Simulated Annealing for Constructing Binary Covering Arrays of Variable Strength

Jose Torres-Jimenez and Eduardo Rodriguez-Tello

Abstract— **This paper presents new upper bounds for binary covering arrays of variable strength constructed by using a new Simulated Annealing (SA) algorithm. This algorithm incorporates several distinguished features including an efficient heuristic to generate good quality initial solutions, a compound neighborhood function which combines two carefully designed neighborhoods and a fine-tuned cooling schedule. Its performance is investigated through extensive experimentation over well known benchmarks and compared with other state-of-the-art algorithms, showing that the proposed SA algorithm is able to outperform them.**

I. Introduction

Software systems are becoming ubiquitous in modern society where numerous human activities rely on them. Ensuring that software systems meet people's expectations for quality and reliability is an expensive and highly complex task. Especially, considering that usually those systems have many possible configurations produced by the combination of multiple input parameters, making immediately impractical the use of exhaustive testing. An alternative technique to accomplish this goal is called *software interaction testing* [1]. It is based on constructing economical sized test-suites that provide coverage of the most prevalent configurations. Covering arrays (CAs) are combinatorial structures which can be used to represent these test-suites.

A covering array, $\mathrm{CA}(N;t,k,v)$, 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 contains all ordered subsets from v symbols of size t (t-tuples) at least once. In such an array, each *test configuration* of the analyzed software system is represented by a row. A test configuration is composed by the combination of k parameters taken on v values. This test-suite allows to cover all the t-way combinations of parameter values, (i.e. for each set of t parameters every t-tuple of parameter values is represented). Then, software testing costs can be substantially reduced by minimizing the number of test configurations N in a covering array. The minimum N for which a $\mathrm{CA}(N;t,k,v)$ exists is the *covering array number* and it is defined according to (1).

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

Jose Torres-Jimenez and Eduardo Rodriguez-Tello are with the CINVESTAV-Tamaulipas, Information Technology Laboratory, Km. 5.5 Carretera Victoria-Soto La Marina, 87130 Victoria Tamps., MEXICO (email: {jtj, ertello}@cinvestav.mx).

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

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

There exist only some special cases where it is possible to find the covering array number using polynomial order algorithms. For instance, Rényi, Katona, and Kleitman and Spencer independently proposed constructions, in the 1970s, for finding $\mathrm{CAN}(2,k,2)$; see [2] for references. However, in the general case determining the covering array number is known to be a hard combinatorial optimization problem [3], [4].

Some other applications related to the CAC problem arise in experimental design where it is absolutely necessary to test the interaction of all combinations of t parameters, and hence that all such selections are covered by columns of the array. Some examples of these applications are: drug screening, data compression, regulation of gene expression, authentication, intersecting codes and universal hashing (see [5] for a detailed survey).

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 covering arrays are: a) recursive methods [6], b) algebraic methods [5], c) greedy methods [7] and d) metaheuristics such as Simulated Annealing [8], Genetic Algorithms [9], Ant Colony Algorithms [10] and Tabu Search [11].

This paper aims at developing a powerful Simulated Annealing (SA) algorithm for finding near-optimal solutions for the CAC problem. In particular, we are interested in constructing binary covering arrays of strengths three, four and five and in establishing new upper bounds on the covering array number. Contrary to other existing SA implementations developed for solving the CAC problem, our algorithm has the merit of improving three key features that have a great impact on its performance: a method designed to generate initial solutions containing a balanced number of symbols in each column, a compound neighborhood function combining two carefully designed neighborhoods and a fine-tuned cooling schedule. The performance of the proposed SA algorithm is assessed with a benchmark, composed by 40 binary covering arrays of strengths three, four and five taken from the literature. The computational results are reported and compared with previously published ones, showing that our algorithm is able to find 22 new upper bounds and to equal 18 previous best-known solutions on the selected

benchmark instances.

The rest of this paper is organized as follows. In Section II, a brief review is given to present some representative solution procedures for constructing binary covering arrays. Then, the components of our new SA algorithm are discussed in detail in Section III. Section IV is mainly dedicated to computational experiments and comparisons with respect to previous published upper bounds. This section is also devoted to analyze the influence of the neighborhood function on the performance of the proposed SA implementation as well as the implications of the new upper bounds achieved by our algorithm when they are used as ingredients to recursive constructions [6]. Finally, the last section summarizes the main contributions of this work.

II. Relevant Related Work

Because of the importance of the CAC problem, much research has been carried out in developing effective methods for solving it. Most of the known algorithms are specially designed for constructing strength two and three CAs [5], [8]. However, less is known about CAs with larger strength [12]. In this section, we give a brief review of some representative procedures which were used in our comparisons.

A Simulated Annealing (SA) metaheuristic has been applied by Cohen et al. in [13] for solving the CAC problem. Their SA implementation starts with a randomly generated initial solution A which cost $c(A)$ is 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 SA implementation is able to produce smaller CAs than other computational methods, sometimes improving upon algebraic constructions. However, they also indicate that their SA algorithm fails to match the algebraic constructions for larger problems, especially when $t = 3$ [13].

In [14] a Tabu Search (TS) algorithm is presented by Walker and Colbourn. This algorithm employs a compact representation of CAs based on permutation vectors and covering perfect hash families [15] in order to reduce the size of the search space. Using this algorithm, improved CAs of strength 3 to 5 have been found, as well as the first arrays of strength 6 and 7 found by computational search.

Bryce and Colbourn published in [16] a method called Deterministic Density Algorithm (DDA) which constructs strength two covering arrays 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 covering arrays of strength $t \geq 3$ [17]. 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 covering arrays that are of competitive size, and expending less computational time than other published methods like TS [14] and SA [8].

More recently Forbes *et al.* [18] introduced an algorithm for the efficient production of covering arrays of strength t up to 6, called IPOG-F (In-Parameter Order-Generalized). Contrary to many other algorithms that build covering arrays one row at a time, the IPOG-F strategy constructs them one column at a time. The main idea is that covering arrays 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, which was written in Java. Even if IPOG-F is a very fast algorithm for producing covering arrays it generally provides poorer quality results than other state-of-the-art algorithm like the recursive procedure proposed in [6].

III. An Improved Simulated Annealing Algorithm

Simulated Annealing (SA) 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 [19].

In this section we present a new implementation of a SA algorithm for constructing binary CAs of strength five. It improves three 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, a composed neighborhood function and an effective cooling schedule. Next all the details of the SA implementation proposed are presented.

A. Internal Representation and Search Space

Let A be a potential solution in the search space $\mathscr{A}$, that is a covering array $\mathrm{CA}(N;t,k,v)$ 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:

$$|\mathscr{A}| = v^{Nk} \tag{2}$$

B. Evaluation Function

Previously reported metaheuristic algorithms for the CAC problem have commonly evaluated the quality of a potential solution (covering array) as the change in the number of uncovered t-tuples [8], [9], [10], [11]. This evaluation function is formally defined as follows. Let $A \in \mathscr{A}$ be a potential solution, S^r a $N \times t$ subarray of A representing the r-th subset of t columns taken from k, and ϑ_j a set containing the union[1] of the N t-tuples in S^j denoted by the following expression:

$$\vartheta_j = \bigcup_{i=0}^{N-1} S_i^j, \tag{3}$$

then the evaluation function $\mathscr{F}(A)$ for computing the cost of a potential solution A can be defined using (4).

$$\mathscr{F}(A) = \binom{k}{t} v^t - \sum_{j=0}^{\binom{k}{t}-1} |\vartheta_j| \tag{4}$$

In the proposed SA implementation this evaluation function definition was used. Its computational complexity is equivalent to $O(N\binom{k}{t})$. However, by using appropriate data structures it allows an incremental fitness evaluation of neighboring solutions in $O(2\binom{k-1}{t-1})$ operations.

C. 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 SA implementations reported in the literature [20], [8] the initial solution is randomly generated. In contrast, in our implementation the starting solution is created using a procedure that guarantees a balanced number of symbols in each column of the generated covering array $\mathrm{CA}(N;t,k,v)$. 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.

We have decided to use this particular method for constructing the initial solution because we have observed, from preliminary experiments, that good quality solutions contain a balanced number of symbols in each column.

[1]Please remember that the union operator $\cup$ in set theory eliminates duplicates.

D. Neighborhood Function

Given that our SA implementation 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 a LS algorithm. Formally, a neighborhood relation is a function $\mathscr{N} : \mathscr{A} \rightarrow 2^{\mathscr{A}}$ that assigns to every potential solution (a covering array) $A \in \mathscr{A}$ a set of neighboring solutions $\mathscr{N}(A) \subseteq \mathscr{A}$, which is called the neighborhood of A.

The neighborhood function is therefore a key component which has a great impact on the performance of every LS algorithm. For instance, some neighborhoods allow the search to obtain solution improvements in a quick and important manner, but the improvement occurs only for a limited number of iterations. On the contrary, other neighborhoods only enable small improvements, but for a long time.

In case two or more neighborhoods present complementary characteristics, it is then possible and interesting to create more powerful compound neighborhoods. The advantage of such an approach is well documented in [21], [22], [23]. Following this idea, and based on the results of our preliminary experimentations, a neighborhood structure composed by two different functions is proposed for this SA algorithm implementation.

Let be a function allowing to change the value of the element $a_{i,j}$ by a different legal member of the alphabet in the current solution A, and $W \subseteq \mathscr{A}$ a set containing ω different neighboring solutions of A created by applying the function $switch(A,i,j)$ with different random values of i and j ($0 \leq i < N$, $0 \leq j < k$). Then the first neighborhood $\mathscr{N}_1(A,\omega)$ of a potential solution A, used in our SA implementation can be defined using the following expression:

$$\mathscr{N}_1(A,\omega) = \left\{ A' \in \mathscr{A} : A' = \min_{\forall A'' \in W, |W|=\omega} [\mathscr{F}(A'')] \right\} \tag{5}$$

In (6), the second neighborhood $\mathscr{N}_2(A)$ used in our SA implementation is defined. It requires the use of a function $swap(A,i,j,l)$ which exchanges the values of two elements $a_{i,j}$ and $a_{l,j}$ ($a_{i,j} \neq a_{l,j}$) within the same column of A, and a set $R \subseteq \mathscr{A}$ containing neighboring solutions of A produced by γ successive applications of the function $swap(A,i,j,l)$ using randomly chosen values for the parameters i, j and l ($0 \leq i < N$, $0 \leq l < N$, $0 \leq j < k$).

$$\mathscr{N}_2(A,\gamma) = \left\{ A' \in \mathscr{A} : A' = \min_{\forall A'' \in R, |R|=\gamma} [\mathscr{F}(A'')] \right\} \tag{6}$$

During the search process a combination of both $\mathscr{N}_1(A,\omega)$ and $\mathscr{N}_2(A,\gamma)$ neighborhood functions is employed by our SA algorithm. The former is applied with probability p, while the latter is employed at a $(1-p)$ rate. This combined neighborhood function $\mathscr{N}_3(A,x,\omega,\gamma)$ is defined in (7), where x is a random number in the interval $[0,1]$.

$$\mathscr{N}_3(A,x,\omega,\gamma) = \begin{cases} \mathscr{N}_1(A,\omega) & \text{if } x \leq p \\ \mathscr{N}_2(A,\gamma) & \text{if } x > p \end{cases} \tag{7}$$

E. Cooling Schedule

The cooling schedule determines the degree of uphill movement permitted during the search and is thus critical to the SA algorithm's performance. The parameters that define a cooling schedule are: an initial temperature, a final temperature or a stopping criterion, the maximum number of neighboring solutions that can be generated at each temperature, and a rule for decrementing the temperature.

The literature offers a number of different cooling schedules, see for instance [24], [25], [26]. In our SA implementation we preferred a geometrical cooling scheme 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 $T_k = \alpha T_{k-1}$. 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 three main original features presented previously, our SA algorithm using this simple cooling scheme gives remarkable results.

F. Termination Condition

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

IV. 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 Xeon at 2 GHz, 1 GB of RAM with Linux operating system. Due to the incomplete and non-deterministic nature of the algorithm, 20 independent runs were executed for each of the selected benchmark instances. In all the experiments the following parameters were used for our SA 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 and taking into consideration our experience in solving other combinatorial optimization problems with the use of SA algorithms [22], [27].

In order to assess the performance of the SA algorithm introduced in Section III (called hereafter SA), a benchmark composed of 40 well known instances taken from the literature was used [14], [28], [18]. It includes binary instances of degree $4 \leq k \leq 25$ and strength $3 \leq t \leq 5$. The main criterion used for the comparison is the same as the one commonly used in the literature: the *best size N* found (smaller values are better) given fixed values for k, t, and v.

A. Comparing SA With the State-of-the-art Procedures

The purpose of this experiment is to carry out a performance comparison of the upper bounds achieved by SA with respect to those produced by the following state-of-the-art procedures: Deterministic density algorithm (DDA) [17], Tabu Search (TS) [14], and IPOG-F [18]. The Simulated Annealing implementation reported by Cohen et al. [13] 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 $t \geq 3$.

Table I lists the detailed computational results produced by this experiment. The first two columns in the table indicate the strength t and degree k of the instances. Next 3 columns show, in terms of the size N of the CAs, the best solution found by DDA [17], TS [14] and IPOG-F [18], respectively. Column 6 presents the smallest (Best) CAs published in the literature [28], while column 7 reports the best solutions achieved by our SA. The computational times T, in seconds, consumed by it are listed in column 8. Finally, the difference ($\Delta_{SA-Best}$) between the best result produced by SA and the previous best-known solution is depicted in the last column.

From Table I we can observe that SA compares very favorably with respect to the state-of-the-art procedures summarized in column 6. Indeed, our SA implementation is able to improve on 22 previous best-known solutions and to equal these results for the other 18 instances in the benchmark (see column $\Delta_{SA-Best}$). Remark that for certain instances, like covering array CA$(N;5,14,2)$, a significant reduction of the size N, of 60.94%, is accomplished by our algorithm when compared with the previous best-known solution.

We can also observe that DDA is dominated by IPOG-F on the selected benchmark instances. However, IPOG-F produces covering arrays which are in average 37.77% bigger than those constructed by SA. It is also clear that the solutions provided by our SA algorithm are better than those created with the use of TS [14] since they are in average 57.32% worst than ours.

The favorable results achieved by SA are more evident over the selected strength five benchmark instances. This is illustrated in Figure 1. The plot represents the degree k of the instance (abscissa) against the size N attained by the compared procedures (ordinate). The bounds provided by IPOG-F [18] are shown with squares, the previous best-known solutions [28] are depicted as circles, while the bounds computed with SA are shown as triangles. From this figure it can be seen that SA consistently outperforms IPOG-F, obtaining also important improvements with respect to the previous best-known solutions on CAN$(5,k,2)$ for $6 \leq k \leq 24$.

Even if the results attained by SA are very competitive, we have observed that the average computing time consumed by our approach, to produce these excellent results, is greater than that used by some recursive [29], [6] and algebraic

TABLE I

EXPERIMENTAL COMPARISON AMONG SA AND THREE STATE-OF-THE-ART PROCEDURES OVER 40 BINARY COVERING ARRAYS OF STRENGTHS THREE, FOUR AND FIVE TAKEN FROM THE LITERATURE.

		N						
t	*k*	DDA	TS	IPOG-F	Best	SA	T	$\Delta_{SA-Best}$
3	4	8	8	8	8	8	0.001	0
	5	10	10	11	10	10	0.001	0
	11	20	12	18	12	12	0.001	0
	12	21	15	19	15	15	0.003	0
	14	27	16	21	16	16	52.450	0
	16	27	17	22	17	17	21.720	0
	20	30	18	25	18	18	59.100	0
	22	32	19	26	19	19	31.700	0
	23	34	22	26	22	20	38.920	-2
	25	34	23	27	23	21	19.770	-2
4	5	16	16	22	16	16	0.001	0
	6	26	21	26	21	21	0.001	0
	12	52	48	47	24	24	0.001	0
	13	53	53	49	34	32	725.670	-2
	17	67	54	57	39	35	1876.450	-4
	18	73	55	60	39	36	1467.890	-3
	20	74	55	65	39	39	1476.170	0
	21	85	80	68	42	42	1534.890	0
	22	85	80	69	44	44	1675.450	0
	24	89	80	71	46	46	1765.790	0
	25	91	80	74	50	50	1894.450	0
5	6	32	32	42	32	32	0.001	0
	7	52	56	57	42	42	0.001	0
	8	76	56	68	56	52	0.880	-4
	9	90	62	77	62	54	20.140	-8
	10	102	62	87	62	56	649.590	-6
	11	108	92	95	82	64	872.100	-18
	12	126	92	105	89	64	1233.300	-25
	13	136	110	111	95	64	1349.780	-31
	14	146	110	119	103	64	1534.460	-39
	15	153	152	127	110	79	1890.780	-31
	16	171	152	134	117	99	2350.780	-18
	17	176	176	140	123	104	5823.650	-19
	18	197	176	144	127	109	13807.900	-18
	19	205	176	148	135	117	18675.670	-18
	20	205	194	155	136	120	20679.190	-16
	21	229	261	160	136	122	22876.390	-14
	22	236	261	163	136	124	24935.870	-12
	23	257	261	168	136	132	37923.270	-4
	24	260	261	175	136	132	39679.390	-4
Avg.		97.78	88.10	77.15	61.73	54.28	5173.59	-7.45

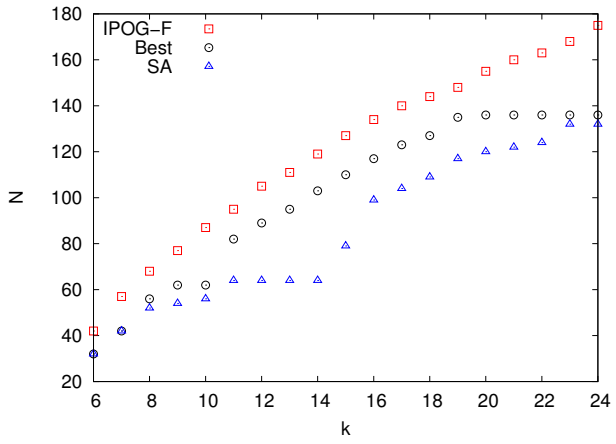

Fig. 1. Previous best-known and improved bounds on CAN$(5,k,2)$.

methods [30], [5], [31]. However, since SA outperforms some of the state-of-the-art procedures, finding 22 new bounds, we believe that the extra consumed computing time is fully justified. Especially, if we consider that for this kind of experiments the objective is to compare the best bounds achieved by the studied algorithms. Furthermore, compared with TS [14], which consumed between an hour and a day of CPU time (on a 2.66 GHz Pentium 4 PC) for computing the CAs reported here, SA is reasonably competitive because it is able to solve the largest instance in the selected benchmark, CA$(N;5,24,2)$, by employing only 10.84 hours of CPU time on the computer described in Section IV.

It is important to point out that, in general, authors of the algorithms used in our comparisons only provide the best solution quality achieved by them. Thus, these algorithms cannot be statistically compared with our SA algorithm.

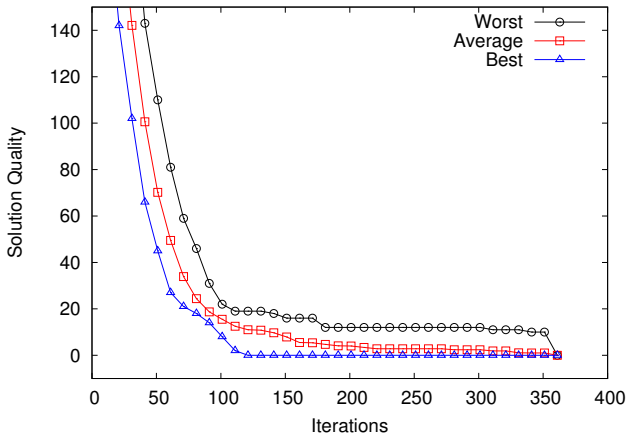

Fig. 2. Execution profiles of SA over the instance CA(52;5,8,2).

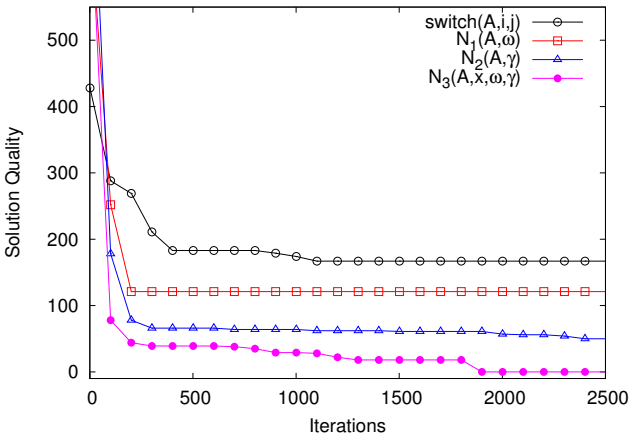

Fig. 3. Four neighborhood functions and its influence on the performance of SA over the instance CA(56;5,10,2).

Nevertheless, the average behavior of SA can be illustrated and analyzed from the execution profiles produced during the 20 independent executions of this comparative experiment. Figure 2 shows three SA's execution profiles which represent the evolution of the worst, average and best values of the evaluation function $\mathscr{F}(A)$ during the search process for constructing the covering array CA(56;5,10,2).

From Figure 2 we can observe a relatively small gap between the worst and the best solution found during these executions. This is a good indicator of the algorithm's precision and robustness since it shows that in average the performance of our SA implementation does not present important fluctuations. For this particular instance the average standard deviation is 6.54. This figure allows to summarize the overall behavior of SA since similar results were obtained with all the other tested instances.

B. Analysis of SA

In order to further examine the behavior of our SA implementation we have performed an additional experiment for analyzing the influence of the following neighborhood functions (described in Section III-D) on its performance:

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

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

C. Some Implications of the Results

The new upper bounds obtained with SA in the experimental comparison presented in Section IV-A by themselves represent an important achievement in the study of these combinatorial structures. However, they also have the additional value of being excellent ingredients to recursive constructions used to produce other new bounds for binary CAs of higher degrees.

For instance, by employing the construction reported in [32] over the CAs listed in column 4 of Table II, we can construct the binary CAs of strength five indicated by the first and third columns of this table. Those new results improve the previous best-known solutions (Best) reported in the literature for the given values k, t, and v [28]. The last column Δ indicates the improvement achieved by the combination of SA and this recursive construction (SA+Recursive) with respect to the best-known size N depicted in column 2.

From Table II one observes that using recursive constructions taking as input certain CAs found with our SA algorithm, permits to find 5 new upper bounds which are considerably better (Δ up to -158) than those reported in [28].

V. Conclusions

In this paper, we have introduced a highly effective Simulated Annealing (SA) algorithm, which integrates three key features that importantly determines its performance. First, an efficient method to generate initial solutions containing a balanced number of symbols in each column. This initialization method allows us to replace some SA actions occurring at the highest temperatures saving thus important computing

TABLE II

IMPROVED UPPER BOUNDS FOR BINARY CAS OF STRENGTH FIVE AND DEGREE $k > 24$ PRODUCED BY COMBINING SA AND RECURSIVE CONSTRUCTIONS [32].

	N			
k	Best	SA+Recursive	Using	Δ
1014	946	788	CA(64;5,13,2), CA(32;5,6,2)	-158
1183	946	797	CA(64;5,13,2), CA(42;5,7,2)	-149
1352	946	807	CA(64;5,13,2), CA(52;5,8,2)	-139
1521	946	809	CA(64;5,13,2), CA(54;5,9,2)	-137
1690	946	811	CA(64;5,13,2), CA(56;5,10,2)	-135
Avg.	946.00	802.40		-143.60

time. 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. Third, an effective cooling schedule allowing our SA algorithm to converge faster, producing at the same time good quality solutions.

To assess the practical effectiveness of this SA algorithm, we have carried out extensive experimentation using a set of 40 benchmark instances taken from the literature. In these experiments our SA algorithm was carefully compared with other three state-of-the-art algorithms. The results show that SA was able to find 22 new upper bounds and to equal 18 previous best-known solutions on the selected benchmark. Furthermore, it was demonstrated that certain of these new upper bounds can be used as input to recursive construction methods in order to produce other new bounds for binary CAs of higher degrees.

The results obtained with the SA implementation presented opens up an exciting range of possibilities for future research. We are currently considering that one fruitful possibility is to adapt our SA algorithm for the efficient construction of CAs of strength $t \geq 6$ and order $v > 2$.

ACKNOWLEDGMENT

The authors would like to thank Renée C. Bryce and Charles J. Colbourn for sharing their DDA source code.

REFERENCES

[1] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, 1996.

[2] N. J. A. Sloane, "Covering arrays and intersecting codes," *Journal of Combinatorial Designs*, vol. 1, no. 1, pp. 51–63, 1993.

[3] G. Seroussi and N. Bshouty, "Vector sets for exhaustive testing of logic circuits," *IEEE Transactions on Information Theory*, vol. 34, pp. 513–522., 1988.

[4] Y. Lei and K. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *In Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 254–261.

[5] M. A. Chateauneuf and D. L. Kreher, "On the state of strength-three covering arrays," *Journal of Combinatorial Design*, vol. 10, no. 4, pp. 217–238, 2002.

[6] S. S. Martirosyan and T. Van Trung, "On t-covering arrays," *Designs, Codes and Cryptography*, vol. 32, no. 1-3, pp. 323–339, 2004.

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, pp. 437–444, 1997.

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

[9] J. Stardom, "Metaheuristics and the search for covering and packing arrays," Master's thesis, Simon Fraser University, Burnaby, Canada, 2001.

[10] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proceedings of the IEEE 28th Annual International Computer Software and Applications Conference*, vol. 1. IEEE Computer Society, 2004, pp. 72–77.

[11] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, no. 1-2, pp. 143–152, 2004.

[12] C. J. Colbourn, "Combinatorial aspects of covering arrays," *Le Matematiche*, vol. 58, pp. 121–167, 2004.

[13] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and 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.

[14] R. A. Walker II 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.

[15] 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.

[16] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 159–182, 2007.

[17] ——, "A density-based greedy algorithm for higher strength covering arrays," *Software Testing, Verification and Reliability*, vol. (in press), 2008.

[18] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, vol. 113, no. 5, pp. 287–297, 2008.

[19] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning," *Operations Research*, vol. 37, no. 6, pp. 865–892, 1989.

[20] K. J. Nurmela and P. R. J. Östergård, "Constructing covering designs by simulated annealing," Department of Computer Science, Helsinki University of Technology, Otaniemi, Finland, Tech. Rep. 10, Jan. 1993.

[21] L. Cavique, C. Rego, and I. Themido, "Subgraph ejection chains and tabu search for the crew scheduling problem," *Journal of the Operational Research Society*, vol. 50, no. 6, pp. 608–616, 1999.

[22] E. Rodriguez-Tello, J. K. Hao, and J. Torres-Jimenez, "An improved simulated annealing algorithm for bandwidth minimization," *European Journal of Operational Research*, vol. 185, no. 3, pp. 1319–1335, Mar. 2008.

[23] Z. Lü, J. K. Hao, and F. Glover, "Neighborhood analysis: a case study on curriculum-based course," *Journal of Heuristics*, vol. In Press, 2009.

[24] E. H. L. Aarts and P. J. M. Van Laarhoven, "A new polynomial-time cooling schedule," in *Proceedings of the IEEE International Conference on Computer Aided Design*. Santa Clara, CA, USA: IEEE Computer Society, 1985, pp. 206–208.

[25] M. D. Huang, F. Romeo, and A. L. Sangiovanni-Vincentelli, "An efficient general cooling schedule for simulated annealing," in *Proceedings of the IEEE International Conference on Computer Aided Design*. Santa Clara, CA, USA: IEEE Computer Society, 1986, pp. 381–384.

[26] J. M. Varanelli and J. P. Cohoon, "A fast method for generalized starting temperature determination in homogeneous two-stage simulated annealing systems," *Computers & Operations Research*, vol. 26, no. 5, pp. 481–503, 1999.

[27] E. Rodriguez-Tello, J. K. Hao, and J. Torres-Jimenez, "An effective two-stage simulated annealing algorithm for the minimum linear arrangement problem," *Computers & Operations Research*, vol. 35, no. 10, pp. 3331–3346, Oct. 2008.

[28] C. J. Colbourn, "Covering Array Tables," http://www.public.asu.edu/~ccolbou/src/tabby/catable.html, Accessed on April 29, 2009.

[29] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," *Discrete Mathematics*, vol. 284, no. 1-3, pp. 149–156, 2004.

[30] A. S. Hedayat, N. J. A. Sloane, and J. Stufken, *Orthogonal Arrays, Theory and Applications*. Berlin: Springer, 1999.

[31] A. Hartman, "Software and hardware testing using combinatorial covering suites," in *Graph Theory, Combinatorics and Algorithms*. Springer-Verlag, 2005, ch. 10, pp. 237–266.

[32] C. J. Colbourn and J. Torres-Jimenez, "Heterogeneous hash families and covering arrays," *Contemporary Mathematics*, vol. (submitted for review on November, 2009), 2009.