SUBSECRETARÍA DE EDUCACIÓN SUPERIOR
DIRECCIÓN GENERAL DE EDUCACIÓN SUPERIOR TECNOLÓGICA
INSTITUTO TECNOLÓGICO DE CD. VICTORIA

SECRETARÍA DE EDUCACIÓN PÚBLICA

TecnoINTELECTO
Órgano de Divulgación Científica

Una Publicación del Instituto Tecnológico de Cd. Victoria

Volumen 7 No. 1 Abril 2010 ISSN 1665-983X

CIENCIAS EXACTAS Y NATURALES

INGENIERÍA Y TECNOLOGÍA

TecnoINTELECTO (ISSN 1665-983X y reserva: 04-2004-072626452400-102) es un órgano de divulgación científica de forma semestral del Instituto Tecnológico de Cd. Victoria. Boulevard Emilio Portes Gil No. 1301, C. P. 87010, Cd. Victoria, Tamaulipas, México; Tels. (834) 3130662 al 64; Fax: (834) 3133646. La responsabilidad del contenido y la sintaxis de los artículos presentados son responsabilidad del autor (es). Editor Principal: División de Estudios de Posgrado e Investigación. Apoyo editorial-informático: Blanca Rosa Méndez Gómez & Aurora Yazmín Rocha Sánchez. Envío de documentos, consultas y sugerencias al correo electrónico: ludivinab@yahoo.com, ludibarrientos@prodigy.net.mx

TecnoINTELECTO, Vol. 7 No. 1. Cd. Victoria, Tamaulipas, México.

CONSTRUCTION OF BINARY COVERING ARRAYS USING SIMULATED ANNEALING

E. Covarrubias-Flores[1], J. Torres-Jiménez[2], E. Rodríguez-Tello[2], N. Rangel-Valdez[2], R. Infante-Ventura[1]

[1]Instituto Tecnológico de Cd. Victoria. Blvd. Emilio Portes Gil # 1301,CP 87010, Cd. Victoria, Tamaulipas, México.
[2]Centro de Investigación y de Estudios Avanzados del IPN. Km. 5.5 carr. Victoria-Soto La Marina CP. 87130 Cd.Victoria,Tam.
{esmeralda,reginoi}@itcv.edu.mx,{jtj,ertello,nrangel}@tamps.cinvestav.mx

ABSTRACT: Empirical studies on software testing indicate that software failures can be reduced if all the interactions among parameters, produced by each subset of parameters of a size ***t***, are included during the software testing. When the size ***t*** of the subsets grows then the software will be more reliable. The set of Covering Array (CA) may improve the testing software process by creating a minimum set of test cases with the maximum possibility of testing the functionality of the developed software. The set of test cases includes the desired interactions between parameters, where the size ***t*** of the subsets is called the strength of the CA. In this paper a Simulated Annealing approach (SA) that constructs binary CA of variable strength is presented. This approach is based on a variable neighborhood function and on a parameter tuning phase (specifically we tuned the neighborhood function and the cooling schedule). The implemented algorithm solved a benchmark with 57 cases of strengths 3 and 4 reaching the best known results.

KEYWORDS: Simulated Annealing, Parameter Tuning, Variable Neighborhood Function, Covering Arrays.

RESUMEN: Estudios empíricos en pruebas de software indican que las fallas de software pueden reducirse si todas las interacciones entre parámetros, producidas por cada subconjunto de parámetros de tamaño t, son incluidas durante las pruebas de software. El conjunto del Covering Array (CA) puede mejorar el proceso de pruebas de software, creando un conjunto mínimo de casos de prueba, incluyendo las interacciones deseadas entre parámetros, donde el tamaño ***t*** del subconjunto es llamado la *fuerza* del CA. En este artículo se propone un enfoque del Simulated Annealing (SA), que construye CA binarios de fuerza variable. Esta propuesta está basada en una función de vecindad variable y en una fase de sintonización de parámetros (específicamente sintonizamos la función de vecindad y el esquema de enfriamiento). El algoritmo implementado resuelve un conjunto de 57 casos de fuerza 3 y 4 alcanzando los mejores resultados conocidos.

PALABRAS CLAVE: Recocido Simulado, sintonización de parámetros, función de vecindad variable, Covering Arrays.

1. INTRODUCTION

Making tests is a widely used way to assure the quality of the software that will be offered to the market. According to a NIST (National Institute of Standards and Technology) research report written in 2002 by Gregory Tassey (Tassey 2002), the national annual estimated cost of an inadequate infrastructure for software testing is $59.5 billion dollar; the potential cost reduction using an adequate infrastructure is $22.2 billion. These quantities respectively represent the 0.6 and 0.2 percent of the U.S. $10 trillion dollar GDP. Column 2 of the table 1 describes the costs of software failures; 40 percent of the costs are generated during the development of software while the rest occurs when the software is already in use. In column 3 the savings achieved by improving the software testing infrastructure are shown. It is possible to see that a third of the costs can be reduced. Assuming that we have a software with 10 parameters and each one has 2 possible values, to test that this software is failures free,

we must test 2^{10} combinations. With this simple example we can see that increasing the number of parameters makes impractical to use this exhaustive approach.

An alternative to improve the testing software process is the creation of tools that generate test cases in an automatic way. Through the computation of CA's, it is possible to obtain a minimum set of test cases, with the maximum possibility of testing the functionality of the developed software.

The set of test cases includes the desired interactions between parameters. The size ***t*** of the subsets from which the interactions are produced is called the strength of the CA. When the size ***t*** of the subsets grows then the software will be more reliable (but the size of the CA also grows).
In this paper, an implementation of a Simulated Annealing algorithm (Kirkipatrick 1983, van Laarhoven 1999) for the construction of variable strength binary CA's is presented. This algorithm integrates several important features such as the initialization of the columns of the CA with balanced symbols, a tuning approach of its parameters and the use of 2 perturbation operators as the neighborhood function. The rest of the paper is organized as follows. In section 2 some concepts about the CA are described. Section 3 expounds the Simulated Annealing algorithm. In section 4 the implementation of the Simulated Annealing algorithm is presented; in this section the tuning process and the variable neighborhood function are described. In section 5 the computational results are shown. And finally, in section 6 the conclusions are reported.

2. COVERING ARRAYS

The first use of combinatorial objects for testing and/or experimental design was made through the mathematical objects called Orthogonal Arrays (OA) (Hedayat 1999). An OA is defined using the notation OA λ (***N; k, v, t***): it is an array of items with values {0, 1,..., v-1}, where:

- ***N*** : Indicates the number of test cases.
- ***k*** : Corresponds to the number of parameters.
- ***v*** : Is the number of values for each parameter.
- ***t*** : Is the interaction degree, known as strength.
- λ : Is the number of times each v^t combination must appear.

Each subarray of size $N \times t$ contains all the possible v^t combinations exactly λ times.

A CA is a special case of an OA. This is represented by ***CA(N; k, v, t)*** and is a matrix of size $N \times k$ elements, in which each $N \times t$ subarray contains all the v^t combinations at least once. A ***CA(N;k,v,t)*** is optimal if the value of ***N*** is the minimum possible given ***k, v, t*** values. The determination of an optimal Covering Array ***CA(N;k,v,t)*** has been studied by many researchers (Hnich 2006, Meagher 2005, Nurmela 2004), but only in a few cases are known polynomial order algorithms, for example: the case ***CA(N;k,2,2)*** (Torres-Jimenez 2004) and the case $CA(p^{2n}, p^n + 1, p^n, 2)$, (Laywine 1999, Lidl 1997) in which the alphabet $v = p^n$ is a power of a prime number.

Table 1. Costs of an Inadequate Software Testing Infrastructure on the U.S.A. Economy

	The Cost of Inadequate Software Testing Infrastructure (billions)	Potential Cost Reduction from Feasible Infrastructure Improvements (billions)
Software developers	$21.2	$10.6
Software users	$38.3	$11.7
Total	$59.5	$22.2

Generally, constructing CA is an NP-Complete problem (Garey 1979)[1], this implies that there are not known polynomial order algorithms to solve the CA and an exhaustive algorithm would require exponential computing resources.
Nowadays, there are several algorithms published, which are useful in the generation of test cases for software. These algorithms can be classified as heuristics, metaheuristics and recursive constructions. The heuristic algorithms try to find good solutions as fast as possible, but do not guarantee to find an optimal solution. Two heuristic algorithms used to solve binary CA of variable strength are the Automatic Efficient Testcase Generator (AETG) (Cohen 1997) and the In-Parameter-Order-General (IPOG) (Lei 2007). The AETG algorithm is a commercial software, it constructs CA by adding one row to the solution until all t-tuples have been covered. The row added to the CA is the one that covers the greatest number of t-tuples from a set of candidates. The IPOG algorithm is an strategy that represents the generalization of the algorithm proposed in (Lei 1998). It generates a t-wise test set for the first t parameters. Then, it extends it for the first t+1 parameters and continues to do so for each additional parameter. The extension of a test set includes two steps: horizontal growth, which means to add one value of the new parameter; and vertical growth, which adds new tests after the completion of the horizontal growth. The most recent and available tool for constructing CA of strength $2 \leq t \leq 6$ is FireEye (Lei 2007). This tool implements five algorithms based on the IPOG strategy. These algorithms are IPOG, PaintBall, IPOG D, IPOD F, e IPOG F2.

The metaheuristic algorithms generate better CA (with less rows) but it consumes more CPU time than the heuristic algorithms. Some metaheuristics that have been used to solve the CA problem are: Simulated Annealing (SA) (Cohen 2003, 2008, Stevens 1999), Tabu Search (TS) (Nurmela 2004) and Genetic Algorithms (GA) (Stardom 2001).

[1] NP are problems for which there is not known algorithm that can solve them in polynomial time in a Deterministic Turing Machine

The algorithms that use recursive constructions generate a number of small CA and with them build CA of greater size. These algorithms have been used in (Cohen 2003, 2008, Stevens 1999) to construct CA. In the next section the SA approach used to solve the CA problem will be presented.

3. SIMULATED ANNEALING ALGORITHM

The Simulated Annealing Algorithm (SA) (Kirkipatrick 1983, van Laarhoven 1999) is based on the analogy between the simulation of the annealing of solids and the problem of solving large combinatorial optimization problems. In condensed matter physics, annealing denotes a physical process in which: firstly, a solid is heated up by increasing its temperature to a maximum value; and secondly the solid is cooled so that its particles arrange in the low energy ground state of a corresponding lattice. This low energy state is achieved once that the temperature of the solid in its liquid phase is slowly lowered.

The analogy between the optimization of a problem and the process already described can be seen in this way: the solid becomes the problem, the energy is represented by the objective function being optimized, and the low energy state is achieved through a cooling process defined by an initial and final temperatures and a factor for decreasing the initial temperature. During the optimization of a problem, the SA performs random movements over the system. When the energy of the system decreases, the movements are accepted immediately. On the other hand, when a movement produces an increase in the energy, it is accepted according to the Boltzman distribution. Following this way, while the temperatures remain high, the acceptance probability is almost 1; consequently, the probability of being caught in a local optimum or in a plateau is minimal. In this way the algorithm produces a Markov chain which approximate to a thermal equilibrium. The algorithm 3.1 shows the simulated annealing pseudocode.

Algorithm 3.1: Simulated annealing

SIMULATED ANNEALING()
u ← initial solution
T_i ← initial temperature
T_f ← final temperature
L ← length of Markov chain
α ← cooling factor

do
  for l ← to ***L***
    generate neighbor ***v*** ← ***u***
    if ***f(v)***>= ***f(u)*** or random[0,1] < $e^{-(f(v)-f(u))/v}$
      do
        then ***u*** ← ***v***
          if ***f(u)*** = 0
            then ***I*** ← ***L***
  l±+

$T_i \leftarrow T_i * \alpha$
while (***f(u)*** = 0 or $T_i = T_f$)

of the implementation of the SA for the purpose of this paper are described in the next section. Next related work that uses simulated annealing to solve the CA problem is described.

Brett Stevens (Stevens 1999) uses a SA to build small transversal covers (TC)[2]. A TC is formed by blocks, which are rows when it is seen as a CA. The algorithm initially creates a random solution for the TC instance and after that it performs a binary search for the best TC. The neighborhood function randomly chooses a block and modifies one of its elements. The evaluation function minimizes the number of uncovered pairs in the TC. This SA algorithm implementation was used to solve TC instances with alphabet $3 \leq g \leq 7$ and $k \leq 50$ parameters. Given that TC and CA of strength 2 have equivalent formulations, the results of this SA algorithm are valid for CA's with the same alphabet and number of parameters.
(Cohen et al. 2003, 2008, Stevens1999) uses SA in combination with recursive combinatorial constructions to build CA of strength $t = 3$ and homogeneous alphabets $v >= 3$. In these approaches, the initial solution is created by filling a bidimensional array ***A*** of size $N \times k$ with random chosen values of the defined alphabet ***v***. The neighborhood function randomly selects an element $a_{ij} \in A$ and modifies its value. The evaluation function minimizes the number of uncovered t-sets, i.e., the missing 3-combinations in the CA. Next section presents implementation details of our SA used to construct variable strength binary CA.

4. IMPLEMENTATION OF THE SA

The main features of the implemented SA are: a) an initialization with balanced symbols; b) a cooling schedule and neighborhood function tuned using a small set of possible choices.

4.1 Initial solution

A good CA initialization is very important. It was observed that good binary CA's have a balance in the number of different symbols that appear in the columns. In this research work a CA was randomly initialized generating an array ***M*** of size $N \times k$ where the number of symbols (zeros and ones) in each column of the array ***M*** are balanced; i.e., it has $\frac{N}{2}$ 1's and the same number of 0's in each column. For the cases in which N is odd, it can have $\left\lfloor \frac{N}{2} \right\rfloor + 1$ ones and $\left\lfloor \frac{N}{2} \right\rfloor$ zeros or vice versa.

4.2 Evaluation Function

The evaluation function used in this work is the most common function reported in the literature (Cohen 2003, 2008, Stevens 1999), i.e., the number of missing t-wise combinations. Let ***M*** represent the array in the SA of size $N \times k$; let **m** be a subarray of ***M*** of size $N \times t$; let ***r*** be an array of size **1×t** in ***m***, representing part of a row in ***M***; and let $c_m = \{r | r \in \{0, 1, ..., v-1\}^t, r \in m\}$. The evaluation function cost(***M***) can be defined as shown in the Equation 1 (duplicated ***r***'s are eliminated).

[2] TC are mathematical objects similar to CA. (Stevens 1999) claims that a TC(k,v:n) can be represented as a CA by listing the blocks of the covers one by one on top of each other as row vectors, producing a N × k arrays of v-ary values

$$\mathbf{cost(M)} = \binom{\mathbf{k}}{\mathbf{t}}\mathbf{v^t} - \sum_{\mathbf{m} \subseteq \mathbf{M}} |\mathbf{c_m}| \quad (1)$$

a	**b**	c
0	**0**	0
0	**1**	1
1	**1**	1
1	**1**	0

(a)

a	b	**c**
0	0	**0**
0	1	**1**
1	1	**1**
1	1	**0**

(b)

a	**b**	**c**
0	**0**	**0**
0	**1**	**1**
1	**1**	**1**
1	**1**	**0**

(c)

$$cost(M) = \binom{k}{t}v^t - (|\mathbf{c_{ab}}| + |\mathbf{c_{ac}}| + |\mathbf{c_{bc}}|)$$

$$cost(M) = (\mathbf{3})(\mathbf{2^2}) - (3 + 4 + \mathbf{3}) = 12 - 10 = 2$$

(d)

Fig. 1. Example of the function cost(***M***) over a matrix ***M*** of size 4 × 3.

In Figure 1 is shown an example of the evaluation function cost(***M***) over a particular array ***M*** of size 4×3. The subarrays ***m*** are shown in figures 1(a), 1(b) and 1(c) in bolted fonts. The cardinality of the sets c_m corresponding to each subarray ***m*** are: c_{ab} = {(0, 0), (0, 1), (1, 1)}, c_{ac} = {(0, 0), (0, 1), (1, 0), (1, 1)} and c_{bc} = {(0, 0), (1, 0), (1, 1)}. The result from the evaluation function is 2 shown in figure 1(d), meaning that there are only two missing combinations in the array ***M*** ((1, 0) and (0, 1). These combinations corresponds to the subarray ***m*** formed by columns a, b; and to the subarray ***m*** composed by columns b, c, respectively.

4.3 Neighborhood function

The purpose of a neighborhood function is to make small perturbations to the solution in order to explore close solutions. It has been observed that hard optimization problems can be benefitted when more than one neighborhood function are used to solve them, as reported in (Avanthay 2003, Mladenovirk 1997, Rodriguez 2008). Following this ideas, in this work five neighborhood functions for the binary CA problem are proposed. In order to describe the neighborhood functions, the following notation will be used. The array ***M*** will represent the current solution in the SA. The function ***complement(M, i, j)*** over an array ***M*** produces a new array ***M'*** whose values are exactly the same in all the elements except by m_{ij} . The element m_{ij} will change its value in ***M'***. The function ***exchange(M, i, j, k)*** creates a new array ***M'*** containing the same values that the array ***M*** but with the values of the elements $m_{ij} \in M$ and $m_{kj} \in M$ exchanged ($m_{ij} \neq m_{kj}$).

N_1(M, i, j) Random exchange over the solution array M In this neighborhood function the value of the element $m_{ij} \in M$ is changed, as suggested in the Equation 2.

$$\mathbf{N_1}(\mathbf{M}, \mathbf{i}, \mathbf{j}) = \{\mathbf{M'} | \mathbf{M'} = \mathbf{complement}(\mathbf{M}, \mathbf{i}, \mathbf{j})\} \quad (2)$$

where the values $0 \leq i \leq N - 1$ and $0 \leq j \leq k - 1$ are randomly selected. Figure 2 shows an example of the use of this neighborhood function.

0	**0**	0
0	1	1
1	0	1
1	1	0

(a)M

0	**1**	0
0	1	1
1	0	1
1	1	0

(b)M'

Fig. 2. Neighborhood function $N_1(M, i, j)$ were with values $i = 0$ and $j = 1$.

N_2(M, β) The best change of β . Given a matrix M, this neighborhood function creates a new solution by selecting the best one resulting from β evaluations of the function $N_1(M, i, j)$. The best solution will be the one that minimizes the value resulting from the evaluation function cost(***M***). This function can be formally defined as shown in Equation 3, where ***B*** is a set of β randomly chosen elements $m_{ij} \in M$. An example of this perturbation operator with $\beta = 3$ can be seen in Figure 3.

$$\mathbf{N_2\,(M,\beta) = \{M' \,|\, \min(cost(M')), M' = N_1(M,i,j), m_{ij} \in B\}} \quad (3)$$

0	**1**	0
0	1	1
1	0	1
1	1	**0**

(a) The β changes

N_1 (M,i,j) **neighbor cost**

M_{01}	1	→ 0	1
M_{20}	1	→ 0	0
M_{32}	0	→ 1	3

(b) Possible neighbors

0	**0**	0
0	1	1
1	0	1
1	1	0

(c) Neighbor $N_2(M,\beta)$

Fig. 3. Example of the use of the neighborhood function ***$N_2(M, \beta)$***. (a) elements ***m_{ij}*** to be changed; (b) information about the changes; (c) neighbor with the best evaluation.

$N_3(M, i, j, k)$ The exchange between two elements within the same column. In this perturbation, a randomly chosen element $\mathbf{m_{ij}} \in$ ***M*** exchanges its value with another randomly chosen element ***m_{kj}*** $\in$ ***M*** that has a different value. This function is shown in Equation 4. Figure 4 shows an example of the use of this neighborhood function

$$\mathbf{N_3\,(M) = \{M' | M' = exchange(M,i,j,k)\}} \quad (4)$$

0	0	**0**
0	1	1
1	0	**1**
1	1	0

(a)M

0	0	**1**
0	1	1
1	0	**0**
1	1	0

(b)M'

Fig. 4. Neighborhood function ***$N_3(M, i, j, k)$***. (a) Exchange between elements ***$m_{0,2}$*** and ***$m_{2,2}$*** of ***M***; (b) Array ***M'*** resulting from neighborhood function ***$N_3(M, 0, 2, 2)$***.

N_4(M, i, j) The best exchange between an element $m_{i,j}$ and the $\frac{N}{2}$ elements within the same column with different values. In this neighborhood function, an element ***m_{ij}*** $\in$ ***M*** is chosen randomly. After that, all the possible exchanges that can be done with the elements in the same column of ***m_{ij}*** are evaluated. Then, the exchange that minimizes the evaluation function is selected as the neighbor produced by ***$N_4(M, i, j)$***. The Equation 5 defines this neighborhood function and Figure 5 shows an example of its use.

Cost = 0

0	0	**1**
0	1	**0**
1	0	1
1	1	0

(a)

Cost=2

0	0	**1**
0	1	0
1	0	1
1	1	**0**

(b)

0	0	0
0	1	1
1	0	1
1	1	0

(c)

Fig. 5. Neighborhood function ***$N_4(M, i, j)$*** applied to element ***$m_{0,2}$***. (a) Evaluation of the function ***$N_3(M, 0, 2, 1)$***. (b) Evaluation of the function ***$N_3(0, 2, 3)$***. (c) The new neighbor produced by ***$N_4(M, 0, 2)$***.

$N_5(M, j)$ The best exchange within a column This neighborhood function is the generalization of the neighborhood function ***$N_4(M, i, j)$***. In this perturbation, a new neighbor is created by doing all the possible exchanges in a randomly chosen column ***j***. The neighbor will be the one that minimizes the evaluation function. The Equation 6 formally describes this neighborhood function. An example of the use of neighborhood function ***N_5*** is shown in Figure 6.

$$\mathbf{N_4\,(M,i,j) = \{M' \,|\, \min(cost(M')), M' = N_3(M,i,j,k), m_{ij} \neq m_{kj}\}} \quad (5)$$

$$\mathbf{N_5\,(M,j) = \{M' \,|\, \min(cost(M')), M' = N_4(M,i,j), 0 \leq i \leq\}} \quad (6)$$

0 0 **1**	0 0 **1**	0 0 1	0 0 1
0 1 **0**	0 1 0	0 1 **0**	0 1 0
1 0 1	1 0 1	1 0 **1**	1 0 **1**
1 1 0	1 1 **0**	1 1 0	1 1 **0**
(a) Cost = 0	(b) Cost = 2	(c) Cost = 2	(d)Cost = 0

Fig. 6. All the possible exchanges produced using neighborhood function ***$N_5(M, 2)$***. Figure 6(a) is the neighbor produced by ***$N_5(M, j)$***.

Tuning process for the neighborhood function. In the approach presented in this work, a SA is used to solve the CA problem. So far, five neighborhood functions for this algorithm have been described. These neighborhood functions were subject to a tuning process in order to determine the best combination of them for the SA. This process is described next.

During the tuning process, a set ***P*** = {0.1, 0.2,...,1.0} of probabilities were defined for each neighborhood function. A combination φ is a set of five values $\varphi_i \in \boldsymbol{P}$ were $\sum_{i=1}^{5} c_i = 1.0$. Each combination represents the degree of participation of a neighborhood function $\boldsymbol{N_i}$ in the SA. A total of 1001 combinations were tested during the tuning process.

The best combination φ was empirically determined by solving the benchmark composed by 10 instances: the instances {(8, 4), (12, 11), (16, 14), (18, 20), (22, 24)} were of strength t = 3 while the instances {(16, 5), (24, 12), (44, 14), (48, 16), (52, 18)} were of strength t = 4 (each instance is represented by a pair of values, the first one denoting the number of rows and the second the columns). Every instance was solved fifteen times by each of the 1001 combinations generated.

The best combination φ was $\boldsymbol{N_2(M, i, j, k) = 0.6}$ and $\boldsymbol{N_5(M,j) = 0.4}$. This means that the best combination is to use the neighborhood function $\boldsymbol{N_2}$ with a probability of 0.6 and the neighborhood function $\boldsymbol{N_5}$ with a probability of 0.4. This combination maximizes the number of times that the SA solves the proposed benchmark. In the next experiments, the SA will use exclusively these neighborhood functions.

4.4 Cooling schedule

It is well known that the performance of a SA algorithm is sensitive to parameter tuning. The cooling schedule determines the degree of uphill movement permitted during the search; this uphill movement is critical for the algorithm performance. In this section is presented the methodology used to do the tuning process of the cooling schedule. This schedule is described through an initial and final temperature, a cooling factor, a Markov chain length and a stop criterion. The stop criterion in this implementation of the SA algorithm can be achieved for any of the following reasons: a solution for the CA instance is found; or the final temperature $\boldsymbol{T_f}$ is reached. The initial and final temperatures, cooling factor and Markov chain length were subject to a tuning process. This process is based on the CA(25; 4, 5, 2) shown in the table 2.

The CA is built in order to test not the functionality of the SA but its performance subject to different configurations of its parameters. The CA then represent a set of possible configurations of the SA, where all the interactions between two of its four parameters are involved. The alphabet represents the values to be tested for each parameter; these values were selected in accordance with the common values used in the literature and small variations over them. The possible values of the Markov chain length were defined using a subset of the combinations obtained using the number of rows ***N***, the number of columns ***k***, and the size of the alphabet ***v*** of the instances been solved. In the table 3 are shown the parameters values selected for the implementation of the SA.

Table 2. Parameters of the SA and their values in the tuning process.

CA values	α	T_i	T_f	L
0	0.80	1	0.001	Nk
1	0.85	2	0.0001	Nv
2	0.90	3	0.00001	Nkv
3	0.95	4	0.000001	Kv
4	0.99	5	0.0000001	N

Table 3. CA(25; 4, 5, 2) used in the cooling schedule tuning.

Test	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
α	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	4	3	4	0	1	2	3	4
T_i	0	1	2	3	4	1	2	3	4	0	2	3	4	0	1	3	4	0	1	2	4	0	1	2	3
T_f	0	1	2	3	4	2	3	4	0	1	4	0	1	2	3	1	2	1	4	0	3	4	0	1	2
L	0	1	2	3	4	3	4	0	1	2	1	2	3	4	0	4	0	1	2	3	2	3	4	0	1

In order to evaluate the SA performance subject to the value configurations of its parameters, ten instances from the Charlie Colbourn webpage[3] were chosen. These instances are the same used in the neighborhood function tuning process described in section 4.3.

The number of configurations tested were ***vt*** = 5^2 = 25. Each instance were solved fifteen times by the SA using each of the 25 different configurations. In the table 4 are reported the results of this experiment summarized by configurations.

The 25 different configurations are listed in the table 4. In this list the configurations are sorted according to the number of instances solved. Each configuration is described using columns 2 - 5. The rest of the columns is used to describe the performance of the SA, when using that specific parameter configuration. The SA was executed 150 times per configuration, 15 times per instance. The column 6 shows the number of times that the SA found the best known solution. The column 7 shows the number of instances solved at least once by the SA. The column 8 shows the summation of the average time spent by the SA when solving each CA instance. The column 9 shows the total of the average number of evaluations of the objective function performed by the SA when solving each CA instance.

The configuration chosen for the SA was the one which solved all the benchmark instances using the less time. Then, according with the results shown in the table 4 this configuration was α ***= 0.95, T_i = 5, T_f = 0.0001*** and ***L = Nk***.

5. COMPUTATIONAL EXPERIMENTS

In this section the SA, tunned according with the results obtained in section 4, is used to solve the benchmark shown in tables 6 and 7. This benchmark consists of 57 binary CA of strengths ***t*** = 3 and ***t*** = 4 (taken from Charlie Colbourn's webpage). The results obtained by the SA proposed in this work are compared with the best reported results. The algorithm was coded in C language. It was compiled using g++ under linux operating system. The instances were solved in a PC with the following characteristics: a 2.8 Ghz Pentium IV processor and 512 RAM memory.

In particular, the SA used to solve the indicated benchmark in this section uses the neighborhood function ***N_2(M, i, j, k)*** and ***N_5(M,j)***. The parameters of the SA were: an initial temperature ***T_i*** = 5, a final temperature ***T_f*** = 0.0001, a cooling factor α = 0.95 and a Markov chain length ***L = Nk***. The comparison of the results of the SA with the best known solutions reported in the literature are shown in tables 6

[3] http://www.public.asu.edu/~ccolbou/src/tabby/3-2-ca.html y /4-2-ca.html

and 7. The nomenclature used in these tables is described in the table 5.

Table 4. Result from de cooling schedule tunning.

Number of	Configuration				Instance		Total	Total
Test	α	T_i	T_f	L	Global	Solved	Time	Evaluation
23	0.95	3	0.0010000	Nkv	127	10	2974.68	35410.47
8	**0.95**	**5**	**0.0001000**	**Nk**	**122**	**10**	**2594.44**	**19113.51**
17	0.99	1	0.0001000	Nkv	118	10	3085.43	33502.85
10	0.80	3	0.0001000	Nv	114	10	2963.67	28537.73
11	0.85	4	0.0000100	Nk	113	10	3040.52	31345.39
22	0.90	2	0.0000001	Nk	112	10	2992.09	34694.48
18	0.95	2	0.0000100	Nv	128	9	3097.50	35434.56
9	0.99	1	0.0000100	Nkv	119	9	2927.95	23935.81
15	0.80	4	0.0000001	Nkv	115	9	3126.02	33265.54
1	0.85	2	0.0001000	Nkv	114	9	2411.33	29873.11
12	0.90	5	0.0000010	Nkv	113	9	3123.56	33253.46
19	0.99	3	0.0000010	Nk	110	9	3147.90	36904.99
7	0.90	4	0.0010000	Nv	108	8	2570.75	18083.17
0	0.80	1	0.0010000	Nk	103	8	925.32	13641.87
6	0.85	3	0.0000001	N	103	8	2663.19	19456.21
3	0.95	4	0.0000010	N	103	8	2553.91	26220.46
14	0.99	2	0.0010000	N	100	8	3178.50	33650.70
20	0.80	5	0.0000100	N	96	8	3157.03	36741.44
4	0.99	5	0.0000001	Nv	86	8	2666.05	23421.53
16	0.85	5	0.0010000	kv	85	8	3171.75	33914.37
2	0.90	3	0.0000100	kv	84	8	2791.27	32921.62
5	0.80	2	0.0000010	kv	79	8	2811.06	21608.99
24	0.99	4	0.0001000	kv	71	8	2993.57	36379.78
21	0.85	1	0.0000010	Nv	89	7	3067.42	35459.43
13	0.95	1	0.0000001	kv	85	7	3318.11	35496.14

Table 5. Nomenclature used in tables 6 and 7.

OTAR	Orthogonal Array (Hedayat 1999)
JOEN	Johnson-Entringer
SLOA	Sloane (Sloane 1993)
KANU	Kari Nurmela (Nurmela 2004)
CHKR	Chateauneuf-Kreher (Kreher 2002)
YANJ	Yan Jun
SORI	Soriano
KULI	Kuliamin
CMTW	Colbourn-Martirosyan-Trung-Walker

Tables 6, 7 are organized as follows: in the column 1 is shown the number of columns involved in each CA instances solved. The column 2 shows the best reported number of rows found for each instance. Column 3 names the author of the results shown in column 2, if it exists. In the next five columns are shown the results obtained by FireEye 1.0 (the most recent approach created to solve binary CA instances of variable strength). In the column 9 are reported the results obtained by the SA implemented in this work.

6. CONCLUSIONS

In this paper was presented a SA approach to construct binary CA of strength $t = 3$ and $t = 4$. The SA algorithm was improved by using an initial solution with balanced number of symbols per column, a variable neighborhood function and a tuned cooling schedule (composed by initial temperature, final temperature, cooling factor and Markov chain length).

Five neighborhood functions were implemented and experimentally tunned to determine its participation in the SA approach. The neighbor_hood functions $N_2(M, i, j, k)$ with value 0.6 and $N_5(M, j)$ with value 0.4 formed the combination with the best performance for the SA.

The cooling schedule of the SA was tunned using a CA of strength $t = 2$ and alphabet $v = 5$. The CA gave configurations for the cooling process that covers all the interactions between pairs of the parameters initial and final temperatures, cooling factor, and Markov chain. The SA approach proposed got the best reported results in the literature for the binary CA cases of strength $t = 3$ and $t = 4$ provided in the webpage of Charlie Colbourn.

Acknowledgements This research was partially funded by the following projects: CONACyT 58554-Calculo de Covering Arrays, 51623-Fondo Mixto CONACyT y Gobierno del Estado de Tamaulipas, CONACyT 74521 Repatriación.

Table 6. Results obtained by the SA when solving ***CA(N; k, 2, 3)*** instances.

K	N	Best reported	IPOG	Paint Ball	IPOG D	IPOG F	IPOG F2	Simulated Annealing
4	8	OTAR	8	10	8	8	9	8
5	10	JOEN	12	12	12	13	12	10
6	12	SLOA	12	14	12	13	13	12
7	12	SLOA	15	16	14	12	15	12

8	12	SLOA	18	18	14	16	16	12
9	12	SLOA	20	18	18	17	17	12
10	12	SLOA	20	18	18	17	18	12
11	12	SLOA	22	20	20	19	21	12
12	15	KANU	22	22	20	19	19	15
13	16	SLOA	22	24	24	20	19	16
14	16	SLOA	24	24	24	23	20	16
15	17	SLOA	24	26	26	20	23	17
16	17	SLOA	25	26	26	21	21	17
17	18	CHKR	26	26	28	22	23	18
18	18	CHKR	26	30	28	24	24	18
19	18	CHKR	27	30	30	24	25	18
20	18	CHKR	27	30	30	25	24	18
21	19	CHKR	29	32	32	26	25	19
22	19	CHKR	29	32	32	26	27	19
23	22	CHKR	29	34	32	27	27	22
24	22	CHKR	29	32	32	26	27	22
25	23	CHKR	30	34	32	28	28	23
26	23	CHKR	31	36	32	27	29	23
27	23	CHKR	32	36	34	30	29	23
28	23	CHKR	32	36	34	28	31	23
29	25	CHKR	32	36	34	29	31	25
30	25	CHKR	32	38	34	30	30	25
31	25	CHKR	34	38	35	30	31	25
32	25	CHKR	34	40	35	30	31	25

Table 7. Results obtained by the SA when solving ***CA(N; k, 2, 4)*** instances.

K	N	Best reported	IPOG	Paint Ball	IPOG D	IPOG F	IPOG F2	Simulated Annealing
5	16	OTAR	16	17	34	17	20	16
6	21	JOEN	28	29	44	27	26	21
7	24	YANJ	35	32	64	33	31	24
8	24	YANJ	38	34	64	33	35	24
9	24	YANJ	41	38	80	40	36	24
10	24	YANJ	42	42	80	41	41	24
11	24	YANJ	46	47	104	43	44	24
12	24	YANJ	48	50	104	48	47	24
13	34	SORI	51	54	127	49	50	34
14	44	KULI	52	54	127	49	52	44
15	46	KULI	58	59	54	54	54	46
16	48	KULI	58	65	144	56	56	48
17	49	KULI	61	69	161	57	59	49
18	52	KULI	64	69	161	61	61	52
19	55	CMTW	65	71	174	62	64	55
20	55	CMTW	66	72	174	64	64	55
21	57	CMTW	68	78	192	66	68	57
22	57	CMTW	69	80	192	68	69	57
23	61	CMTW	70	81	206	71	71	61
24	61	KULI	72	84	206	71	73	61
25	65	CMTW	74	87	221	74	72	65

26	72	KULI	75	89	221	76	77	72
27	73	KULI	76	90	238	75	75	73
28	73	KULI	76	91	238	76	77	73
29	73	KULI	79	93	254	78	79	73
30	73	KULI	81	97	254	79	79	73
31	73	KULI	81	99	267	80	81	73
32	73	KULI	82	100	267	80	84	73

7. LITERATURE CITED

Avanthay, C., A. Hertz, & N. Zufferey. 2003. A variable neighborhood search for graph coloring. European Journal of Operational Research. 151(2):379–388.

Cohen, D. M. , S. R. Dalal, M. L. Fredman, & G. C. Patton. 1997. The AETG system: An approach to testing based on combinatiorial design. Software Engineering. 23(7):437–444

Cohen, M.B., C.J. Colbourn, & A.C.H. Ling. 2003. Augmenting simulated annealing to build interaction test suites. Software Reliability Engineering, 2003. 14th International Symposium on, pages 394–405.

Cohen, M.B., C.J. Colbourn, & A.C.H. Ling. 2008. Constructing strength three covering arrays with augmented annealing. Discrete Mathematics, Combinatorial Designs. 308(13):2709–2722.

Cohen, M.B., P.B. Gibbons, W.B. Mugridge, C.J. Colbourn & J.S. Collofello. 2003. A variable strength interaction testing of components. Computer Software and Applications Conference, 2003. Proceedings. 27th Annual International, pages 413–418.

Garey, M. R. & D. S. Johnson. 1979. Computers and Intractability A Guide to the Theory of NP-Completeness. Bell Laboratories. 1-38 pp. ISBN 0-7167-1044-7.

Hedayat, A.S., N.J.S. Sloane & J. Stufken. 1999. Orthogonal Arrays Theory and Applications. Springer Series and Statistics, 1999. ISBN 0387-98766-5.

Hnich, B., S. Prestwich, E. Selensky & B. M. Smith. 2006. Constraints models for the covering test problem. 11:199-219.

Kirkpatrick, S., C. D. Gelatt, & M. P. Vecchi. 1983. Optimization by simulated annealing. Science 220(4598): 671–680.

Kreher, D.L. & M. Chateauneuf. 2002. On the state of strength-three covering arrays. Journal of Combinatorial Designs 10(4): 217–238.

Laywine, Ch. F. & G. L. Mullen. 1998. Discrete Mathematics Using Latin Squares. Wiley Interscience Series in Discrete Mathematics and Optimization. 3-39 pp. ISBN 0471-24064-8.

Lei, Y. & K. C. Tai. 1998. In-parameter-order: A test generation strategy for pairwise testing. Third IEEE International High-Assurance Systems Engineering Symposium: 254–261.

Lei, Yu, R. Kacker, D. R. Kuhn, V. Okun & J. Lawrence. 2007. Ipog: A general strategy for t-way software testing. Engineering of Computer-Based Systems. 14th Annual IEEE International Conference and Workshops: 549–556.

Lidl, R. & H. Niederreiter. 1997. Finite Fields. Cambridge University. 83-100. Press. ISBN 0521-39231-4.

Meagher, K. & B. Stevens. 2005. Covering arrays on graphs. Journal of Combinatorial Theory, 95(1):134 – 151.

Mladenovirc, N. & E Hansen. 1997. Variable neighborhood search. Computers and Operations Research, 24(11):1097–1100.

Nurmela, K.J. 2004. Upper bounds for covering arrays by tabu search. Discrete Applied Mathematics, 138: 143-152.

Rodriguez-Tello, E., J. Kao Hao, and J. Torres-Jimenez. 2008. An effective two-stage simmulated annealing algorithm for the minimum linear arrangement problem. Computers and Operations Research. 35 (10): 3331-3310.

Sloane, N. J. A. 1993. Covering arrays and intersecting codes. Journal of Combinatorial Designs 1:51–63.

Stardom, Jhon. 2001. Metaheuristics and the Search for Covering and Packing Arrays. PhD thesis, Trent University.

Stevens, B. & E. Mendelsohn. 1999. New recursive methods for transversal covers. Journal of Combinatorial Designs. John Wiley Sons, Inc., 7(3):185–203.

Tassey, G. 2002. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology. http://www.nist.gov/public affairs/releases/n02-10.htm.

Torres-Jiménez, J., C. De Alfonso & V. Hernández. 2004. Computation of ternary covering arrays using a grid. Springer-Verlag Berlin Heidelberg, LNCS 3285:240-246.

Van Laarhoven, P. J. M. & E. H. L. Arts 1992.. Simulated Annealing: Theory and Applications. Editor's Preface. Philips Research Laboratories, 88 pp. Eindhoven, The Netherlands. ISBN 90-277-2513-6 7.