12-8-2022

Algorithmic Methods for Covering Arrays of Higher Index

Ryan Dougherty
United States Military Academy, ryan.dougherty@westpoint.edu

Kristoffer Kleine
MATRIS, SBA Research, kris.kleine@yahoo.de

Michael Wagner
MATRIS, SBA Research, mwagner@sba-research.org

Charles J. Colbourn
Arizona State University, charles.colbourn@asu.edu

Dimitris E. Simos
MATRIS, SBA Research, DSimos@sba-research.org

Journal of Combinatorial Optimization (2023) 45:28
https://doi.org/10.1007/s10878-022-00947-x

Algorithmic methods for covering arrays of higher index

Ryan E. Dougherty[1] · Kristoffer Kleine[2] · Michael Wagner[2] · Charles J. Colbourn[3] · Dimitris E. Simos[2]

Accepted: 27 September 2022

Abstract
Covering arrays are combinatorial objects used in testing large-scale systems to increase confidence in their correctness. To do so, each interaction of at most a specified number t of factors is represented in at least one test; that is, the covering array has strength t and index 1. For certain systems, the outcome of running a test may be altered by variability of the interaction effect or by measurement error of the test result. To improve the efficacy of testing, one can ensure that each interaction of t or fewer factors is represented in at least λ tests. When $\lambda > 1$, this leads to covering arrays of higher index. We explore two algorithmic methods for constructing covering arrays of higher index. One is based on the in-parameter-order algorithm, and the other employs a conditional expectation paradigm. We compare these two by performing experiments on real-world benchmarks and on uniform parameter sets.

Keywords Covering array · Conditional expectation · In-parameter-order algorithm · Software testing

✉ Dimitris E. Simos
DSimos@sba-research.org

Ryan E. Dougherty
ryan.dougherty@westpoint.edu

Kristoffer Kleine
kris.kleine@yahoo.de

Michael Wagner
mwagner@sba-research.org

Charles J. Colbourn
Charles.Colbourn@asu.edu

1 Department of Electrical Engineering and Computer Science, United States Military Academy, West Point, NY, USA

2 MATRIS, SBA Research, Floragasse 7, 1040 Vienna, Austria

3 School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Published online: 08 December 2022

1 Introduction

Informally, we are concerned with systems that have k *factors* $\{F_1, \ldots F_k\}$ that may affect correctness, individually or by interactions among the factors. Each factor F_i has a fixed, finite number v_i of *levels* (or *values* or *options*) that determine the manner in which the factor is set; we always have $v_i \geq 2$. When each of the k factors is assigned one of its admissible levels, we obtain a *test*. Our objective is to choose a set of N tests so that, by running each test and examining the responses, we can gain confidence that the system is operating as required.

A formal model is developed next. Let N, t, k, and λ be positive integers with $k \geq t \geq 2$ and $\lambda \geq 1$. Let $v_1, \ldots, v_k$ be positive integers with $v_i \geq 2$ for $1 \leq i \leq k$. Let A be an array with N rows, k columns, in which each column contains symbols from a v_i-ary alphabet Σ_i. (Symbols in the alphabets are arbitrary, and can be driven by the intended application.) For every t-tuple $(c_1, \ldots, c_t)$ of distinct column indices and every t-tuple $(a_1, \ldots, a_t) \in \Sigma_{c_1} \times \cdots \times \Sigma_{c_t}$, the set $I = \{(c_i, a_i) : 1 \leq i \leq t\}$ is an *interaction* of *strength* t, or a t*-way interaction.* Array A s*-covers* interaction I if there exist (at least) s distinct row indices $r_1, \cdots, r_s$ such that $A(r_m, c_i) = a_i$ for all $1 \leq i \leq t$ and $1 \leq m \leq s$. When every t-way interaction is λ-covered, A is a *mixed-level covering array* of *strength* t and *index* λ, denoted by $\mathsf{MCA}_\lambda(N; t, (v_1, \ldots, v_k))$. When $v_1 = \cdots = v_k = v$, A is *uniform*, and is a *covering array*, denoted $\mathsf{CA}_\lambda(N; t, k, v)$. The most frequently studied situations arise when $\lambda = 1$, for which we adopt the simpler notation of $\mathsf{CA}(N; t, k, v)$. Naturally we are concerned with running as few tests as possible. The *covering array number*, $\mathsf{CAN}_\lambda(t, k, v)$, is the smallest N for which a $\mathsf{CA}_\lambda(N; t, k, v)$ exists.

An example is provided in Table 1, in which $N = 14, t = 3, k = 6, v = 2$, and $\lambda = 1$. In columns 2, 4, and 5, we box all 8 interactions that must appear in these columns; because $N > 8$, some interactions appear more than once. In each set of 3 columns all 8 interactions are covered at least once, and so we have a $\mathsf{CA}(14; 3, 6, 2)$. This example establishes that $\mathsf{CAN}_1(3, 6, 2) \leq 14$.

Table 1 illustrates the application of covering arrays to testing of large-scale systems; each column corresponds to a factor, and each row represents a configuration of the system that can be used as a test. To evaluate the system, a tester runs each test, resulting in either "Pass" or "Fail" for each. If a fault of size at most t exists within the system, then the covering array will detect that such a fault exists.

Such an outcome does not ensure that we can determine the number of faults, or the set of faulty interactions.

Detecting and locating arrays were introduced in Colbourn and McClary (2008) as a specialization of covering arrays to support the localization of faults; see Colbourn and Syrotiuk (2018) and Martínez et al. (2009/10), and for an application see Aldaco et al. (2015). Although detecting arrays impose further combinatorial requirements, a primary requirement is that the underlying covering array have higher index.

There are further reasons to ask for higher index. For example, executing tests may fail to produce a viable response due to environmental issues. Then if an interaction is covered in a single test, its effect cannot be observed. More seriously, an interaction effect may cause intermittent faults; then the probability of its being detected depends on the number of tests in which it is covered. In order to guard against the loss of test

Table 1 An example test suite (left), and the corresponding covering array (right), provided by NIST

OS	Processor	Browser	Display Ports	Network	Battery Level		
Win	Intel	Chrome	DVI	Ethernet	Low		0 0 0 1 0 1
Win	Intel	Firefox	HDMI	Wifi	High		0 0 1 0 1 0
Win	AMD	Chrome	HDMI	Ethernet	Low		0 1 0 0 0 1
Win	AMD	Firefox	DVI	Ethernet	High		0 1 1 1 0 0
Mac	Intel	Chrome	HDMI	Wifi	Low		1 0 0 0 1 1
Mac	Intel	Firefox	HDMI	Ethernet	High		1 0 1 0 0 0
Mac	AMD	Chrome	DVI	Wifi	High	$\Leftrightarrow$	1 1 0 1 1 0
Mac	AMD	Firefox	HDMI	Wifi	Low		1 1 1 0 1 1
Mac	Intel	Firefox	DVI	Wifi	Low		1 0 1 1 1 1
Mac	AMD	Chrome	DVI	Ethernet	Low		1 1 0 1 0 1
Win	AMD	Chrome	DVI	Wifi	Low		0 1 0 1 1 1
Win	Intel	Chrome	DVI	Ethernet	High		0 0 0 1 0 0
Win	$\star$	Firefox	$\star$	Ethernet	Low		0 $\star$ 1 $\star$ 0 1
$\star$	AMD	Chrome	HDMI	$\star$	High		$\star$ 1 0 0 $\star$ 0

The array has 14 rows, 6 columns, two values for each column, and covers all 3-way combinations at least once. Entries that are $\star$ indicate "don't care" entries (i.e., they can be set to any value and would not affect whether the array is a covering array) results and the effects of intermittent interactions, covering arrays of higher index are of interest.

There is a substantial literature on the construction of covering arrays with index 1; see, for example, Colbourn (2011), Kuhn et al. (2013) and Nie and Leung (2011). Current research on arrays of higher index is limited (see, for example, Akhtar et al. (2021)). In order to address needs for additional interaction coverage, naturally one could simply repeat each test λ times using a covering array of index 1. In Sect. 2 we show that this naïve strategy often runs far more tests than needed; in the process we determine the extent of potential improvements over simple replication in order to identify parameter sets on which to evaluate construction techniques. In Sect. 3 we develop an IPO-style (one-column-at-a-time) algorithm, and in Sect. 4 we develop a conditional expectation (one-row-at-a-time) algorithm. In Sect. 5 we present computational results from these algorithms applied to a variety of parameter sets, both from real-world applications and from a range of uniform situations. In Sect. 6 we evaluate the results obtained.

2 Asymptotic upper bounds on higher-index covering arrays

In order to understand the relationship between the number of tests in covering arrays of index one and those of index λ, we first examine the asymptotics of the sizes for MCA_λs.

Very few exact covering array numbers are known; consequently, these are typically bounded by considering the probability that a random array is a covering array with the desired parameters.

We employ the well-known probabilistic method (Alon and Spencer 2004); see also Deng et al. (2004). Let N, t, k, and λ be positive integers with $t \leq k$, and let $v_1, \ldots, v_k$ be positive integers, each at least 2. Let A be an $N \times k$ array in which the entries of column i are chosen independently, and uniformly at random, from a set of v_i symbols. What is the probability that a specific t-way interaction I on columns $\mathcal{I_C} = (c_1, \ldots, c_t)$ is not λ-covered in A? The probability that I is covered in a single row is $\phi_{t,I} = \prod_{c \in \mathcal{I_C}} \frac{1}{v_c}$, and so the probability that it is not covered in any row is $(1 - \phi_{t,I})^N$. We write ϕ and $1 - \phi$ when the parameters are clear from the context. The number of rows that do not cover interaction I is equal to ρ with probability $\binom{N}{\rho}(1-\phi)^\rho \phi^{N-\rho}$. Define $\psi_{N,t,I,\lambda}$ to be $\sum_{\rho=0}^{\lambda-1} \binom{N}{\rho}(1-\phi)^\rho \phi^{N-\rho}$. By the linearity of expectation, the expected number of interactions that are not λ-covered in A is precisely $\sum_I \psi_{N,t,I,\lambda}$. When this expected number is strictly less than 1, an $\mathsf{MCA}_\lambda(N; t, (v_1, \ldots, v_k))$ exists.

We outline the proof of an asymptotically optimal bound, referring the reader to Dougherty (2019) for further discussion.

Theorem 1 *Let $v_1, \ldots, v_k, t$ be fixed. For $\lambda \geq 1$, k sufficiently large, and any $MCA_\lambda(N; t, (v_1, \ldots, v_k))$ with N minimum, N is $\Theta(\log k + \lambda)$, where the hidden constants are independent of k, λ (but may depend on $v_1, \ldots, v_k, t$).*

Proof (Sketch) For the lower bound, $\mathsf{CAN}_1(t, k, v)$ is $\Omega(\log k)$ Colbourn (2004), and deleting any $\lambda - 1$ rows from covering array of index λ yields a covering array of index one. Hence $\mathsf{CAN}_\lambda(t, k, v)$ is $\Omega(\log k + \lambda)$. For the upper bound, the expected number of interactions that are not λ-covered in an $N \times k$ array, with entries chosen uniformly at random, is $\binom{k}{t} v^t \psi_{N,t,v,\lambda}$, where $\psi_{N,t,v,\lambda} = \sum_{\rho=0}^{\lambda-1} \binom{N}{\rho} (1-\phi)^\rho \phi^{N-\rho}$. We obtain an upper bound on $\psi_{N,t,v,\lambda}$ by applying the Cauchy-Schwarz inequality.

It suffices to obtain an upper bound on N in the following equation:

$$\binom{k}{t} v^t (1-\phi)^{N-\lambda} \left(\frac{eN}{\lambda}\right)^\lambda = 1.$$

We use the Lambert W-function $W(x)$ (W is the inverse of the function $f(W) = We^W$) to obtain:

$$N \le \frac{\lambda}{\log(1-\phi)} W\left(\frac{\log(1-\phi)}{e(\binom{k}{t} v^t (1-\phi))^{1/\lambda}}\right).$$

By Alzahrani and Salem (2018), N is at most $C \log k + D\lambda$ (for constants C, D depending only on v and t), and hence is $O(\log k + \lambda)$. The extension to MCA_λs is routine, by considering covering arrays on $\min(v_1, \ldots, v_k)$ and on $\max(v_1, \ldots, v_k)$ symbols. □

For certain parameters, one could instead exploit concentration inequalities as in Alon and Gutner (2007) to establish that within a t-set of columns, the difference between the number of times that one interaction is covered and the number of times that another is covered is 'small' with high probability. Requiring this to hold for all t-sets of columns, each with large enough probability, would establish a lower bound on the number of times each interaction is covered for some covering array. We do not pursue this approach here.

In Sect. 4, our conditional expectation algorithm guarantees to meet the asymptotic bound of Theorem 1. However, the asymptotics of Theorem 1 can be quite misleading when $k \approx t$. Indeed, using an approach from Ray-Chaudhuri and Singhi (1988), one obtains:

Theorem 2 *Let $k, v_1, \ldots, v_k, t$ be fixed integers such that $v_1 \ge \cdots \ge v_k$ and $k \ge t$. Then there is a sufficiently large constant λ_0 such that for any $\lambda \ge \lambda_0$, there is an $\mathsf{MCA}_\lambda(N; t, (v_1, \ldots, v_k))$ having $N = \lambda \prod_{i=1}^t v_i$.*

In summary, if the number of columns is sufficiently large, relatively few additional rows are needed; and if the number of columns is sufficiently small, relatively many more rows are needed. Scenarios of most interest therefore arise when k is "moderately large." Moreover, by considering Theorem 2 and also the need to run relatively few tests, cases of most interest arise when λ is a "small" constant.

3 IPOG family

The In-Parameter-Order (IPO) strategy for covering array generation was introduced in Lei and Tai (1998) for pairwise testing and later generalized to arbitrary strengths (Lei et al. 2007). With this strategy, a covering array is constructed incrementally using horizontal and vertical extension steps; see Algorithm 1. In horizontal extension, a new column is added to the array and its values are assigned greedily to maximize the number of newly covered interactions. If uncovered interactions remain after horizontal extension, the algorithm attempts to cover all missing interactions by performing vertical extension. This process is repeated until a CA with the desired number of columns is constructed. In this paper, we consider three prominent algorithms of the IPO family: IPOG, IPOG- F and IPOG- F2, as detailed in Forbes et al. (2008).

Algorithm 1 IPO Algorithm

```
Array ← cross-product of first t columns
for i ← t + 1, . . . , k do
    Array ← HorizontalExtension(Array, i)
    if there are uncovered interactions then
        Array ← VerticalExtension(Array, i)
    end if
end for
```

Vertical Extension

The vertical extension step is equivalent for all considered IPO variants. Its purpose is to make sure all interactions are covered, if necessary by adding additional rows. First, for each missing interaction, existing rows are examined in an effort to find one to which the interaction can be added. This can be done if all entries in the row and columns of the t-selection either match with the corresponding values in the missing interaction or were not assigned previously by the algorithm. Such unassigned values are called *don't-care* or *star-values*. If such a row exists, the interaction gets added to it. Otherwise, a new row is added and the interaction is placed in it. Upon completion of vertical extension, a CA with the current number of columns has been constructed.

Horizontal Extension

The horizontal extension step is used to expand the array until the desired number of columns is reached. Initially an empty column i is added to the CA with $i - 1$ columns. IPOG iterates over all rows in order and greedily assigns the values in the new column that maximize the number of newly covered interactions. IPOG- F extends this by greedily selecting the order in which the rows are treated and the new values are assigned. In IPOG- F2 the selection of values is done heuristically, removing the need to search through uncovered interactions entirely. This can result in smaller run times, but generally produces larger arrays.

3.1 Adaptations for higher-index

For the IPO algorithms to support the generation of covering arrays of higher index, multiple adaptations are undertaken. First, the exact numbers of occurrences of interactions must be taken into account. For CAs with $\lambda = 1$, the IPO algorithms can represent the current coverage status of an interaction by a single bit, using 1 to indicate an interaction that is already covered and 0 for one that is yet to be covered. The

$$\begin{array}{cccccc}
c_0 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 \\
* & 0 & 0 & 1 & 1 & \\
* & 1 & 1 & 0 & 0 &
\end{array}$$

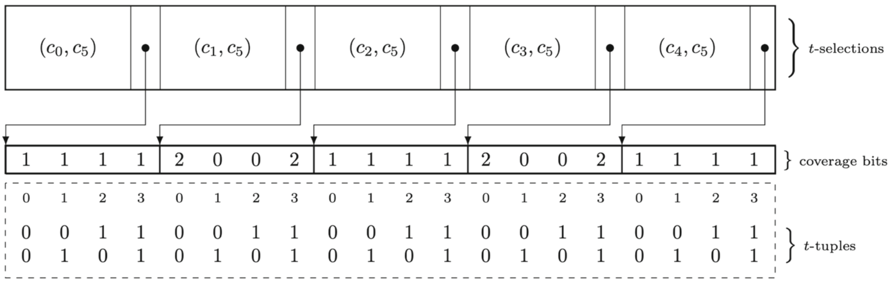


Fig. 1 State of the λ-*coverage-map* after the first four rows have been assigned a value for column c_5 while constructing a $\mathsf{CA}_\lambda(N; 2, 5, 2)$

coverage status of all interactions is stored in a data structure called *coverage map*, in which the state of each interaction is stored in a bit vector. In order to track the coverage information of all possible t-tuples for each of the $\binom{k-1}{t-1}$ different t-selections of columns that contain the newly added column, $\prod_{i=1}^{t} v_i$ bits are reserved in the bit vector, where v_i is the cardinality of the alphabet of the i-th column of the t-selection. States can be updated by *packing* the values of the interaction into an integer used to index the entry in the bit vector. For a detailed description of the coverage map and the packing function, see Kleine and Simos (2018).

For higher-index arrays this is insufficient, so we track exactly how many times an interaction is covered. Therefore, to support higher indices, the bit vector is replaced by a vector of integers, each entry storing the total number of occurrences of each interaction. The size of the integer can be chosen with as few bits as needed, e.g., to support arrays of index ≤ 255 a vector of bytes is sufficient. An example of such a coverage map can be found in Figure 1, which showcases the coverage status of all interactions when column c_5 is appended to a binary CA of strength two by means of horizontal extension. For example, in the 2-selection consisting of columns c_1 and c_5, both the (0, 0) tuple, which is packed into the integer 0, as well as the (1, 1) tuple, encoded to 3, occur two times in the CA in rows 1 and 3 as well as 2 and 4 respectively, while the (0, 1) and (1, 0) tuple do not currently appear in the CA.

Horizontal extension only requires small adjustments to the calculation of coverage gains. In the IPOG algorithm, the selection of values based on the objective function remains as in the $\lambda = 1$ case, including the tie-breaking behavior. However, the objective function is modified to consider how often the interactions are covered. Algorithm 2 describes the modified objective function. If an interaction is already covered λ times or more, additional occurrences cannot improve the solution, and hence it returns a gain of 0. For all other interactions, the difference between λ and the number of occurrences of the interaction is returned. This allows the algorithm to prioritize interactions that need to be covered more frequently.

Algorithm 2 Coverage Gain

```
procedure CoverageGain(i)
    gain ← 0
    for all interactions in the row i being extended do
        count ← coverage_count(interaction)
        if count < λ then
            gain ← gain + λ − count
        end if
    end for
end procedure
```

Adapting the horizontal extension steps of the algorithms IPOG- F and IPOG- F2 to support higher index arrays is even simpler. Neither algorithm uses the coverage map for calculating the coverage gain; instead they maintain a separate data structure for counting/estimating the coverage gain for each row and value pair. When $\lambda = 1$, the estimates start at $\binom{\ell}{t-1}$ where ℓ is the number of assigned columns in each row. This is an upper bound on the number of interactions that could be covered by selecting a value in the new column. Whenever a candidate value is selected for one of the rows, the estimates are updated for all row/value pairs based on the newly-covered interactions as well as the number of columns in which the rows share the same value. While IPOG-F does this exhaustively in order always maintain a precise value for the potential coverage gain of each row/value pair, IPOG-F2 uses a heuristic for this update step and therefore can only provide an estimate. For an in-depth explanation of the two algorithms as well as their differences we refer the interested reader to Forbes et al. (2008). In order to handle $\lambda > 1$, it is sufficient to multiply this estimate by λ. This invalidates the interpretation of the estimate as the number of potentially coverable interactions in that row, however, since each interaction now needs to be covered λ times, this approach still tracks which and how many interactions remain to be covered. Moreover, this objective function prioritizes interactions that have been covered fewer times in absolute terms.

Vertical extension requires the most substantial adaptation. First, interactions might still need to be added multiple times, specifically λ minus the number of times it already occurs in the array. Therefore, when merging interactions into existing rows, we cannot limit our search to the first compatible row, because we could fall into the trap of merging the interaction into a prior occurrence of itself and not increasing the coverage. Thus, we skip occurrences of interactions that already appear in the array when selecting a compatible row and only consider rows that contain a *partial* match, where the corresponding entries in the row and t-selection of columns match or are star values and at least one of the entries is a star value. Lastly, when adding an interaction to an existing row, care is needed to not mark interactions in the existing row multiple times. This can be achieved by only considering newly added interactions. A schematic of the procedure is given in Algorithm 3.

Algorithm 3 Vertical Extension Algorithm for IPOG Methods

```
procedure VERTICALEXTENSION(Array, i)
    for all uncovered interactions tuple do
        count ← coverage_count(tuple)
        while count < λ do
            if ∃ row such that its entries partially match tuple then
                add tuple to Array[row]
                increase coverage count of new interactions in row by one
            else
                add new row to Array containing only don't-care values
                add tuple to new row
            end if
            count ← count + 1
        end while
    end for
end procedure
```

4 Conditional expectation

One-row-at-a-time methods for constructing covering arrays were pioneered in the AETG approach (Cohen et al. 1997). Using conditional expectations, Bryce and Colbourn (2007, 2009) established that such methods underlie polynomial time algorithms to produce covering arrays meeting the asymptotic bounds. Such conditional expectation methods, also called density algorithms, have been extended to employ compact representations of uniform covering arrays (Colbourn 2014; Colbourn et al. 2017) in order to improve both the asymptotic guarantee on the size and accelerate the computations. They have also been explored for mixed levels and variable strength (Moura et al. 2019), but our extension to higher index appears to be new.

The covering array is generated one-row-at-a-time, and the algorithm by Bryce and Colbourn is both deterministic and efficient. Let $\mathcal{F} = \{F_1, \ldots, F_k\}$ be a set of k factors, and $\mathcal{I}$ be the set of all t-way interactions with values from the factors in $\mathcal{F}$. Generate a row R of indeterminates, and consider each factor F_c and each t-way interaction $T \in \mathcal{I}$, both in any arbitrary order. Iterate through all levels $\ell_1, \ldots, \ell_{v_c}$ that can be set for factor F_c. For each assignable level ℓ_i, calculate the probability that T would be covered in this row R if we fix F_c to ℓ_i. If among the columns that are determined of R it is the case that T's values disagree with them, this probability is 0. Otherwise, let f be the number of columns of T not fixed to an entry; if we set a level to F_c in R (which is not fixed at this point), the probability that T is covered in R is $1/v^{f-1}$, if all other entries are independently randomly chosen. Finally, fix the level ℓ_{max} that maximizes the number of interactions covered for the first time in R if ℓ_{max} is the entry in factor F_c of R. When R has all of its entries determined, update $\mathcal{I}$ by removing all interactions covered in R for the first time. The *density* of R at each step is the expected number of t-way interactions that are covered for the first time in R.

When $\lambda \geq 2$, the existing method is insufficient because coverage of an interaction in a row does not imply that this interaction has been covered λ times. Indeed, the probability of being λ-covered depends on how many times it has been covered in earlier rows. Although here we also employ the idea of conditional expectation, we must correctly determine these probabilities. We first determine an upper bound for the number of rows N; for example, we could employ the elementary upper bound $\lambda\binom{k}{t}v^t$, or compute the smallest value consistent with the analysis of Theorem 1. We adopt the second approach mainly for efficiency.

4.1 Conditional expectation for higher index

We focus on several changes to the methods for $\lambda = 1$. We generate an array one-row-at-a-time; suppose the currently formed array is A, and the factors are $\mathcal{F} = \{F_1, \ldots, F_k\}$. At each point in the construction, let M denote the number of rows already constructed, and let $\mathcal{T}$ be the set of interactions that are not (yet) λ-covered. We generate a row R of indeterminates. Then we examine each factor F_c in arbitrary order, and iterate through its levels one at a time in any order. However, instead of measuring the expected number of interactions covered for the first time, we require a finer measure of progress. Provided that A is not already a covering array of index λ, let $T \in \mathcal{T}$ be any interaction not λ-covered. Determine the probability that T would be covered in R one more time if we fix F_c to ℓ_i in R. If T was covered μ times prior to row R, we now have the probability that it is covered $\mu + 1$ times after row R, and the complementary probability that it is covered μ times. Use these to calculate the probability that T is (at least) λ-covered when the remaining $N - M - 1$ rows are selected uniformly at random. This is the probability that T is λ-covered if the termination of the algorithm occurs once N rows are constructed.

Summing such probabilities for all interactions in $\mathcal{T}$ produces the expected number of uncovered interactions assuming all N rows are to be completed. Therefore we choose a level for factor F_c in row R that minimizes this expectation. Having chosen a level, we increase the coverage count for each interaction in $\mathcal{T}$ that is covered in R. We also recalculate N to again be the smallest value such that the total expectation is strictly less than 1; in this way, we may reduce the target number of rows but never increase it. A more formal description is presented in Algorithm 4.

Lemma 1 *Each row generated by* MAKENEXTROW *in Algorithm* 4 *covers at least one interaction that is i-covered for some $i < \lambda$.*

Proof Rows are only generated when at least one uncovered interaction remains. Suppose to the contrary that a row R is generated that does not cover any interaction in $\mathcal{T}$. Let A be the array before the addition of R, and let A' be the result of appending R to A, so that A' has one more row than A does. All values in R are chosen so that they do not increase the expectation of the number of uncovered interactions at the end of the algorithm. If R fails to cover an i-covered interaction for some $0 \leq i < \lambda$, this expectation must increase, a contradiction. $\square$

Even though Lemma 1 guarantees that at least one uncovered interaction is covered in each row generated, this may not ensure that the number of rows created does not exceed the N bound calculated at the start. We address this next.

Lemma 2 *The* CONDITIONALEXPECTATION *algorithm, presented as Algorithm 4, generates an* $MCA_\lambda(N; t, (v_1, \ldots, v_k))$ *where* N *is asymptotically optimal (i.e., it asymptotically meets the bound from Theorem 1).*

Algorithm 4 Conditional Expectation (CE) Algorithm to Produce CAs of higher index.

Factor levels $L_1, \ldots, L_k$, with $|L_i| = \ell_i$; Factors $\mathcal{F} = (L_1, \ldots, L_k)$
Interaction $T = \{(\gamma_i, \nu_i) : 1 \leq i \leq t\}$, $\lambda(T)$ is #times left to cover
function UNCOVERPROB($row, N, T = \{(\gamma_i, \nu_i) : 1 \leq i \leq t\}$)
  $cr \leftarrow 1$; $p \leftarrow \left(\prod_{c=1}^{t} \frac{1}{\ell_{\gamma_c}}\right)$
  for c from 1 to t **do**
    $cr \leftarrow cr \times \begin{cases} \frac{1}{\ell_{\gamma_c}} & if \;\; row[\gamma_c] = \star \\ 1 & if \;\; row[\gamma_c] = \nu_c \\ 0 & if \;\; row[\gamma_c] \notin \{\nu_c, \star\} \end{cases}$
  end for
  return $\sum_{i=0}^{\lambda(T)-2} \left(\binom{N-1}{i} p^i (1-p)^{N-1-i}\right) + (1-cr)\binom{N-1}{\lambda(T)-1} p^{\lambda(T)-1} (1-p)^{N-\lambda(T)}$
end function
function MAKENEXTROW($\mathcal{T}$)
  Initialize row to be a vector of k entries equal to $\star$
  $N \leftarrow \min(M : \sum_{T \in \mathcal{T}} \text{UNCOVERPROB}(row, M, T) < 1)$
  for $c \in \{1, \ldots, k\}$ **do**
    for $s \in L_c$ **do**
      Form row_s from row by setting column c to s
      $uc_s \leftarrow \sum_{T \in \mathcal{T}} \text{UNCOVERPROB}(row_s, N, T)$
    end for
    $row[c] \leftarrow \sigma$ for some σ such that $uc_\sigma = \min(uc_s : s \in L_c)$
  end for
  return row
end function
procedure CONDITIONALEXPECTATION($t, \mathcal{F}, \lambda$)
  $\mathcal{T} \leftarrow$ all t-way interactions on $\mathcal{F}$, each having $\lambda(T) = \lambda$
  while $\mathcal{T} \neq \emptyset$ **do**
    $row \leftarrow$ MAKENEXTROW($\mathcal{T}$)
    Output row
    for each interaction $T \in \mathcal{T}$ that appears in row **do**
      $\lambda(T) \leftarrow \lambda(T) - 1$
      if $\lambda(T) = 0$ **then**
        Remove T from $\mathcal{T}$
      end if
    end for
  end while
end procedure

Proof This follows from two crucial observations. First, the selection of a value for a factor in row R cannot decrease the expectation for the current target value of N. Secondly, treating the expected number of uncovered interactions as a function of N, as N increases, the expected number may decrease or remain unchanged. Hence, because selections of levels never increase the expected number, the target number of rows is never increased. □

Theorem 3 *Let $t, v_1, \ldots, v_k, \lambda$ be fixed integers. Then Algorithm* 4 *generates an* $\mathsf{MCA}_\lambda(N; t, (v_1, \ldots, v_k))$ *in time polynomial in k.*

Proof ConditionalExpectation invokes MakeNextRow once per row constructed, which by Theorem 2 is $O(\log k + \lambda)$ times. In addition, it maintains the coverage status of each interaction, of which there are polynomially many because $t, v_1, \ldots, v_k$, and λ are fixed.

MakeNextRow calls UnCoverProb for each member of $\mathcal{T}$, again a polynomial number. In addition, it calculates (and recalculates) the target number N of rows. This can be efficiently handled by a binary search for the smallest value of N. UnCoverProb can be computed in $O(\log N)$ time because λ is fixed.

Hence an $\mathsf{MCA}_\lambda(N; t, (v_1, \ldots, v_k))$ is produced in time polynomial in k. □

5 Computational results

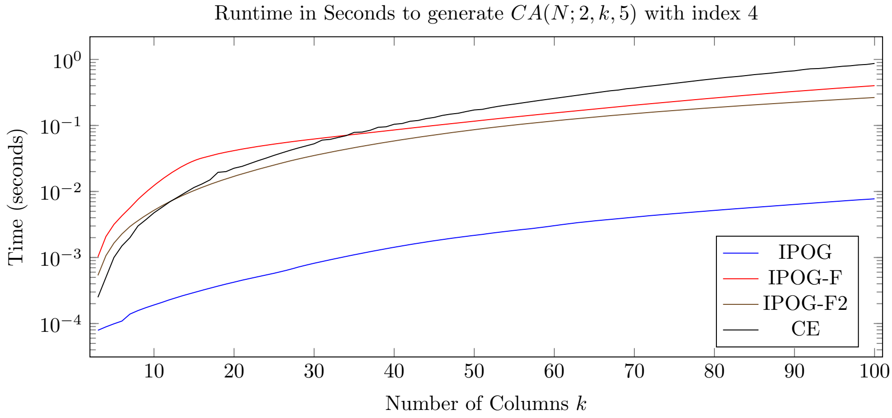
For the experiments using the In-Parameter-Order strategy, the algorithms FIPOG, FIPOG-F and FIPOG-F2 were used, which implement the algorithmic and implementation-level enhancements proposed in Kleine and Simos (2018). All three algorithms are available as part of the tool (Wagner et al. 2020).

For all algorithms, we generated uniform covering arrays of higher index whenever $2 \le t \le 4$, $2 \le v \le 5$, $k \in \{10, 15, 20, 50, 100\}$ and $1 \le \lambda \le 4$. These parameters have been chosen to demonstrate the logarithmic growth of covering array sizes, and to show that the relative difference in the number of additional rows for higher λ decreases as k increases. Additionally, they illustrate differences between the implemented algorithms, because each has its own advantages and disadvantages.

In the presentation of results, we abbreviate FIPOG, FIPOG-F and FIPOG-F2 to G, F, and F2, respectively; CE denotes the density/conditional expectation method. Covering array sizes are reported in Table 2 for $t \in \{2, 3\}$, and in Table 3 for $t = 4$.

We also generate mixed-level covering arrays of higher index for certain parameter sets arising from real-world scenarios; the notation v_i^j indicates that there are j columns with v_i symbols.

- mobile: $10^8 9^1 8^4 7^5 6^{10} 5^4 4^6 3^9 2^{28}$
- wireless: $5^9 4^5 3^7 2^3$
- flex: $5^2 3^4 2^{23}$
- make: $6^1 5^1 4^2 3^4 2^{14}$
- grep: $21^1 13^1 10^1 7^1 5^1 4^1 3^3 2^1 1^4$
- sed: $10^1 8^2 6^1 5^3 4^3 3^1 2^7 1^1$
- gzip: $34^1 6^1 5^1 4^2 3^8 2^8 1^4$



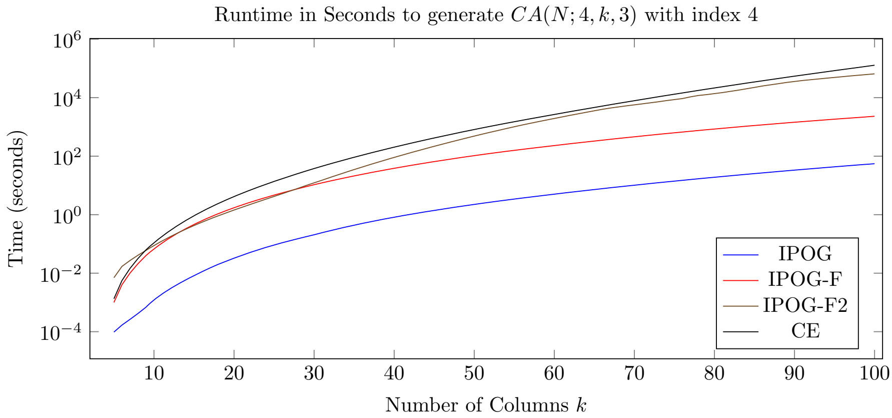


Fig. 2 Runtimes in seconds of the IPO algorithms are depicted for different values of k on a logarithmic scale

- nanoxml: $6^1 4^1 3^6 2^{11} 1^2$

The resulting (mixed-level) array sizes are reported in Table 4; we again report their sizes for $1 \leq \lambda \leq 4$ and $t \in \{2, 3, 4, 5\}$ (Fig. 2).

The experiments for the IPO family of algorithms were performed on a machine with an Intel Core i7-4770 CPU clocked at 3.40 GHz with 64 GB of RAM; for the conditional expectation algorithm, they were performed on a machine with an Intel Core i9 CPU clocked at 3.6 GHz with 16 GB of RAM. While the infrastructures used to evaluate the algorithms differ slightly, the run time results should still provide a good estimation on the performance and scalability of the different algorithms. The λ-coverage was verified using the CAMETRICS combinatorial coverage measurement tool (Leithner et al. 2018).

Table 2 Results for uniform CAs of strength $t = 2$ and $t = 3$

	$\lambda = 1$				$\lambda = 2$				$\lambda = 3$				$\lambda = 4$			
	G	F	F2	CE	G	F	F2	CE	G	F	F2	CE	G	F	F2	CE
$t = 2$																
2^{10}	10	10	10	8	14	14	15	12	20	18	17	17	24	24	24	20
2^{15}	10	10	12	9	14	14	19	12	20	20	21	19	24	24	28	22
2^{20}	12	12	13	10	16	16	20	14	24	20	23	20	28	28	32	24
2^{50}	14	14	17	13	20	20	26	16	28	26	34	24	32	28	42	29
2^{100}	16	16	21	15	24	22	29	20	32	28	40	27	36	32	49	30
3^{10}	15	18	19	16	24	30	26	26	27	27	37	35	42	44	44	44
3^{15}	21	18	23	20	33	33	34	31	45	45	46	40	51	46	52	49
3^{20}	23	21	28	21	39	33	43	33	48	45	52	43	53	50	62	53
3^{50}	28	26	39	28	44	41	61	40	56	49	78	51	69	61	95	62
3^{100}	33	30	47	30	51	43	69	44	62	56	95	58	79	68	115	70
4^{10}	40	32	33	30	49	49	51	47	64	60	64	62	82	80	64	80
4^{15}	40	36	42	34	57	52	64	53	71	72	79	70	85	82	64	88
4^{20}	40	37	49	37	60	55	73	58	77	76	91	83	95	88	112	96
4^{50}	50	46	72	42	74	69	107	61	97	86	142	87	124	107	165	98
4^{100}	56	53	84	52	82	77	131	68	108	98	173	92	136	118	209	117
5^{10}	45	45	48	45	80	73	72	72	105	97	98	97	120	123	123	124
5^{15}	48	49	61	52	87	77	90	83	112	102	117	109	120	127	141	135
5^{20}	51	53	74	58	93	86	110	92	114	110	139	119	120	136	166	147
5^{50}	68	70	110	75	113	105	161	114	159	136	209	143	195	163	254	175
5^{100}	81	83	133	90	127	118	197	127	173	152	262	160	211	184	321	195

Table 2 continued

	$\lambda = 1$				$\lambda = 2$				$\lambda = 3$				$\lambda = 4$			
	G	F	F2	CE	G	F	F2	CE	G	F	F2	CE	G	F	F2	CE
$t = 3$																
2^{10}	20	20	22	16	24	24	28	24	30	40	39	30	32	32	48	32
2^{15}	24	22	29	23	37	32	45	30	40	41	62	36	48	48	57	42
2^{20}	28	28	35	26	41	36	51	35	48	43	70	44	68	56	85	51
2^{50}	44	40	49	41	53	48	72	54	65	56	93	64	105	70	117	71
2^{100}	52	50	61	49	63	57	86	61	77	68	110	74	108	80	138	85
3^{10}	68	65	76	64	109	96	117	99	171	126	152	129	162	150	181	157
3^{15}	80	80	99	79	126	114	161	116	171	148	222	150	188	174	266	182
3^{20}	91	89	117	92	134	125	181	134	192	160	249	168	205	191	312	203
3^{50}	132	126	169	134	181	169	249	177	227	204	330	215	268	242	409	252
3^{100}	165	158	209	168	217	205	295	216	270	243	386	257	304	280	475	296
4^{10}	159	149	172	160	235	229	290	242	314	299	354	317	385	367	256	393
4^{15}	198	189	233	198	287	272	387	288	366	352	537	375	450	425	623	457
4^{20}	228	217	275	228	321	307	440	326	408	389	605	415	489	464	767	493
4^{50}	320	303	397	319	428	410	587	433	523	506	805	524	612	595	1024	616
4^{100}	400	379	499	399	516	493	700	520	619	594	937	624	711	689	1179	718
5^{10}	305	316	367	322	521	440	587	489	649	584	759	638	699	719	909	775
5^{15}	355	369	472	395	587	536	771	576	755	688	1063	738	977	834	1355	894
5^{20}	393	419	558	450	631	605	876	646	826	768	1199	818	1045	911	1531	974
5^{50}	590	603	802	633	819	812	1187	855	1067	1001	1601	1048	1307	1174	2056	1232
5^{100}	762	744	988	783	998	975	1395	1027	1212	1179	1856	1233	1429	1367	2361	1427

Table 3 Results for $t = 4$

	$\lambda = 1$				$\lambda = 2$				$\lambda = 3$				$\lambda = 4$			
	G	F	F2	CE	G	F	F2	CE	G	F	F2	CE	G	F	F2	CE
2^{10}	44	45	57	32	68	75	72	58	93	80	91	75	128	102	96	94
2^{15}	56	55	74	51	87	82	107	71	107	94	150	95	155	113	184	113
2^{20}	66	63	85	59	97	91	122	74	118	104	173	104	166	126	222	122
2^{50}	101	99	125	97	135	126	170	118	150	147	235	143	209	169	288	171
2^{100}	132	124	158	132	166	153	210	160	186	178	276	184	238	201	342	205
3^{10}	220	220	248	231	363	337	449	335	405	422	593	433	566	510	686	525
3^{15}	301	303	356	304	476	418	616	426	598	525	844	529	689	625	1088	633
3^{20}	374	362	429	381	530	483	694	512	675	593	946	626	799	698	1220	732
3^{50}	580	558	669	575	732	708	958	735	912	842	1262	872	1031	965	1586	997
3^{100}	731	715	855	745	902	884	1156	903	1103	1031	1494	1056	1208	1168	1847	1174
4^{10}	763	731	892	755	1108	1064	1456	1106	1436	1374	1925	1418	1739	1652	1841	1728
4^{15}	1032	981	1255	1044	1425	1364	1982	1452	1781	1703	2767	1793	2109	2023	3540	2130
4^{20}	1226	1171	1478	1235	1660	1584	2258	1665	2035	1947	3099	2045	2386	2294	4011	2397
4^{50}	1895	1823	2244	1889	2422	2320	3115	2419	2869	2762	4125	2871	3289	3165	5195	3287
4^{100}	2412	2337	2804	–	2991	2900	–	–	3484	3388	–	–	3944	3833	–	–
5^{10}	1702	1806	2092	1917	2851	2619	3387	2795	3954	3370	4660	3576	5000	4090	5604	4339
5^{15}	2220	2432	2975	2563	3514	3374	4622	3572	4665	4242	6541	4452	6437	5031	8254	5273
5^{20}	2698	2878	3594	3047	4073	3920	5363	4143	5232	4833	7439	5094	6742	5685	9495	5967
5^{50}	4486	4504	5577	4794	5945	5763	7531	6103	7274	6842	10051	7286	8646	7847	12509	8523
5^{100}	5963	5784	–	–	7409	7191	–	–	8645	8398	–	–	9782	9503	–	–

Table 4 Results of generated arrays for testing real world applications

t		$\lambda = 1$				$\lambda = 2$				$\lambda = 3$				$\lambda = 4$			
		G	F	F2	CE	G	F	F2	CE	G	F	F2	CE	G	F	F2	CE
mobile	2	157	151	203	170	255	243	314	275	357	351	406	377	448	446	497	481
	3	2236	2085	2792	2389	3356	3144	4753	3530	4352	4142	6287	4609	5382	5162	8130	5703
	4	27,538	25,747	34,303	–	39,263	36,872	52,905	–	49,946	47,066	75,069	–	60,191	57,052	95,115	–
	5	304,887	–	–	–	421,857	–	–	–	528,744	–	–	–	629,708	–	–	–
wireless	2	45	45	54	48	80	72	83	72	105	96	106	101	120	123	128	127
	3	315	309	378	315	524	421	611	468	646	557	826	618	684	686	1039	750
	4	1841	1764	2153	1882	2739	2462	3378	2780	3388	3162	4811	3351	3825	3885	6026	4109
	5	11,064	10,116	11,612	10,467	15,906	14,579	18,033	14,876	19,403	17,713	25,640	17,923	21,768	20,014	32,468	20128
flex	2	26	25	26	25	50	50	50	50	75	75	75	75	100	100	100	100
	3	91	100	102	101	153	151	187	166	225	225	274	230	300	300	362	300
	4	347	341	439	409	521	502	704	633	681	675	1085	803	900	900	1389	987
	5	1164	1121	1362	1338	1679	1598	2357	2056	2128	2071	3594	2552	2711	2702	4869	2986
make	2	30	30	33	31	60	60	60	61	90	90	90	90	120	120	122	120
	3	138	142	140	147	244	240	255	245	360	360	411	360	480	480	533	480
	4	607	588	619	692	997	967	1186	1017	1446	1440	1794	1441	1920	1920	2280	1920
	5	2208	2170	2293	2480	3356	3154	4350	3562	4515	4372	6207	4584	5848	5763	8556	5901
grep	2	273	273	273	273	546	546	546	546	819	819	819	819	1092	1092	1092	1092

Table 4 continued

	t	$\lambda = 1$				$\lambda = 2$				$\lambda = 3$				$\lambda = 4$			
		G	F	F2	CE	G	F	F2	CE	G	F	F2	CE	G	F	F2	CE
	3	2732	2730	2730	2730	5460	5460	5460	5460	8190	8190	8190	8190	10,920	10,920	10,920	10,920
	4	191,75	19,110	19,144	19,115	38,220	38,220	38,297	38,220	57,330	57,330	57,425	57,330	76,440	76,440	76,440	76440
	5	97,024	95,680	99,010	97,249	191,100	191,100	197,161	192,315	286,650	286,650	293,320	287,302	382,200	382,200	387683	382200
sed	2	81	80	81	85	160	160	160	162	240	240	241	240	320	320	323	320
	3	679	642	677	647	1284	1280	1317	1280	1920	1920	1957	1920	2560	2560	2586	2560
	4	4546	4289	4692	4694	7885	7684	9189	7745	11,530	11,520	13,418	11,535	15,360	15,360	17,736	15,365
	5	25,846	25,716	28,985	26,993	42,485	39,739	50,448	43,586	58,839	57,619	81,536	59,014	76,860	76,800	112,054	76800
gzip	2	204	204	204	207	408	408	408	409	612	612	612	617	816	816	816	822
	3	1038	1085	1331	1187	2041	2040	2263	2103	3060	3060	3527	3187	4080	4080	4977	4103
	4	5138	5251	6698	5825	8538	8231	12,124	9628	12,263	12,240	18,731	12,462	16,320	16,320	25,122	16,320
	5	23,690	22,876	29,524	–	36,592	34,708	52,812	–	50,004	49,049	77,445	–	65,360	65,280	107,927	–
nanoxml	2	25	24	25	24	48	48	52	48	72	72	74	72	96	96	97	96
	3	102	107	115	115	161	153	211	177	222	222	327	233	288	288	448	297
	4	387	379	448	429	603	569	784	659	776	780	1198	849	1028	1009	1660	1033
	5	1320	1255	1383	1432	2120	1884	2535	2178	2563	2417	3785	2782	3338	3141	5248	3319

6 Discussion and conclusion

In all instances, when $k > t$, all algorithms were able to create higher-index arrays whose size is smaller than the size of a $\mathsf{CA}_1(N; t, k, v)$, found via the same methods, times the index λ.

This behavior is apparent in both the uniform and mixed-level experiments. This is not surprising, because in our experiments the largest t is somewhat smaller than the smallest k, and we focus on small values of v. When k is larger than $\max(t, v)$, even when $\lambda = 1$ some interactions necessarily must be covered multiple times. In these situations, further rows may not need to consider as many interactions.

What is more remarkable is how *few* additional rows are needed even when k is small. Consider the situation when $t = 4$, $k = 10$, and $v = 3$. All four algorithms report a covering array size between 220 and 248 for $\lambda = 1$; but for $\lambda = 2$, the average increase was 61%, and the increase diminishes for $\lambda \in \{3, 4\}$. We expect that if k is smaller, but still larger than t, these increases are more pronounced.

There are obvious differences between CE and the IPOG algorithms. CE builds rows of full length k, one-row-at-a-time. In contrast, the IPOG algorithms add columns during the construction. One advantage of CE over the IPOG methods is that an upper bound is determined at the start (which may improve as rows are built), whereas the IPOG methods do not make this determination. Nevertheless, neither method knows in advance the actual number of rows to-be-generated.

One advantage of the IPOG methods over CE is that they are faster in practice. CE repeatedly employs the number of times each interaction has been covered so far. Either this information is stored, or is recomputed whenever needed by iterating through the rows of the currently constructed array. This calculation cannot be avoided if one wants to achieve the guaranteed upper bound on the number of rows. IPOG deals with a substantially smaller number of interactions: $\binom{k}{t-1}v^t$ versus $\binom{k}{t}v^t$ for CE.

Both types of algorithms contain both *local* and *global* heuristics. Locally, each algorithm chooses a value in a single row and column that maximizes some quantity, but the choice made is based on what can occur after all interactions are considered. For the IPOG methods, this is maximizing the coverage gain; and for CE, this is maximizing the decrease of the expectation.

We have explored two algorithms for constructing higher-index covering arrays; one is based on the in-parameter-order algorithm, and the other is based on conditional expectation. Naturally, other methods for index one can (and should) be extended to treat higher index. For uniform arrays, one promising direction is to extend the methods of this paper to a very compact representation of (uniform) covering arrays, the covering perfect hash families introduced in Sherwood et al. (2006) and extensively explored in Colbourn et al. (2017). The extension to higher index is natural, and IPO-like strategies are quite effective on this compact representation (Wagner et al. 2021). In a similar way, the extension of the cyclotomic constructions (Colbourn 2010) to higher index is routine; some steps in this direction are taken in Akhtar et al. (2021). Finally, a major paradigm in recursive constructions employs other types of hash families in column replacement methods (Colbourn 2011), and generalizations of these to higher index have recently been considered (Dougherty and Colbourn 2020). We expect that for some parameter sets, such extensions can lead to covering arrays with fewer rows than are found by our two sets of algorithms. However, the proposed directions all concentrate on uniform covering arrays. For mixed-level covering arrays, algorithms like those developed here appear likely to remain the most effective for quickly generating tests.

Acknowledgements The views expressed in this article are those of the author(s) and do not reflect the official policy or position of the Department of the Army, Department of Defense, or the U.S. Government. This research of KK, DES, and MW was carried out partly in the context of the Austrian COMET K1 program and publicly funded by the Austrian Research Promotion Agency (FFG) and the Vienna Business Agency (WAW). Research of CJC is funded by the U.S. National Science Foundation Grant #1813729.

References

Akhtar Y, Colbourn CJ, Syrotiuk VR (2021) Mixed covering, locating, and detecting arrays via cyclotomy Springer Nature PROMS, accepted Jul 22

Aldaco AN, Colbourn CJ, Syrotiuk VR (2015) Locating arrays: a new experimental design for screening complex engineered systems. SIGOPS Oper Syst Rev 49(1):31–40

Alon N, Spencer JH (2004) The probabilistic method. Wiley, Hoboken

Alon N, Gutner S (2007) Balanced families of perfect hash functions and their applications. In: Automata, languages and programming. Lecture Lecture Notes in Computer Science, vol 4596, pp 435–446. Springer, Berlin

Alzahrani F, Salem A (2018) Sharp bounds for the Lambert W function. Integr Transform Spec Funct 29(12):971–978

Bryce RC, Colbourn CJ (2007) The density algorithm for pairwise interaction testing. Softw Test Verif Reliab 17:159–182

Bryce RC, Colbourn CJ (2009) A density-based greedy algorithm for higher strength covering arrays. Softw Test Verif Reliab 19(1):37–53

Cohen DM, Dalal SR, Fredman ML, Patton GC (1997) The AETG system: an approach to testing based on combinatorial design. IEEE Trans Softw Eng 23(7):437–444

Colbourn CJ (2011) Covering arrays and hash families. In: Information security, coding theory, and related combinatorics. NATO Science for Peace and Security Series. IOS Press, Amsterdam, pp 99–135

Colbourn CJ, Lanus E, Sarkar K (2017) Asymptotic and constructive methods for covering perfect hash families and covering arrays. Designs Codes Cryptogr 86(4):1–31

Colbourn CJ (2004) Combinatorial aspects of covering arrays. Le Matematiche (Catania) 58:121–167

Colbourn CJ (2010) Covering arrays from cyclotomy. Des Codes Cryptogr 55:201–219

Colbourn CJ (2014) Conditional expectation algorithms for covering arrays. J Comb Math Comb Comput 90:97–115

Colbourn CJ, McClary DW (2008) Locating and detecting arrays for interaction faults. J Comb Optim 15:17–48

Colbourn CJ, Syrotiuk VR (2018) On a combinatorial framework for fault characterization. Math Comput Sci 12(4):429–451

Deng D, Stinson DR, Wei R (2004) The Lovász local lemma and its applications to some combinatorial arrays. Design Codes Cryptogr 32(1–3):121–134

Dougherty RE (2019) Hash families and applications to t-restrictions. PhD thesis, Arizona State University

Dougherty RE, Colbourn CJ (2020) Perfect hash families: the generalization to higher indices. In: Raigorodskii AM, Rassias MT (eds) Discrete mathematics and applications. Springer, Cham

Forbes M, Lawrence J, Lei Y, Kacker RN, Kuhn DR (2008) Refining the in-parameter-order strategy for constructing covering arrays. J Res Nat Inst Stand Technol 113(5):287

Kleine K, Simos DE (2018) An efficient design and implementation of the in-parameter-order algorithm. Math Comput Sci 12(1):51–67

Kuhn DR, Kacker RN, Lei Y (2013) Introduction to combinatorial testing. Chapman & Hall/CRC Innovations in software engineering and software development series. Taylor & Francis, Boca Raton

Lei Y, Kacker R, Kuhn DR, Okun V, Lawrence J (2007) IPOG: a general strategy for t-way software testing. In: 14th annual IEEE international conference and workshops on the engineering of computer-based systems, 2007. ECBS'07, IEEE. pp 549–556

Lei Y, Tai K-C (1998) In-parameter-order: a test generation strategy for pairwise testing. In: High-assurance systems engineering symposium, 1998. Proceedings. Third IEEE International, pp. 254–261. IEEE

Leithner M, Kleine K, Simos DE (2018) CAMETRICS: a tool for advanced combinatorial analysis and measurement of test sets. In: 2018 IEEE international conference on software testing, verification and validation workshops (ICSTW), pp 318–327

Martínez C, Moura L, Panario D, Stevens B (2009/10) Locating errors using ELAs, covering arrays, and adaptive testing algorithms. SIAM J Discrete Math 23:1776–1799

Moura L, Raaphorst S, Stevens B (2019) Upper bounds on the sizes of variable strength covering arrays using the Lovász local lemma. Theoret Comput Sci 800:146–154

Nie C, Leung H (2011) A survey of combinatorial testing. ACM Comput Surv 43:1–29

Ray-Chaudhuri DK, Singhi NM (1988) On existence and number of orthogonal arrays. J Comb Theory Ser A 47(1):28–36

Sherwood GB, Martirosyan SS, Colbourn CJ (2006) Covering arrays of higher strength from permutation vectors. J Comb Des 14(3):202–213

Wagner M, Colbourn CJ, Simos DE (2021) In-parameter-order strategies for covering perfect hash families. Appl Math Comput. https://doi.org/10.1016/j.amc.2022.126952

Wagner M, Kleine K, Simos DE, Kuhn R, Kacker R (2020) CAGEN: a fast combinatorial test generation tool with support for constraints and higher-index arrays. In: Proceedings of 2020 IEEE international conference on software testing, verification and validation workshops (ICSTW)