Separating Interaction Effects Using Locating and Detecting Arrays

Stephen A. Seidel, Kaushik Sarkar, Charles J. Colbourn(✉), and Violet R. Syrotiuk

School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA
{stephen.seidel,ksarkar1,colbourn,syrotiuk}@asu.edu

Abstract. The correctness and performance of complex engineered systems are often impacted by many factors, each of which has many possible levels. Performance can be affected not just by individual factor-level choices, but also by interactions among them. While covering arrays have been employed to produce combinatorial test suites in which every possible interaction of a specified number of factor levels arises in at least one test, in general they do not identify the specific interaction(s) that are significant. Locating and detecting arrays strengthen the requirements to permit the identification of a specified number of interactions of a specified size. Further, to cope with outliers or missing responses in data collected from real engineered systems, a further requirement of *separation* is introduced. In this paper, we examine two randomized methods for the construction of locating and detecting arrays, the first based on the Stein-Lovász-Johnson paradigm, and the second based on the Lovász Local Lemma. Each can be derandomized to yield efficient algorithms for construction, the first using a conditional expectation method, and the second using Moser-Tardos resampling. We apply these methods to produce upper bounds on sizes of locating and detecting arrays for various numbers of factors and levels, when one interaction of two factor levels is to be detected or located, for separation of up to four. We further compare the sizes obtained with those from more targeted (and more computationally intensive) heuristic methods.

1 Introduction

Complex engineered systems are critical, engineered, large-scale systems, such as transportation networks, power grids, and wireless communication systems. The correct operation of such systems often depends not just on the choices made for numerous parameters in their configuration, but also on interaction effects among these choices. Moreover, the performance of such a system can be dramatically affected by the choices and their interactions, even when the system is operating.

We examine a formal testing model. There are k factors $F_1, \ldots, F_k$. Each factor F_i has a set $S_i = \{v_{i1}, \ldots, v_{is_i}\}$ of s_i possible values (*levels*). A *test* is an

C. Iliopoulos et al. (Eds.): IWOCA 2018, LNCS 10979, pp. 349–360, 2018.
https://doi.org/10.1007/978-3-319-94667-2_29

assignment, for each $i = 1, \ldots, k$, of a level from $v_{i1}, \ldots, v_{is_i}$ to F_i. The execution of a test yields a measurement of a *response*. When $\{i_1, \ldots, i_t\} \subseteq \{1, \ldots, k\}$ and $\sigma_{i_j} \in S_{i_j}$, the set $\{(i_j, \sigma_{i_j}) : 1 \leq j \leq t\}$ is a *t-way interaction*. (The interaction has *strength* t.) A test on k factors *covers* $\binom{k}{t}$ t-way interactions. A *test suite* is a collection of tests. Usually such a test suite is represented as an array: Suppose that $A = (\sigma_{i,j})$ is an $N \times k$ array for which $\sigma_{i,j} \in S_j$ when $1 \leq i \leq N$ and $1 \leq j \leq k$. This is a *test suite* of *size* N and *type* $(s_1, \ldots, s_k)$. Types can be written in exponential notation: $g_1^{u_1} \cdots g_r^{u_r}$ means that there are u_i factors with g_i levels for $1 \leq i \leq r$. Tests are recorded as rows of A, and factors correspond to columns.

Combinatorial testing [20,28] is concerned with the design and analysis of test suites in order to assess correctness and performance of a system. The focus has been on test suites known as covering arrays, which ensure that every t-way interaction is covered by a test. We define these precisely next. Let $A = (\sigma_{i,j})$ be a test suite of size N and type $(s_1, \ldots, s_k)$. Let $T = \{(i_j, \sigma_{i_j}) : 1 \leq j \leq t\}$ be a t-way interaction. Denote by $\rho(A, T)$ the set $\{r : a_{r,i_j} = \sigma_{i_j}, 1 \leq j \leq t\}$ of rows of A in which the interaction is covered. A *mixed covering array* $\mathrm{MCA}(N; t, (s_1, \ldots, s_k))$ is a test suite A for which every t-way interaction T has $\rho_A(T) \neq \emptyset$, i.e., every t-way interaction is covered in at least one row. When used for testing correctness, covering arrays reveal the presence of an interaction that causes faulty behaviour, but in general does not identify the specific faulty interaction(s); see [10,11].

We consider two motivating examples. In [1], a software simulation of a mobile wireless network is studied. There, 75 factors are identified among the controllable parameters in the protocol stack, ranging from 2 to 10 levels. (The type is $10^8 9^1 8^4 7^5 6^{10} 5^4 4^6 3^9 2^{28}$.) Throughput is measured as a response, and the objective is to determine which factors, and interactions among them, significantly affect the response. In this setting, it is of little interest to determine whether some interactions significantly affect the response; the goal is to ascertain which do. One could, of course, obtain the responses for a MCA, and design further testing based on the interactions covered in tests whose responses deviate most widely from the mean. In this way, a MCA could be used to narrow the set of t-way interactions that might have a significant effect on performance. Because the study employs a software simulation, a second round of testing could be conducted in the same environment as the first, and an adaptive method that uses results of earlier tests to choose later ones may be suitable.

In [12], a testbed evaluation of a conferencing scenario in a wireless network is conducted, measuring voice quality and exposure as responses. There are 24 controllable factors, ranging from 2 to 5 levels. (The type is $5^9 4^5 3^7 2^3$.) Hence the testing environment seems somewhat simpler than the simulation. Unlike the well-controlled environment in which the simulation study is conducted, however, the conferencing scenario is impacted by factors in the physical environment, including the hardware used directly in the experiment, interference from other communications in the vicinity, and the like. Despite best efforts to shield the testbed from such effects, measurements taken far apart in time can

be significantly affected by environmental factors that cannot be controlled, and may not be measured. In this setting, it is desirable that all tests be conducted in the same environment, and the significant interactions be identified without further testing. Hence we want a *nonadaptive* approach to testing.

The nonadaptive identification of faults or significant interactions can be accomplished by strengthening covering arrays. A combinatorial set of requirements was first identified in [10]; see also [11,23]. We develop this next.

1.1 Locating and Detecting Arrays

Let A be a test suite of size N and type $(s_1, \ldots, s_k)$. Let $\mathcal{I}_t$ be the set of all t-way interactions for A, and let $\overline{\mathcal{I}_t}$ be the set of all interactions of strength *at most* t. When interaction $T \in \overline{\mathcal{I}_t}$ has strength less than t and the t-way interaction T' contains T, it must hold that $\rho(A, T') \subseteq \rho(A, T)$; there can be no row covering T' but not T. A set $\mathcal{T}' \subseteq \overline{\mathcal{I}_t}$ is *independent* if there do not exist $T, T' \in \mathcal{T}'$ with $T \subset T'$. Our objective is to identify a set $\mathcal{T} \subseteq \mathcal{I}_t$ (or perhaps $\overline{\mathcal{I}_t}$) that have significant effects on the response. If no limitation is placed on $\mathcal{T}$, the design of a test suite can be impossible [23]; even when possible, the size of the test suite grows as the number of interactions in $\mathcal{T}$ increases [10]. We assume that a number d of interactions is to be identified. When at most d are to be identified, we employ the notation $\overline{d}$. Despite this limit, in the intended applications, often many more than d significant interactions can be found by iterative analysis of the response data, adjusting the responses after each selection of significant interactions, without the need for further experimentation; see [31] for details.

Locating arrays for identifying sets of interactions can be defined in this framework [10]. For a set $\mathcal{T}$ of interactions, define $\rho(A, \mathcal{T}) = \bigcup_{T \in \mathcal{T}} \rho(A, T)$. A test suite A is (d, t)-*locating* if $\rho(A, \mathcal{T}_1) = \rho(A, \mathcal{T}_2) \Leftrightarrow \mathcal{T}_1 = \mathcal{T}_2$ whenever $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{I}_t$, $|\mathcal{T}_1| = d$, and $|\mathcal{T}_2| = d$. When $\mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{I}_t}$, and $\mathcal{T}_1$ and $\mathcal{T}_2$ are independent, the array is $(d, \overline{t})$-*locating*. When instead $|\mathcal{T}_1| \leq d$ and $|\mathcal{T}_2| \leq d$, the array is $(\overline{d}, t)$-*locating* or $(\overline{d}, \overline{t})$-*locating*.

Using a locating array, knowing the tests that yield a significant deviation in response, there can be at most one set of (at most) d interactions of strength (at most) t covered in the same sets of tests that account for these deviations. By enumeration of all sets of d interactions of strength t, the location of the interactions causing the faults can be calculated from the outcomes. However, determining the interactions involved may require enumeration of sets of interactions. Determining the tests that exhibit a substantial deviation in response does not ensure that any interaction that is covered only within these tests must be significant. To deal with this, additional requirements are needed [10].

An array A is (d, t)-*detecting* if $\rho(A, T) \subseteq \rho(A, \mathcal{T}_1) \Leftrightarrow T \in \mathcal{T}_1$ whenever $\mathcal{T}_1 \subseteq \mathcal{I}_t$, $|\mathcal{T}_1| = d$, and $T \in \mathcal{I}_t \setminus \mathcal{T}_1$. When instead $\mathcal{T}_1 \subseteq \overline{\mathcal{I}_t}$, $T \in \overline{\mathcal{I}_t}$, and $\mathcal{T}_1 \cup \{T\}$ is independent, the array is $(d, \overline{t})$-*detecting*. For detecting arrays, we can also consider a set of at most d interactions, to obtain arrays that are $(\overline{d}, t)$-*detecting* or $(\overline{d}, \overline{t})$-*detecting*.

Detecting arrays underlie an efficient algorithm for the recovery of the set of significant interactions [10], but necessitate a larger number of tests.

This framework of eight variants of testing arrays for identifying significant interactions suggests many existence questions. However, relationships among them enable a unified examination. In [10], the relevant relationships are established (provided that $s_1 > 1$, $k \geq t$, and d is not larger than the number of possible interactions):

$$\begin{array}{ccccc}
(\overline{d},\overline{t})\text{-detecting} & \Rightarrow & (\overline{d},t)\text{-detecting} & & \\
\Updownarrow & & \Updownarrow & & \\
(d,\overline{t})\text{-detecting} & \Rightarrow & (d,t)\text{-detecting} & & \\
\Downarrow & & \Downarrow & & \\
(\overline{d},\overline{t})\text{-locating} & \Rightarrow & (\overline{d},t)\text{-locating} & \Rightarrow & (d-1,t)\text{-detecting} \\
\Downarrow & & \Downarrow & & \\
(d,\overline{t})\text{-locating} & \Rightarrow & (d,t)\text{-locating} & \Rightarrow & (d-1,t)\text{-locating}
\end{array}$$

Like covering arrays, locating and detecting arrays scale well to large numbers of factors. Indeed when the strength, number d of potentially significant interactions, and maximum number of levels are fixed, the number of tests required is $O(\log k)$ [10]. Unlike covering arrays, however, constructions for locating and detecting arrays have been much less studied. Although locating and detecting arrays are mixed covering arrays of strength t, the extension of covering array constructions requires substantial information about the tests in which interactions are covered. Naturally the objective is to employ as few tests as possible.

Only when $d = t = 1$ is the minimum number of tests in (d,t)-, $(\overline{d},t)$-, $(d,\overline{t})$-, and $(\overline{d},\overline{t})$-locating arrays known precisely [8]. The analogous situation for detecting arrays is explored in [21,24], and strong bounds are established that are exact infinitely often.

When $t \geq 2$, exact results are known for locating arrays when k is very small [32,34]; for larger numbers of factors, a small set of recursive constructions is available when $d = 1$ and $t = 2$ for locating arrays [7]. Beyond these few direct and recursive methods, computational methods have been developed for $(1,2)$-locating arrays [17,19,26] using constraint satisfaction techniques and one-row-at-a-time greedy methods. In order to address concerns with infeasible tests, Jin and Tsuchiya [17] extend the definition of locating arrays to account for testing constraints. These algorithmic methods do not treat detecting arrays; for locating arrays they limit the number of significant interactions, the strength, and the number of factors to small values. In our motivating problems, limiting the number and strength of interactions can be worthwhile, but techniques are needed to construct locating and detecting arrays for larger numbers of factors.

2 The Need for Separation

Consider the use of a locating or detecting array in an experimental setting. In principle, the responses for each test can identify the set of significant interactions whenever the assumptions on number and strengths of interactions are met. In practice, however, a problem arises. Suppose that two sets of (at most) d interactions, $\mathcal{T}_1$ and $\mathcal{T}_2$, have $|\rho(\mathcal{T}_1) \setminus \rho(\mathcal{T}_2)| = 1$. If this occurs, the response

measured in a single test is the sole 'witness' to the difference between the two. In the absence of noise or measurement error, one such witness suffices to differentiate. In our experiments, however, outliers and missing responses do occur. These compromise our ability to analyze the response data. This can be mitigated by exploring a number of possible sets of significant interactions, as in [31], rather than identifying a single set. As we have discussed, it cannot be effectively handled by simply running the test for the outlier or missing response again, without strong (and unjustified) assumptions about the stability of environmental factors.

Therefore we argue that effective locating and detecting arrays must allow for outliers and missing responses. Fortunately this can be treated by further requirements on the testing array, by enforcing a separation between sets of rows for different sets of interactions. We make this precise next. Let $\delta \geq 1$ be an integer, the *distance*. A test suite A is (d, t, δ)-*locating* if whenever $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{I}_t$, $|\mathcal{T}_1| = d$, and $|\mathcal{T}_2| = d$, we have that

$$|(\rho(A, \mathcal{T}_1) \cup \rho(A, \mathcal{T}_2)) \setminus (\rho(A, \mathcal{T}_1) \cap \rho(A, \mathcal{T}_2))| < \delta \Leftrightarrow \mathcal{T}_1 = \mathcal{T}_2.$$

This requires that at least δ tests witness the difference. The variants for $\overline{d}$ and $\overline{t}$ are immediate.

Similarly, an array A is (d, t, δ)-*detecting* if whenever $\mathcal{T}_1 \subseteq \mathcal{I}_t$, $|\mathcal{T}_1| = d$, and $T \in \mathcal{I}_t \setminus \mathcal{T}_1$, we have that $T \in \mathcal{T}_1$ whenever $|\rho(A, T) \setminus \rho(A, \mathcal{T}_1)| < \delta$ or $|\rho(A, \mathcal{T}_1) \setminus \rho(A, T)| < \delta$. Again, definitions of the variants for $\overline{d}$ and $\overline{t}$ are straightforward.

Separation by distance δ ensures that any $\delta - 1$ or fewer tests can fail to provide a response, or provide an outlier response, without losing the differentiation supported by the locating or detecting array. In our motivating examples among many others, requiring larger distance for separation is desirable, but only if it can be accomplished without a dramatic increase in the number of tests.

The simplest technique to make a (d, t, δ)-locating array might be to concatenate the rows of δ (d, t)-locating arrays, or indeed to replicate each row of a single (d, t)-locating array, each δ times. This would enable the use of the few available methods for locating arrays, while increasing the separation as required. However, this appears to necessitate far too many tests.

3 Randomized and Derandomized Algorithms

We require algorithms to construct (d, t, δ)-locating arrays and (d, t, δ)-detecting arrays. The methods of most interest to us must not only handle a range of small values of d, t, and δ (each between, say, 1 and 4), but – more importantly – must handle reasonable numbers of factors (in the range of 50–100 at least). We do not expect to (or need to) produce the fewest tests possible, but naturally we prefer arrays with fewer tests.

Every $(\overline{1}, t)$-locating array is a mixed covering array of strength t, and hence the algorithmic paradigms that have been most effective for covering arrays appear to be natural candidates for construction of locating and detecting arrays. Among these, integer programming, constraint satisfaction, tabu search, and

simulated annealing have provided the best available upper bounds on the number of tests when the number of factors is small [6]. One-column-at-a-time [14,20] and one-row-at-a-time [5] greedy algorithms extend the range of numbers of factors treated, but do not outperform more sophisticated methods for few factors and small strength. Indeed for both methods, a post-optimization strategy [27] can often reduce the number of tests by investing more computation.

For larger numbers of factors and larger strengths yet, the best available upper bounds arise from randomized methods based on the Stein-Lovász-Johnson framework [18,22,33], and derandomized algorithms using conditional expectations [3,4]; and on the Lovász Local Lemma [2,13] with Moser-Tardos resampling [25] to yield both efficient construction techniques and the best asymptotic bounds [9,29,30].

We fully expect that computationally intensive methods and storage intensive methods can and will produce detecting and locating arrays with fewer tests than randomized and derandomized methods when the number of factors is relatively small and the search is tailored to specific choices of d, t, and δ (all 'small'); see, for example, [19]. In exploring randomized techniques, our objective is instead to develop algorithms that can be effectively used for a wide variety of construction problems, without undue time and storage requirements.

3.1 The Stein-Lovász-Johnson Framework and Conditional Expectation

Suppose that an $N \times k$ array A is specified. When A does not meet the requirements to be a locating or detecting array of the kind intended, it is because certain requirements are not met. For example, for (d, t, δ)-locating, when $\mathcal{T}_1 \neq \mathcal{T}_2$, $|\mathcal{T}_1| = d$, and $|\mathcal{T}_2| = d$, but $|(\rho(A, \mathcal{T}_1) \cup \rho(A, \mathcal{T}_2)) \setminus (\rho(A, \mathcal{T}_1) \cap \rho(A, \mathcal{T}_2))| = \mu < \delta$, the requirement is not met, and has *deficiency* $\delta - \mu$. (When a requirement is met, its deficiency is 0.) This notion of deficiency can be extended to requirements for other locating and detecting arrays in a straightforward manner. Then the *deficiency* of A is the sum of the deficiencies of all requirements.

When the deficiency of A is nonzero, a next test can be chosen to reduce the deficiency. Indeed if a test were chosen at random among all possible tests, the *expected reduction* in deficiency can be calculated. The Stein-Lovász-Johnson framework dictates that a next test be chosen to reduce the deficiency by at least this expectation. Choosing such a test at each stage ensures that no more tests are needed than in an entire array chosen at random whose expected deficiency is less than 1, and indeed the one-test-at-a-time method often employs much fewer tests.

An effective implementation of this strategy requires not only that the expected reduction in deficiency be calculated, but that a test be found to achieve this reduction. In [3,4], conditional expectation methods are used to demonstrate for covering arrays that the expectation can be efficiently calculated, and more importantly that the test needed can be constructed by choosing one entry at a time so as never to decrease the expectation. Although the details for locating and detecting arrays differ from the simpler requirements for covering arrays,

the strategy of [4] can be applied here as well. In the interests of space, we do not here repeat the details needed in order to explore the differences.

Every time a new test is to be added, deficiencies for each requirement until this point are needed. A storage intensive method can maintain this information as tests are added, but the number of pairs of sets of interactions can quickly exceed available storage even for arrays with a moderate number of factors. Instead a time-intensive version could recalculate the deficiency for each requirement when it is needed, incurring a substantial amount of recomputation. Recomputation may be feasible; however, the deficiency for a restriction may be recalculated many times in the construction of a single test.

3.2 The Lovász Local Lemma and Moser-Tardos Resampling

For covering arrays, asymptotic results based on the Lovász Local Lemma (such as [15,16,30]) improve upon those based on the Stein-Lovász-Johnson paradigm [3,5]. Despite this, the latter have typically yielded fewer tests in practice. Hence one might expect, for locating and detecting arrays, that the conditional expectation methods would be the preferred ones. As with covering arrays, however, the column resampling methods based on the Lovász Local Lemma again lead to methods that avoid the time/storage tradeoff incurred by the conditional expectation methods so they again provide viable construction algorithms, which we outline next.

Suppose that an $N \times k$ array A is specified. When A does not meet the requirements to be a locating or detecting array of the kind intended, some requirement has deficiency greater than 0. Following [25], we consider each requirement in an arbitrary but fixed order. If none is encountered that has nonzero deficiency, the array is the desired solution. Otherwise, the first time a requirement with nonzero deficiency is found, we identify all columns involved in all interactions of both sets, and randomly resample all of the entries in the same column. When this resampling occurs, checking is restarted at the first requirement and continues in the fixed order.

Moser and Tardos [25] show that the number of times resampling occurs is expected to be polynomial when the number of tests is that specified by the bound in the Lovász Local Lemma. As noted in [9], resampling can be applied when the number of tests is less than the bound, but in that case there is no guarantee that a solution can be found in a small expected number of resamplings (or indeed found at all). Nevertheless, resampling underlies a construction algorithm that typically reduces the number of tests below the bound.

To accomplish this, a random array is chosen with a number of tests equal to the bound. Column resampling proceeds as described until the array has deficiency 0. At this point, one test is removed, typically making the deficiency again nonzero, and a new round of column resampling is begun with the smaller number of tests.

In order to ensure termination, a threshold on the total number of column resamplings is set. Once this threshold is exceeded, the computation terminates

with the current number of tests. The deficiency of each requirement is recalculated every time this requirement is checked; no status information is stored for the requirements. Whereas the conditional expectation methods can employ the deficiency of an interaction up to k times for each test added, the column resampling methods limit this recalculation to the threshold. But the actual behaviour is much better than this worst case limit. Indeed, requirements that are later in the fixed order are checked only when all earlier requirements have deficiency 0. Hence although all requirements must be verified to ensure that the array is of the intended kind, typically a much smaller number is examined before we discover a requirement demonstrating that it is not. In practice, this results in a much smaller number of recalculations of deficiencies for requirements than one might have anticipated.

The method is flexible enough to permit construction for the variety of locating and detecting arrays described here, requiring less time and less storage than the conditional expectation methods (and less than any method that stores a status for each requirement). Naturally the question is whether such column resampling methods can yield useful test suites of an acceptable size. We address this in Sect. 4.

3.3 Local Optimization

Column resampling makes no explicit effort to reduce the deficiency, instead relying on the likelihood that a random replacement of the columns reduces the deficiency more than it increases it. When provided with an initial array of very low (but nonzero) deficiency having few tests, column resampling often increases the deficiency far more than it reduces it.

In order to study the effects of this, we examine a local optimization technique. At each iteration, we again consider each requirement in an arbitrary but fixed order. If none is encountered that has nonzero deficiency, the array is the desired solution. Otherwise, the first time a requirement with nonzero deficiency is found, we identify and randomly resample a column that is involved. If the resulting array has deficiency no larger than it did before, the new array replaces the old. Then no iteration increases the deficiency.

This shares the low storage footprint of the column resampling methods. The time invested is harder to compare. Although column resampling may make many resamplings that make the deficiency higher, each such resampling is triggered typically after few requirements are checked. In order to retain at least the progress made, local optimization checks every requirement at every iteration.

4 Some Computational Results

We implemented a (storage-intensive) conditional expectation method (Sect. 3.1), a column resampling method (Sect. 3.2), and a local optimization method (Sect. 3.3) for $(\overline{1}, \overline{2})$-locating arrays and for $(1, \overline{2})$-detecting arrays.

Our primary concerns are to (1) assess the effect of requiring larger separation on the numbers of tests required and (2) determine the feasibility of constructing locating and detecting arrays for scenarios with tens to hundreds of factors. In the discussion to follow, we repeatedly refer to Table 1.

The first column of Table 1 lists the types for which we applied one or more of the computational methods. These range from few factors (10) to a larger number (100). We select primarily types in which are numbers of levels are

Table 1. Numbers of tests in generated locating and detecting arrays. Testbed has type $5^9 4^5 3^7 2^3$; Simulation has type $10^8 9^1 8^4 7^5 6^{10} 5^4 4^6 3^9 2^{28}$.

Type	$(\overline{1}, \overline{2})$-locating					$(1, \overline{2})$-detecting				
	$\delta = 1$		$\delta = 2$	$\delta = 3$	$\delta = 4$	$\delta = 1$		$\delta = 2$	$\delta = 3$	$\delta = 4$
2^{10}	15	14	19	24	30	25	21	32	42	54
2^{15}	19	17	22	29	34	30	29	41	57	63
2^{20}	21	19	26	31	37	37	32	44	57	70
2^{50}	29	26	33	40	47	52	46	63	76	89
2^{75}	32	28	36	44	50	58	51	68		
2^{100}	34	31	39	46	53	62	57	74		
3^{10}	34	30	46	57	66	71	60	87	109	128
3^{15}	42	40	52	65	73	82	75	101	124	146
3^{20}	49	44	57	69	79	96	83	110	136	157
3^{50}	60	57	70	83	95	122	110	139	167	
3^{75}	66	62	76	90	103	135	121			
3^{100}	70	67	81	94	107	138	132			
4^{10}	71	65	86	104	122	135	118	161	201	235
4^{15}	78	76	96	116	133	153	139	185	219	
4^{20}	91	82	104	122	141	170	152	198		
4^{50}	113	106	129	148	168	217	196	244		
4^{75}	120	116	138	159	178	236	212			
4^{100}	127	123	146	166	187	248	236			
5^{10}	118	110	141	165	194	220	192	263	315	367
5^{15}	133	126	156	185	212	247	223	293		
5^{20}	150	138	169	198	225	275	243	313		
5^{50}	182	173	208	236		342	310	391		
5^{75}	198	189	223	256		367	337			
5^{100}	211	202	235	266		390	357			
$5^{10}2^{10}$	123	110	139	172	197	370	316	415	504	597
Testbed	117	114	144	169	194	313	266	367	450	533
Simulation	580	532	654			1883	1712			

equal, to observe the growth in number of tests as a function of the number of factors. We also report results for one mixed type, $5^{10}2^{10}$ to demonstrate how detecting arrays and locating arrays differ (more on this soon), and the two types from our motivating examples.

Sizes from the column resampling algorithm for separation distance 1 are reported in the first of the two columns under '$\delta = 1$', both for locating and detecting arrays, using a threshold of 1000 resamplings. As expected, the method runs relatively quickly, completing in less than a minute for type 3^{75}.

We employed the local optimization algorithm both to compare with column resampling for distance 1, and to extend to larger distances (up to $\delta = 4$ for many types). It proved to be somewhat more time consuming to complete 1000 iterations, for example taking approximately 6 min for the locating array of type 3^{75} with distance 1. Nevertheless, in all computations it yields a size smaller than that from column resampling.

Most interesting is the effect of increasing the separation distance. The sizes obtained suggest that one can do much better than replicating a solution with distance 1 δ times; in fact, for most of the types examined, enforcing distance 4 no more than doubles the number of tests used for distance 1 for locating arrays. This suggests that one can better cope with outliers and missing responses in experimentation using locating and detecting arrays, incurring a modest amount of additional testing.

Table 1 also illustrates substantial differences between locating arrays and detecting arrays. The efficient recovery algorithm for the latter appears to come at a high price. For types in which all factors have the same number of levels, the number of tests in a detecting array appears to be nearly twice the number for the corresponding locating array. However, for types in which factors have widely different numbers of levels, detection appears to cause a much larger increase; see the last three rows in Table 1. This is as one would anticipate, because when a factor has few levels, each level appears in many more tests on average. The likelihood that this larger set of tests contains all tests in which a much less frequently occurring 2-way interaction appears is consequently larger, necessitating a larger number of tests.

Finally we mention some results from the conditional expectation method. With our current implementation for detecting arrays with distance 1, storage and time limitations make it infeasible to handle large numbers of factors, so we report only a handful of results for few factors. For $k \in \{10, 15, 20\}$, while local optimization produces detecting arrays of type 3^k with 60, 75, and 83 tests, conditional expectation produces much smaller arrays of sizes 41, 48, and 54, respectively. Similar differences are found in other cases with few factors.

Nagamoto *et al.* [26] describe a greedy algorithm for $(1, 2)$-locating arrays of distance 1 that bears some resemblance to the conditional expectation method outlined here and apply it to a limited set of types with at most 20 factors. For $k \in \{10, 15, 20\}$, while local optimization produces locating arrays of type 5^k with 110, 136, and 148 tests, their greedy approach produces much smaller arrays of sizes 91, 105, and 113, respectively.

Evidently, column resampling and local optimization, at least within the number of iterations performed, produce numbers of tests that are far from minimum. Despite this, we have found them to yield acceptable results, better than expected in a randomly chosen array, for much larger numbers of factors than appear to be handled by existing methods.

Acknowledgements. This work is supported in part by the U.S. National Science Foundation grant #1421058, and in part by the Software Test & Analysis Techniques for Automated Software Test program by OPNAV N-84, U.S. Navy.

References

1. Aldaco, A.N., Colbourn, C.J., Syrotiuk, V.R.: Locating arrays: a new experimental design for screening complex engineered systems. SIGOPS Oper. Syst. Rev. **49**(1), 31–40 (2015)
2. Alon, N., Spencer, J.H.: The Probabilistic Method. Wiley-Interscience Series in Discrete Mathematics and Optimization, 3rd edn. Wiley, Hoboken (2008)
3. Bryce, R.C., Colbourn, C.J.: The density algorithm for pairwise interaction testing. Softw. Testing Verification Reliab. **17**, 159–182 (2007)
4. Bryce, R.C., Colbourn, C.J.: A density-based greedy algorithm for higher strength covering arrays. Softw. Testing Verification Reliab. **19**, 37–53 (2009)
5. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. **23**, 437–444 (1997)
6. Colbourn, C.J.: Covering array tables: $2 \leq v \leq 25$, $2 \leq t \leq 6$, $t \leq k \leq 10000$ (2005–2017). www.public.asu.edu/~ccolbou/src/tabby
7. Colbourn, C.J., Fan, B.: Locating one pairwise interaction: three recursive constructions. J. Algebra Comb. Discrete Struct. Appl. **3**, 125–134 (2016)
8. Colbourn, C.J., Fan, B., Horsley, D.: Disjoint spread systems and fault location. SIAM J. Discrete Math. **30**, 2011–2016 (2016)
9. Colbourn, C.J., Lanus, E., Sarkar, K.: Asymptotic and constructive methods for covering perfect hash families and covering arrays. Des. Codes Crypt. **86**, 907–937 (2018)
10. Colbourn, C.J., McClary, D.W.: Locating and detecting arrays for interaction faults. J. Comb. Optim. **15**, 17–48 (2008)
11. Colbourn, C.J., Syrotiuk, V.R.: Coverage, location, detection, and measurement. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 19–25. IEEE Press (2016)
12. Compton, R., Mehari, M.T., Colbourn, C.J., De Poorter, E., Syrotiuk, V.R.: Screening interacting factors in a wireless network testbed using locating arrays. In: IEEE INFOCOM International Workshop on Computer and Networking Experimental Research Using Testbeds (CNERT) (2016)
13. Erdős, P., Lovász, L.: Problems and results on 3-chromatic hypergraphs and some related questions. In: Infinite and Finite Sets, Colloq., Keszthely, vol. 2, pp. 609–627 (1973). Colloq. Math. Soc. János Bolyai, vol. 10, North-Holland, Amsterdam (1975)
14. 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**, 287–297 (2008)

15. Francetić, N., Stevens, B.: Asymptotic size of covering arrays: an application of entropy compression. J. Combin. Des. **25**, 243–257 (2017)
16. Godbole, A.P., Skipper, D.E., Sunley, R.A.: t-covering arrays: upper bounds and Poisson approximations. Comb. Probab. Comput. **5**, 105–118 (1996)
17. Jin, H., Tsuchiya, T.: Constrained locating arrays for combinatorial interaction testing. CoRR abs/1801.06041 (2018). http://arxiv.org/abs/1801.06041
18. Johnson, D.S.: Approximation algorithms for combinatorial problems. J. Comput. Syst. Sci. **9**, 256–278 (1974)
19. Konishi, T., Kojima, H., Nakagawa, H., Tsuchiya, T.: Finding minimum locating arrays using a SAT solver. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, 13–17 March 2017, pp. 276–277 (2017)
20. Kuhn, D.R., Kacker, R., Lei, Y.: Introduction to Combinatorial Testing. CRC Press, Boca Raton (2013)
21. Li, P.C., Meagher, K.: Sperner partition systems. J. Combin. Des. **21**(7), 267–279 (2013)
22. Lovász, L.: On the ratio of optimal integral and fractional covers. Discrete Math. **13**(4), 383–390 (1975)
23. Martínez, C., Moura, L., Panario, D., Stevens, B.: Locating errors using ELAs, covering arrays, and adaptive testing algorithms. SIAM J. Discrete Math. **23**, 1776–1799 (2009/2010)
24. Meagher, K., Moura, L., Stevens, B.: A Sperner-type theorem for set-partition systems. Electron. J. Combin. **12**, Note 20, 6 (2005). (Electronic)
25. Moser, R.A., Tardos, G.: A constructive proof of the general Lovász local lemma. J. ACM **57**(2), Article no. 11, 15 (2010)
26. Nagamoto, T., Kojima, H., Nakagawa, H., Tsuchiya, T.: Locating a faulty interaction in pair-wise testing. In: 20th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2014, Singapore, 18–21 November 2014, pp. 155–156 (2014)
27. Nayeri, P., Colbourn, C.J., Konjevod, G.: Randomized postoptimization of covering arrays. Eur. J. Comb. **34**, 91–103 (2013)
28. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2), #11 (2011)
29. Sarkar, K., Colbourn, C.J.: Two-stage algorithms for covering array construction. CoRR abs/1606.06730 (2016). http://arxiv.org/abs/1606.06730
30. Sarkar, K., Colbourn, C.J.: Upper bounds on the size of covering arrays. SIAM J. Discrete Math. **31**, 1277–1293 (2017)
31. Seidel, S.A., Mehari, M.T., Colbourn, C.J., De Poorter, E., Moerman, I., Syrotiuk, V.R.: Analysis of large-scale experimental data from wireless networks. In: IEEE INFOCOM International Workshop on Computer and Networking Experimental Research Using Testbeds (CNERT) (2018)
32. Shi, C., Tang, Y., Yin, J.: Optimal locating arrays for at most two faults. Sci. China Math. **55**(1), 197–206 (2012)
33. Stein, S.K.: Two combinatorial covering theorems. J. Comb. Theory Ser. A **16**, 391–397 (1974)
34. Tang, Y., Colbourn, C.J., Yin, J.: Optimality and constructions of locating arrays. J. Stat. Theory Pract. **6**(1), 20–29 (2012)