Sequence Covering Arrays and Linear Extensions

Patrick C. Murray and Charles J. Colbourn(✉)

School of Computing, Informatics, and Decision Systems Engineering,
Arizona State University, P.O. Box 878809, Tempe, AZ 85287, USA
colbourn@asu.edu

Abstract. Covering subsequences by sets of permutations arises in numerous applications. Given a set of permutations that cover a specific set of subsequences, it is of interest not just to know how few permutations can be used, but also to find a set of size equal to or close to the minimum. These permutation construction problems have proved to be computationally challenging; few explicit constructions have been found for small sets of permutations of intermediate length, mostly arising from greedy algorithms. A different strategy is developed here. Starting with a set that covers the specific subsequences required, we determine local changes that can be made in the permutations without losing the required coverage. By selecting these local changes (using linear extensions) so as to make one or more permutations less 'important' for coverage, the method attempts to make a permutation redundant so that it can be removed and the set size reduced. A post-optimization method to do this is developed, and preliminary results on sequence covering arrays show that it is surprisingly effective.

1 Introduction

In order to motivate our study, consider the following question from [2]: Given an n-vertex m-edge graph G, what is the smallest number k of dimensions so that m axis-parallel k-dimensional boxes in $\mathbb{R}^k$ can be found whose intersection graph is the line graph of G? Remarkably, they recast this as a question about permutations: What is the smallest number of permutations so that for every two vertex-disjoint edges $\{w, x\}$ and $\{y, z\}$ of G, in at least one of the permutations w and x both precede y and z, or w and x both follow y and z? Similar problems abound. In [17], in a problem in event sequence testing, one asks for the fewest permutations so that for each of the $t!$ orders of each subset of t elements, some permutation contains the specified elements in the specified order.

In numerous problems of this type, strong asymptotic bounds on the minimum number of permutations as a function of the length of the permutations have been established. Our concern here is quite different; for any practical application we must explicitly construct a set of permutations, and asymptotic results are often not well suited to addressing construction problems for moderate lengths. Probabilistic arguments typically establish that choosing a certain number of permutations uniformly at random can yield a non-zero probability of success; yet for practical purposes this is not satisfactory, because the number

J. Kratochvíl et al. (Eds.): IWOCA 2014, LNCS 8986, pp. 274–285, 2015.
DOI: 10.1007/978-3-319-19315-1_24

of permutations required to have a reasonable chance of success is often much more than the minimum. To treat the construction of such sets of permutations, one avenue is to find mechanisms to translate knowledge about related combinatorial structures to underlie a direct construction, and another is to make sets of permutations of greater length recursively from those with smaller. General mechanisms to do this are not known, although in specific cases these can provide fast construction of sets of permutations far smaller than those from random selections. But when such direct or recursive constructions are not available, one resorts to computation.

Not surprisingly, exact methods such as backtracking or integer programming can be applied effectively only for small lengths and simple requirements. Heuristic methods extend the size and complexity of problems for which useful sets of permutations can be found. As lengths increase and requirements become more complex, best known results often arise from greedy algorithms that select one permutation at a time. In the problems being discussed, even deciding what is the best permutation to include next can be challenging, and hence greedy methods often make selections that are sub-optimal even locally.

This is not a good state of affairs. If we want to test sequences of 75 events so that every four of them appears in each of the 24 orders, we need an explicit set of permutations, and the best method in the literature to find them is a greedy method that selects permutations sub-optimally (but efficiently). In cases for which this greedy strategy has been implemented [5], it provides the smallest known sets of permutations in broad ranges of interest, despite the myopic nature of greedy methods and the sub-optimal selection of permutations. How can we do better? Rather than trying to improve the greedy methods further, we propose a different local optimization approach that we call *post-optimization*. Post-optimization repeatedly modifies the set of permutations with the general goal of making one of the permutations "less useful" in meeting the requirements. Ultimately, if it can be made redundant, it is deleted and the set size is reduced.

Our strategy determines what each permutation contributes, expresses this information as a partial order, and chooses a linear extension (which is ensured to contribute at least as much). This strategy can be effectively implemented, but the real surprise is that it reduces the number of permutations in best known solutions for event sequencing, sometimes dramatically.

The remainder of the paper is organized as follows. In Sect. 2 we give a precise formulation of a general set of problems and then discuss a special case, the sequence covering arrays that arise in event sequence testing. Then in Sect. 3 we focus on sequence covering arrays; we describe the current state of computational methods, and motivate the idea of post-optimization. Section 4 develops a framework and some details for the post-optimization method using linear extensions, and Sect. 5 describes preliminary computational results that are quite encouraging.

2 Background

Let $\Sigma = \{0, \ldots, v-1\}$ be a set of *symbols* or *elements*. A *t-subsequence* of Σ is a t-tuple $(x_1, \ldots, x_t)$ with $x_i \in \Sigma$ for $1 \leq i \leq t$, and $x_i \neq x_j$ when $i \neq j$.

A *genus* of t-subsequences for the t-subset $\{x_1, \ldots, x_t\}$ is a non-empty subset of $\{(y_1, \ldots, y_t) : \{y_1, \ldots, y_t\} = \{x_1, \ldots, x_t\}\}$, A permutation π of Σ *covers* the t-subsequence $(x_1, \ldots, x_t)$ if $\pi^{-1}(x_i) < \pi^{-1}(x_j)$ whenever $i < j$. A permutation *covers* a genus when it covers one of the t-subsequences in the genus.

By way of example, suppose that $v = 8$ and $t = 4$. Then $(2, 4, 3, 5)$ is a 4-subsequence, and one genus of subsequences for $\{2, 3, 4, 5\}$ is $\{(y_1, y_2, y_3, y_4) : \{\{y_1, y_2\}, \{y_3, y_4\}\} = \{\{2, 4\}, \{3, 5\}\}\}$. This genus contains eight 4-subsequences ((2,4,3,5), (2,4,5,3), (4,2,3,5), (4,2,5,3), (3,5,2,4), (3,5,4,2), (5,3,2,4), and (5,3,4,2)). The permutation $\pi = 40251376$ has $\pi^{-1}(2) = 2$, $\pi^{-1}(3) = 5$, $\pi^{-1}(4) = 0$, and $\pi^{-1}(5) = 3$. It therefore covers the 4-subsequence $(4, 2, 5, 3)$, and hence covers the genus given. However, the same permutation fails to cover the genus $\{(y_1, y_2, y_3, y_4) : \{\{y_1, y_2\}, \{y_3, y_4\}\} = \{\{2, 3\}, \{4, 5\}\}\}$.

Let $\mathcal{G}$ be a set of genera of t-subsequences on symbols Σ. A set $\Pi = \{\pi_1, \ldots, \pi_N\}$ of N permutations is a *$\mathcal{G}$-permutation covering* if, for every genus $G \in \mathcal{G}$, there exists a permutation $\pi_j \in \Pi$ for which π_j covers genus G. To continue the example, the *box genus*, $box(\{x_1, x_2\}, \{x_3, x_4\})$, is $\{(y_1, y_2, y_3, y_4) : \{\{y_1, y_2\}, \{y_3, y_4\}\} = \{\{x_1, x_2\}, \{x_3, x_4\}\}\}$. Let

$$\mathcal{B}_\Sigma = \{box(\{x_1, x_2\}, \{x_3, x_4\}) : \{x_1, x_2, x_3, x_4\} \text{ is a 4-subset of } \Sigma\}.$$

A $\mathcal{B}_{\{0,\ldots,7\}}$-permutation covering is:

4 3 2 0 6 1 5 7
3 5 0 6 2 1 7 4
7 3 0 1 2 6 4 5
4 0 2 5 1 3 7 6
2 7 5 4 0 6 3 1

Genus $box(\{2, 4\}, \{3, 5\})$ is covered by 40251376. Now $box(\{2, 3\}, \{4, 5\})$ is covered by 73012645 and $box(\{2, 5\}, \{3, 4\})$ is covered by 27540631. In this example, there are 1680 4-subsequences forming 210 box genera, so the remaining verification is best left to a machine.

No matter what genera are to be covered, some number of permutations suffices to cover them, because each t-subsequence has a linear extension to a permutation that necessarily covers the subsequence. Our interest is to determine the minimum number of permutations needed to cover a set $\mathcal{G}$ of genera.

Many permutation covering problems have been explored; we mention a few. Dushnik [8] examined the existence of sets of permutations in which for every subset S of k elements and every element σ not in this subset, one permutation has all elements in S preceding σ. In other words, he examined $\mathcal{G}_k$-permutation coverings with $\mathcal{G}_k = \{(x_1, \ldots, x_k, x_{k+1}) : \{x_1, \ldots, x_k\} = S\}$ for $S \cup \{x_{k+1}\}$ a $(k+1)$-subset of $\Sigma\}$. Spencer [25] established bounds on the number of permutations needed. In defining the "dimension of hypergraphs," Fishburn and Trotter [10] examined coverage of fewer genera in which the set S corresponds to hyperedges of an input hypergraph.

Füredi [11] explored *3-mixing sets*, which are $\mathcal{M}$-permutation coverings with $\mathcal{M} = \{(x_1, x_3, x_2), (x_2, x_3, x_1) : \{x_1, x_2, x_3\}$ is a 3-subset of $\Sigma\}$. In other words, for every pair of elements $\{a, b\}$ and third element c, there must be a permutation in which c is between a and b; see also [23] and [6].

Basaravaju *et al.* [2] discuss the relevance of these permutation coverings to geometric representations of graphs and hypergraphs. In particular, they define the *separation dimension* of a graph in a manner equivalent to the following. Let $G = (V, E)$ be a graph. Let $\mathcal{B}_G$ be the set of all box genera $box(\{x_1, x_2\}, \{x_3, x_4\})$ for which $\{x_1, x_2\}$ and $\{x_3, x_4\}$ are vertex-disjoint edges of G. They establish that the smallest number of permutations in a $\mathcal{B}_G$-permutation covering, which they term the separation dimension of G, is precisely the same as the "boxicity" of the line graph of G.

Recently, applications of permutation coverings in event sequence testing have attracted attention as well. In this case, the genera each contain a single t-subsequence, so the terminology need not refer to genera at all. A *sequence covering array* of *order* v and *strength* t, or $\mathsf{SeqCA}(N; t, v)$, is a set $\Pi = \{\pi_1, \ldots, \pi_N\}$ where π_i is a permutation of Σ, and every t-subsequence of Σ is covered by at least one of the permutations $\{\pi_1, \ldots, \pi_N\}$. Often the permutations are written as an $N \times v$ array. (Every permutation of every t of the v letters appears in the specified order in at least one of the N permutations.) Kuhn *et al.* [17,18] describe the application to testing. Suppose that a process involves a sequence of v tasks or events. The operator may perform the tasks in an incorrect order, resulting in system faults. Often errors result from the improper ordering of a small number of events. When each permutation of a sequence covering array is used as a test order for the events, if faults result from the improper ordering of t or fewer tasks and are not masked, the presence of a fault will be detected. To reduce testing cost, we examine $\mathsf{SeqCAN}(t, v)$, the smallest N for which a $\mathsf{SeqCA}(N; t, v)$ exists.

Spencer [25] examined equivalent sets of permutations, *completely t-scrambling permutations*, obtained by interchanging the roles of symbols and columns in a sequence covering array [5]. For subsequent research, see Füredi [11], Ishigami [14,15], Radhakrishnan [24], and Tarui [26]. Chee *et al.* [5] explore the relationship to so-called "directed t-coverings" as well.

From this point onwards, we restrict to cases when, for some strength t, every t-subsequence appears in one of the genera to be covered. Even then, for each of the permutation covering problems mentioned thus far, few exact results for the fewest permutations needed are known. More precisely, when each genus (of strength t) contains all $t!$ orderings, a single permutation suffices to cover all genera. When the genera can be named as $G_1, \ldots G_g, H_1, \ldots H_g$ so that for $1 \leq i \leq g$, $G_i \cup H_i$ contains all $t!$ orderings of t symbols, and both G_i and H_i contain each t-subsequence of these t symbols or its reversal, two permutations suffice: Simply take any permutation and its reversal. In these "trivial" situations, there is no dependence on the size of Σ. When the strength t is at most two, only these trivial cases arise. When Σ is "small," exact values are also sometimes known. For example, $\mathsf{SeqCAN}(t, t+1) = t!$ [19].

Unfortunately, except in these situations, current knowledge of sizes of minimum permutation coverings has focussed on asymptotic results, determining the relationship between the size of the covering and the number of symbols,

as the latter goes to infinity. While informative, these methods typically do not provide explicit solutions for small numbers of symbols. Yet in the testing application, the construction of sequence covering arrays is essential. In [26], an elegant direct construction for $\mathsf{SeqCA}(N; 3, v)$ is given that, while typically the smallest known, is known not to realize the minimum when v is small [5]. In [5], a direct construction produces a $\mathsf{SeqCA}(N + M; 3, vw)$ from a $\mathsf{SeqCA}(N; 3, v)$ and a $\mathsf{SeqCA}(M; 3, w)$; occasionally this produces the smallest sequence covering array that is known, but it does not do so in general. For strength $t \geq 4$, no such direct or recursive methods are known. Hence we turn to computational methods. Although we focus on sequence covering arrays to make the presentation more self-contained, most of the method to be described operates *mutatis mutandis* for permutation coverings with more complicated genera.

3 Computational Constructions

In [18], a simple greedy method is used to compute upper bounds on $\mathsf{SeqCAN}(t, v)$ for $t \in \{3, 4\}$ and small values of v. A more sophisticated greedy method was developed by Erdem *et al.* [9]. A conditional expectation greedy algorithm that derandomizes a randomized method establishes:

Theorem 1. [5] *For fixed t and input v, there is an algorithm to construct a $\mathsf{SeqCA}(N; t, v)$ having at most $N \leq 2(\log(\frac{v!}{(v-t)!}))/(\log(\frac{t!}{t!-2}))$ permutations in time that is polynomial in v.*

Chee *et al.* [5] observe that the bound in Theorem 1 is quite pessimistic. Implementation of the conditional expectation methods yields substantially better results for $t \in \{3, 4, 5\}$ than guaranteed by the theorem. One might expect that a greedy method can fare well, but it appears unlikely that it will yield optimum coverings. Indeed, using answer set programming, improvements on the greedy methods have been developed when $t \in \{3, 4\}$ [1,3,9]. A cooperative search strategy (the "bees algorithm") is explored in [12]. These methods are compared in [5], and we summarize the conclusions here.

For strength $t = 3$, the answer set programming methods outperform all of the greedy methods. Nevertheless when $v \geq 30$ they do not fare as well as Tarui's direct construction or the recursive construction. Tarui's direct construction is not optimum, however; for small values of v, answer set programming wins, and for certain values of v (such as $v = 128$), the recursive method wins.

Proceeding to strengths four and five, no direct or recursive method is available. Surprisingly, the answer set programming methods do not report the best known results except when v is very small; the conditional expectation greedy method yields the best known result. The explanation is almost certainly that the time and storage required for the answer set programming methods and the cooperative search methods are prohibitive.

15	2	3	0	4	1	5	15	5	4	0	1	3	2
15	3	1	4	0	2	5	15	1	0	5	2	3	4
15	4	3	5	2	1	0	15	2	0	5	1	4	3
15	4	1	2	5	3	0	15	1	3	5	0	4	2
15	5	0	2	4	3	1	15	0	3	2	1	5	4
15	3	2	4	5	0	1	14	1	2	4	0	3	5
15	5	3	1	2	0	4	14	0	3	4	5	1	2
13	0	4	2	1	3	5	13	5	2	1	3	4	0
12	4	2	0	5	3	1	12	0	1	4	5	3	2
12	2	3	5	4	1	0	11	4	3	0	1	5	2
10	2	1	5	0	3	4	9	1	5	4	2	3	0
10	4	5	1	0	2	3	9	5	3	4	0	2	1
7	3	0	5	2	1	4	7	0	1	3	2	4	5
5	2	1	0	4	5	3	5	4	1	3	2	0	5
4	0	2	5	4	1	3	4	5	2	0	3	1	4
3	3	1	4	5	2	0	3	0	5	1	4	2	3
2	1	0	3	2	5	4	1	2	5	4	3	0	1

The conditional expectation algorithm is greedy, and hence it is reasonable to expect that the later permutations chosen are less useful in the coverage of t-subsequences. Let us examine this more carefully. Consider the SeqCA(34;4,6) shown at left; the permutations are shown in the last six columns. Although the permutation covering need not order the permutations, in the sequence covering array they are ordered, and the greedy method added these permutations in this order. Therefore we can count, for each permutation, the 4-subsequences covered by this permutation that are covered by no earlier one (this is precisely the quantity that the greedy algorithm attempts to maximize). These counts are shown in the first column on the left.

No permutation in this example can cover more than $\binom{6}{4} = 15$ 4-subsequences. Mathon and Tran Van Trung [21] give a SeqCA(24;4,6) in which every permutation necessarily covers 15 4-subsequences for the first time. However, the myopic nature of the greedy method has resulted in permutations that cover fewer and fewer 4-subsequences for the first time, so that the last only covers a single 4-subsequence.

When the permutations are listed in this order, the last appears to be less useful. Can we avoid using some of these later permutations? The last permutation covers only the 4-subsequence $(2, 4, 3, 0)$ for the first time, and hence any of 30 different permutations would serve as well. A similar problem arises in the construction of related combinatorial objects known as covering arrays. In that setting, Nayeri *et al.* [22] devised a post-optimization method, which repeatedly reorders the array, attempting to reduce the amount of coverage required from the last row. If the last row can be made to provide no coverage not provided by an earlier row, it can be deleted to yield a solution with fewer rows. Surprisingly, this works well! For a variety of covering arrays from different constructions, such post-optimization eliminates many rows, sometimes more than 10 % of the rows in an initial (best known) solution. Arguably this is because the best known solutions can often be far from optimal due to deficiencies in the constructions that we know; nevertheless in that context such post-optimization has proved useful. Indeed covering arrays arise as a type of "t-restriction" problem, and post-optimization is effective more generally for such problems [7].

4 Post-Optimization and Linear Extensions

The main contribution of this paper is to develop a post-optimization technique for sequence covering arrays. Define the *effective coverage* of a permutation in an ordered list of permutations to be the number of t-subsequences covered by this permutation but by no earlier one. The basic algorithm follows:

repeat until "termination condition"
  choose an ordering of the permutations $\pi_1, \ldots, \pi_N$.
  compute the effective coverage for each permutation.
  select a permutation with least effective coverage and place it last in the order.
  repeat until "iteration condition"
    if any permutation has effective coverage zero, delete it.
    for each permutation π, replace π by any permutation that covers all of the t-subsequences covered by π for the first time.
    reorder the first $N-1$ permutations
    recompute the effective coverage for each permutation.
    if any permutation has effective coverage lower than the last one, interchange it with the last one.

As an iteration condition, we terminate the inner loop when at least one permutation is removed or when an iteration limit is exceeded. The termination condition enforces a limit on the number of times a complete reordering of the array is undertaken. The determination of specific iteration and termination conditions dictate the number of times that an improvement is attempted, and can be set based on experimentation (that we do not describe here). We adopt random reordering. While it is reasonable to instead reorder so that permutations with larger effective coverage appear earlier in the ordering, we found that the method appeared more likely to be trapped in a local optimum and fail to make improvement.

The key aspect of the algorithm is to determine when a permutation can be replaced by another without loss of (effective) coverage. Patterned on the approach for covering arrays [7], call an entry σ in a permutation π_j *necessary* if there is some t-subsequence containing σ that is covered by π_j for the first time, and *flexible* otherwise. By iterating through all t-subsequences, finding their first occurrence in the array, and marking the t corresponding symbols as necessary, any symbol left unmarked is flexible. Any permutation π that contains the necessary elements in π_j in the same order can be used to replace π_j without reducing the effective coverage. Hence a basic post-optimization method can locate all flexible symbols in permutations, and move them (perhaps randomly) within the permutation; the result remains a sequence covering array.

Despite the fact that the SeqCA(34;4,6) is far from optimal, it has only two flexible symbols (1 and 5), both in the last permutation. Nevertheless, it has much more flexibility; we pursue this next. For each permutation π_j define a partial order $\prec_j$ on Σ so that whenever $(x_1, \ldots, x_t)$ is covered for the first time by π_j, we have $x_i \prec_j x_{i+1}$ for $1 \leq i < t$. Evidently π_j is a linear extension of $\prec_j$, but *any* linear extension of $\prec_j$ serves to provide at least the same effective coverage. In our SeqCA(34;4,6) example, the last permutation has partial order $2 \prec 4 \prec 3 \prec 0$ incomparable to 1 and 5. (This just restates the earlier observation about flexible symbols.) The second last permutation has effective coverage 2 (covering (1,0,3,2) and (0,3,5,4)) and partial order $1 \prec 0 \prec 3 \prec 5 \prec 4$ and $3 \prec 2$. While there are no flexible symbols, the partial order has three linear extensions. The third last permutation, with effective coverage of 3 (covering

(0,1,2,3), (0,5,1,4), and (5,1,2,3)), has partial order $0 \prec 5 \prec 1 \prec 2 \prec 3$ and $1 \prec 4$, again with three linear extensions. The fourth last permutation, with effective coverage of 3 (covering (3,1,2,0), (3,1,5,0), and (3,4,5,2)), has partial order $3 \prec 1 \prec 5 \prec 2 \prec 0$ and $3 \prec 4 \prec 5$, with two linear extensions.

The partial orders $\prec_j$ are computed at the same time as the effective coverage. Replacement of permutations is carried out by choosing a linear extension of $\prec_j$ to replace π_j. A strategy for choosing the 'best' linear extension for each of the partial orders is not clear; deterministic rules appear to stall the method in local optima. Hence one might prefer random linear extensions. Counting linear extensions is #P-complete [4], but there is a polynomial expected time algorithm for generating a random linear extension [13,16]. We do not need such sophisticated machinery, because we have no need for extensions to be selected uniformly at random. For our purposes, any method that has non-zero probability of obtaining each linear extension suffices. Such a method is easy: A simple greedy algorithm that repeatedly selects a minimum element and removes it exhibits this behaviour.

One improvement is worth mentioning. Rather than computing all of the partial orders $\prec_j$ and then forming a linear extension of each, once each partial order $\prec_j$ is determined, we immediately replace it by a linear extension. This can sometimes cover an additional t-subsequence previously only covered by a later permutation, reducing its effective coverage. Indeed, if in considering partial order $\prec_j$ some t-subsequence T covered only in the last permutation is consistent with $\prec_j$, we add the comparability relations from T to $\prec_j$ before choosing a linear extension; in this way, we guarantee that the last permutation has less effective coverage than before (making it a better candidate for removal). Further effort could be made to search for t-subsequences covered only after the jth permutation that are consistent with $\prec_j$ to make some permutation other than the last have lower effective coverage, but this can require checking a large number of t-subsequences for consistency with $\prec_j$. We have concentrated instead on reducing the effective coverage of the last permutation.

With these implementation decisions, the algorithm requires only $O(vN)$ storage for the array and $O(v^2)$ storage for the partial orders. The time is dominated by the time to determine the partial orders, which involves examining up to N permutations for $O(v^t)$ different t-subsequences. While this appears to be quite large, a single iteration of the post-optimization involves essentially the same effort as checking that the array is indeed a sequence covering array. In practice, the execution time depends not only on the time per iteration of the inner loop, but the termination and iteration conditions that determine the number of iterations. Our interest is not in theoretical efficiency, although our decisions have been guided by ensuring that a single iteration is not too computationally intensive; rather our concern is with whether the post-optimization method can be used in a practical sense to improve our knowledge about sequence covering arrays. For this, we turn to some computational results.

5 Some Computations

The objective is to find best explicit constructions for small sequence covering arrays, so the 'acid test' for post-optimization is whether it can improve upon the current best sequence covering numbers, and to what extent. We have implemented the method, and performed a number of preliminary experiments for strengths 3, 4, and 5 in the ranges of lengths reported in [5]. Results are reported in Table 1.

Table 1. Post-optimization for strengths 3, 4, 5, and 6

$t = 3$				$t = 4$				$t = 5$				$t = 6$		
v	Best	In	Out	v	Best	In	Out	v	Best	In	Out	v	In	Out
4	6	8	6	5	24	26	24	6	120	148	122	7	991	836
5	7	8	7	6	24	34	24	7	198	198	175	8	1342	1179
6	8	10	8	7	38	41	37	8	242	242	218	9	1662	1535
7	8	12	8	8	44	47	42	9	282	284	261	10	1970	1873
8	9	12	9	9	50	52	46	10	318	322	294			
9	9	12	9	10	55	57	53	11	354	354	330			
10	9	14	10	15	78	78	67	12	384	386	360			
15	10	15	12	20	92	92	80	13	416	419	390			
20	12	16	13	25	104	104	90	14	446	446	418			
25	14	18	14	30	113	113	98	15	470	475	448			
30	14	19	15	40	128	128	112	16	496	501	474			
40	16	21	16	50	141	141	123	17	518	518	496			
50	16	23	17	60	151	151	133	18	540	547	520			
60	16	26	18	70	160	160	142	19	560	570	541			
70	16	28	19	80	168	168	150	20	582	590	568			
80	17	30	20	90	176	180	162	25	674	674	656			
90	18	30	21					30	748	748	725			

For each strength, different lengths are examined. For each, `Best` reports the best result known from [5], `In` reports the size of the array – usually produced by the conditional expectation greedy algorithm – to which post-optimization is applied, and `Out` reports the size when post-optimization was terminated.

First consider the results for strength $t = 3$. In this case, more sophisticated computational methods have earlier been applied, and useful direct and recursive constructions are known. Perhaps then it is no surprise that post-optimization has not improved upon any of the best known sizes. The improvements upon the sizes produced by the greedy method are nevertheless substantial. When $v = 80$, for example, the conditional expectation method gives a SeqCA(30;3,80);

ensuring that reversals are present yields a SeqCA(26;3,80) [5]. The answer set programming techniques give a SeqCA(24;3,80) [1] and a SeqCA(23;3,80) [3]. Post-optimization improves the SeqCA(30;3,80) by removing ten permutations, yielding a SeqCA(20;3,80). While Tarui's method [26] yields a SeqCA(18;3,80) and the recursive construction a SeqCA(17;3,80) [5], it remains striking that post-optimization fares so well.

Turning to strengths 4 and 5 shows the potential of post-optimization. In each of the cases examined, post-optimization matches or improves upon the best known result. In light of the comparison with the direct and recursive constructions for strength 3, it is unlikely that post-optimization has produced optimal arrays except when v is quite small. Nevertheless, in the absence of such constructions for strength greater than 3, it does yield the best known sizes, giving a non-trivial improvement on other methods applied.

Even when powerful direct or recursive constructions (as for strength three) are known, post-optimization may nevertheless prove useful. If only some of the t-subsequences are to be covered ("partial coverage"), the direct and recursive constructions do not exploit this, whereas post-optimization can eliminate permutations while ensuring that every t-subsequence that is covered initially remains covered. In principle, there is no obstacle to incorporating constraints as well. A *constraint* is an ℓ-subsequence that is not permitted to appear in any permutation. Sets of constraints may make it impossible to find an array covering some specified t-subsequences consistent with the constraints [5,20]. When some array with appropriate coverage meeting all constraints does exist, however, post-optimization can be applied by ensuring that every linear extension chosen does not violate any constraint. We have conducted limited experiments with partial coverage and with constraints; the extensions are natural. We have also conducted a more thorough set of experiments with covering various genera, such as the box genus. Our computational results, not discussed here, demonstrate that post-optimization is practical and effective in these problems. We argue that the ability to cope with such variants is a positive feature of post-optimization.

Finally we remark on execution times. Every iteration of post-optimization on a SeqCA(180;4,90) examines 61,324,560 4-subsequences, determining for each the first permutation in which it is covered and forming 180 partial orders on 90 elements each. Linear extensions of each order are then selected and a random reordering of the permutations is done before going on to the next iteration. Thus each iteration can be substantial. The results shown reflect computations after between 1000 and 10000 iterations for the most part, yielding times that are comparable to the initial construction cost by the conditional expectation method. Because our concern until this point has been with the extent of improvement possible, we have not optimized execution times. We plan a more detailed examination of the time to conduct one iteration, and the numbers of iterations employed to see different reductions.

As a proof of concept for post-optimization, we believe that the results shown succeed in demonstrating its potential.

6 Conclusions

In describing and implementing post-optimization, we have concentrated on sequence covering arrays. The extensions to requiring specified partial coverage, and to incorporating constraints on the ordering of pairs, are immediate. The extension to permutation coverings with more complicated genera follows similar lines. Both will be reported fully elsewhere.

At the outset, we asked a question: How ought one, in practice, construct a 'small' set of permutations of specified length with specified coverage properties? When the length and the strength of coverage are small enough, exhaustive methods will do. For somewhat larger strength and length, clever metaheuristic methods apply. For large enough length, randomized methods can be used. But sadly there typically remains a substantial intermediate range of lengths for which none of these methods applies. In these cases, randomized methods yield far too many permutations. Random selection gives a SeqCA(361;4,90), but greedy methods produce a SeqCA(176;4,90). In any real application the reduction from 361 to 176 is important. Our post-optimization provides a mechanism to obtain even smaller solutions, in this case a SeqCA(162;4,90).

Naturally one prefers powerful explicit constructions in the intermediate range of interest, as has been done in part for sequence covering arrays of strength 3. However, for most of the permutation coverage problems mentioned here, such powerful explicit constructions remain elusive; this is particularly the case when considering partial coverage or constraints. Our argument is that a sensible and practical strategy in these situations is to first apply a greedy method to get a 'reasonably sized' initial array, and then to post-optimize it.

Acknowledgments. Thanks to Sunil Chandran, Marty Golumbic, Rogers Mathew, and Deepak Rajendraprasad for interesting discussions about permutation coverings and geometric representations of graphs and hypergraphs.

References

1. Banbara, M., Tamura, N., Inoue, K.: Generating event-sequence test cases by answer set programming with the incidence matrix. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012), pp. 86–97 (2012)
2. Basavaraju, M., Chandran, L.S., Golumbic, M.C., Mathew, R., Rajendraprasad, D.: Boxicity and separation dimension. In: Kratsch, D., Todinca, I. (eds.) WG 2014. LNCS, vol. 8747, pp. 81–92. Springer, Heidelberg (2014)
3. Brain, M., Erdem, E., Inoue, K., Oetsch, J., Pührer, J., Tompits, H., Yilmaz, C.: Event-sequence testing using answer-set programming. Int. J. Adv. Softw. **5**(3–4), 237–251 (2012)
4. Brightwell, G., Winkler, P.: Counting linear extensions. Order **8**(3), 225–242 (1991)
5. Chee, Y.M., Colbourn, C.J., Horsley, D., Zhou, J.: Sequence covering arrays. SIAM J. Discrete Math. **27**(4), 1844–1861 (2013)
6. Chor, B., Sudan, M.: A geometric approach to betweenness. SIAM J. Discrete Math. **11**(4), 511–523 (1998)

7. Colbourn, C.J., Nayeri, P.: Randomized Post-optimization for t-Restrictions. In: Aydinian, H., Cicalese, F., Deppe, C. (eds.) Ahlswede Festschrift. LNCS, vol. 7777, pp. 597–608. Springer, Heidelberg (2013)
8. Dushnik, B.: Concerning a certain set of arrangements. Proc. Amer. Math. Soc. **1**, 788–796 (1950)
9. Erdem, E., Inoue, K., Oetsch, J., Pührer, J., Tompits, H., Yilmaz, C.: Answer-set programming as a new approach to event-sequence testing. In: Proceedings of the Second International Conference on Advances in System Testing and Validation Lifecycle, pp. 25–34. Xpert Publishing Services (2011)
10. Fishburn, P.C., Trotter, W.T.: Dimensions of hypergraphs. J. Combin. Theory Ser. B **56**(2), 278–295 (1992)
11. Füredi, Z.: Scrambling permutations and entropy of hypergraphs. Random Struct. Alg. **8**(2), 97–104 (1996)
12. Hazli, M.M.Z., Zamli, K.Z., Othman, R.R.: Sequence-based interaction testing implementation using bees algorithm. In: 2012 IEEE Symposium on Computers and Informatics, pp. 81–85. IEEE (2012)
13. Huber, M.: Fast perfect sampling from linear extensions. Discrete Math. **306**(4), 420–428 (2006)
14. Ishigami, Y.: Containment problems in high-dimensional spaces. Graphs Combin. **11**(4), 327–335 (1995)
15. Ishigami, Y.: An extremal problem of d permutations containing every permutation of every t elements. Discrete Math. **159**(1–3), 279–283 (1996)
16. Karzanov, A., Khachiyan, L.: On the conductance of order Markov chains. Order **8**(1), 7–15 (1991)
17. Kuhn, D.R., Higdon, J.M., Lawrence, J.F., Kacker, R.N., Lei, Y.: Combinatorial methods for event sequence testing. CrossTalk: J. Defense Software Eng. **25**(4), 15–18 (2012)
18. Kuhn, D.R., Higdon, J.M., Lawrence, J.F., Kacker, R.N., Lei, Y.: Combinatorial methods for event sequence testing. In: IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 601–609 (2012)
19. Levenshteĭn, V.I.: Perfect codes in the metric of deletions and insertions. Diskret. Mat. **3**(1), 3–20 (1991)
20. Margalit, O.: Better bounds for event sequence testing. In: The 2nd International Workshop on Combinatorial Testing (IWCT 2013), pp. 281–284 (2013)
21. Mathon, R.: Tran Van Trung: Directed t-packings and directed t-Steiner systems. Des. Codes Cryptogr. **18**(1–3), 187–198 (1999)
22. Nayeri, P., Colbourn, C.J., Konjevod, G.: Randomized postoptimization of covering arrays. Eur. J. Comb. **34**, 91–103 (2013)
23. Opatrný, J.: Total ordering problem. SIAM J. Comput. **8**(1), 111–114 (1979)
24. Radhakrishnan, J.: A note on scrambling permutations. Random Struct. Alg. **22**(4), 435–439 (2003)
25. Spencer, J.: Minimal scrambling sets of simple orders. Acta Math. Acad. Sci. Hungar. 22, 349–353 (1971/72)
26. Tarui, J.: On the minimum number of completely 3-scrambling permutations. Discrete Math. **308**(8), 1350–1354 (2008)