33	-2.98
	15	276	6.39	1.88	-4.51
	16	292	10.16	4.44	-5.72
	17	309	15.43	9.82	-5.61
	18	327	26.23	16.30	-9.93
	19	343	38.45	29.67	-8.78
	20	363	60.72	41.73	-18.99
	21	375	92.39	99.32	6.93
Avg.		246.13	17.81	13.70	

following state-of-the-art procedures: orthogonal array constructions [8], Roux type constructions [54], doubling constructions [10], algebraic constructions [27], Deterministic Density Algorithm (DDA) [6], Tabu Search (TS) [58], and IPOG-F [23].

The SA implementation reported by Cohen et al. (SAC) [16] for solving the CAC problem was intentionally omitted from this comparison because as their authors recognize this algorithm fails to produce competitive results when the strength of the arrays is $t \geq 3$.

For this experiment we have fixed the maximum computational time expended by ISA for constructing a CA to 8 hours. Due to the incomplete and non-deterministic nature of our ISA algorithm, 20 independent runs were executed over a subset of 62 benchmark instances of strengths $3 \leq t \leq 6$ and degrees $4 \leq k \leq 56$ (see Section 4). The detailed computational results produced by this experiment are listed in Table 3. The first two columns in this table indicate the strength t and degree k of the selected instances. Next 2 columns show, in terms of the size N of the CAs, the best solution found by TS [58] and IPOG-F [23]. Column 5 presents the smallest (Best) CAs published in the literature [18], while column 6 reports the best solutions achieved by ISA. Column 7 lists the computational time T, in seconds, consumed by ISA. The difference between the best result produced by ISA and the previous best-known solution (Δ = ISA − Best) is depicted in the last column. In the case of the strength 6 benchmark instances, the column corresponding to the TS results was omitted since they were not furnished by the authors [58].

Results in bold in column 6 indicate that the produced CAs contain two constant rows. A covering array $A = N \times k$ contains a constant row a_i if $a_{i,j} = a_{i,0}, \forall i, j \,|\, i, j \in \{0, \ldots, k-1\}, a_{i,0} \in \{0, \ldots, v-1\}$. This characteristic of the CAs

Table 3: Improved bounds on CAN$(t, k, 2)$ for strengths $3 \leq t \leq 6$ and degrees $4 \leq k \leq 56$ produced by ISA using a computational time limited to 8 hours. For each instance of strength t and degree k, the best solution, in terms of the size N, found by TS [58], IPOG-F [23] and ISA are listed. T represents the computational time consumed by ISA, in seconds, and Best the best-known solution reported in the literature [18]. Last column depicts the difference between the best result produced by ISA and the previous best-known solution (Δ = ISA − Best). Results in bold indicate that the produced CAs contain two constant rows.

		N					
t	k	TS	IPOG-F	Best	ISA	T	Δ
3	4	8	9	8	**8**	0.01	0
	5	10	11	10	**10**	0.01	0
	8	12	17	12	**12**	0.01	0
	11	12	18	12	12	0.01	0
	12	15	19	15	**15**	0.03	0
	14	16	21	16	16	52.45	0
	16	17	22	17	17	21.72	0
	20	18	25	18	18	59.10	0
	22	19	26	19	19	31.70	0
	23	22	26	22	20	38.92	-2
	25	23	27	23	21	19.77	-2
	26	23	27	23	22	67.48	-1
	28	23	28	23	23	55.30	0
	38	25	33	26	24	129.20	-1
	44	27	33	27	25	143.98	-2
	46	30	34	30	26	4644.39	-4
	49	31	36	31	27	3204.58	-4
	52	31	37	31	28	241.49	-3
	54	31	37	31	29	378.71	-2
	56	31	37	31	30	122.31	-1
Avg.		21.20	26.15	21.25	20.10	460.56	

		N					
t	k	TS	IPOG-F	Best	ISA	T	Δ
4	5	16	22	16	16	0.01	0
	6	21	26	21	21	0.01	0
	12	48	47	24	**24**	0.01	0
	13	53	49	34	32	725.67	-2
	17	54	57	39	35	1876.45	-4
	18	55	60	39	36	1467.89	-3
	20	55	65	39	39	1476.17	0
	21	80	68	42	42	1534.89	0
	22	80	69	44	44	1675.45	0
	24	80	71	46	46	1765.79	0
	25	80	74	50	50	1894.45	0
	26	91	74	52	51	2076.56	-1
	28	91	77	54	53	2300.67	-1
	30	92	80	58	56	2543.78	-2
	32	94	83	59	58	3745.78	-1
	33	96	83	64	61	3356.23	-3
	35	96	85	66	64	3723.56	-2
	37	96	88	67	65	8610.35	-2
Avg.		71.00	65.44	45.22	44.06	2154.10	

		N					
t	k	TS	IPOG-F	Best	ISA	T	Δ
5	6	32	42	32	**32**	0.01	0
	7	56	57	42	**42**	0.01	0
	8	56	68	56	**52**	0.88	-4
	9	62	77	62	**54**	20.14	-8
	10	62	87	62	**56**	649.59	-6
	14	110	119	103	64	1534.46	-39
	15	152	127	110	79	1890.78	-31
	16	152	134	117	**99**	2350.78	-18
	17	176	140	123	**104**	5823.65	-19
	18	176	144	127	107	13807.90	-20
	19	176	148	135	**116**	18675.67	-19
	20	194	155	136	**119**	20679.19	-17
	21	261	160	136	**122**	22876.39	-14
	22	261	163	136	**124**	24935.87	-12
Avg.		137.57	115.79	98.36	83.57	8088.95	

		N				
t	k	IPOG-F	Best	ISA	T	Δ
6	7	79	64	64	0.01	0
	8	118	85	85	0.03	0
	9	142	111	**108**	179.32	-3
	10	165	116	**116**	243.87	0
	11	192	128	118	329.60	-10
	15	276	160	128	3356.05	-32
	16	292	230	**179**	432.79	-51
	17	309	276	235	395.80	-41
	18	327	291	280	218.00	-11
	19	343	308	299	763.01	-9
Avg.		224.30	176.90	161.20	591.85	

is particularly important when they are used as ingredients to some recursive procedures [10, 28, 41], since constant rows allow us to reduce the final size N of the arrays constructed with these methods which can lead to new bounds.

The analysis of the data presented in Table 3 leads us to the following main observations. First, the solution quality attained by ISA is very competitive with respect to that produced by the state-of-the-art procedures summarized in column 5 (Best). In fact, it is able to improve on 39 previous best-known solutions and to equal these results for the other 23 benchmark instances. Please notice that for several instances, like covering array CA$(N; 6, 16, 2)$, a significant reduction of the size N is accomplished by our algorithm when compared with the previous best-known solution (Δ up to -51).

Second, one observes that the procedures IPOG-F [23] and TS [58] consistently return poorer quality solutions than ISA. Indeed, ISA produces CAs which are on average 28.44% and 35.71% smaller than those constructed by IPOG-F and TS, respectively.

Regarding the computational effort we would like to point that, in general, authors of the algorithms used in our comparisons did not provide detailed information about their expended CPU times. Thus, the running times from

these algorithms cannot be directly compared with ours.

Even if the results attained by our ISA algorithm are very competitive, we have observed that the average computing time consumed by our approach, to produce these results, is greater than that used by some recursive [28, 41] and algebraic methods [29, 10, 27]. However, since ISA outperforms some of the state-of-the-art procedures, finding 39 new bounds, we think that the extra consumed computing time is fully justified. Specially, if we consider that in this kind of experiments the main objective is to compare the best bounds achieved by the studied algorithms.

4.4. Comparing ISA on higher degree instances

This experiment aims at comparing the performance of our ISA algorithm with respect to state-of-the-art procedures over a set of 65 higher degree instances ($20 \leq k \leq 1712$) of strengths $3 \leq t \leq 6$. In order to better understand the limits of our ISA implementation, the maximum computational time allowed for each one of the 20 independent executions of this experiment was fixed to 72 hours.

Table 4 summarizes the computational results from this experimental comparison. Columns 1 and 2 depict the strength t and degree k of the benchmark instances. Next 2 columns list the best solution found by TS [58] and IPOG-F [23], in terms of the size N of the CAs, while column 5 presents the best-known solution (Best) reported in the literature [18]. In column 6 the best results reached by ISA, within the maximum computational time allowed, are indicated. Results in bold in this column denotes that the resulting covering array contains two constant rows. Column 7 displays the difference between the best result produced by ISA and the previous best-known solution (Δ = ISA − Best). Once again, we have omitted the TS column in the case of the strength 6 instances because we do not have the results furnished by TS [58] over those CAs. The IPOG-F method does not report a result for CAN$(5, 192, 2)$ [23].

From Table 4, it is observed that ISA outperforms all the other compared algorithms in terms of solution quality. ISA is able to improve the previous best-known solutions for the 65 tested instances within the maximum CPU time fixed. When comparing the algorithms TS, IPOG-F and ISA with each other, over the instances of strengths 3 through 5, one finds that the average size N of the CAs produced by them is 331.59, 186.33 and 150.84. Therefore, for the selected instances the CAs produced by ISA are 54.51% and 19.04% smaller than those produced by TS and IPOG-F, respectively. In the case of the strength 6 benchmark instances, the average size N of the CAs produced by ISA is 7.06% smaller than that of IPOG-F (481.00 vs. 447.05).

These results demonstrate that our ISA implementation is highly effective and capable of surpassing the best procedures reported in the literature in terms of solution quality. It is particularly important to note that ISA was able to construct the covering array CA$(66; 3, 1712, 2)$. To the best of our knowledge this is the highest degree covering array of strength three ever found by computational search.

Next section is dedicated to analyze the implications of the results achieved by our ISA algorithm when they are used as ingredients to some recursive constructions [10, 28, 41].

4.5. Some implications of the results

As we have seen, from the previous experimental comparisons, ISA is able to find new bounds on CAN$(t, k, 2)$ for strengths $3 \leq t \leq 6$ and degrees $4 \leq k \leq 1712$. These results, by themselves, represent an important achievement in the study of this kind of combinatorial structures. However, they also have the additional value of being excellent ingredients to recursive procedures which can be used to attain other new bounds.

For instance, by using the CAs CA$(12; 3, 11, 2)$ and CA$(15; 3, 12, 2)$ (built with ISA) as ingredients to the recursive method reported in [21] we can construct CA$(61; 3, 1452, 2)$, which improves the previous best-known solution CA$(69; 3, 1452, 2)$ [18]. Using this combination of ISA and recursive constructions permits to find 20 new bounds for binary CAs of strength three and degrees $88 \leq k \leq 10648$. The interested reader is referred to [21] for a detailed description of those new bounds.

5. Analyzing the performance of ISA

The main objective of this section is to experimentally analyze the global performance of the ISA algorithm and the influences that some of its key features have on it. Next, we present the results of the experiments carried out for this purpose.

Table 4: Improved bounds on CAN($t, k, 2$) for strengths $3 \leq t \leq 6$ and degrees $20 \leq k \leq 1712$ produced by ISA using a computational time limited to 72 hours. For each instance of strength t and degree k, the best solution, in terms of the size N, found by TS [58], IPOG-F [23] and ISA are listed. Best represents the best-known solution reported in the literature [18], while last column depicts the difference between the best bound produced by ISA and the previous best-known solution (Δ = ISA − Best). Results in bold indicate that the produced CAs contain two constant rows.

		N				
t	k	TS	IPOG-F	Best	ISA	Δ
3	68	33	39	32	31	-1
	123	41	46	42	35	-6
	127	41	46	42	36	-5
	140	42	48	43	37	-5
	141	44	48	43	39	-4
	144	44	48	43	40	-3
	156	44	48	44	41	-3
	177	49	50	44	42	-2
	242	51	55	44	43	-1
	243	51	55	46	44	-2
	246	51	55	46	45	-1
	256	52	55	49	46	-3
	257	53	55	52	47	-5
	1712	86	77	69	66	-3
Avg.		48.71	51.79	45.64	42.29	

		N				
t	k	TS	IPOG-F	Best	ISA	Δ
4	38	96	89	67	66	-1
	89	181	120	89	87	-2
	97	182	123	97	94	-3
	101	182	125	101	98	-3
	107	182	127	107	102	-5
	109	182	127	109	105	-4
	113	183	130	113	107	-6
	145	189	140	138	123	-15
	151	189	142	142	129	-13
	156	189	145	145	130	-15
	169	249	147	147	135	-12
	184	252	151	151	150	-1
	196	257	155	155	151	-4
	208	257	157	157	154	-3
	213	257	158	158	156	-2
	217	257	159	159	157	-2
	222	257	160	160	159	-1
Avg.		208.29	138.53	129.12	123.71	

		N				
t	k	TS	IPOG-F	Best	ISA	Δ
5	24	261	175	136	**132**	-4
	128	644	366	254	252	-2
	132	644	370	262	260	-2
	138	644	374	274	272	-2
	140	644	376	278	274	-4
	150	734	386	298	289	-9
	152	734	387	302	292	-10
	158	734	390	314	305	-9
	164	770	394	326	310	-16
	168	770	398	334	318	-16
	174	1002	402	346	331	-15
	180	1005	406	358	338	-20
	192	1005	—	359	352	-7
Avg.		737.77	368.67	295.46	286.54	

		N			
t	k	IPOG-F	Best	ISA	Δ
6	20	363	327	314	-13
	21	375	341	330	-11
	22	382	355	344	-11
	23	397	371	357	-14
	24	411	384	372	-12
	25	426	397	385	-12
	26	438	409	399	-10
	27	449	422	410	-12
	28	463	432	421	-11
	29	474	444	435	-9
	30	481	454	444	-10
	31	489	464	457	-7
	32	500	473	468	-5
	33	511	496	480	-16
	34	522	502	488	-14
	35	528	510	496	-14
	36	535	525	505	-20
	37	544	534	514	-20
	38	552	545	530	-15
	40	566	560	545	-15
	60	695	695	694	-1
Avg.		481.00	459.05	447.05	

5.1. ISA overall performance

In general, authors of the algorithms used in our experimental comparisons (Sections 4.3 and 4.4) only provide the best solution quality, in terms of the size N, achieved by them. Thus, these algorithms cannot be statistically compared with ISA. Nevertheless, we can illustrate and analyze the average behavior of ISA using the execution profiles produced during the 20 independent executions of those comparative experiments. Figure 1 shows three execution profiles which represent the evolution of the worst, average and best values of the evaluation function $\mathscr{F}(A)$ attained by ISA during the search process for constructing the CAs CA(16; 3, 14, 2) and CA(52; 5, 8, 2). Each iteration in the graphs represents a call to the evaluation function described in Section 3.2.

From Figure 1 we can observe a relatively small gap between the worst and the best solution found during these executions. It is an indicator of the algorithm's precision and robustness since it shows that on average the performance of ISA does not present important fluctuations. For these particular instances the average standard deviation is 4.75

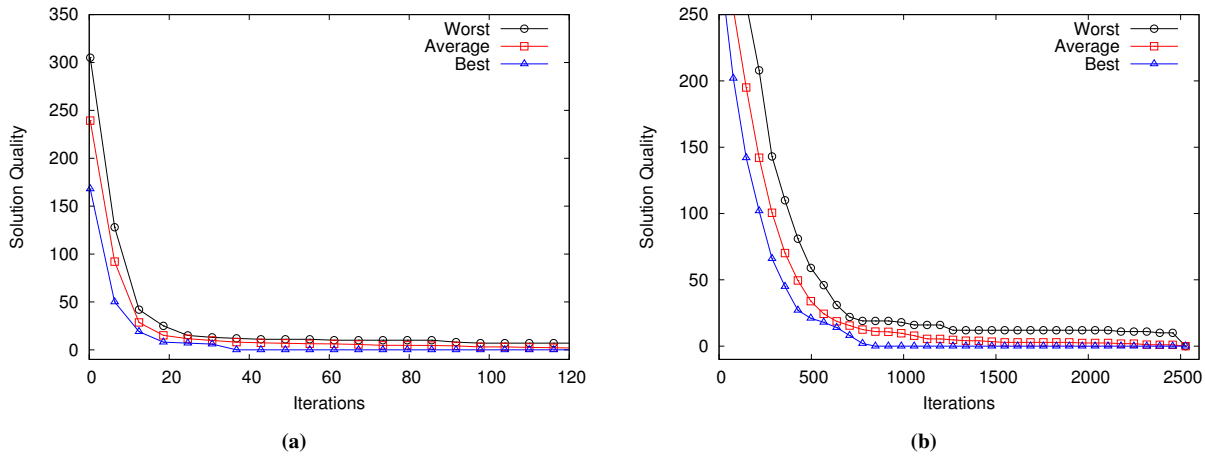

Figure 1: Execution profiles of ISA over the instances: (a) CA(16; 3, 14, 2) and (b) CA(52; 5, 8, 2).

and 9.91, respectively. This figure allows us to summarize the overall behavior of ISA since similar results were obtained with all the other tested instances.

5.2. Influence of the initial solution

In this experiment we compare the performance of two different procedures for constructing the initial solution of ISA. The first one is commonly used in the literature [43, 15], and creates the initial solution by assigning randomly a symbol in v at each element $a_{i,j}$ of the array. The second one is the procedure described in Section 3.3 which guarantees a balanced number of symbols in each column of the generated covering array CA($N; t, k, v$) (Balanced).

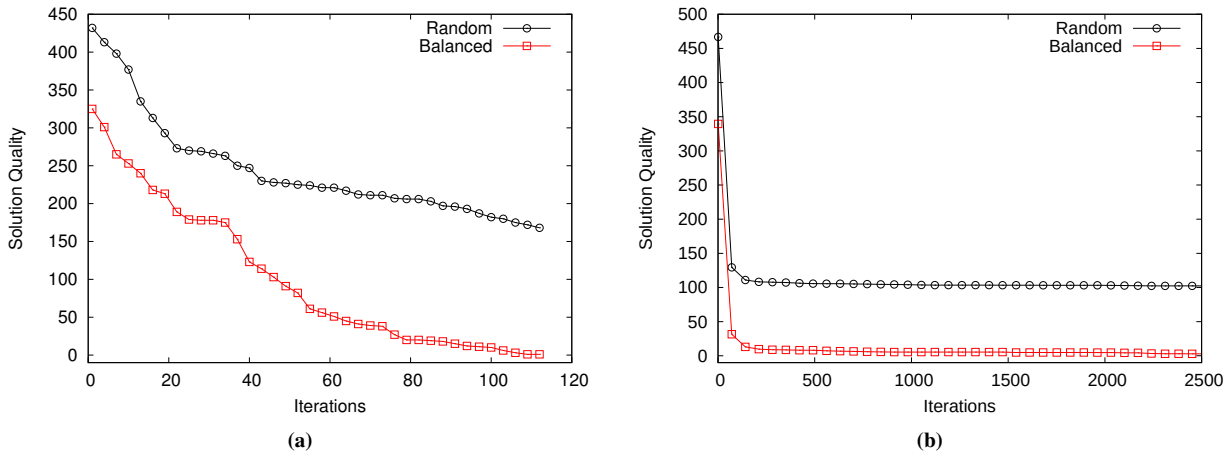

Figure 2: Performance comparison of two different initialization procedures for ISA over the instances: (a) CA(16; 3, 14, 2) and (b) CA(56; 5, 10, 2).

Both initialization procedures (called here Random and Balanced, respectively) were integrated into the ISA source code and executed 20 times over the benchmark instances described in Section 4. The results achieved by ISA over the instances CA(16; 3, 14, 2) and CA(56; 5, 10, 2) are illustrated in Figure 2. The plot represents the iterations of ISA (abscissa) against the average solution quality attained from the starting arrays generated with the compared initialization procedures (ordinate). Figure 2 discloses that ISA using balanced initial solutions performs much better than the ISA algorithm that starts from a randomly generated solution. Very similar results were obtained with the rest of the analyzed benchmark instances, thus this figure correctly summarizes the behavior of the compared initialization procedures.

5.3. Influence of the neighborhood functions

The neighborhood function is a critical element for the performance of any local search algorithm. In order to further examine the influence of this element on the global performance of our ISA implementation we have performed some experimental comparisons using the following neighborhood functions (described in Section 3.4):

- $switch(A, i, j)$
- $\mathcal{N}_1(A, \omega)$
- $\mathcal{N}_2(A, \gamma)$
- $\mathcal{N}_3(A, x, \omega, \gamma)$

For this experiment each one of the studied neighborhood functions was implemented within ISA, compiled and executed independently 20 times over the selected benchmark instances using the set of parameter values listed in Section 4. The results of this experiment are summarized in Figure 3. It shows the differences in terms of average solution quality attained by ISA, when each one of the studied neighborhood relations is used to solve the instances CA(16; 3, 14, 2) and CA(56; 5, 10, 2) (comparable results were obtained with all the other tested instances). From this graph it can be observed that the worst performance is attained by ISA when the neighborhood function called $switch(A, i, j)$ is used. The functions $\mathcal{N}_1(A, \omega)$ and $\mathcal{N}_2(A, \gamma)$ produce better results compared with $switch(A, i, j)$ since they improve the solution quality faster. However, they also cause that our ISA algorithm gets stuck on some local minima. Finally, the best performance is attained by ISA when it is employed the neighborhood function $\mathcal{N}_3(A, x, \omega, \gamma)$, which is a compound neighborhood combining the complementary characteristics of both $\mathcal{N}_1(A, \omega)$ and $\mathcal{N}_2(A, \gamma)$.

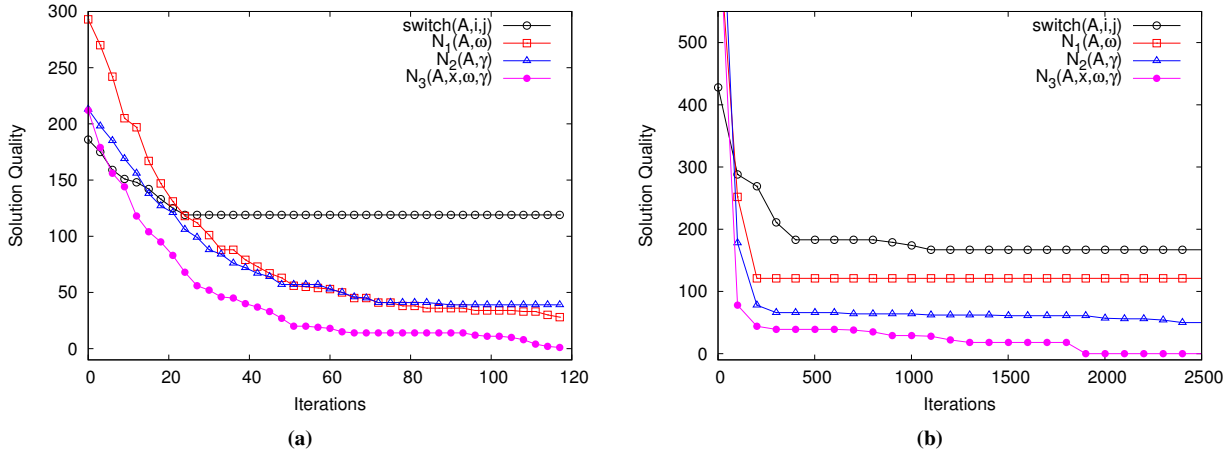

Figure 3: Performance comparison of four neighborhood functions using ISA over the instances: (a) CA(16; 3, 14, 2) and (b) CA(56; 5, 10, 2).

6. Conclusions

Software systems are becoming ubiquitous in modern society where numerous human activities rely on them to fulfill their needs for information processing, storage, search, and retrieval. These software systems are usually highly complex and contain millions of lines of source code. Furthermore, they usually have many possible configurations which are produced by the combination of their multiple input parameters. In order to ensure the quality of these software products any such configuration should be tested [9]. However, as the number of input values grows and the number of discrete values for each datum item increases, exhaustive functional testing becomes impractical or impossible due to time and cost limitations [22]. Hence, different criterion for selecting test cases were proposed in

the literature to deal with this issue [44, 4, 37, 3, 7]. Software interaction testing distinguishes among them, because it employs combinatorial structures called covering arrays (CAs) for representing economical sized test suites that provide coverage of all the t-way combinations of component values [13].

In this paper, we have introduced an improved implementation of a Simulated Annealing algorithm (ISA) for constructing binary CAs of different strengths. This algorithm integrates two key features that importantly determine its performance. First, an efficient method to generate initial solutions containing a balanced number of symbols in each column. Second, a carefully designed composed neighborhood function which allows the search to quickly reduce the total cost of candidate solutions, while avoiding to get stuck on some local minima.

To assess the practical effectiveness of our ISA algorithm, we have carried out extensive experimentation using a set of 127 benchmark instances taken from the literature [54, 10, 42, 23]. In these experiments our algorithm was carefully compared with an existing simulated annealing implementation (SAC) [16], a greedy method called IPOG-F, and other five state-of-the-art algorithms. The results show that ISA improves 11.11% on average the best results found by SAC, expending for that a small fraction of the computational time employed by this existing algorithm. Regarding IPOG-F [23] we have observed that on average it consumes more computational time than our ISA algorithm while returning on average worse quality solutions. Compared with the representative state-of-the-art algorithms, our ISA algorithm was able to improve on 104 previous best-known solutions and to equal these results on the other 23 selected benchmark instances, expending a reasonable amount of time. Furthermore, it was demonstrated that certain of these new bounds on $\mathrm{CAN}(3, k, 2)$ can be used as input to recursive construction methods in order to produce 20 new bounds for binary CAs of strength three and degrees $88 \leq k \leq 10648$ [21].

These experimental results confirm the practical advantages of using our algorithm in the software testing area. It is a robust algorithm yielding smaller test suites than other representative state-of-the-art algorithms at competitive computational time, which allows reducing software testing costs.

The introduction of this new simulated annealing implementation opens up an exciting range of possibilities for future research. Currently we are working on applying directly the ISA algorithm presented here for constructing CAs of strength $t > 6$ and order $v > 2$. More generally, we think that it would be interesting to adapt the data structures currently used in our ISA algorithm for allowing it to construct mixed level covering arrays, i.e., CAs where the number of component values varies.

Finally we would like to point out that the CAs constructed using the ISA algorithm reported in this paper are available under request at the following address: `http://www.tamps.cinvestav.mx/~jtj`.

Acknowledgements

The authors would like to thank Myra B. Cohen for running her simulated annealing implementation on some instances and for giving us her results in the appropriate format for reporting our experiments. We also would like to express our gratitude to Renée C. Bryce and Charles J. Colbourn for sharing their DDA source code. The 5 reviewers of the paper are greatly acknowledged for their constructive comments which have aided to improve the presentation of this work.

References

[1] E.H.L. Aarts, P.J.M. Van Laarhoven, A new polynomial-time cooling schedule, in: Proceedings of the IEEE International Conference on Computer Aided Design, IEEE Press, 1985, pp. 206–208.

[2] D. Abramson, M. Krishnamoorthy, H. Dang, Simulated annealing cooling schedules for the school timetabling problem, Asia-Pacific Journal of Operational Research 16 (1999) 1–22.

[3] A. Arcuri, X. Yao, Search based software testing of object-oriented containers, Information Sciences 178 (2008) 3075–3095.

[4] B. Beizer, Software Testing Techniques, International Thompson Computer Press, Boston, USA, second edition, 1990.

[5] R.C. Bryce, C.J. Colbourn, The density algorithm for pairwise interaction testing, Software Testing, Verification and Reliability 17 (2007) 159–182.

[6] R.C. Bryce, C.J. Colbourn, A density-based greedy algorithm for higher strength covering arrays, Software Testing, Verification and Reliability 19 (2009) 37–53.

[7] P. Bueno, M. Jino, W. Wong, Diversity oriented test data generation using metaheuristic search techniques, Information Sciences In Press, Corrected Proof. DOI 10.1016/j.ins.2011.01.025 (2011).

[8] K.A. Bush, Orthogonal arrays of index unity, Annals of Mathematical Statistics 23 (1952) 426–434.

[9] K.Y. Cai, Z. Dong, K. Liu, Software testing processes as a linear dynamic system, Information Sciences 178 (2008) 1558–1597.

[10] M.A. Chateauneuf, D.L. Kreher, On the state of strength-three covering arrays, Journal of Combinatorial Design 10 (2002) 217–238.
[11] C.T. Cheng, The test suite generation problem: Optimal instances and their implications, Discrete Applied Mathematics 155 (2007) 1943–1957.
[12] D.M. Cohen, S.R. Dalal, M.L. Fredman, G.C. Patton, The AETG system: An approach to testing based on combinatorial design, IEEE Transactions on Software Engineering 23 (1997) 437–444.
[13] D.M. Cohen, S.R. Dalal, J. Parelius, G.C. Patton, The combinatorial design approach to automatic test generation, IEEE Software 13 (1996) 83–88.
[14] M.B. Cohen, Personal communication, 2009.
[15] M.B. Cohen, C.J. Colbourn, A.C.H. Ling, Constructing strength three covering arrays with augmented annealing, Discrete Mathematics 308 (2008) 2709–2722.
[16] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, C.J. Colbourn, Constructing test suites for interaction testing, in: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, 2003, pp. 38–48.
[17] C.J. Colbourn, Combinatorial aspects of covering arrays, Le Matematiche 59 (2004) 121–167.
[18] C.J. Colbourn, Covering Array Tables, `http://www.public.asu.edu/~ccolbou/src/tabby/catable.html`, Accessed on July 18, 2011.
[19] C.J. Colbourn, G. Kéri, Binary covering arrays and existentially closed graphs, Lecture Notes in Computer Science 5557 (2009) 22–33.
[20] C.J. Colbourn, S.S. Martirosyan, T. Van Trung, R.A. Walker II, Roux-type constructions for covering arrays of strengths three and four, Designs, Codes and Cryptography 41 (2006) 33–57.
[21] C.J. Colbourn, J. Torres-Jimenez, Heterogeneous hash families and covering arrays, Contemporary Mathematics 523 (2010) 3–15.
[22] Z. Ding, K. Zhang, J. Hu, A rigorous approach towards test case generation, Information Sciences 178 (2008) 4057–4079.
[23] M. Forbes, J. Lawrence, Y. Lei, R.N. Kacker, D.R. Kuhn, Refining the in-parameter-order strategy for constructing covering arrays, Journal of Research of the National Institute of Standards and Technology 113 (2008) 287–297.
[24] F. Glover, Ejection chains, reference structures and alternating path methods for traveling salesman problems, Discrete Applied Mathematics 65 (1996) 223–253.
[25] L. Gonzalez-Hernandez, J. Torres-Jimenez, MiTS: A new approach of tabu search for constructing mixed covering arrays, Lecture Notes in Artificial Intelligence 6438 (2010) 382–392.
[26] B. Hajek, Cooling schedules for optimal annealing, Mathematics of Operations Research 13 (1988) 311–329.
[27] A. Hartman, Software and hardware testing using combinatorial covering suites, in: Graph Theory, Combinatorics and Algorithms, Springer-Verlag, 2005, pp. 237–266.
[28] A. Hartman, L. Raskin, Problems and algorithms for covering arrays, Discrete Mathematics 284 (2004) 149–156.
[29] A.S. Hedayat, N.J.A. Sloane, J. Stufken, Orthogonal Arrays, Theory and Applications, Springer, Berlin, 1999.
[30] D.S. Johnson, C.R. Aragon, L.A. McGeoch, C. Schevon, Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning, Operations Research 37 (1989) 865–892.
[31] D.S. Johnson, C.R. Aragon, L.A. McGeoch, C. Schevon, Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning, Operations Research 39 (1991) 378–406.
[32] D.J. Kleitman, J. Spencer, Families of k-independent sets, Discrete Mathematics 6 (1973) 255–262.
[33] D.R. Kuhn, M.J. Reilly, An investigation of the applicability of design of experiments to software testing, in: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop, IEEE Computer Society, 2002, p. 91.
[34] D.R. Kuhn, D.R. Wallace, A.M. Gallo, Software fault interactions and implications for software testing, IEEE Transactions on Software Engineering 30 (2004) 418–421.
[35] J. Lawrence, R.N. Kacker, Y. Lei, D.R. Kuhn, M. Forbes, A survey of binary covering arrays, The Electronic Journal of Combinatorics 18 (2011) P84.
[36] Y. Lei, R.N. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, Ipog: A general strategy for t-way software, in: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, IEEE Computer Society, 2007, pp. 549–556.
[37] J. Lin, P. Yeh, Automatic test data generation for path testing using gas, Information Sciences 131 (2001) 47–64.
[38] Z. Lü, J.K. Hao, F. Glover, Neighborhood analysis: a case study on curriculum-based course, Journal of Heuristics In Press (2009).
[39] H.B. Mann, The construction of orthogonal latin squares, Annals of Mathematical Statistics 13 (1942) 418–423.
[40] S.S. Martirosyan, Perfect Hash Families, Identifiable Parent Property Codes and Covering Arrays, Ph.D. thesis, Department of Mathematics, University Duisburg-Essen, Germany, 2003.
[41] S.S. Martirosyan, T. Van Trung, On t-covering arrays, Designs, Codes and Cryptography 32 (2004) 323–339.
[42] K.J. Nurmela, Upper bounds for covering arrays by tabu search, Discrete Applied Mathematics 138 (2004) 143–152.
[43] K.J. Nurmela, P.R.J. Östergård, Constructing Covering Designs by Simulated Annealing, Technical Report 10, Department of Computer Science, Helsinki University of Technology, Otaniemi, Finland, 1993.
[44] T.J. Ostrand, M.J. Balcer, The category-partition method for specifying and generating functional tests, Communications of the ACM 31 (1988) 676–686.
[45] N. Rangel-Valdez, J. Torres-Jimenez, J. Bracho-Rios, P. Quiz-Ramos, Problem and algorithm fine-tuning - a case of study using bridge club and simulated annealing, in: Proceedings of the International Joint Conference on Computational Intelligence, INSTICC Press, 2009, pp. 302–305.
[46] C.R. Rao, Factorial arrangements derivable from combinatorial arrangements of arrays, Journal of the Royal Statistical Society 9 (1947) 128–139.
[47] A. Rényi, Foundations of Probability, Wiley, New York, USA, 1971.
[48] E. Rodriguez-Tello, J.K. Hao, J. Torres-Jimenez, An effective two-stage simulated annealing algorithm for the minimum linear arrangement problem, Computers & Operations Research 35 (2008) 3331–3346.
[49] E. Rodriguez-Tello, J.K. Hao, J. Torres-Jimenez, An improved simulated annealing algorithm for bandwidth minimization, European Journal of Operational Research 185 (2008) 1319–1335.

[50] E. Rodriguez-Tello, J. Torres-Jimenez, Memetic algorithms for constructing binary covering arrays of strength three, Lecture Notes in Computer Science 5975 (2010) 86–97.
[51] G. Roux, *k*-propriétés dans des tableaux de *n* colonnes; cas particulier de la *k*-surjectivité et de la *k*-permutivité, Ph.D. thesis, Université de Paris 6, France, 1987.
[52] G. Seroussi, N. Bshouty, Vector sets for exhaustive testing of logic circuits, IEEE Transactions on Information Theory 34 (1988) 513–522.
[53] G.B. Sherwood, S.S. Martirosyan, C.J. Colbourn, Covering arrays of higher strength from permutation vectors, Journal of Combinatorial Designs 14 (2006) 202–213.
[54] N.J.A. Sloane, Covering arrays and intersecting codes, Journal of Combinatorial Designs 1 (1993) 51–63.
[55] J. Stardom, Metaheuristics and the Search for Covering and Packing Arrays, Master's thesis, Simon Fraser University, Burnaby, Canada, 2001.
[56] P.J.M. Van Laarhoven, E.H.L. Aarts, Simulated Annealing: Theory and Applications, Kluwer Academic Publishers, 1988.
[57] J.M. Varanelli, J.P. Cohoon, A fast method for generalized starting temperature determination in homogeneous two-stage simulated annealing systems, Computers & Operations Research 26 (1999) 481–503.
[58] R.A. Walker II, C.J. Colbourn, Tabu search for covering arrays using permutation vectors, Journal of Statistical Planning and Inference 139 (2009) 69–80.
[59] K.Z. Zamli, M.F.J. Klaib, M.I. Younis, N.A.M. Isa, R. Abdullah, Design and implementation of a t-way test data generation strategy with automated execution tool support, Information Sciences 181 (2011) 1741–1758.