10

Using Grid Computing for Constructing Ternary Covering Arrays

Himer Avila-George[1], Jose Torres-Jimenez[2],
Abel Carrión[1] and Vicente Hernández[1]
[1]*Instituto de Instrumentación para Imagen Molecular (I3M), Centro Mixto CSIC - Universitat Politècnica de València - CIEMAT, Valencia*
[2]*CINVESTAV-Tamaulipas, Information Technology Laboratory, Km. 5.5 Carretera Victoria-Soto La Marina, Ciudad Victoria, Tamaulipas*
[1]*Spain*
[2]*Mexico*

1. Introduction

To continue at the forefront in this fast paced and competitive world, companies have to be highly adaptable and to suit such transforming needs customized software solutions play a key role. To support this customization, software systems must provide numerous configurable options. While this flexibility promotes customizations, it creates many potential system configurations, which may need extensive quality assurance.

A good strategy to test a software component involves the generation of the whole set of cases that participate in its operation. While testing only individual values may not be enough, exhaustive testing of all possible combinations is not always feasible. An alternative technique to accomplish this goal is called combinatorial testing. Combinatorial testing is a method that can reduce cost and increase the effectiveness of software testing for many applications. 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 (CA) is a combinatorial object, denoted by $CA(N;t,k,v)$ which can be described like a matrix with $N \times k$ elements, such that every $N \times t$ subarray contains all possible combinations of v^t symbols at least once. N represents the rows of the matrix, k is the number of parameters, which has v possible values and t represents the strength or the degree of controlled interaction.

To illustrate the CA approach applied to the design of software testing, consider the Web-based system example shown in Table 1, the example involves four parameters each with three possible values. A full experimental design ($t = 4$) should cover $3^4 = 81$ possibilities, however, if the interaction is relaxed to $t = 2$ (pair-wise), then the number of possible combinations is reduced to 9 test cases.

	Browser	OS	DBMS	Connections
0	Firefox	Windows 7	MySQL	ISDN
1	Chromium	Ubuntu 10.10	PostgreSQL	ADSL
2	Netscape	Red Hat 5	MaxDB	Cable

Table 1. Parameters of Web-based system example.

Fig. 1 shows the CA corresponding to $CA(9;2,4,3)$; given that its strength and alphabet are $t = 2$ and $v = 3$, respectively, the combinations that must appear at least once in each subset of size $N \times 2$ are $\{0,0\}$, $\{0,1\}$, $\{0,2\}$, $\{1,0\}$, $\{1,1\}$, $\{1,2\}$, $\{2,0\}$, $\{2,1\}$, $\{2,2\}$.

$$\begin{pmatrix} 0&0&0&0 \\ 0&1&1&1 \\ 0&2&2&2 \\ 1&0&1&2 \\ 1&1&2&0 \\ 1&2&0&1 \\ 2&0&2&1 \\ 2&1&0&2 \\ 2&2&1&0 \end{pmatrix}$$

Fig. 1. A combinatorial design, $CA(9;2,4,3)$.

Finally, to make the mapping between the CA and the Web-based system, every possible value of each parameter in Table 1 is labeled by the row number. Table 2 shows the corresponding pair-wise test suite; each of its nine experiments is analogous to one row of the CA shown in Fig. 1.

	Experiments			
1	Firefox	Windows 7	MySQL	ISDN
2	Firefox	Ubuntu 10.10	PostgreSQL	ADSL
3	Firefox	Red Hat 5	MaxDB	Cable
4	Chromium	Windows 7	PostgreSQL	Cable
5	Chromium	Ubuntu 10.10	MaxDB	ISDN
6	Chromium	Red Hat 5	MySQL	ADSL
7	Netscape	Windows 7	MaxDB	ADSL
8	Netscape	Ubuntu 10.10	MySQL	Cable
9	Netscape	Red Hat 5	PostgreSQL	ISDN

Table 2. Test-suite covering all 2-way interactions, $CA(9;2,4,3)$.

When a CA contains the minimum possible number of rows, it is optimal and its size is called the *Covering Array Number* (CAN). The CAN is defined according to

$$CAN(t,k,v) = \min_{N \in \mathbb{N}} \{N : \exists\, CA(N;t,k,v)\}.$$

The trivial mathematical *lower bound* for a covering array is $v^t \leq CAN(t,k,v)$, however, this number is rarely achieved. Therefore determining achievable bounds is one of the main research lines for CAs. Given the values of t, k, and v, the optimal CA construction problem (CAC) consists in constructing a $CA(N;t,k,v)$ such that the value of N is minimized.

The construction of $CAN(2,k,2)$ can be efficiently done according with Kleitman & Spencer (1973); the same is possible for $CA(2,k,v)$ when the cardinality of the alphabet is $v = p^n$, where p is a prime number and n a positive integer value (Bush, 1952). However, in the general case determining the *covering array number* is known to be a hard combinatorial problem (Colbourn, 2004; Lei & Tai, 1998). This means that there is no known efficient algorithm to find an optimal CA for any level of interaction t or alphabet v. For the values of t and v that no efficient algorithm is known, we use approximated algorithms to construct them. Some of these approximated strategies must verify that the matrix they are building is a CA. If the matrix is of size $N \times k$ and the interaction is t, there are $\binom{k}{t}$ different combinations which implies a cost of $O(N \times \binom{k}{t})$ for the verification (when the matrix has $N \geq v^t$ rows, otherwise it will never be a CA and its verification is pointless). For small values of t and v the verification of CAs is overcame through the use of sequential approaches; however, when we try to construct CAs of moderate values of t, v and k, the time spent by those approaches is impractical. This scenario shows the necessity of Grid strategies to construct and verify CAs.

Grid Computing is a technology which allows sharing resources between different administration domains, in a transparent, efficient and secure way. The resources comprise: computation hardware (supercomputers or clusters) or storage systems, although it is also possible to share information sources, such as databases or scientific equipment. So, the main concept behind the Grid paradigm is to offer a homogeneous and standard interface for accessing these resources. In that sense, the evolution of Grid Middlewares has enabled the deployment of Grid e-Science infrastructures delivering large computational and data storage capabilities. The current infrastructures rely on Globus Toolkit (Globus Alliance, 2011), UNICORE (Almond & Snelling, 1999), GRIA (Surridge et al., 2005) or gLite (gLite, 2011) mainly as core middleware supporting several central services dedicated to: user management, job metascheduling, data indexing (cataloguing) and information system, providing consolidated virtual view of the whole or larger parts of the infrastructure. The availability of hundreds and thousands of processing elements (PEs) and the efficient storage of Petabyes of data is expanding the knowledge on areas such as particle physics, astronomy, genetics or software testing. Thus, Grid Computing infrastructures are the cornerstone in the current scientific research.

In this work is reported the use of Grid Computing by means of the use of the European production infrastructure provided by the European Grid Infrastructure (EGI) (EGI, 2011) project. The availability of this kind of computing platforms makes feasible the execution of computing-intensive applications, such as the construction and verification of CAs. In this work we focus on the construction of ternary CAs when $5 \leq k \leq 100$ and $2 \leq t \leq 4$.

The chapter is structured as follows. First, Section 2 offers a review of the relevant related work. Then, the algorithm for the verification of CAs is exposed in Section 3. Moreover, Section 4 details the algorithm for the construction of CAs by using a simulated annealing algorithm. Next, the Section 5 explains how to parallelize the previous algorithm using a master-slave approach. Taking the previous parallelization, Section 6 describes how to develop a Grid implementation of the construction of CAs. The results obtained in the

experiments performed in the Grid infrastructure are showed in Section 7. Finally, Section 8 presents the conclusions derived from the research presented in this work.

2. Relevant related work

Because of the importance of the construction of (near) optimal CAs, much research has been carried out in developing effective methods for construct them. There are several reported methods for constructing these combinatorial models. Among them are: (a) direct methods, (b) recursive methods, (c) greedy methods, and d) meta-heuristics methods. In this section we describe the relevant related work to the construction of CAs.

Direct methods construct CAs in polynomial time and some of them employ graph or algebraic properties. There exist only some special cases where it is possible to find the covering array number using polynomial order algorithms. Bush (1952) reported a direct method for constructing optimal CAs that uses Galois finite fields obtaining all $CA(q^t; t, q+1, q)$ where q is a prime or a prime power and $q \leq t$. Rényi (1971) determined sizes of CAs for the case $t = v = 2$ when N is even. Kleitman & Spencer (1973) and Katona (1973) independently determined covering array numbers for all N when $t = v = 2$. Williams & Probert (1996) proposed a method for constructing CAs based on algebraic methods and combinatorial theory. Sherwood (2008) described some algebraic constructions for strength-2 CAs developed from index-1 orthogonal arrays, ordered designs and CAs. Another direct method that can construct some optimal CAs is named zero-sum (Sherwood, 2011). Zero-sum leads to $CA(v^t; t, t+1, v)$ for any $t > 2$; note that the value of degree is in function of the value of strength. Recently, cyclotomic classes based on Galois finite fields have been shown to provide examples of binary CAs, and more generally examples are provided by certain Hadamard matrices (Colbourn & Kéri, 2009).

Recursive methods build larger CAs from smaller ones. Williams (2000) presented a tool called TConfig to construct CAs. TConfig constructs CAs using recursive functions that concatenate small CAs to create CAs with a larger number of columns. Moura et al. (2003) introduced a set of recursive algorithms for constructing CAs based on CAs of small sizes. Some recursive methods are product constructions (Colbourn & Ling, 2009; Colbourn et al., 2006; Martirosyan & Colbourn, 2005). Colbourn & Torres-Jimenez (2010) presented a recursive method to construct CAs using *perfect hash families* for CAs contruction. The advantage of the recursive algorithms is that they construct almost minimal arrays for particular cases in a reasonable time. Their basic disadvantage is a narrow application domain and impossibility of specifying constraints.

The majority of commercial and open source test data generating tools use greedy algorithms for CAs construction (AETG (Cohen et al., 1996), TCG (Tung & Aldiwan, 2000), IPOG (Lei et al., 2007), DDA (Bryce & Colbourn, 2007) and All-Pairs (McDowell, 2011)). AETG popularized greedy methods that generate one row of a covering array at a time, attempting to select the best possible next row; since that time, TCG and DDA algorithms have developed useful variants of this approach. IPOG instead adds a factor (column) at a time, adding rows as needed to ensure coverage. The greedy algorithms provide the fastest solving method.

A few Grid approaches has been found in the literature. Torres-Jimenez et al. (2004) reported a mutation-selection algorithm over Grid Computing, for constructing ternary CAs. Younis et al. (2008) presented a Grid implementation of the modified IPOG algorithm (MIPOG).

Calvagna et al. (2009) proposed a solution for executing the reduction algorithm over a set of Grid resources.

Metaheuristic algorithms are capable of solving a wide range of combinatorial problems effectively, using generalized heuristics which can be tailored to suit the problem at hand. Heuristic search algorithms try to solve an optimization problem by the use of heuristics. A heuristic search is a method of performing a minor modification of a given solution in order to obtain a different solution.

Some metaheuristic algorithms, such as TS (Tabu Search) (Gonzalez-Hernandez et al., 2010; Nurmela, 2004), SA (Simulated Annealing) (Cohen et al., 2003; Martinez-Pena et al., 2010; Torres-Jimenez & Rodriguez-Tello, 2012), GA (Generic Algorithm) and ACA (Ant Colony Optimization Algorithm) (Shiba et al., 2004) provide an effective way to find approximate solutions. Indeed, a SA metaheuristic has been applied by Cohen et al. (2003) for constructing CAs. Their SA implementation starts with a randomly generated initial solution M which cost $E(M)$ 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 M' is generated by changing the value of the element $a_{i,j}$ by a different legal member of the alphabet in the current solution M. The cost of this iteration is evaluated as $\Delta E = E(M') - E(M)$. If ΔE is negative or equal to zero, then the neighboring solution M' is accepted. Otherwise, it is accepted with probability $P(\Delta E) = e^{-\Delta E/T_n}$, 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 $t = 3$.

Some of these approximated strategies must verify that the matrix they are building is a CA. If the matrix is of size $N \times k$ and the interaction is t, there are $\binom{k}{t}$ different combinations which implies a cost of $O(N \times \binom{k}{t})$ (given that the verification cost per combination is $O(N)$). For small values of t and v the verification of CAs is overcame through the use of sequential approaches; however, when we try to construct CAs of moderate values of t, v and k, the time spent by those approaches is impractical, for example when $t = 5, k = 256, v = 2$ there are $8,809,549,056$ different combinations of columns which require days for their verification. This scenario shows the necessity of grid strategies to solve the verification of CAs.

The next section presents an algorithm for the verification of a given matrix is a CA. The design of algorithm is presented for its implementation in grid architectures.

3. An algorithm for the verification of covering arrays

In this section we describe a grid approach for the problem of verification of CAs. See (Avila-George et al., 2010) for more details.

A matrix $\mathcal{M}$ of size $N \times k$ is a $CA(N; t, k, v)$ iff every t-tuple contains the set of combination of symbols described by $\{0, 1, ..., v-1\}^t$. We propose a strategy that uses two data structures

called P and J, and two injections between the sets of t-tuples and combinations of symbols, and the set of integer numbers, to verify that $\mathcal{M}$ is a CA.

Let $\mathcal{C} = \{c_1, c_2, ..., c_{\binom{k}{t}}\}$ be the set of the different t-tuples. A t-tuple $c_i = \{c_{i,1}, c_{i,2}, ..., c_{i,t}\}$ is formed by t numbers, each number $c_{i,1}$ denotes a column of matrix $\mathcal{M}$. The set $\mathcal{C}$ can be managed using an injective function $f(c_i) : \mathcal{C} \rightarrow \mathcal{I}$ between $\mathcal{C}$ and the integer numbers, this function is defined in Eq. 1.

$$f(c_i) = \sum_{j=1}^{t} \binom{c_{i,j} - 1}{i + 1} \tag{1}$$

Now, let $\mathcal{W} = \{w_1, w_2, ..., w_{v^t}\}$ be the set of the different combination of symbols, where $w_i \in \{0, 1, ..., v-1\}^t$. The injective function $g(w_i) : \mathcal{W} \rightarrow \mathcal{I}$ is defined as done in Eq. 2. The function $g(w_i)$ is equivalent to the transformation of a v-ary number to the decimal system.

$$g(w_i) = \sum_{j=1}^{t} w_{i,j} \cdot v^{t-i} \tag{2}$$

The use of the injections represents an efficient method to manipulate the information that will be stored in the data structures P and J used in the verification process of $\mathcal{M}$ as a CA. The matrix P is of size $\binom{k}{t} \times v^t$ and it counts the number of times that each combination appears in $\mathcal{M}$ in the different t-tuples. Each row of P represents a different t-tuple, while each column contains a different combination of symbols. The management of the cells $p_{i,j} \in P$ is done through the functions $f(c_i)$ and $g(w_j)$; while $f(c_i)$ retrieves the row related with the t-tuple c_i, the function $g(w_i)$ returns the column that corresponds to the combination of symbols w_i. The vector J is of size t and it helps in the enumeration of all the t-tuples $c_i \in \mathcal{C}$.

Table 3 shows an example of the use of the function $g(w_j)$ for the Covering Array $CA(9; 2, 4, 3)$ (shown in Fig. 1). Column 1 shows the different combination of symbols. Column 2 contains the operation from which the equivalence is derived. Column 3 presents the integer number associated with that combination.

$\mathcal{W}$	$g(w_i)$	$\mathcal{I}$
w_1 ={0,0}	$0 \cdot 3^1 + 0 \cdot 3^0$	0
w_2 ={0,1}	$0 \cdot 3^1 + 1 \cdot 3^0$	1
w_3 ={0,2}	$0 \cdot 3^1 + 2 \cdot 3^0$	2
w_4 ={1,0}	$1 \cdot 3^1 + 0 \cdot 3^0$	3
w_5 ={1,1}	$1 \cdot 3^1 + 1 \cdot 3^0$	4
w_6 ={1,2}	$1 \cdot 3^1 + 2 \cdot 3^0$	5
w_7 ={2,0}	$2 \cdot 3^1 + 0 \cdot 3^0$	6
w_8 ={2,1}	$2 \cdot 3^1 + 1 \cdot 3^0$	7
w_9 ={2,2}	$2 \cdot 3^1 + 2 \cdot 3^0$	8

Table 3. Mapping of the set $\mathcal{W}$ to the set of integers using the function $g(w_j)$ in $CA(9; 2, 4, 3)$ shown in Fig. 1.

The matrix P is initialized to zero. The construction of matrix P is direct from the definitions of $f(c_i)$ and $g(w_j)$; it counts the number of times that a combination of symbols $w_j \in \mathcal{W}$ appears

in each subset of columns corresponding to a t-tuple c_i, and increases the value of the cell $p_{f(c_i),g(w_j)} \in P$ in that number.

Table 4(a) shows the use of injective function $f(c_i)$. Table 4(b) presents the matrix P of $CA(9;2,4,3)$. The different combination of symbols $w_j \in \mathcal{W}$ are in the first rows. The number appearing in each cell referenced by a pair (c_i, w_j) is the number of times that combination w_j appears in the set of columns c_i of the matrix $CA(9;2,4,3)$.

(a) Applying $f(c_i)$.

c_i		
index	t-tuple	$f(c_i)$
c_1	{1,2}	0
c_2	{1,3}	1
c_3	{1,4}	3
c_4	{2,3}	2
c_5	{2,4}	4
c_6	{3,4}	5

(b) Matrix P.

	$g(w_j)$								
$f(c_i)$	{0,0}	{0,1}	{0,2}	{1,0}	{1,1}	{1,2}	{2,0}	{2,1}	{2,2}
0	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1

Table 4. Example of the matrix P resulting from $CA(9;2,4,3)$ presented in Fig. 1.

In summary, to determine if a matrix $\mathcal{M}$ is or not a CA the number of different combination of symbols per t-tuple is counted using the matrix P. The matrix $\mathcal{M}$ will be a CA iff the matrix P contains no zero in it.

The grid approach takes as input a matrix $\mathcal{M}$ and the parameters N, k, v, t that describe the CA that $\mathcal{M}$ can be. Also, the algorithm requires the sets $\mathcal{C}$ and $\mathcal{W}$. The algorithm outputs the total number of missing combinations in the matrix $\mathcal{M}$ to be a CA. The Algorithm 1 shows the pseudocode of the grid approach for the problem of verification of CAs; particularly, the algorithm shows the process performed by each core involved in the verification of CAs. The strategy followed by the algorithm 1 is simple, it involves a block distribution model of the set of t-tuples. The set $\mathcal{C}$ is divided into n blocks, where n is the processors number; the size of block $\mathcal{B}$ is equal to $\lceil \frac{\mathcal{C}}{n} \rceil$. The block distribution model maintains the simplicity in the code; this model allows the assignment of each block to a different core such that SA can be applied to verify the blocks.

Algorithm 1: Grid approach to verify CAs. This algorithm assigns the set of t-tuples $\mathcal{C}$ to *size* different cores.

Input: A covering array file, the number of processors (*size*) and the current processor id (*rank*).
Result: A file with the number of missing combination of symbols.

```
1 begin
2     readInputFile()                          /* Read M, N, k, v and t parameters.  */
3     B ← ⌈ (k choose t) / size ⌉              /* t-tuples per processor.            */
4     K_l ← rank · B          /* The scalar corresponding to the first t-tuple.      */
5     K_u ← (rank + 1) · B − 1  /* The scalar corresponding to the last t-tuple.     */
6     Miss ← t_wise(M, N, k, v, t, K_l, K_u)   /* Number of missing t-tuples.        */
7 end
```

The *t_wise* function first counts for each different t-tuple c_i the times that a combination $w_j \in \mathcal{W}$ is found in the columns of $\mathcal{M}$ corresponding to c_i. After that, it calculates the missing combinations $w_j \in \mathcal{W}$ in c_i. Finally, it transforms c_i into c_{i+1}, i.e. it determines the next t-tuple to be evaluated.

The pseudocode for *t_wise* function is presented in Algorithm 2. For each different t-tuple (lines 5 to 28) the function performs the following actions: counts the expected number of times a combination w_j appears in the set of columns indicated by J (lines 6 to 14, where the combination w_j is the one appearing in $\mathcal{M}_{n,J}$, i.e. in row n and t-tuple J); then, the counter *covered* is increased in the number of different combinations with a number of repetitions greater than zero (lines 10 to 12). After that, the function calculates the number of missing combinations (line 15). The last step of each iteration of the function is the calculation of the next t-tuple to be analyzed (lines 16 to 27). The function ends when all the t-tuples have been analyzed (line 5).

Algorithm 2: Function to verify a CA.

```
   Output: Number of missing t-tuples.
1  t_wise(M, N, k, v, t, K_l, K_u)
2  begin
3      Miss ← 0, iMax ← t − 1, P ← 0
4      J ← getInitialTuple(k, t, K_l)
5      while J_t ≤ k and f(J) ≤ K_u do
6          covered ← 0
7          for n ← 1 to N do
8              P_{f(J),g(M_{n,J})} ← P_{f(J),g(M_{n,J})} + 1
9          end for
10         for j ← 1 to v^t do
11             if P_{f(J),j} > 0 then
12                 covered ← covered + 1
13             end if
14         end for
15         Miss ← Miss + v^t − covered
           /* Calculates the next t-tuple                                  */
16         J_t ← J_t + 1
17         if J_t > k and iMax > 0 then
18             J_{iMax} ← J_{iMax} + 1
19             for i ← iMax + 1 to t do
20                 J_i ← J_{i−1} + 1
21             end for
22             if J_{iMax} > k − t + iMax then
23                 iMax ← iMax − 1
24             else
25                 iMax ← t
26             end if
27         end if
28     end while
29     return Miss
30 end
```

To make the distribution of work, it is necessary to calculate the initial t-tuple f for each core according to its ID (denoted by *rank*), where $F = rank \cdot \mathcal{B}$. Therefore it is necessary a method to

convert the scalar F to the equivalent t-tuple $c_i \in \mathcal{C}$. The sequential generation of each t-tuple c_i previous to c_F can be a time consuming task. There is where lies the main contribution of our grid approach; its simplicity is combined with a clever strategy for computing the initial t-tuple of each block.

We propose the *getInitialTuple* function as a method that generates c_F (see Algorithm 3), according to a lexicographical, without generating its previous t-tuples c_i, where $i < F$. To explain the purpose of the *getInitialTuple* function, lets consider the $CA(9;2,4,3)$ shown in Fig. 1. This CA has as set $\mathcal{C}$ the elements found in column 1 of Table 4(a). The *getInitialTuple* function with input $k = 4,\ t = 2,\ F = 3$ must return $J = \{1,4\}$, i.e. the values of the t-tuple c_3. The *getInitialTuple* function is optimized to find the vector $J = \{J_1, J_2, ..., J_t\}$ that corresponds to F. The value J_i is calculated according to

$$J_i = \min_{j \geq 1} \left\{ \Delta_i \leq \sum_{l=J_{i-1}+1}^{j} \binom{k-l}{t-i} \right\}$$

where

$$\Delta_i = F - \sum_{m=1}^{i-1} \sum_{l=J_{m-1}+1}^{J_m-1} \binom{k-l}{t-m}$$

and

$$J_0 = 0.$$

Algorithm 3: Get initial t-tuple to PA.

Input: Parameters k and t; the scalar corresponding to the first t-tuple ($\mathcal{K}_l$).
Output: The initial t-tuple.

```
1  getInitialTuple ( k, t, K_l )
2  begin
3      Θ ← K_l, iK ← 1, iT ← 1
4      kint ← C(k−iK, t−iT)
5      for i ← 1 to t do
6          while Θ > kint do
7              Θ ← Θ − kint
8              iK ← iK + 1
9              kint ← C(k−iK, t−iT)
10         end while
11         J_i ← iK
12         iK ← iK + 1
13         iT ← iT + 1
14         kint ← C(k−iK, t−iT)
15     end for
16     return J
17 end
```

In summary, the Algorithm 3 only requires the computation of $O(t \times k)$ binomials to compute the n initial t-tuples of the PA. This represents a great improvement in contrast with the naive approach that would require the generation of all the $\binom{k}{t}$ t-tuples, as done in the SA.

The next three sections presents a simulated annealing approach to construct CAs. Section 4 describes in depth the components of our algorithm. Section 5 presents a method to

parallelizing our SA algorithm. Section 6 describes how to implement our algorithm on a grid architecture.

4. An algorithm for the construction of covering arrays using a simulated annealing technique

Often the solution space of an optimization problem has many local minima. A simple local search algorithm proceeds by choosing random initial solution and generating a neighbor from that solution. The neighboring solution is accepted if it is a cost decreasing transition. Such a simple algorithm has the drawback of often converging to a local minimum. The simulated annealing algorithm (SA), though by itself it is a local search algorithm, avoids getting trapped in a local minimum by also accepting cost increasing neighbors with some probability. SA is a general-purpose stochastic optimization method that has proven to be an effective tool for approximating globally optimal solutions to many types of NP-hard combinatorial optimization problems. In this section, we briefly review SA algorithm and propose an implementation to solve CAC problem.

SA is a randomized local search method based on the simulation of annealing of metal. The acceptance probability of a trial solution is given by Eq. 3, where T is the *temperature* of the system, ΔE is the difference of the costs between the trial and the current solutions (the cost change due to the perturbation), Eq. 3 means that the trial solution is accepted by nonzero probability $e^{(-\Delta E/T)}$ even though the solution deteriorates (*uphill move*).

$$(P) = \begin{cases} 1 & if \Delta E < 0 \\ e^{(-\frac{\Delta E}{T})} & otherwise \end{cases} \tag{3}$$

Uphill moves enable the system to escape from the local minima; without them, the system would be trapped into a local minimum. Too high of a probability for the occurrence of uphill moves, however, prevents the system from converging. In SA, the probability is controlled by temperature in such a manner that at the beginning of the procedure the temperature is sufficiently high, in which a high probability is available, and as the calculation proceeds the temperature is gradually decreased, lowering the probability (Jun & Mizuta, 2005).

4.1 Internal representation

The following paragraphs will describe each of the components of the implementation of our SA. The description is done given the matrix representation of an CA. An CA can be represented as a matrix M of size $N \times k$, where the columns are the parameters and the rows are the cases of the test set that is constructed. Each cell $m_{i,j}$ in the array accepts values from the set $\{1, 2, ..., v_j\}$ where v_j is the cardinality of the alphabet of j^{th} column.

4.2 Initial solution

The *initial solution* M is constructed by generating M as a matrix with maximum Hamming distance. The Hamming distance $d(x, y)$ between two rows $x, y \in M$ is the number of elements in which they differ. Let r_i be a row of the matrix M. To generate a random matrix M of maximum Hamming distance the following steps are performed:

1. Generate the first row r_1 at random.
2. Generate two rows c_1, c_2 at random, which will be candidate rows.
3. Select the candidate row c_i that maximizes the Hamming distance according to Eq. 4 and added to the i^{th} row of the matrix M.

$$g(r_i) = \sum_{s=1}^{i-1} \sum_{v=1}^{k} d(m_{s,v}, m_{i,v}), \textbf{where } d(m_{s,v}, m_{i,v}) = \begin{cases} 1 \textbf{ if } m_{s,v} \neq m_{i,v} \\ 0 \textbf{ Otherwise} \end{cases} \quad (4)$$

4. Repeat from step 2 until M is completed.

An example is shown in Fig. 2; the number of symbols different between rows r_1 and c_1 are 4 and between r_2 and c_1 are 3 summing up 7. Then, the hamming distance for the candidate row c_1 is 7.

$$Rows \begin{cases} r_1 = \{2\ 1\ 0\ 1\} \\ r_2 = \{1\ 2\ 0\ 1\} \\ c_1 = \{0\ 2\ 1\ 0\} \end{cases} \qquad Distances \begin{cases} d(r_1, c_1) = 4 \\ d(r_2, c_1) = 3 \\ g(c_1) = 7 \end{cases}$$

Fig. 2. Example of the hamming distance between two rows r_1, r_2 that are already in the matrix $\mathcal{M}$ and a candidate row c_1.

4.3 Evaluations function

The *evaluation function* $E(M)$ is used to estimate the goodness of a candidate solution. Previously reported metaheuristic algorithms for constructing CA have commonly evaluated the quality of a potential solution (covering array) as the number of combination of symbols missing in the matrix M (Cohen et al., 2003; Nurmela, 2004; Shiba et al., 2004). Then, the expected solution will be zero missing.

In the proposed SA implementation this evaluation function definition was used. Its computational complexity is equivalent to $O(N\binom{k}{t})$.

4.4 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. 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 (Cavique et al., 1999). 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.

Two *neighborhood functions* were implemented to guide the local search of our SA algorithm. The neighborhood function $\mathcal{N}_1(s)$ makes a random search of a missing t-tuple, then tries by setting the j^{th} combination of symbols in every row of M. The neighborhood function $\mathcal{N}_2(s)$ randomly chooses a position (i, j) of the matrix M and makes all possible changes of symbol. During the search process a combination of both $\mathcal{N}_1(s)$ and $\mathcal{N}_2(s)$ neighborhood functions is employed by our SA algorithm. The former is applied with probability P, while the latter

is employed at an $(1 - P)$ rate. This combined neighborhood function $\mathcal{N}_3(s, x)$ is defined in Eq. 5, where x is a random number in the interval [0, 1).

$$\mathcal{N}_3(s,x) = \begin{cases} \mathcal{N}_1(s) & \text{if } x \leq P \\ \mathcal{N}_2(s) & \text{if } x > P \end{cases} \tag{5}$$

In the next section it is presented our parallel simulated annealing approach for solving CAC problem.

4.5 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 (Aarts & Van Laarhoven, 1985; Atiqullah, 2004). 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 is the maximum cardinality of M) of the studied covering array. This is because more moves are required for CAs with alphabets of greater cardinality.

4.6 Termination condition

The *stop criterion* for our SA is 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.

4.7 SA Pseudocode

The Algorithm 4 presents the simulated annealing heuristic as described above. The meaning of the four functions is obvious: INITIALIZE computes a start solution and initial values of the parameters T and L; GENERATE selects a solution from the neighborhood of the current solution, using the neighborhood function $\mathcal{N}_3(s, x)$; CALCULATE_CONTROL computes a new value for the parameter T (cooling schedule) and the number of consecutive temperature decrements with no improvement in the solution.

5. Parallel simulated annealing

In this section we propose a parallel strategy to construct CAs using a simulated annealing algorithm.

A common approach to parallelizing simulated annealing is to generate several perturbations in the current solution simultaneously. Some of them, those with small variance, locally explore the region around the current point, while those with larger variances globally explore the feasible region. If each process has got different perturbation or move generation, each

Algorithm 4: Sequential simulated annealing for the CAC problem

```
1  INITIALIZE(M,T,L);                                  /* Create the initial solution. */
2  M* ← M;                                             /* Memorize the best solution. */
3  repeat
4      for i ← 1 to L do
5          M_i ← GENERATE(M);                          /* Perturb current state. */
6          ΔE ← E(M_i) − E(M);                         /* Evaluate cost function. */
7          x ← random;                                 /* Range [0,1). */
8          if ΔE < 0 or e^(−ΔE/T) > x then
9              M ← M_i;                                /* Accept new state. */
10             if E(M) < E(M*) then
11                 M* ← M;                             /* Memorize the best solution. */
12             end if
13         end if
14     end for
15     CALCULATE_CONTROL(T,φ)
16 until termination condition is satisfied;
```

process will probably get a different solution at the end of iterations. This approach may be described as follows:

1. The *master* node set $T = T_0$, generates an *initial_solution* using the Hamming distance algorithm (See Section 4.2) and distributes them to each workers.
2. At the current temperature T, each worker begins to execute iterative operations (L).
3. At the end of iterations, the *master* is responsible for collecting the solution obtained by each process at current temperature and broadcasts the best solution of them among all participating processes.
4. If the termination condition is not met, each process reduces the temperature and goes back to step 2, else algorithm terminates.

Algorithm 5 shows the pseudocode for *master* node. The function INITIALIZE computes a start solution (using Hamming distances algorithm) and initial values of the parameters T and L. The *master* node distributes the initial parameters to slave nodes, and awaits the results. Each L iterations, the slaves send their results to the master node (See Algorithm 6). The master node selects the best solution. If the termination criterion is not satisfied, the master node computes a new value for the parameter T (cooling schedule) and the number of consecutive temperature decrements with no improvement in the solution.

Algorithm 5: Simulated annealing for the master node

```
1  INITIALIZE(M,T,L);                                  /* Create the initial solution. */
2  M* ← M;                                             /* Memorize the best solution. */
3  repeat
4      M ← annealing_worker(M*,T,L);                   /* Call workers */
5      if E(M) < E(M*) then
6          M* ← M;                                     /* Memorize the best solution. */
7      end if
8      CALCULATE_CONTROL(T,φ)
9  until termination condition is satisfied;
```

Algorithm 6: Worker algorithm

Input: *best_solution*, Temperature (T) and the number of perturbations (L).
Output: *best_local_solution*.

```
1  annealing_worker( M, T, L )
2  for i ← 1 to L do
3      M_i ← GENERATE(M)                          /* Perturb current state.  */
4      ΔE ← E(M_i) − E(M)                         /* Evaluate cost function. */
5      x ← random                                 /* Range [0,1).            */
6      if ΔE < 0 or e^(−ΔE/T) > x then
7          M ← M_i                                /* Accept new state.       */
8      end if
9  end for
10 return M
```

6. Grid Computing approach

Simulated annealing (SA) is inherently sequential and hence very slow for problems with large search spaces. Several attempts have been made to speed up this process, such as development of special purpose computer architectures (Ram et al., 1996). As an alternative, we propose a Grid deployment of the parallel SA algorithm for constructing CAs, introduced in the previous section. In order to fully understand the Grid implementation developed in this work, this subsection will introduce all the details regarding the Grid Computing Platform used and then, the different execution strategies will be exposed.

6.1 Grid Computing Platform

The evolution of Grid Middlewares has enabled the deployment of Grid e-Science infrastructures delivering large computational and data storage capabilities. Current infrastructures, such as the one used in this work, EGI, rely on gLite mainly as core middleware supporting several services in some cases. World-wide initiatives such as EGI, aim at linking and sharing components and resources from several European NGIs (National Grid Initiatives).

In the EGI infrastructure, jobs are specified through a job description language (Pacini, 2001) or JDL that defines the main components of a job: executable, input data, output data, arguments and restrictions. The restrictions define the features a resource should provide, and could be used for meta-scheduling or for local scheduling (such as in the case of MPI jobs). Input data could be small or large and job-specific or common to all jobs, which affects the protocols and mechanisms needed. Executables are either compiled or multiplatform codes (scripts, Java, Perl) and output data suffer from similar considerations as input data.

The key resources in the EGI infrastructure are extensively listed in the literature, and can be summarized as:

1. WMS / RB (Workload Management System / Resource Broker): Meta-scheduler that coordinates the submission and monitoring of jobs.
2. CE (Computing Elements): The access point to a farm of identical computing nodes, which contains the LRMS (Local Resource Management System). The LRMS is responsible for scheduling the jobs submitted to the CE, allocating the execution of a job in one

(sequential) or more (parallel) computing nodes. In the case that no free computing nodes are available, jobs are queued. Thus, the load of a CE must be considered when estimating the turnaround of a job.

3. WN (Working Nodes): Each one of the computing resources accessible through a CE. Due to the heterogeneous nature of Grid infrastructure, the response time of a job will depend on the characteristics of the WN hosting it.
4. SE (Storage Element): Storage resources in which a task can store long-living data to be used by the computers of the Grid. This practice is necessary due to the size limitation imposed by current Grid Middlewares in the job file attachment (10 Megabytes in the gLite case). So, use cases which require the access to files which exceed that limitation are forced to use these Storage Elements. Nevertheless, the construction of CAs is not a data-intensive use case and thus the use of SEs can be avoided.
5. LFC (Logic File Catalog): A hierarchical directory of logical names referencing a set of physical files and replicas stored in the SEs.
6. BDII (Berkley Database Information System): Service point for the Information System which registers, through LDAP, the status of the Grid. Useful information relative to CEs, WNs and SEs can be obtained by querying this element.
7. R-GMA (Relational Grid Monitoring Architecture). Service for the registration and notification of information in most of the EGI services.
8. VOMS (Virtual Organisation Management System). Authorization infrastructure to define the access rights to resources.

Some of this terms will be referenced along the following sections.

6.2 Preprocessing task: Selecting the most appropriate CEs

A production infrastructure such as EGI involves tens of thousands of resources from hundreds of sites, involving tens of countries and a large human team. Since it is a general-purpose platform, and although there is a common middleware and a recommended operating system, the heterogeneity in the configuration and operation of the resources is inevitable. This heterogeneity, along with other social and human factors such as the large geographical coverage and the different skills of operators introduces a significant degree of uncertainty in the infrastructure. Even considering that the service level required is around 95%, it is statistically likely to find in each large execution sites that are not working properly. Thus, prior to beginning the experiments, it is necessary to do empirical tests to define a group of valid computing resources (CEs) and this way facing resource setup problems. These tests can give some real information like computational speed, primary and secondary memory sizes and I/O transfer speed. These data, in case there are huge quantities of resources, will be helpful to establish quality criteria choosing resources.

6.3 Asynchronous schema

Once the computing elements, where the jobs will be submitted, have been selected, the next step involves correctly specifying the jobs. In that sense, it will be necessary to produce the specification using the job description language in gLite. An example of a JDL file can be seen in the Fig. 3.

```
Type = "Job";
VirtualOrganisation = "biomed";
Executable = "test.sh";
Arguments = "16 21 3 2";
StdOutput = "std.out";
StdError = "std.err";
InputSandbox = {"/home/CA_experiment/gridCA.c","/home/CA_experiment/N16k21v3t2.ca","/home/CA_experiment/test.sh"};
OutputSandbox = {"std.out","std.err","N16k21v3t2.ca"};
```

Fig. 3. JDL example for the case of $N = 16, k = 21, v = 3, t = 2$.

As it can be seen in Fig. 3, the specification of the job includes: the virtual organisation where the job will be launched (VirtualOrganisation), the main file that will start the execution of the job (Executable), the arguments that will used for invoking the executable (Arguments), the files in which the standard outputs will be dumped (StdOutput y StdError), and finally the result files that will be returned to the user interface (OutputSandBox).

So, the most important part of the execution lies in the program (a shell-script) specified in the *Executable* field of the description file. The use of a shell-script instead of directly using the executable (gridCA) is mandatory due to the heterogeneous nature present in the Grid. Although the conditions vary between different resources, as it was said before, the administrators of the sites are recommended to install Unix-like operative systems. This measure makes sure that all the developed programs will be seamlessly executed in any machine of the Grid infrastructure. The source code must be dynamically compiled in each of the computing resources hosting the jobs. Thus, basically, the shell-script works like a wrapper that looks for a *gcc*-like compiler (the source code is written in the *C* language), compiles the source code and finally invokes the executable with the proper arguments (values of N, k, v and t respectively).

One of the most crucial parts of any Grid deployment is the development of an automatic system for controlling and monitoring the evolution of an experiment. Basically, the system will be in charge of submitting the different gLite jobs (the number of jobs is equal to the value of the parameter N), monitoring the status of these jobs, resubmitting (in case a job has failed or it has been successfully completed but the SA algorithm has not already converged) and retrieving the results. This automatic system has been implemented as a master process which periodically (or asynchronously as the name of the schema suggests) oversees the status of the jobs.

This system must possess the following properties: completeness, correctness, quick performance and efficiency on the usage of the resources. Regarding the completeness, we have take into account that an experiment will involve a lot of jobs and it must be ensured that all jobs are successfully completed at the end. The correctness implies that there should be a guarantee that all jobs produce correct results which are comprehensive presented to the user and that the data used is properly updated and coherent during the whole experiment (the master must correctly update the file with the .ca extension showed in the JDL specification in order the Simulated Annealing algorithm to converge). The quick performance property implies that the experiment will finish as quickly as possible. In that sense, the key aspects are: a good selection of the resources that will host the jobs (according to the empirical tests performed in the preprocessing stage) and an adequate resubmission policy (sending new jobs to the resources that are being more productive during the execution of the experiment). Finally, if the on-the-fly tracking of the most productive computing resources is correctly done, the efficiency in the usage of the resources will be achieved.

Due to the asynchronous behavior of this schema, the number of slaves (jobs) that can be submitted (the maximum size of N) is only limited by the infrastructure. However, other schemas such as the one showed in the next point, could achieve a better performance in certain scenarios.

6.4 Synchronous schema

This schema a sophisticated mechanism known, in Grid terminology, as submission of *pilot jobs*. The submission of pilot jobs is based on the master-worker architecture and supported by the DIANE (DIANE, 2011) + Ganga (Moscicki et al., 2009) tools. When the processing begins a master process (a server) is started locally, which will provide tasks to the worker nodes until all the tasks have been completed, being then dismissed. On the other side, the worker agents are jobs running on the Working Nodes of the Grid which communicate with the master. The master must keep track of the tasks to assure that all of them are successfully completed while workers provide the access to a CPU previously reached through scheduling, which will process the tasks. If, for any reason a task fails or a worker losses contact with the master, the master will immediately reassign the task to another worker. The whole process is exposed in Fig. 4. So, in contrast to the asynchronous schema, in this case the master is continuously in contact with the slaves.

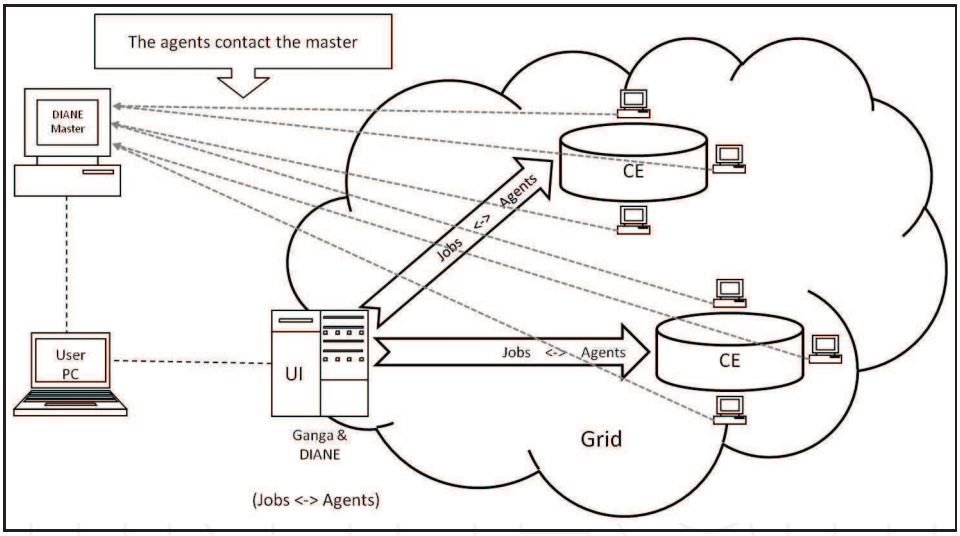


Fig. 4. Pilot jobs schema offered by DIANE-Ganga.

However, before initiating the process or execution of the master/worker jobs, it is necessary to define their characteristics. Firstly, the specification of a run must include the master configuration (workers and heartbeat timeout). It is also necessary to establish master scheduling policies such as the maximum number of times that a lost or failed task is assigned to a worker; the reaction when a task is lost or fails; and the number of resubmissions before a worker is removed. Finally, the master must know the arguments of the tasks and the files shared by all tasks (executable and any auxiliary files).

At this point, the master can be started using the specification described above. Upon checking that all is right, the master will wait for incoming connections from the workers.

Workers are generic jobs that can perform any operation requested by the master which are submitted to the Grid. In addition, these workers must be submitted to the selected CEs in the pre-processing stage. When a worker registers to the master, the master will automatically assign it a task.

This schema has several advantages derived from the fact that a worker can execute more than one task. Only when a worker has successfully completed a task the master will reassign it a new one. In addition, when a worker demands a new task it is not necessary to submit a new job. This way, the queuing time of the task is intensively reduced. Moreover, the dynamic behavior of this schema allows achieving better performance results, in comparison to the asynchronous schema.

However, there are also some disadvantages that must be mentioned. The first issue refers to the unidirectional connectivity between the master host and the worker hosts (Grid node). While the master host needs inbound connectivity, the worker node needs outbound connectivity. The connectivity problem in the master can be solved easily by opening a port in the local host; however the connectivity in the worker will rely in the remote system configuration (the CE). So, in this case, this extra detail must be taken into account when selecting the computing resources. Another issue is defining an adequate timeout value. If, for some reason, a task working correctly suffers from temporary connection problems and exceeds the timeout threshold it will cause the worker being removed by the master. Finally, a key factor will be to identify the rightmost number of worker agents and tasks. In addition, if the number of workers is on the order of thousands (i.e. when N is about 1000) bottlenecks could be met, resulting on the master being overwhelmed by the excessive number of connections.

7. Experimental results

This section presents an experimental design and results derived from testing the approach described in the section 6. In order to show the performance of the SA algorithm, two experiments were developed. The first experiment had as purpose to fine tune the probabilities of the neighborhood functions to be selected. The second experiment evaluated the performance of SA over a new benchmark proposed in this chapter. The results were compared against two of the well-known tools in the literature that constructs CAs, the TConfig[1] (recursive constructions) and ACTS[2] (a greedy algorithm named IPOG-F) respectively.

In all the experiments the following parameters were used for our SA implementation:

1. Initial temperature $T_i = 4.0$
2. Final temperature $T_f = 1.0E - 10$
3. Cooling factor $\alpha = 0.99$
4. Maximum neighboring solutions per temperature $L = (N \times k \times v)^2$

[1] TConfig: `http://www.site.uottawa.ca/~awilliam/TConfig.jar`
[2] ACTS: `http://csrc.nist.gov/groups/SNS/acts/index.html`

5. Frozen factor $\phi = 11$
6. According to the results shown in section 7.1, the neighborhood function $\mathcal{N}_3(s, x)$ is applied using a probability $P = 0.3$

Moreover, the characteristics of the Grid infrastructure employed for carrying the experiments are:

1. Infrastructure's name: EGI (European Grid infrastructure)
2. Virtual Organisation: biomed
3. Middleware: gLite
4. Total number of WNs available: 281.530

7.1 Fine tuning the probability of execution of the neighborhood functions

It is well-known that the performance of a SA algorithm is sensitive to parameter tuning. In this sense, we follow a methodology for a fine tuning of the two neighborhood functions used in our SA algorithm. The fine tuning was based on the next linear Diophantine equation,

$$P_1 x_1 + P_2 x_2 = q$$

where x_i represents a neighborhood function and its value set to 1, P_i is a value in $\{0.0, 0.1, .., 1.0\}$ that represents the probability of executing x_i , and q is set to 1.0 which is the maximum probability of executing any x_i. A solution to the given linear Diophantine equation must satisfy

$$\sum_{i=1}^{2} P_i x_i = 1.0$$

This equation has 11 solutions, each solution is an experiment that test the degree of participation of each neighborhood function in our SA implementation to accomplish the construction of an CA. Every combination of the probabilities was applied by SA to construct the set of CAs shows in Table 5(a) and each experiment was run 31 times, with the data obtained for each experiment we calculate the median. A summary of the performance of SA with the probabilities that solved the 100% of the runs is shown in Table 5(b).

Finally, given the results shown in Fig. 5 the best configuration of probabilities was $P_1 = 0.3$ and $P_2 = 0.7$ because it found the CAs in smaller time (median value). The values $P_1 = 0.3$ and $P_2 = 0.7$ were kept fixed in the second experiment.

In the next subsection, we will present more computational results obtained from a performance comparison carried out among our SA algorithm, a well-known greedy algorithm (IPOG_F) and a tool named TConfig that constructs CAs using recursive functions.

7.2 Comparing SA with the state-of-the-art algorithms

For the second of our experiments we have obtained the ACTS and TConfig software. We create a new benchmark composed by 60 ternary CAs instances where $5 \leq k \leq 100$ and $2 \leq t \leq 4$.

The SA implementation reported by (Cohen et al., 2003) 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 $t \geq 3$.

(a)

Id	CA description
ca_1	$CA(19;2,30,3)$
ca_2	$CA(35;3,5,3)$
ca_3	$CA(58;3,10,3)$
ca_4	$CA(86;4,5,3)$
ca_5	$CA(204;4,10,3)$
ca_6	$CA(243;5,5,3)$
ca_7	$CA(1040;5,15,3)$

(b)

p_1	p_2	ca_1	ca_2	ca_3	ca_4	ca_5	ca_6	ca_7
0	1	4789.763	3.072	46.989	12.544	3700.038	167.901	0.102
0.1	0.9	1024.635	0.098	0.299	0.236	344.341	3.583	0.008
0.2	0.8	182.479	0.254	0.184	0.241	173.752	1.904	0.016
0.3	0.7	224.786	0.137	0.119	0.222	42.950	1.713	0.020
0.4	0.6	563.857	0.177	0.123	0.186	92.616	3.351	0.020
0.5	0.5	378.399	0.115	0.233	0.260	40.443	1.258	0.035
0.6	0.4	272.056	0.153	0.136	0.178	69.311	2.524	0.033
0.7	0.3	651.585	0.124	0.188	0.238	94.553	2.127	0.033
0.8	0.2	103.399	0.156	0.267	0.314	81.611	5.469	0.042
0.9	0.1	131.483	0.274	0.353	0.549	76.379	4.967	0.110
1	0	7623.546	15.905	18.285	23.927	1507.369	289.104	2.297

Table 5. (a) A set of 7 CAs configurations; (b) Performance of SA with the 11 combinations of probabilities which solved the 100% of the runs to construct the CAs listed in (a).

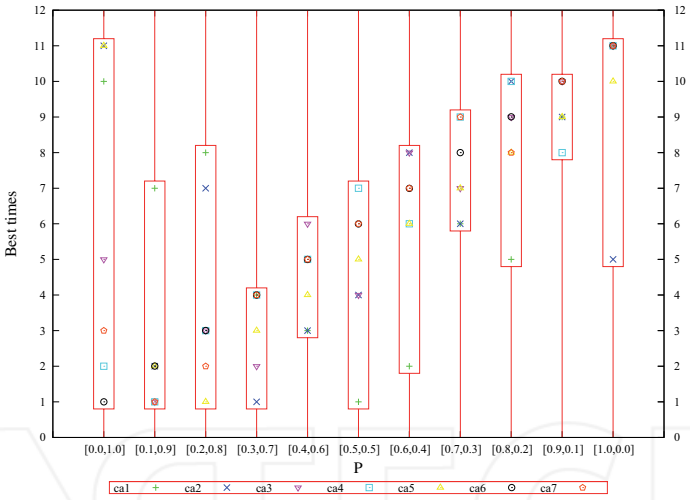


Fig. 5. Performance of our SA with the 11 combinations of probabilities.

The results from this experiment are summarized in Table 6, which presents in the first two columns the strength t and the degree k of the selected benchmark instances. The best size N found by the TConfig tool, IPOG-F algorithm and our SA algorithm are listed in columns 3, 4 and 5 respectively. Next, Fig. 6 compares the results shown in Table 6.

From Table 6 and Fig. 6 we can observe that our SA algorithm gets solutions of better quality than the other two tools. Finally, each of the 60 ternary CAs constructed by our SA algorithm have been verified by the algorithm described in Section 3 . In order to minimize the execution time required by our SA algorithm, the following rule has been applied when choosing the

rightmost Grid execution schema: experiments involving a value of the parameter N equal or less than 500 have been executed with the synchronous schema while the rest have been performed using the asynchronous schema.

(a) $CAN(2,k,3)$

t	k	TConfig	IPOG-F	Our SA
2	5	15	13	11
	10	15	19	14
	15	17	20	15
	20	21	21	15
	25	21	21	17
	30	21	23	18
	35	21	23	19
	40	21	24	20
	45	23	25	20
	50	25	26	21
	55	25	26	21
	60	25	27	21
	65	27	27	21
	70	27	27	21
	75	27	28	21
	80	27	29	21
	85	27	29	21
	90	27	30	21
	95	27	30	22
	100	27	30	22

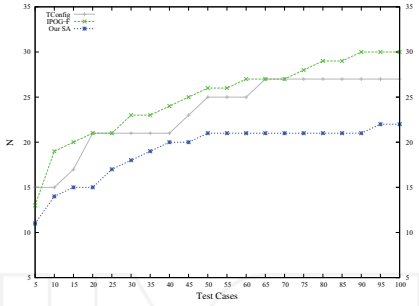
(b) $CAN(3,k,3)$

t	k	TConfig	IPOG-F	Our
3	5	40	42	33
	10	68	66	45
	15	83	80	57
	20	94	92	59
	25	102	98	72
	30	111	106	87
	35	117	111	89
	40	123	118	89
	45	130	121	90
	50	134	126	101
	55	140	131	101
	60	144	134	104
	65	147	138	120
	70	150	141	120
	75	153	144	120
	80	155	147	129
	85	158	150	130
	90	161	154	130
	95	165	157	132
	100	167	159	133

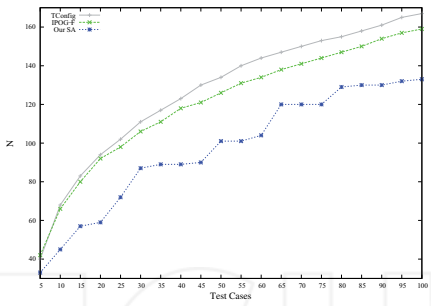
(c) $CAN(4,k,3)$

t	k	TConfig	IPOG-F	Our
4	5	115	98	81
	10	241	228	165
	15	325	302	280
	20	383	358	330
	25	432	405	400
	30	466	446	424
	35	518	479	475
	40	540	513	510
	45	572	533	530
	50	581	559	528
	55	606	581	545
	60	621	596	564
	65	639	617	581
	70	657	634	597
	75	671	648	610
	80	687	663	624
	85	699	678	635
	90	711	690	649
	95	723	701	660
	100	733	714	669

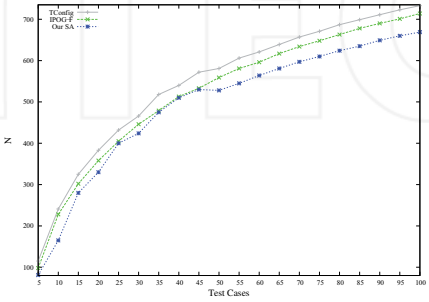
Table 6. Comparison among TConfig, IPOG-F and our SA to construct ternary CAs when $5 \leq k \leq 100$ and $2 \leq t \leq 4$.



(a) $CAN(2,k,3)$



(b) $CAN(3,k,3)$



(c) $CAN(4,k,3)$

Fig. 6. Graphical comparison of the performance among TConfig, IPOG-F and our SA to construct ternary CAs when $5 \leq k \leq 100$ and $2 \leq t \leq 4$.

8. Conclusions

In large problem domains, testing is limited by cost. Every test adds to the cost, so CAs are an attractive option for testing.

Simulated annealing (SA) is a general-purpose stochastic optimization method that has proven to be an effective tool for approximating globally optimal solutions to many types of NP-hard combinatorial optimization problems. But, the sequential implementation of SA algorithm has a slow convergence that can be improved using Grid or parallel implementations

This work focused on constructing ternary CAs with a new approach of SA, which integrates three key features that importantly determines its performance:

1. An efficient method to generate initial solutions with maximum Hamming distance.
2. 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 minimal.
3. An effective cooling schedule allowing our SA algorithm to converge faster, producing at the same time good quality solutions.

The empirical evidence presented in this work showed that SA improved the size of many CAs in comparison with the tools that are among the best found in the state-of-the-art of the construction of CAs.

To make up for the time the algorithm takes to converge, we proposed an implementation of our SA algorithm for Grid Computing. The main conclusion extracted from this point was the possibility of using two different schemas (asynchronous and synchronous) depending on the size of the experiment. On the one hand, the synchronous schema achieves better performance but is limited by the maximum number of slave connections that the master can keep track of. On the other hand, the asynchronous schema is slower but experiments with a huge value of N can be seamlessly performed.

As future work, we aim to extend the experiment where $100 \leq k \leq 20000$ and $2 \leq t \leq 12$, and compare our results against the best upper bounds found in the literature (Colbourn, 2011).

Finally, the new CAs are available in CINVESTAV Covering Array Repository (CAR), which is available under request at `http://www.tamps.cinvestav.mx/~jtj/CA.php`.

9. Acknowledgments

The authors thankfully acknowledge the computer resources and assistance provided by Spanish Supercomputing Network (TIRANT-UV). This research work was partially funded by the following projects: CONACyT 58554, Calculo de Covering Arrays; 51623 Fondo Mixto CONACyT y Gobierno del Estado de Tamaulipas.

10. References

Aarts, E. H. L. & Van Laarhoven, P. J. M. (1985). Statistical Cooling: A General Approach to Combinatorial Optimization Problems, *Philips Journal of Research* 40: 193–226.

Almond, J. & Snelling, D. (1999). Unicore: Uniform access to supercomputing as an element of electronic commerce, *Future Generation Computer Systems* 613: 1–10. `http://dx.doi.org/10.1016/S0167-739X(99)00007-2`.

Atiqullah, M. (2004). An efficient simple cooling schedule for simulated annealing, *Proceedings of the International Conference on Computational Science and its Applications - ICCSA 2004*, Vol. 3045 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 396–404. `http://dx.doi.org/10.1007/978-3-540-24767-8_41`.

Avila-George, H., Torres-Jimenez, J., Hernández, V. & Rangel-Valdez, N. (2010). Verification of general and cyclic covering arrays using grid computing, *Proceedings of the Third international conference on Data management in grid and peer-to-peer systems - GLOBE 2010*, Vol. 6265 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 112–123. `http://dx.doi.org/10.1007/978-3-642-15108-8_10`.

Bryce, R. C. & Colbourn, C. J. (2007). The density algorithm for pairwise interaction testing, *Softw Test Verif Rel* 17(3): 159–182. `http://dx.doi.org/10.1002/stvr.365`.

Bush, K. A. (1952). Orthogonal arrays of index unity, *Ann Math Stat* 23(3): 426–434. `http://dx.doi.org/10.1214/aoms/1177729387`.

Calvagna, A., Gargantini, A. & Tramontana, E. (2009). Building T-wise Combinatorial Interaction Test Suites by Means of Grid Computing, *Proceedings of the 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises - WETICE 2009*, IEEE Computer Society, pp. 213–218. `http://dx.doi.org/10.1109/WETICE.2009.52`.

Cavique, L., Rego, C. & Themido, I. (1999). Subgraph ejection chains and tabu search for the crew scheduling problem, *The Journal of the Operational Research Society* 50(6): 608–616. http://www.ingentaconnect.com/content/pal/01605682/1999/00000050/00000006/2600728.

Cohen, D. M., Dalal, S. R., Parelius, J. & Patton, G. C. (1996). The combinatorial design approach to automatic test generation, *IEEE Software* 13(5): 83–88. http://dx.doi.org/10.1109/52.536462.

Cohen, M. B., Colbourn, C. J. & Ling, A. C. H. (2003). Augmenting simulated annealing to build interaction test suites, *Proceedings of the 14th International Symposium on Software Reliability Engineering - ISSRE 2003*, IEEE Computer Society, pp. 394–405. http://dx.doi.org/10.1109/ISSRE.2003.1251061.

Colbourn, C. (2004). Combinatorial aspects of covering arrays, *Le Matematiche* 58: 121–167.

Colbourn, C. J. (2011). Covering array tables for t=2,3,4,5,6. Accessed on April 20, 2011. URL: *http://www.public.asu.edu/ ccolbou/src/tabby/catable.html*

Colbourn, C. J. & Kéri, G. (2009). Binary covering arrays and existentially closed graphs, *Proceedings of the 2nd International Workshop on Coding and Cryptology - IWCC 2009*, Vol. 5557 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 22–33. http://dx.doi.org/10.1007/978-3-642-01877-0_3.

Colbourn, C. J. & Ling, A. C. H. (2009). A recursive construction for perfect hash families, *Journal of Mathematical Cryptology* 3(4): 291–306. http://dx.doi.org/10.1515/JMC.2009.018.

Colbourn, C. J., Martirosyan, S. S., Mullen, G. L., Shasha, D., Sherwood, G. B. & Yucas, J. L. (2006). Products of mixed covering arrays of strength two, *J. Combin. Designs* 12(2): 124–138. http://dx.doi.org/10.1002/jcd.20065.

Colbourn, C. J. & Torres-Jimenez, J. (2010). Error-Correcting Codes, Finite Geometries and Cryptography. Chapter: Heterogeneous Hash Families and Covering Arrays, *Contemporary Mathematics* 523: 3–15. ISBN-10 0-8218-4956-5.

DIANE (2011). Distributed analysis environment. Accessed on June 6, 2011. URL: *http://it-proj-diane.web.cern.ch/it-proj-diane/*

EGI (2011). European grid initiative. Accessed on September 6, 2011. URL: *http://www.egi.eu/*

gLite (2011). Lightweight middleware for grid computing. Accessed on June 6, 2011. URL: *http://glite.cern.ch/*

Globus Alliance (2011). Globus toolkit. Accessed on May 21, 2011. URL: *http://www.globus.org/toolkit/*

Gonzalez-Hernandez, L., Rangel-Valdez, N. & Torres-Jimenez, J. (2010). Construction of mixed covering arrays of variable strength using a tabu search approach, *Proceedings of the 4th international conference on Combinatorial optimization and applications - COCOA 2010*, Vol. 6508 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 51–64. http://dx.doi.org/10.1007/978-3-642-17458-2_6.

Jun, Y. & Mizuta, S. (2005). Detailed analysis of uphill moves in temperature parallel simulated annealing and enhancement of exchange probabilities, *Complex Systems* 15(4): 349–358. URL: *http://www.complexsystems.com/abstracts/v15_i04_a04.html*

Katona, G. O. H. (1973). Two applications (for search theory and truth functions) of sperner type theorems, *Periodica Math.* 3(1-2): 19–26. http://dx.doi.org/10.1007/BF02018457.

Kleitman, D. J. & Spencer, J. (1973). Families of k-independent sets, *Discrete Math* 6(3): 255–262. http://dx.doi.org/10.1016/0012-365X(73)90098-8.

Lei, Y., Kacker, R., Kuhn, D. R., Okun, V. & Lawrence, J. (2007). IPOG: A general strategy for t-way software testing, *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems - ECBS 2007*, IEEE Computer Society, pp. 549–556. http://dx.doi.org/10.1109/ECBS.2007.47.

Lei, Y. & Tai, K.-C. (1998). In-parameter-order: A test generation strategy for pairwise testing, *Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering - HASE 1998*, IEEE Computer Society, pp. 254–261.

Martinez-Pena, J., Torres-Jimenez, J., Rangel-Valdez, N. & Avila-George, H. (2010). A heuristic approach for constructing ternary covering arrays using trinomial coefficients, *Proceedings of the 12th Ibero-American Conference on Artificial Intelligence - IBERAMIA 2010*, Vol. 6433 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 572–581. http://dx.doi.org/10.1007/978-3-642-16952-6_58.

Martirosyan, S. S. & Colbourn, C. J. (2005). Recursive constructions of covering arrays, *Bayreuth. Math. Schr.* 74: 266–275.

McDowell, A. G. (2011). All-pairs testing. Accessed on June 21, 2011. URL: *http://www.mcdowella.demon.co.uk/allPairs.html*

Moscicki, J., Brochu, F., Ebke, J., Egede, U., Elmsheuser, J., Harrison, K., Jones, R., Lee, H., Liko, D., Maier, A., Muraru, A., Patrick, G., Pajchel, K., Reece, W., Samset, B., Slater, M., Soroko, A., Tan, C., van der Ster, D. & Williams, M. (2009). Ganga: A tool for computational-task management and easy access to grid resources, *Comput Phys Commun* 180(11): 2303–2316. http://dx.doi.org/10.1016/j.cpc.2009.06.016.

Moura, L., Stardom, J., Stevens, B. & Williams, A. (2003). Covering arrays with mixed alphabet sizes, *Journal of Combinatorial Designs* 11(6): 413–432. http://dx.doi.org/10.1002/jcd.10059.

Nurmela, K. J. (2004). Upper bounds for covering arrays by tabu search, *Discrete Appl. Math.* 138: 143–152. http://dx.doi.org/10.1016/S0166-218X(03)00291-9.

Pacini, F. (2001). Job description language howto. Accessed on October 10, 2011. URL: *http://server11.infn.it/workload-grid/docs/DataGrid01TEN01020_2-Document.pdf*

Ram, D. J., Sreenivas, T. H. & Subramaniam, K. G. (1996). Parallel simulated annealing algorithms., *Journal of Parallel and Distributed Computing* 37(2): 207–212.

Rényi, A. (1971). *Foundations of Probability*, John Wiley & Sons, New York, USA.

Sherwood, G. (2011). On the construction of orthogonal arrays and covering arrays using permutation groups. Accessed on June 20, 2011. URL: *http://testcover.com/pub/background/cover.htm*

Sherwood, G. B. (2008). Optimal and near-optimal mixed covering arrays by column expansion, *Discrete Mathematics* 308(24): 6022 – 6035. http://dx.doi.org/10.1016/j.disc.2007.11.021.

Shiba, T., Tsuchiya, T. & Kikuno, T. (2004). Using artificial life techniques to generate test cases for combinatorial testing, *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01 - COMPSAC 2004*, IEEE Computer Society, pp. 72–77. http://dx.doi.org/10.1109/CMPSAC.2004.1342808.

Surridge, M., Taylor, S., Roure, D. D. & Zaluska, E. (2005). Experiences with gria - industrial applications on a web services grid, *Proceedings of the First International Conference on*

e-Science and Grid Computing, IEEE Computer Society, pp. 98–105. http://dx.doi.org/10.1109/E-SCIENCE.2005.38.

Torres-Jimenez, J., De Alfonso, C. & Hernández, V. (2004). Computation of ternary covering arrays using a grid, *Proceedings of the Second Asian Applied Computing Conference - AACC 2004*, Vol. 3285 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 240–246. http://dx.doi.org/10.1007/978-3-540-30176-9_31.

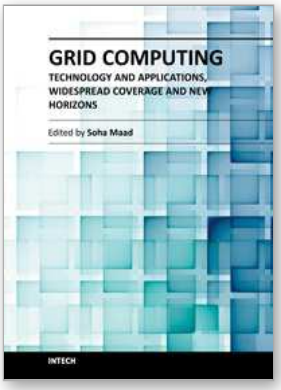
Torres-Jimenez, J. & Rodriguez-Tello, E. (2012). New bounds for binary covering arrays using simulated annealing, *Information Sciences*, 185(1): 137–152. http://dx.doi.org/10.1016/j.ins.2011.09.020.

Tung, Y. & Aldiwan, W. S. (2000). Automating test case generation for the new generation mission software system, *Proceedings of the IEEE Aerospace Conference*, Vol. 1, IEEE Press, pp. 431–437. http://dx.doi.org/10.1109/AERO.2000.879426.

Williams, A. W. (2000). Determination of test configurations for pair-wise interaction coverage, *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques - TestCom 2000*, Kluwer, pp. 59–74.

Williams, A. W. & Probert, R. L. (1996). A practical strategy for testing pair-wise coverage of network interfaces, *Proceedings of the The Seventh International Symposium on Software Reliability Engineering - ISSRE 1996*, IEEE Computer Society, pp. 246–256. http://dx.doi.org/10.1109/ISSRE.1996.558835.

Younis, M., Zamli, K. & Mat Isa, N. (2008). IRPS - An Efficient Test Data Generation Strategy for Pairwise Testing, *Proceedings of the 12th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems - KES 2008*, Vol. 5177 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 493–500. http://dx.doi.org/10.1007/978-3-540-85563-7_63.



Grid Computing - Technology and Applications, Widespread Coverage and New Horizons

Edited by Dr. Soha Maad

ISBN 978-953-51-0604-3
Hard cover, 354 pages
Publisher InTech
Published online 16, May, 2012
Published in print edition May, 2012

Grid research, rooted in distributed and high performance computing, started in mid-to-late 1990s. Soon afterwards, national and international research and development authorities realized the importance of the Grid and gave it a primary position on their research and development agenda. The Grid evolved from tackling data and compute-intensive problems, to addressing global-scale scientific projects, connecting businesses across the supply chain, and becoming a World Wide Grid integrated in our daily routine activities. This book tells the story of great potential, continued strength, and widespread international penetration of Grid computing. It overviews latest advances in the field and traces the evolution of selected Grid applications. The book highlights the international widespread coverage and unveils the future potential of the Grid.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821