Randomized Postoptimization of Covering Arrays

Peyman Nayeri, Charles J. Colbourn, and Goran Konjevod

Computer Science and Engineering, Arizona State University,
P.O. Box 878809, Tempe, AZ 85287, U.S.A.
{peyman.nayeri,charles.colbourn,goran}@asu.edu

Abstract. The construction of covering arrays with the fewest rows remains a challenging problem. Most computational and recursive constructions result in extensive repetition of coverage. While some is necessary, some is not. By reducing the repeated coverage, metaheuristic search techniques typically outperform simpler computational methods, but they have been applied in a limited set of cases. Time constraints often prevent them from finding an array of competitive size. We examine a different approach. Having used a simple computation or construction to find a covering array, we employ a postoptimization technique that repeatedly adjusts the array in order to (sometimes) reduce its number of rows. At every stage the array retains full coverage. We demonstrate its value on a collection of previously best known arrays by eliminating, in some cases, 10% of their rows. In the well-studied case of strength two with twenty factors having ten values each, postoptimization produces a covering array with only 162 rows, improving on a wide variety of computational and combinatorial methods. We identify certain important features of covering arrays for which postoptimization is successful.

1 Introduction

Covering arrays are combinatorial models of test suites used to detect faulty interactions among components in software, hardware, and networked systems. They are intimately related to orthogonal arrays and related experimental designs; to surjective codes; and to qualitatively independent partitions. As a consequence of these and other connections, the construction of covering arrays has been a topic of substantial interest. See [1,2] for surveys that are now somewhat dated. Despite the extensive effort expended, finding the smallest test suites for given testing scenarios remains an unsolved problem in general. We first introduce a purely combinatorial formulation.

Let N, k, t, and v be positive integers. An $N \times k$ array, each column of which contains v distinct symbols, is a *covering array* $\mathsf{CA}(N; t, k, v)$ of *strength* t when, for every way to select t columns, each of the v^t possible tuples of symbols arises in at least one row. When used for testing, columns of the array form *factors*, and the symbols in the column form *values* or *levels* for the factor. Each row specifies the values to which to set the factors for an experimental run. A *t-tuple*

J. Fiala, J. Kratochvíl, and M. Miller (Eds.): IWOCA 2009, LNCS 5874, pp. 408–419, 2009.

or *t-way interaction* is a set of t of the factors, and an admissible level for each. The array is 'covering' in the sense that every t-way interaction is represented by at least one run. Now $\mathsf{CAN}(1,k,v) = v$ and $\mathsf{CAN}(t,k,1) = 1$. So to avoid trivial cases, we suppose that $k \geq t \geq 2$ and $v \geq 2$. In this paper, we always take the value set of each factor to be $\{0,\ldots,v-1\}$.

111202111221001202022222 1
000110212122211001121011 22
102122212202012110102000 11
012221111111210020010200 02
121101102100000220222211 11
210201200211022121112011 20
020010222021012020002222 10
101022000110112222011021 02
1⋆000212111220221102011000
010020020200100011101212 11
202202021001011011200121 02
222021010020121100221100 20
121210102022120012110020 01
201111120102220112000222 20
021212201210220201100101 12
0⋆010022120⋆⋆0210221200202
212110002012201222⋆211211
⋆0⋆10⋆2⋆1⋆⋆1101⋆⋆⋆⋆121⋆⋆1⋆

At left is shown a $\mathsf{CA}(18;2,26,3)$. The entries shown as $\star$ can be chosen arbitrarily, and every pair of columns contains each of the nine possible pairs.

In testing applications, *the fundamental problem is to determine* $\mathsf{CAN}(t,k,v)$. Evidently, $\mathsf{CAN}(t,k,v) \geq v^t$, and when equality holds the CA is an *orthogonal array* $\mathsf{OA}(v^t;t,k,v)$; see [3] for a textbook treatment. Such orthogonal arrays can exist only when $k \leq \max(v+2,t+1)$ [3], and hence they provide no examples beyond 'small' values of k. For fixed v and t, probabilistic methods establish that $\mathsf{CAN}(v,k,t) = \Theta(\log k)$ [4]. Nevertheless, only in the case when $t = v = 2$ is this function of k known exactly [5,6]. When $\mathsf{CAN}(t,k,v)$ is not known exactly, most effort has been invested in *producing 'good' upper bounds*. This is the problem considered here.

Explicit constructions of covering arrays are needed in concrete testing applications. *Recursive* methods build larger covering arrays from smaller ones. Some recursive methods are product constructions; see, for example, [7] for $t = 2$, [8,9] for $t = 3$, [9] for $t = 4$, and [10,11] for $t \geq 5$. Although these all rely on a similar strategy, their use of numerous smaller covering arrays can result in substantial duplication of coverage; the specific variants result from efforts to reduce this duplication, and have been most successful to date when $t \in \{2,3\}$. A second class of recursive methods are column replacement constructions, which use a second array as a pattern for selecting columns from a covering array; see [12] for the most general one at present. Again these suffer from substantial repetition of coverage. Every recursive method also requires that ingredient covering arrays be known.

Direct methods construct covering arrays without recourse to smaller ingredient covering arrays. Some methods employ geometric, algebraic, or number-theoretic properties. The orthogonal arrays constructed from the finite fields [3] provide a prototype for these. By exploiting the structure of automorphisms of the OAs, compact representations of covering arrays accelerate computational search [13, 14, 15, 16]. Recently, cyclotomic classes in the finite field have been shown to provide examples of binary covering arrays, and more generally examples are provided by certain Hadamard matrices [17]. Block designs have been used to make a few specific covering arrays [18]. Other easily constructed examples are

provided by taking all vectors of specified weights to form the rows of a covering array [19,20,21]. Each of these constructions provides useful examples of covering arrays, but each is quite restricted in its application. Therefore by far the most popular general methods are computational techniques.

Exhaustive computation has proved ineffective except in a handful of small cases. Therefore heuristic and metaheuristic strategies have been the norm. Techniques such as simulated annealing [22], tabu search [23], and constraint satisfaction [24] are very effective for small existence problems, but the time taken for convergence to a solution has limited their range of application. As a consequence, the most prevalent computational methods have been greedy. `AETG` [25] popularized greedy methods that generate one row of a covering array at a time, attempting to select a best possible next row; since that time, `TCG` [26] and density algorithms [27,28] have developed useful variants of this approach. For strength two, `IPO` [29] instead adds a factor (column) at a time, adding rows as needed to ensure coverage; the generalization to t-way coverage in [30,31] is the method that has been run on the largest set of parameters to date. When the arrays to be produced are very large, just checking the properties of the array is challenging; therefore, random methods have also been examined [32].

Unfortunately, at the present time, based on the current best known upper bounds for $\mathsf{CAN}(t,k,v)$ for $2 \le t \le 6$, $2 \le v \le 25$, and $t \le k \le 10000$ at [33], no single construction can be applied generally while yielding the best, or close to the best, known results. This leaves the tester with the problem of how to generate a covering array quickly that is not 'far' from optimum. We examine a new approach, that of improving a covering array after it is constructed; we call this process *postoptimization*. To the best of our knowledge, the only previous effort to improve an existing covering array is the elimination of redundant rows in CATS [34].

2 Postoptimization

In any covering array $\mathsf{CA}(N;t,k,v)$, the number of t-way interactions to be covered is $\binom{k}{t}v^t$, while the number actually covered is $N\binom{k}{t}$. Except possibly when $k \le \max(v+2, t+1)$, some duplication of coverage is necessary. All of the recursive and direct techniques attempt to limit this duplication, but cannot hope to eliminate it completely. Our objective is to eliminate some of the duplication, if possible. Every entry of a $\mathsf{CAN}(t,k,v)$ participates in $\binom{k-1}{t-1}$ t-way interactions. Some of these interactions may be covered elsewhere, while others may be covered only in this row. In principle, a specific t-way interaction could be covered as many as $N - v^t + 1$ times or as little as once. When all of the $\binom{k-1}{t-1}$ t-way interactions involving a specific entry are covered more than once, the entry can be changed arbitrarily, or indeed omitted in the determination of coverage, and the array remains a covering array. Hence such an entry is a *possible don't care position*. If we replace such an entry by $\star$ to indicate that t-way interactions involving this entry are not to be used for coverage, we select it as a *don't care position*. This replacement can cause other possible don't care positions to appear in t-way interactions that are now covered only once – such positions are

no longer possible don't care positions. On the other hand, replacing a $\star$ by an element from $\{0, \ldots, v-1\}$ can result in new positions for which all of their t-way interactions are covered more than once, i.e. new possible don't care positions.

Our strategy is to exploit the presence of don't care positions in covering arrays. By choosing a specific set of such positions to change to $\star$, and then replacing these by elements again, we form a new covering array with a possibly different collection of possible don't care positions. By itself this is of no use other than to produce many covering arrays with the same parameters. However, in some cases we can form an entire row containing only don't care positions. When this occurs, the row is not needed and can be deleted. This is the sense in which the covering array is improved, by the deletion of rows.

2.1 Finding Don't Care Positions

To find possible don't care positions, it suffices to determine the numbers of times that the $\binom{k}{t}v^t$ t-way interactions are covered. For each of the Nk entries, check whether the entry appears in any t-way interaction that is covered only once. If not, it is a possible don't care position. While conceptually simple, this requires space proportional to $\binom{k}{t}v^t$, which is too much in practice. Instead initially mark each of the Nk entries as a possible don't care. Then for each of the $\binom{k}{t}$ sets of columns in turn, use a vector of length v^t to record the number of times each of the t-way interactions arises in the t chosen columns. Then for each that arises only once, mark all t positions in it to be no longer don't care. This requires only $Nk + v^t$ space, but still requires time proportional to $tN\binom{k}{t}$. At the same time, one can verify that the array is in fact a covering array, by ensuring that every t-way interaction is seen at least once. Unfortunately, if we change any one of the possible don't care positions to $\star$, some recomputation is then needed.

To find a set of don't care positions that can all be simultaneously changed to $\star$, we use the fact that rows are recorded in a specific order. For every set of t columns we consider the rows of the CA in order; when a t-tuple is covered for the first time we mark its t positions as *necessary*. After every possible set of t factors is treated, all positions that are not necessary can be changed to $\star$. This can be done in the same time and space as the identification of all possible don't care positions.

Once done, each row may have any number of $\star$ entries from 0 to $k-t$ or may consist entirely of don't care positions. When the latter occurs, this row can be removed without reducing the strength of the CA.

2.2 Choosing a Row to Eliminate

In some cases, simply identifying don't care positions enables us to remove a row, but this is atypical unless the CA is very far from optimal. Therefore we attempt to produce more don't care positions in one row by using don't care positions in others, with the objective of generating an entire row of don't care positions. Thus we wish to select a row that can be 'easily' removed. A natural selection is a row that has the most don't care positions already. Perhaps a more

appropriate selection would be the row in which the number of multiply covered t-tuples is largest. When $\star$ entries are present, however, replacing the $\star$ by a value results in a substantial change in this statistic. For this reason, one should calculate, for a row with ℓ $\star$ entries, the quantity $\sum_{i=1}^{\ell} \binom{\ell}{i}\binom{k-\ell}{t-i}$ *plus* the number of multiply covered t-tuples, and select a row that maximizes this quantity. This would require substantially more computation, so a simple count of don't care positions is used here.

2.3 Algorithm

Having nominated a row for possible elimination, we move the nominated row to be the last row of the CA. We now use don't care positions in other rows in an attempt to introduce (eventually) further don't care positions in the nominated row. A simple strategy is to consider each entry of the nominated row that is not $\star$, locate all $\star$ positions in the same column, and replace each by the entry in the nominated row. This can result in t-way interactions that were covered only in the last row also being covered earlier, and can therefore result in new don't care positions in the last row. In our experiments we found this simple strategy to be too restrictive; while it can produce new don't care positions in the last row, it often fails to produce much change in the pattern of don't care positions in the rest of the array. We therefore adopt a less restrictive approach. For each $\star$ position, if the nominated row does not contain $\star$ in that column, we replace the $\star$ with the value from the nominated row; otherwise we select a value at random to replace the $\star$.

One iteration typically produces a different covering array from the one given as input. However, if we simply find don't care positions again, often the set is very similar to the one just used, and consequently the method stalls quickly. Instead we randomly reorder all rows except the last. Then finding don't care positions typically yields a different set – but in the last row, all positions that were don't care positions remain so. Of course, another row that previously had fewer don't care positions than the nominated row may now have more; if it does, it becomes the nominated row and is moved to the bottom.

Arguably, one should be more clever in filling the don't care positions, and in reordering the rows. Perhaps this is so, but in our experience the randomness of these two choices is crucial. Whatever choices are made, it can happen that the same row is nominated at each step, but no row reordering of the remaining rows yields a set of $\star$ positions that result in an improvement of the nominated row (i.e., more $\star$ positions).

2.4 Escaping Local Optima

The decision that the CA is unlikely to be improved from its current state can be done by monitoring the total number of don't care positions in the array, or the number in the nominated row, and abandoning the nominated row when it is 'too long' since the number has improved. We use the number in the nominated row, and set a threshold on the number of iterations permitted without improvement. When we exceed the threshold, we take this as evidence that the search has

converged to a local optimum. We employ a simple method of escaping. We move the nominated row along with any other row that contains a don't care position to the top of the CA, fill all the $\star$ positions with random values and start with this revised array. This could result in a major change in the state of the CA, and indeed the next row nominated may have substantially fewer don't care positions than the one just abandoned.

2.5 Implementation and Scalability

The escape from local optima permits us to start from one CA and produce a very different one. Therefore multiple processes can execute simultaneously, all working from a single start point and exploring different areas of the search space. Once an improvement has been made by one of the tasks the result can be shared with the others as the new starting point. An effective way to check for improvements among all processes uses an 'Allgather' operation, in which every process shares its current number of rows with the others. If there is a difference between the minimum and maximum of the values then the best result is broadcast from the lowest ranking process with the best result. A reasonable amount of time, at least sufficient for one iteration to complete, must be dedicated to searching for an improvement before communicating with other processes. We have implemented the method both in a sequential setting and in a parallel one as outlined.

3 Results

Perhaps the biggest surprise is that the algorithm works at all. Previously the best result for CAN(6,8,5) is the upper bound 32822 from IPOG-F [31]. Starting with this array, our method eliminates 4034 rows to show that CAN(6,8,5)$\leq$ 28788 in *one minute* of computation; in ten minutes it reduces to 27909 rows; in one hour to 27772; and in five hours to 27717. While five hours may be longer than one wishes to spend, one minute to remove 12.3% of the rows appears well worth the effort! (All times reported here are for an 8-core Intel Xeon processor clocked at 2.66GHz with 4MB of cache, bus speed 1.33GHz, and 16GB of memory. Only one core is used when timing is reported. The program is coded in C++.)

A striking example is the well studied case $\mathsf{CA}(N; 2, 20, 10)$. In the announcement of AETG [25], $\mathsf{CAN}(2, 20, 10) \leq 180$ is stated, but no explicit description is given. Yet the commercial implementation of AETG reports 198 rows. A recent paper by Calvagna and Gargantini [35] reports bounds on CAN(2,20,10) from ten methods; other than the bound of 180 reported by AETG [25], the remaining methods give bounds of 193, 197, 201, 210, 210, 212, 220, 231, and 267. Metaheuristic search using simulated annealing [22] yields 183 rows. Two combinatorial constructions both using a 1-rotational automorphism [13, 14] yield 181 rows. Finally it was shown that $\mathsf{CAN}(2, 20, 10) \leq 174$ using a double projection technique [13]. In Table 1 we apply postoptimization to seven covering arrays; we give the method used to produce a $\mathsf{CA}(N_{\text{old}}; 2, 20, 10)$, the number

Table 1. Postoptimization on $\mathsf{CA}(N; 2, 20, 10)$s

Method	N_{old}	N_{new}	Poss. $\star$	$\star$
`TCG`	217	198	444	256
`IPO`	212	196	449	285
density	203	195	170	79
`AETG`	198	190	195	132
annealing	183	183	13	3
1-rotational	181	181	0	0
double projection	178	162	415	146

N_{new} of rows after postoptimization, and the numbers of possible don't care and don't care positions. The best establishes that $\mathsf{CAN}(2, 20, 10) \leq 162$; five of the seven improve, but those from simulated annealing and the 1-rotational solution see no improvement. The improvement on $\mathsf{CAN}(2,20,10)$ is remarkable, given the variety of methods that have been previously applied to try to improve this bound.

We therefore consider projection further. In [13], a construction of Stevens, Ling, and Mendelsohn [36] is generalized to a projection technique that produces a $\mathsf{CA}(q^2 - x; 2, q + 1 + x, q - x)$ from an $\mathsf{OA}(q^2; 2, q + 1, q)$ when q is a prime power and $x \geq 0$. It is so named because x symbols of the OA are 'projected' to form x new columns. (See [13] for details.) There it is observed that x symbols can be projected to form $2x$ new columns (a 'double projection'), but the result is no longer a covering array. Rather it is a partial covering array that leaves many pairs uncovered, but also contains many don't care positions. A general pattern to complete this partial array while adding few rows is elusive, if indeed one exists at all. We therefore employ this partial covering array as a 'seed' and complete it using the density algorithm [27]. We found that treating all uncovered pairs equally, as density does, results in the addition of many rows (for example, for the partial $\mathsf{CA}(166; 2, 20, 10)$, as many as 50 new rows). Therefore we modified the greedy selection in density to weight uncovered pairs on columns $\{q+1, \ldots, q+2x\}$ highest, pairs with one column from $\{q+1, \ldots, q+2x\}$ next, and pairs with neither column from $\{q+1, \ldots, q+2x\}$ least; then density selects the largest total weight of uncovered pairs. This remains a greedy heuristic; nevertheless, it adds as few as 12 rows to complete the partial $\mathsf{CA}(166; 2, 20, 10)$.

Using projection and double projection on the $\mathsf{OA}(q^2; 2, q + 1, q)$ for $q \in \{13, 16, 17, 19\}$ and completing with the weighted density method, we formed numerous covering arrays and applied postoptimization to each. When $x > 1$, each saw a reduction in the number of rows, sometimes dramatic. In Table 2 we report the new bounds obtained. The value in parentheses is the number of rows of the CA prior to postoptimization.

One expects that the rows added by density are less effective in the coverage of pairs than the rows of the OA to which double projection are applied. Surprisingly, postoptimization can succeed in eliminating so many rows that at the end fewer than $q^2 - x$ remain!

Table 2. Covering Arrays from Double Projection

t	v	k	$\mathsf{CAN}(t,k,v) \leq$	Old Bound	t	v	k	$\mathsf{CAN}(t,k,v) \leq$	Old Bound
2	10	17	152 (166)	154 [14]	2	10	18	155 (178)	163 [14]
2	10	19	159 (178)	172 [14]	2	10	20	162 (178)	174 [13]
2	10	21	171 (189)	190 [14]	2	10	22	184 (195)	191 [7]
2	11	18	180 (193)	181 [14]	2	12	16	192 (219)	199 [14]
2	13	20	246 (253)	253 [14]	2	14	19	253 (254)	254 [13]
2	14	21	279 (286)	286 [13]	2	14	24	310 (387)	313 [14]
2	15	24	343 (357)	357 [13]	2	16	23	353 (358)	358 [13]

Table 3. Covering Arrays from Density

k	New	Old	k	New	Old	k	New	Old	k	New	Old	k	New	Old
$\mathsf{CA}(N;4,k,3)$														
11	211	230	17	300	312	24	377	389	31	440	446	32	445	454
33	454	461	34	462	468	40	499	504	41	506	510	42	509	513
43	518	522	44	522	526	45	526	530	46	530	534	47	534	538
48	542	546	52	560	562	53	565	567	54	568	572	55	572	575
56	578	581	57	581	584	58	585	588	59	589	592	61	598	601
63	604	607	64	612	614	66	618	620	70	627	629			
$\mathsf{CA}(N;5,k,2)$														
11	82	86	12	89	92	13	95	103	14	103	110	15	110	115
16	117	123	18	127	135									
$\mathsf{CA}(N;6,k,2)$														
9	118	120	10	144	150	11	167	178	12	184	190	16	258	270
18	294	309	19	309	323	20	327	337	21	341	352	22	355	362
23	371	377	33	496	503	34	502	508	35	510	516	36	525	529
37	534	541												

Now we consider arrays from the density method [28,37]. We treat a few specific values of t and v. In Table 3, each input array $\mathsf{CA}(N;4,k,3)$, $\mathsf{CA}(N;5,k,2)$, and $\mathsf{CA}(N;6,k,2)$ is from density [28,37], and postoptimization is run for 10 minutes (on a single core). The wall clock time limit results in many more iterations being completed when k is small; we expect that this is the primary reason for the larger improvements for few factors. Two of the 'old' bounds ($\mathsf{CAN}(5,12,2) \leq 92$ and $\mathsf{CAN}(5,14,2) \leq 110$) are from [16]. For $t = 6$, the 'old' bounds are from [20] when $k = 9$, a greedy method of Kuliamin [38] when $k = 10$, PaintBall [32] when $k \in \{11,12,16\}$, and density [28,37] otherwise. All of the new bounds are obtained by postoptimization of CAs from density.

It appears that postoptimization is applicable to covering arrays from a number of sources, but there are cases where it has no effect. Indeed we applied postoptimization to all of the arrays found by Nurmela [23] using tabu search, and none improved. We applied postoptimization to numerous arrays found by Cohen [22] using simulated annealing, and none improved.

We report one more successful application next. Colbourn and Kéri [17] recently employed Hadamard matrices to establish that $\mathsf{CAN}(4, 20, 2) \leq 40$, $\mathsf{CAN}(4, 32, 2) \leq 64$, and $\mathsf{CAN}(4, 36, 2) \leq 72$; previously the best known bounds were $\mathsf{CAN}(4, 20, 2) \leq 55$ [9], $\mathsf{CAN}(4, 32, 2) \leq 73$ [38], and $\mathsf{CAN}(4, 36, 2) \leq 95$ [9]. Applying postoptimization to the Hadamard matrix solutions improve these to establish that $\mathsf{CAN}(4, 20, 2) \leq 39$, $\mathsf{CAN}(4, 32, 2) \leq 59$, and $\mathsf{CAN}(4, 36, 2) \leq 66$.

4 Using Postoptimization in Practice

Arguably the success of postoptimization is evidence of our limited understanding of covering arrays. Indeed the restrictions on applicability of combinatorial constructions have forced us to consider computational search for 'small' covering arrays both to provide best known small arrays, and to serve as ingredient arrays in recursions. However our ability to carry out computations is limited. To illustrate this, consider strength $t = 4$ using [33]. Among the best known arrays, only the bounds $\mathsf{CAN}(4, 13, 2) \leq 34$ [39], $\mathsf{CAN}(4, 6, 3) \leq 111$ [39], $\mathsf{CAN}(4, 7, 3) \leq 126$ [39], $\mathsf{CAN}(4, 8, 3) \leq 153$ [22], $\mathsf{CAN}(4, 6, 4) \leq 375$ [22], $\mathsf{CAN}(4, 7, 6) \leq 1893$ [39], and $\mathsf{CAN}(4, 8, 6) \leq 2068$ [39] are produced by simulated annealing. *None* have been produced by tabu search, constraint satisfaction, or other metaheuristic search techniques. The workhorses of computation are the greedy methods; both density [28] and IPO [30, 31] yield numerous best known covering arrays of strength four. IPO, for example, yields the best known $\mathsf{CA}(207; 4, 599, 2)$, $\mathsf{CA}(1050; 4, 445, 3)$, $\mathsf{CA}(3170; 4, 308, 4)$, $\mathsf{CA}(7145;4,208,5)$, and $\mathsf{CA}(13983;4,163,6)$, along with many arrays with fewer columns. Some direct constructions that limit or eliminate the computation provide sporadic results, but the rest of our knowledge rests on recursions.

What explains the prevalence of greedy computations among the best known results? It is very unlikely that simulated annealing or tabu search would not yield better results, if either could be run for an adequate period of time. That is precisely the problem, however. Neither has been implemented so as to find competitive solutions starting from scratch within a time frame that anyone is willing to invest. Yet neither is configured so as to take an existing covering array and improve it by removing rows. Indeed both have been devised to improve a partial covering array to make it cover more and more t-way interactions within a specified number of rows. Hence if the time allocated is insufficient, these metaheuristic search methods end with an array that is still not a covering array. The fundamental difference in postoptimization is that at every stage we are dealing with a covering array, not a partial one. This focuses the search much more than is typically done with simulated annealing or tabu search.

This suggests the main merit of using postoptimization. In using a greedy approach, or a recursion that may have poor ingredients, we do not expect to produce a covering array whose size is close to the minimum. But we can produce such an array quickly for a wide range of parameters. And having produced it, we can invest time in postoptimizing the array, stopping at any time with the assurance that a covering array is produced. This appears to be a practical

solution to the problem of balancing the time to produce a test suite (covering array) and the time to execute the tests. Within a total time budget for testing, it suggests the feasibility of investing less time in the initial construction of the tests while exploiting the (relatively) fast operation of postoptimization to reduce the time for test execution.

Postoptimization also plays a role in producing the smallest arrays known, as we have seen. Naturally it would be of interest to be able to predict the extent to which postoptimization will be successful. This could help us decide when to try postoptimization. Perhaps more importantly, it would suggest criteria to construct covering arrays that are amenable to postoptimization. Consider Table 1 for the widely studied case $\mathsf{CA}(N; 2, 20, 10)$. Obviously the repetition of coverage in the larger arrays is greater in total, yet the size of the input array does not serve as a good predictor of the improvement seen. In these results, the number of possible don't care positions appears to be the key. Certainly the presence of possible don't care positions is necessary for improvement. However, we believe that the distributions of possible don't care positions among the rows and columns of the array also affect the extent of improvement. Moreover, the patterns of positions that can be realized simultaneously as don't care positions may be more relevant than the pattern of possible don't care positions. Nevertheless, using the number of possible don't care positions as a preliminary indicator of the potential improvement appears worthwhile.

5 Conclusion

It comes as no surprise that many of the covering arrays that are best known at present are far from optimal. In these cases, postoptimization provides a relatively fast method for detecting and exploiting duplication of coverage in order to improve the arrays. More surprising are the cases in which postoptimization improves on a result that is already better than those obtained from heuristic search, as we saw with double projection and with arrays from Hadamard matrices. In these cases, the reason for success does not appear to the poor quality of the initial array. While duplication of coverage is necessary in all arrays with $N > v^t$, the distributions of numbers of times that a t-way interaction is covered can vary widely from interaction to interaction. This can result in certain cells or rows being more effective in coverage than are others. By focusing on arrays in which the contributions of cells or rows are quite unbalanced, postoptimization is sometimes able to eliminate the need for a cell, and perhaps an entire row.

The main benefits of postoptimization are that it does not depend on a particular construction technique; iterations can be executed in approximately the same time as needed to check that the array is in fact a covering array; and that it can be executed for as many iterations as desired, with the assurance that whenever it is stopped, the array is a covering array. At present the main limitations are that it does not appear to be effective for certain covering arrays such as those produced by metaheuristic search; and that the extent of improvement that one can expect cannot be reliably predicted. Despite these limitations,

postoptimization has already proved to be an easy and effective means to improve a wide variety of covering arrays.

Acknowledgements

Research of the second author is supported by DOD grant N00014-08-1-1070.

References

1. Colbourn, C.J.: Combinatorial aspects of covering arrays. Le Matematiche (Catania) 58, 121–167 (2004)
2. Hartman, A.: Software and hardware testing using combinatorial covering suites. In: Golumbic, M.C., Hartman, I.B.A. (eds.) Interdisciplinary Applications of Graph Theory, Combinatorics, and Algorithms, pp. 237–266. Springer, Norwell (2005)
3. Hedayat, A.S., Sloane, N.J.A., Stufken, J.: Orthogonal Arrays. Springer, New York (1999)
4. Godbole, A.P., Skipper, D.E., Sunley, R.A.: t-covering arrays: upper bounds and Poisson approximations. Combinatorics, Probability and Computing 5, 105–118 (1996)
5. Katona, G.: Two applications (for search theory and truth functions) of Sperner type theorems. Periodica Math. 3, 19–26 (1973)
6. Kleitman, D., Spencer, J.: Families of k-independent sets. Discrete Math. 6, 255–262 (1973)
7. Colbourn, C.J., Martirosyan, S.S., Mullen, G.L., Shasha, D.E., Sherwood, G.B., Yucas, J.L.: Products of mixed covering arrays of strength two. Journal of Combinatorial Designs 14(2), 124–138 (2006)
8. Cohen, M.B., Colbourn, C.J., Ling, A.C.H.: Constructing strength three covering arrays with augmented annealing. Discrete Math. 308, 2709–2722 (2008)
9. Colbourn, C.J., Martirosyan, S.S., van Trung, T., Walker II, R.A.: Roux-type constructions for covering arrays of strengths three and four. Designs, Codes and Cryptography 41, 33–57 (2006)
10. Martirosyan, S.S., Colbourn, C.J.: Recursive constructions for covering arrays. Bayreuther Math. Schriften 74, 266–275 (2005)
11. Martirosyan, S.S., van Trung, T.: On t-covering arrays. Des. Codes Cryptogr. 32, 323–339 (2004)
12. Colbourn, C.J.: Distributing hash families and covering arrays. J. Combin. Inf. Syst. Sci. (to appear)
13. Colbourn, C.J.: Strength two covering arrays: Existence tables and projection. Discrete Math. 308, 772–786 (2008)
14. Meagher, K., Stevens, B.: Group construction of covering arrays. J. Combin. Des. 13(1), 70–77 (2005)
15. Sherwood, G.B., Martirosyan, S.S., Colbourn, C.J.: Covering arrays of higher strength from permutation vectors. J. Combin. Des. 14(3), 202–213 (2006)
16. Walker II, R.A., Colbourn, C.J.: Tabu search for covering arrays using permutation vectors. J. Stat. Plann. Infer. 139, 69–80 (2009)
17. Colbourn, C.J., Kéri, G.: Covering arrays and existentially closed graphs. In: Xing, C., et al. (eds.) IWCC 2009. LNCS, vol. 5557, pp. 22–33. Springer, Heidelberg (2009)

18. Chateauneuf, M.A., Kreher, D.L.: On the state of strength-three covering arrays. J. Combin. Des. 10(4), 217–238 (2002)
19. Johnson, K.A., Entringer, R.: Largest induced subgraphs of the n-cube that contain no 4-cycles. J. Combin. Theory Ser. B 46(3), 346–355 (1989)
20. Johnson, K.A., Grassl, R., McCanna, J., Székely, L.A.: Pascalian rectangles modulo m. Quaestiones Math. 14(4), 383–400 (1991)
21. Tang, D.T., Chen, C.L.: Iterative exhaustive pattern generation for logic testing. IBM Journal Research and Development 28(2), 212–219 (1984)
22. Cohen, M.B.: Designing test suites for software interaction testing. PhD thesis, The University of Auckland, Department of Computer Science (2004)
23. Nurmela, K.: Upper bounds for covering arrays by tabu search. Discrete Applied Mathematics 138(9), 143–152 (2004)
24. Hnich, B., Prestwich, S., Selensky, E., Smith, B.M.: Constraint models for the covering test problem. Constraints 11, 199–219 (2006)
25. 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(7), 437–444 (1997)
26. Tung, Y.W., Aldiwan, W.S.: Automating test case generation for the new generation mission software system. In: Proc. 30th IEEE Aerospace Conference, pp. 431–437. IEEE, Los Alamitos (2000)
27. Bryce, R.C., Colbourn, C.J.: The density algorithm for pairwise interaction testing. Software Testing, Verification, and Reliability 17, 159–182 (2007)
28. Bryce, R.C., Colbourn, C.J.: A density-based greedy algorithm for higher strength covering arrays. Software Testing, Verification, and Reliability 19, 37–53 (2009)
29. Tai, K.C., Yu, L.: A test generation strategy for pairwise testing. IEEE Transactions on Software Engineering 28(1), 109–111 (2002)
30. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG: A general strategy for t-way software testing. In: Fourteenth Int. Conf. Engineering Computer-Based Systems, pp. 549–556 (2007)
31. Forbes, M., Lawrence, J., Lei, Y., Kacker, R.N., Kuhn, D.R.: Refining the in-parameter-order strategy for constructing covering arrays. J. Res. Nat. Inst. Stand. Tech. 113(5), 287–297 (2008)
32. Kuhn, D.R., Lei, Y., Kacker, R., Okun, V., Lawrence, J.: Paintball: A fast algorithm for covering arrays of high strength. Internal Tech. Report, NISTIR 7308 (2007)
33. Colbourn, C.J.: Covering array tables, `http://www.public.asu.edu/~ccolbou/src/tabby`, 2005–present
34. Sherwood, G.: Effective testing of factor combinations. In: Proc. 3rd Int'l Conf. Software Testing, Analysis and Review, Software Quality Eng. (1994)
35. Calvagna, A., Gargantini, A.: IPO-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays. In: Proc. Fifth Workshop on Advances in Model Based Testing (to appear, 2009)
36. Stevens, B., Ling, A., Mendelsohn, E.: A direct construction of transversal covers using group divisible designs. Ars Combin. 63, 145–159 (2002)
37. Linnemann, D., Frewer, M.: Computations with the density algorithm (private communication by e-mail) (October 2008)
38. Kuliamin, V.V.: Private communication by e-mail (February 2007)
39. Soriano, P.P.: Private communication by e-mail (March 2008)