Memetic Algorithms for Constructing Binary Covering Arrays of Strength Three[⋆]

Eduardo Rodriguez-Tello and Jose Torres-Jimenez

CINVESTAV-Tamaulipas, Information Technology Laboratory.
Km. 6 Carretera Victoria-Monterrey, 87276 Victoria Tamps., MEXICO
ertello@tamps.cinvestav.mx
jtj@cinvestav.mx

Abstract. This paper presents a new Memetic Algorithm (MA) designed to compute near-optimal solutions for the covering array construction problem. It incorporates several distinguished features including an efficient heuristic to generate a good quality initial population, and a local search operator based on a fine tuned Simulated Annealing (SA) algorithm employing a carefully designed compound neighborhood. Its performance is investigated through extensive experimentation over well known benchmarks and compared with other state-of-the-art algorithms, showing improvements on some previous best-known results.

Key words: Memetic Algorithms, Covering Arrays, Software Testing.

1 Introduction

Software systems play a very important role in modern society, where numerous human activities rely on them to fulfill their needs for information processing, storage, search, and retrieval. Ensuring that software systems meet people's expectations for quality and reliability is an expensive and highly complex task. Especially, considering that usually those systems have many possible configurations produced by the combination of multiple input parameters, making immediately impractical an exhaustive testing approach. An alternative technique to accomplish this goal is called *software interaction testing*. 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$(N; t, k, v)$, of size N, strength t, degree k, and order v is an $N \times k$ array on v symbols such that every $N \times t$ sub-array contains all ordered subsets from v symbols of size t (t-tuples) at least once. In such an array, each *test configuration* of the analyzed software system is represented by

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

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

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

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

There exist only some special cases where it is possible to find the covering array number using polynomial order algorithms. For instance, the case $N = v^t$, $t = 2$, $k = v+1$ was completely solved for $v = p^\alpha$ a prime or a power of prime and $v > t$ (see [1] for references). This case was subsequently generalized by Bush for $t > 2$ [2]. However, in the general case determining the covering array number is known to be NP-complete [3, 4], thus it is unlikely that exact algorithms running in polynomial time exist for this hard combinatorial optimization problem.

Other applications related to the CAC problem arise in fields like: drug screening, data compression, regulation of gene expression, authentication, intersecting codes and universal hashing (see [5] for a detailed survey).

Addressing the problem of obtaining the covering array number in reasonable time has been the focus of much research. Among the approximate methods that have been developed for constructing covering arrays are: a) recursive methods [1, 6], b) algebraic methods [7, 5], c) greedy methods [8] and d) metaheuristics such as Tabu Search [9], Simulated Annealing [10], and Genetic Algorithms [11].

This paper aims at developing a new powerful Memetic Algorithm (MA) for finding near-optimal solutions for the CAC problem. In particular, we are interested in constructing binary covering arrays of strength three and in establishing new bounds on the covering array number $\mathrm{CAN}(3, k, 2)$. To achieve this, the proposed MA algorithm incorporates a fast heuristic to create a good quality initial population and a local search operator based on a fine tuned Simulated Annealing algorithm employing two carefully designed neighborhood functions. The performance of the proposed MA algorithm is assessed with a test-suite, conformed by 20 binary covering arrays of strength three, taken from the literature. The computational results are reported and compared with previously published ones, showing that our algorithm is able to improve on 9 previous best-known solutions and to equal these results on the rest of the selected benchmark instances. It is important to note that for some of those instances the best-known results were not improved since their publication in 1993 [1].

The rest of this paper is organized as follows. In Sect. 2, a brief review is given to present some representative solution procedures for constructing binary covering arrays of strength three. Then, the components of our new Memetic

Algorithm are discussed in detail in Sect. 3. Section 4 is dedicated to computational experiments and comparisons with respect to previous best-known results. Finally, the last section summarizes the main contributions of this work.

2 Relevant Related Work

Because of the importance of the CAC problem, much research has been carried out in developing effective methods for solving it. In this section, we give a brief review of some representative procedures which were used in our comparisons. These procedures were devised for constructing binary CAs of strength three.

Sloane published in [1] a procedure which improves some elements of the work reported in Roux's PhD dissertation [12]. This procedure allows to construct a $\mathrm{CAN}(3, 2k, 2)$ by combining two CAs $\mathrm{CA}(N_2; 2, k, 2)$ and $\mathrm{CA}(N_3; 3, k, 2)$. It first appends $\mathrm{CA}(N_2; 2, k, 2)$ to a $\mathrm{CA}(N_3; 3, k, 2)$, which results in a $k \times (N_2 + N_3)$ array. Then this array is copied below itself, producing a $2k \times (N_2 + N_3)$ array. Finally, the copied strength 2 array is replaced by its bit-complement array (i.e. switch 0 to 1 and 1 to 0).

Following these ideas, Chateauneuf and Kreher presented latter in [5] an algebraic procedure for constructing $\mathrm{CAN}(3, 2k, v)$. This procedure has permitted to attain some of the best-known solutions for binary CAs of strength three. Furthermore, it is a polynomial time algorithm.

In 2001, a study was carried out by Stardom [11] to compare three different metaheuristics: Tabu Search (TS), Simulated Annealing (SA) and Genetic Algorithms (GA). Stardom's GA implementation represents a $\mathrm{CA}(N; t, k, v)$ by using an $N \times k$ array on v symbols and operates as follows: An initial population $100 \leq |P| \leq 500$ is randomly generated. At each generation the original population is randomly partitioned into two groups (male and female) of size $|P|/2$. The members of each group are ordered randomly and then the i-th arrays from each group are mated with each other, for $1 \leq i \leq |P|/2$. The $|P|$ offspring are mutated and then the most fit $|P|$ members of the male, female and offspring subpopulations combined are selected as the new population. The crossover operator randomly selects a point (i, j) for each pair of arrays to be mated. If the pair of mates contain entries A_{mn} and B_{mn} and the pair of offspring contain entries C_{mn} and D_{mn}, then $C_{mn} = A_{mn}(D_{mn} = B_{mn})$ for $m \leq i$ and $n \leq j$; and $C_{mn} = B_{mn}(D_{mn} = A_{mn})$ for $m > i$ and $n > j$. The mutation operator consists in applying a random entry swap. This process is repeated until a predefined maximum of 5000 generations is reached or when a covering array is found. For his comparisons the author employed a set of benchmark instances conformed by binary covering arrays of strength two. The results show that his GA implementation was by far the weakest of the three compared metaheuristics.

A more effective Tabu Search (TS) algorithm than that presented in [11] was devised by Nurmela [9]. This algorithm starts with a $N \times k$ randomly generated matrix that represents a covering array. The number of uncovered t-tuples is used to evaluate the cost of a candidate solution (matrix). Next an uncovered t-tuple is selected at random and the rows of the matrix are verified to find those that require only the change of a single element in order to cover the selected t-tuple. These changes, called *moves*, correspond to the neighboring solutions of the current candidate solution. The variation of cost corresponding to each such move is calculated and the move leading to the smallest cost is selected, provided that the move is not tabu. If there are several equally good nontabu moves, one of them is randomly chosen. Then another uncovered t-tuple is selected and the process is repeated until a matrix with zero cost (covering array) is found or a predefined maximum number of moves is reached. The results produced by Nurmela's TS implementation have demonstrated that it is able to slightly improved some previous best-known solutions, specially the instance $\mathrm{CA}(15; 3, 12, 2)$. However, an important drawback of this algorithm is that it consumes considerably much more computational time than any of the three previously presented algorithms.

More recently Forbes *et al.* [13] introduced an algorithm for the efficient production of covering arrays of strength t up to 6, called IPOG-F (In-Parameter Order-Generalized). Contrary to many other algorithms that build covering arrays one row at a time, the IPOG-F strategy constructs them one column at a time. The main idea is that covering arrays of $k-1$ columns can be used to efficiently build a covering array with degree k. In order to construct a covering array, IPOG-F initializes a $v^t \times t$ matrix which contains each of the possible v^t distinct rows having entries from $\{0, 1, \ldots, v-1\}$. Then, for each additional column, the algorithm performs two steps, called *horizontal growth* and *vertical growth*. Horizontal growth adds an additional column to the matrix and fills in its values, then any remaining uncovered t-tuples are covered in the vertical growth stage. The choice of which rows will be extended with which values is made in a greedy manner: it picks an extension of the matrix that covers as many previously uncovered t-tuples as possible. IPOG-F is currently implemented in a software package called FireEye [14], which was written in Java. Even if IPOG-F is a very fast algorithm for producing covering arrays it generally provides poorer quality results than other state-of-the-art algorithm like the algebraic procedures proposed by Chateauneuf and Kreher [5].

3 A New Memetic Algorithm for Constructing CAs

In this section we present a new Memetic algorithm for solving the CAC problem. Next all the details of its implementation are presented.

3.1 Search Space and Internal Representation

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

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

3.2 Fitness Function

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

Previously reported metaheuristic algorithms for solving the CAC problem have commonly evaluated the quality of a potential solution (covering array) as the change in the number of uncovered t-tuples [15, 10, 11, 9]. We can formally define this fitness function as follows. Let $A \in \mathscr{A}$ be a potential solution, S^r a $N \times t$ subarray of A representing the r-th subset of t columns taken from k, and ϑ_j a set containing the union[1] of the N t-tuples in S^j denoted by the following expression:

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

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

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

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

3.3 General Procedure

Our MA implementation starts building an initial population P, which is a set of configurations having a fixed constant size $|P|$. Then, it performs a series of cycles called generations. At each generation, assuming that $|P|$ is a multiple of four, the population is randomly partitioned into $|P| \bmod 4$ groups of four individuals. Within each group, the two most fit individuals are chosen to become the parents in a recombination operator. The resulting offspring are then improved by using a local search operator for a fixed number of iterations L. Finally, the two worst fit individuals in the group are replaced with the improved offspring.

This mating selection strategy ensures that the fittest individuals remain in the population, but restricts the amount of times they can reproduce to once per generation. At the end of each generation, thanks to the selection for survival, half of the population is turned over, ensuring a wide coverage of the search space through successive mating. The repeated introduction of less fit offspring increases the chance of a less fit individual being involved in the recombination phase, thus maintaining diversity in the population.

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

The iterative process described above stops either when a predefined maximum number of generations (*maxGenerations*) is reached or when a covering array with the predefined parameters N, t, k, and v is found.

3.4 Initializing the Population

In the GA reported in [11] the initial population is randomly generated. In contrast, in our MA implementation the population is initialized using a procedure that guarantees a balanced number of symbols in each column of the generated individuals (CAs). This procedure assigns randomly $\lfloor N/2 \rfloor$ ones and the same number of zeros to each column of the individuals when its size N is even, otherwise it allocates $\lfloor N/2 \rfloor + 1$ ones and $\lfloor N/2 \rfloor$ zeros to each column. Due to the randomness of this procedure, the individuals in the initial population are quite different. This point is important for population based algorithms because a homogeneous population cannot efficiently evolve.

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

3.5 The Recombination Operator

The main idea of the recombination operator is to generate diversified and potentially promising individuals. To do that, a good recombination operator for the CAC problem should take into consideration, as much as possible, the individuals' semantic. After some preliminary experiments for comparing different crossover operators, we have decided to use a row crossover. It randomly selects a row i for each pair of individuals to be mated. If the pair of mates contain entries A_{mn} and B_{mn} and the pair of offspring contain entries C_{mn} and D_{mn}, then $C_{mn} = A_{mn}(D_{mn} = B_{mn})$ for $m \leq i$; and $C_{mn} = B_{mn}(D_{mn} = A_{mn})$ for $m > i$. This recombination operator has the advantage to preserve certain information contained in both parents.

3.6 The Local Search Operator

The purpose of the local search (LS) operator is to improve the offspring (solutions) produced by the recombination operator for a maximum of L iterations before inserting them into the population. In general, any local search method can be used. In our implementation, we have decided to use a Simulated Annealing (SA) algorithm.

In our SA-based LS operator the neighborhood function is a key component which has a great impact on its performance. Formally, a neighborhood relation is a function $\mathcal{N} : \mathscr{A} \rightarrow 2^{\mathscr{A}}$ that assigns to every potential solution (covering array) $A \in \mathscr{A}$ a set of neighboring solutions $\mathcal{N}(A) \subseteq \mathscr{A}$, which is called the neighborhood of A. A wrong selected neighborhood function can lead to a poor exploration of the search space. A well documented alternative to increase the search power of LS methods consists in using compound neighborhood functions [16–18]. Following this idea, and based on the results of our preliminary experimentations, a neighborhood structure composed by two different neighborhood functions is proposed for this SA algorithm.

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

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

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

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

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

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

The SA operator proposed starts at an initial temperature $T_0 = 3$, at each Metropolis round $r = 500$ moves are generated. If the cost of the attempted move decreases then it is accepted. Otherwise, it is accepted with probability $P(\Delta) = e^{-\Delta/T}$ where T is the current temperature and Δ is the increase in cost that would result from that particular move. At the end of each Metropolis round then the current temperature is decremented by a factor of $\alpha = 0.95$. The algorithm stops either if the current temperature reaches $T_f = 0.001$, or when it reaches the predefined maximum of L iterations.

The algorithm memorizes and returns the most recent covering array A^* among the best configurations found: after each accepted move, the current configuration A replaces A^* if $\mathscr{F}(A) \leq \mathscr{F}(A^*)$. The rational to return the last best configuration is that we want to produce a solution which is as far away as possible from the initial solution in order to better preserve the diversity of the population.

4 Computational Experiments

In this section, we present a set of experiments accomplished to evaluate the performance of the MA algorithm presented in Sect. 3. The algorithms was coded in C and compiled with *gcc* using the optimization flag -*O*3. It was run sequentially into a CPU Xeon at 2 GHz, 1 GB of RAM with Linux operating system. Due to the non-deterministic nature of the algorithms, 20 independent runs were executed for each of the selected benchmark instances. In all the experiments the following parameters were used for the MA: a) population size $|P| = 40$, b) recombinations per generation $\mathit{offspring} = |P| \bmod 4$, c) maximal number of local search iterations $L = 50000$, d) the neighborhood function $\mathcal{N}_3(A, x, \omega, \gamma)$ is applied using a probability $p = 0.6$ and parameters $\omega = 10$ and $\gamma = N/2$, and e) maximal number of generations $maxGenerations = 200000$.

These parameter values were chosen experimentally and taking into consideration our experience in solving other combinatorial optimization problems with the use of MA [19].

4.1 Benchmark Instances and Comparison Criteria

To assess the performance of the MA introduced in Sect. 3, a test-suite composed of 20 well known benchmark instances taken from the literature was used [1, 5, 9, 13]. It includes instances of size $8 \leq N \leq 32$. The main criterion used for the comparison is the same as the one commonly used in the literature: the *best degree* k found (bigger values are better) given fixed values for N, t and v.

4.2 Comparison Among MA and the State-of-the-art Procedures

The purpose of this experiment is to carry out a performance comparison of the best bounds achieved by our MA with respect to those produced by the following state-of-the-art procedures: orthogonal array constructions [2], Roux type constructions [1], doubling constructions [7, 5], Tabu Search [9], and IPOG-F [13].

Table 1 displays the detailed computational results produced by this experiment. The first column in the table indicates the size N of the instance. Column 2 shows the best results found by IPOG-F [13] in terms of the degree k, while column 3 (Best) presents the previous best-known degree along with the reference where this result was originally published as indicated in [20]. Next five columns provide the best solutions (k^*), the success rate of finding those best solutions ($Succ.$), the average solution cost ($Avg.$) with respect to (4), its standard deviation ($Dev.$), and the average CPU time (T) in seconds obtained in 20 executions of our MA. Finally, the difference (Δ_{Best-k^*}) between the best result produced by our MA and the previous best-known solution is depicted in the last column. According to [1] the results presented in column 4 for the instances of size $N \leq 12$ are optimal solutions.

From the data presented in Table 1 we can make the following main observations. First, the solution quality attained by the proposed MA is very competitive

Table 1. Improved bounds for CAN$(3, k, 2)$

			MA					
N	IPOG-F	Best	k^*	*Succ.*	*Avg.*	*Dev.*	T	Δ_{Best-k^*}
8	4	4 [7]	4	1.0	0.0	0.0	0.02	0
10	4	5 [7]	5	1.0	0.0	0.0	0.03	0
12	5	11 [1]	11	1.0	0.0	0.0	0.13	0
15	6	12 [9]	12	1.0	0.0	0.0	0.18	0
16	7	14 [1]	14	1.0	0.0	0.0	96.64	0
17	9	16 [1]	16	1.0	0.0	0.0	374.34	0
18	11	20 [5]	20	0.8	0.2	0.4	13430.99	0
19	12	22 [5]	22	0.8	0.4	0.8	10493.26	0
20	13	22 [5]	23	0.4	1.2	1.2	13451.34	1
21	15	22 [5]	24	0.3	1.0	0.9	14793.41	2
22	16	24 [5]	24	0.9	0.2	0.4	5235.06	0
23	16	28 [5]	28	0.7	0.7	1.2	21480.33	0
24	19	30 [5]	36	0.3	2.2	2.0	36609.51	6
25	21	32 [5]	41	0.2	1.8	1.4	49110.23	9
26	24	40 [5]	42	0.4	1.9	2.0	52958.12	2
27	26	44 [5]	45	0.3	1.6	2.1	64939.07	1
28	30	44 [5]	47	0.5	2.2	2.8	71830.49	3
30	31	48 [5]	50	0.6	2.8	3.9	89837.65	2
31	33	56 [5]	56	0.7	2.4	4.0	134914.74	0
32	37	64 [1]	67	0.5	7.5	10.8	262151.32	3
Avg.	16.95	27.90	29.35	0.65	1.29	1.70	42085.34	1.45

with respect to that produced by the state-of-the-art procedures summarized in column 3. In fact, it is able to improve the previous best-known solutions on 9 benchmark instances. It is important to note that for some of these instances the best-known results were not improved since their publication in 1993 [1]. For the rest of the instances in the test-suite our MA equals the previous best-known solutions. Second, one observes that in this experiment the IPOG-F procedure [13] returns poorer quality solutions than our MA in 19 out 20 benchmark instances. Indeed, IPOG-F produces covering arrays which are in average 73.16% worst than those constructed with a MA.

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

Even if the results attained by our MA are very competitive, we have observed that the average computing time consumed by our approach, to produce these excellent results, is greater than that used by some recursive [6, 21] and algebraic methods [7, 5, 22]. However, since MA outperforms some of the state-of-the-art procedures, finding 9 new bounds, we believe that the extra consumed computing time is fully justified. Especially, if we consider that for this kind of experiments the objective is to compare the best bounds achieved by the studied algorithms.

The outstanding results achieved by MA are better illustrated in Fig. 1. The plot represents the size N of the instance (ordinate) against the degree k attained by the compared procedures (abscissa). The bounds provided by IPOG-F [13] are shown with squares, the previous best-known solutions are depicted as circles, while the bounds computed with our MA are shown as triangles. From this figure it can be seen that MA consistently outperforms IPOG-F, obtaining also important improvements with respect to the previous best-known solutions on $\mathrm{CAN}(3, k, 2)$ for $4 \leq k \leq 67$. This is the case of covering array $\mathrm{CA}(25; 3, k, 2)$ for which an increase of 28.13% of the degree k was accomplished by our algorithm.

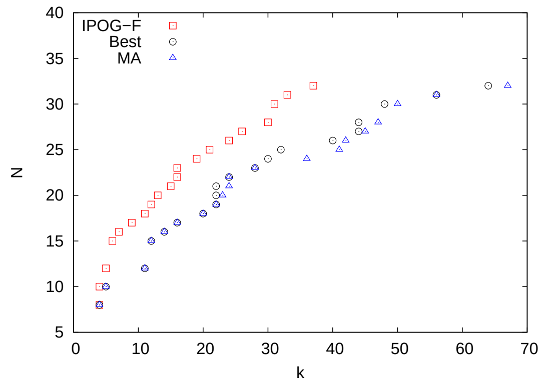

Fig. 1. Previous best-known and improved bounds on $\mathrm{CAN}(3, k, 2)$

4.3 Influence of the Variation Operators

In order to further examine the behavior of our approach we have performed some additional experiments for analyzing the influence of the variation operators used in its implementation. The results obtained with all the benchmark instances described in Sect. 4.1 were similar, so for the reason of space limitation, we have decided to show the product of these experiments using only a representative graph. Figure 2 shows the evolution profile of the fittest individual (ordinate) along the search process (abscissa) when the instance $\mathrm{CA}(17; 3, 16, 2)$ is solved using two variants of the MA described in Sect. 3: a) an algorithm using only the crossover operator, and b) an algorithm using the crossover operator and a simple mutation operator based on the $switch(A, i, j)$ defined in Sect. 3.6.

From Fig. 2 it can be observed that the worst solution quality is provided by the algorithm using only the crossover operator. It gets stuck longer time on some local minima than the algorithm employing also a simple mutation operator. However, neither of these two variants of our MA were able to find a covering array $\mathrm{CA}(17; 3, 16, 2)$. In contrast, the MA using the combination of crossover and LS operators gives better results both in solution quality and computational time expended. These experiments allow us to conclude that the contribution of the crossover operator is less significant than that of the LS operator based on a SA algorithm.

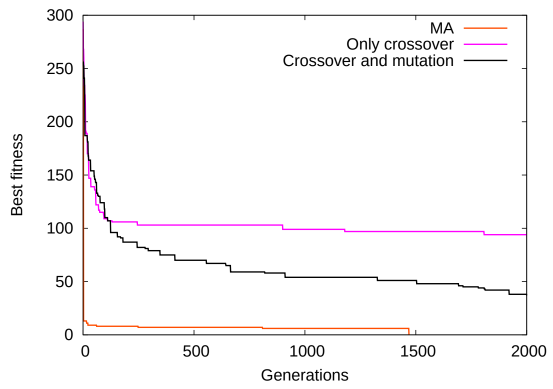

Fig. 2. Influence of the variation operators when solving the instance CA$(17; 3, 16, 2)$.

5 Conclusions

In this paper, a highly effective MA designed to compute near-optimal solutions for the CAC problem was presented. This algorithm is based on an efficient heuristic to generate good quality initial populations, and a SA-based LS operator employing a carefully designed compound neighborhood.

The performance of this MA was assessed through extensive experimentation over a set of well-known benchmark instances and compared with five other state-of-the-art procedures: orthogonal array constructions [2], Roux type constructions [1], doubling constructions [5], Tabu Search [9], and IPOG-F [13]. The results show that our MA was able to improve on 9 previous best-known solutions and to equal these results on the other 11 selected benchmark instances. Furthermore, it is important to note that these new bounds on CAN$(3, k, 2)$ offer the possibility to improve other best-known results for binary CAs of strength three of size $N > 32$ by employing doubling constructions [5].

Finding near-optimal solutions for the CAC problem in order to construct economical sized test-suites for software interaction testing is a very challenging problem. However, the introduction of this new MA opens up an exciting range of possibilities for future research. One fruitful possibility is to develop a multimeme algorithm [23] based on the MA presented here in order to efficiently construct covering arrays of strength $t > 3$ and order $v > 2$.

References

1. Sloane, N.J.A.: Covering arrays and intersecting codes. Journal of Combinatorial Designs **1**(1) (1993) 51–63
2. Bush, K.A.: Orthogonal arrays of index unity. Annals of Mathematical Statistics **23**(3) (1952) 426–434
3. Seroussi, G., Bshouty, N.: Vector sets for exhaustive testing of logic circuits. IEEE Transactions on Information Theory **34** (1988) 513–522.

4. Lei, Y., Tai, K.: In-parameter-order: A test generation strategy for pairwise testing. In: Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering, Washington, DC, USA, IEEE Computer Society (1998) 254–261
5. Chateauneuf, M.A., Kreher, D.L.: On the state of strength-three covering arrays. Journal of Combinatorial Design **10**(4) (2002) 217–238
6. Hartman, A., Raskin, L.: Problems and algorithms for covering arrays. Discrete Mathematics **284**(1-3) (2004) 149–156
7. Hedayat, A.S., Sloane, N.J.A., Stufken, J.: Orthogonal Arrays, Theory and Applications. Springer, Berlin (1999)
8. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering **23** (1997) 437–444
9. Nurmela, K.J.: Upper bounds for covering arrays by tabu search. Discrete Applied Mathematics **138**(1-2) (2004) 143–152
10. Cohen, D.M., Colbourn, C.J., Ling, A.C.H.: Constructing strength three covering arrays with augmented annealing. Discrete Mathematics **308**(13) (2008) 2709–2722
11. Stardom, J.: Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University, Burnaby, Canada (2001)
12. Roux, G.: k-propriétés dans des tableaux de n colonnes; cas particulier de la k-surjectivité et de la k-permutivité. PhD thesis, Université de Paris 6, France (1987)
13. Forbes, M., Lawrence, J., Lei, Y., Kacker, R.N., Kuhn, D.R.: Refining the in-parameter-order strategy for constructing covering arrays. Journal of Research of the National Institute of Standards and Technology **113**(5) (2008) 287–297
14. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG: A general strategy for t-way software. In: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Washington, DC, USA, IEEE Computer Society (2007) 549–556
15. Nurmela, K.J., Östergård, P.R.J.: Constructing covering designs by simulated annealing. Technical Report 10, Department of Computer Science, Helsinki University of Technology, Otaniemi, Finland (January 1993)
16. Mladenović, N., Hansen, P.: Variable neighborhood search. Computers & Operations Research **24**(11) (1997) 1097–1100
17. Urosević, D., Brimberg, J., Mladenović, N.: Variable neighborhood decomposition search for the edge weighted k-cardinality tree problem. Computers & Operations Research **31**(8) (2004) 1205–1213
18. Rodriguez-Tello, E., Hao, J.K., Torres-Jimenez, J.: An effective two-stage simulated annealing algorithm for the minimum linear arrangement problem. Computers & Operations Research **35**(10) (2008) 3331–3346
19. Rodriguez-Tello, E., Hao, J.K., Torres-Jimenez, J.: Memetic algorithms for the MinLA problem. Lecture Notes in Computer Science **3871** (2006) 73–84
20. Colbourn, C.J.: Covering Array Tables. `http://www.public.asu.edu/~ccolbou/src/tabby/catable.html` (Accessed on March 17, 2009)
21. Martirosyan, S.S., Van Trung, T.: On t-covering arrays. Designs, Codes and Cryptography **32**(1-3) (2004) 323–339
22. Hartman, A.: 10. In: Software and Hardware Testing Using Combinatorial Covering Suites. Springer-Verlag (2005) 237–266
23. Krasnogor, N.: Towards robust memetic algorithms. In: Recent Advances in Memetic Algorithms. Volume 166 of Studies in Fuzziness and Soft Computing. Springer (2004) 185–207