Randomized Post-optimization for t-Restrictions

Charles J. Colbourn and Peyman Nayeri

School of Computing, Informatics, and Decision Systems Engineering Arizona State University P.O. Box 878809, Tempe, Arizona 85287-8809
{colbourn,nayeri}@asu.edu

Dedicated to the memory of Rudolf Ahlswede

Abstract. Search, test, and measurement problems in sparse domains often require the construction of arrays in which every t or fewer columns satisfy a simply stated combinatorial condition. Such *t-restriction problems* often ask for the construction of an array satisfying the t-restriction while having as few rows as possible. Combinatorial, algebraic, and probabilistic methods have been brought to bear for specific t-restriction problems; yet in most cases they do not succeed in constructing arrays with a number of rows near the minimum, at least when the number of columns is small. To address this, an algorithmic method is proposed that, given an array satisfying a t-restriction, attempts to improve the array by removing rows. The key idea is to determine the necessity of the entry in each cell of the array in meeting the t-restriction, and repeatedly replacing unnecessary entries, with the goal of producing an entire row of unnecessary entries. Such a row can then be deleted, improving the array, and the process can be iterated. For certain t-restrictions, it is shown that by determining conflict graphs, entries that are necessary can nonetheless be changed without violating the t-restriction. This permits a richer set of ways to improve the arrays. The efficacy of these methods is demonstrated via computational results.

Keywords: covering array, hash family, frameproof code, disjunct matrix.

1 Introduction

In combinatorial search, testing, and measurement problems, numerous problems of the following type arise. An $N \times k$ array is defined. Let Δ be a finite alphabet not containing $\star$. For $1 \le i \le N$, there is a finite alphabet $\Sigma_i \subseteq \Delta$ for which the ith row contains only symbols in $\Sigma_i \cup \{\star\}$. (When $\Sigma_1 = \cdots = \Sigma_N = \Sigma$, the array is *homogeneous*, otherwise it is *heterogeneous.*) For $1 \le j \le k$, there is a finite alphabet Δ_j not containing $\star$ for which the jth column contains only symbols in $\Delta_j \cup \{\star\}$. (When $\Delta_1 = \cdots = \Delta_k = \Delta$, the array is *uniform*, otherwise it is *nonuniform.*) Without loss of generality, $\Sigma_i \subseteq \cup_{j=1}^{k} \Delta_j$ and $\Delta_j \subseteq \cup_{i=1}^{N} \Sigma_i$.

H. Aydinian, F. Cicalese, and C. Deppe (Eds.): Ahlswede Festschrift, LNCS 7777, pp. 597–608, 2013.

If for some i, j with $1 \le i \le N$ and $1 \le j \le k$, we have $\Sigma_i \cap \Delta_j = \varnothing$, the (i, j) cell is permitted only to contain $\star$.

Within this framework, one considers restrictions on what must appear in some row within every subset of t columns. Such 'restriction' problems are considered in [3], but we use a somewhat more general definition here.

Let t be an integer, called the *strength*. A *t-restriction* is a list $(\mathcal{P}_1, \ldots, \mathcal{P}_\tau)$ of subsets of Δ^t, called *demands* [3]. For every selection $S = (i_1, \ldots, i_t)$ of t distinct column indices, the set of possible t-tuples that could arise is $\Delta_{i_1} \times \cdots \times \Delta_{i_t}$. Then the $N \times k$ array $A = (a_{ij})$ *satisfies the t-restriction* $(\mathcal{P}_1, \ldots, \mathcal{P}_\tau)$ if and only if for all t-tuples $(x_1, \ldots, x_t)$ of distinct column indices, and for $1 \le \ell \le \tau$, for each $\mathcal{P}_\ell$ with $\mathcal{P}_\ell \cap (\Delta_{x_1} \times \cdots \times \Delta_{x_t}) \neq \varnothing$, there exists an r with $1 \le r \le N$ for which $(a_{r,x_1}, \ldots, a_{r,x_t}) \in \mathcal{P}_\ell$. The generality of the definition arises from the flexibility in specifying t-restrictions.

We enumerate a few well-studied examples.

Disjunct Matrix [10]: The demand is $\{(\delta_1, \ldots, \delta_t) \in \{0,1\}^t : \delta_1 = \cdots = \delta_{t-1} = 0, \delta_t = 1\}$;

Frameproof Code [15]: The demand is $\{(\delta_1, \ldots, \delta_t) \in \{0,1\}^t : \delta_1 = \cdots = \delta_{t-1},\ \delta_t \neq \delta_1\}$;

Covering Array [6]: Demands are all members of Δ^t;

Perfect Hash Family, PHF [18]: The demand is $\{(\delta_1, \ldots, \delta_t) \in \Delta^t : \delta_i \neq \delta_j \text{ for } i \neq j\}$;

For covering arrays, requiring only a subset $\mathbb{S} \subseteq \Delta^t$ to be covered yields $\mathbb{S}$-*quilting arrays* [8]. For disjunct matrices (equivalently, superimposed codes or cover-free families), numerous t-restriction problems arise in search theory [1,2]. For hash families when the t columns to be separated are partitioned into ℓ classes $C_1, \ldots, C_\ell$ of sizes $w_1, \ldots, w_\ell$ (with $t = \sum_{i=1}^{\ell} w_i$) and we only require $\delta_i \neq \delta_j$ when i and j are in different classes, we obtain a $\{w_1, \ldots, w_\ell\}$-*separating hash family*, $\{w_1, \ldots, w_\ell\}$-SHF [12,16]. When on the t columns, the number of distinct symbols that arise is at most m, we have *m-strengthening hash families* [7]. A hash family that is $\{w_1, \ldots, w_s\}$-separating *for all* $\{w_1, \ldots, w_s\}$ with $\sum_{i=1}^{s} w_i = t$ is a (t, s)-*distributing hash family*, (t, s)-DHF [5]. An s-strengthening (t, s)-DHF is a (t, s)-*partitioning hash family*, (t, s)-PaHF [5].

These examples only scratch the surface. Numerous problems in combinatorial search and group testing [2,10] and in combinatorial cryptography [3,15] fall into this framework. Evidently, treating each such problem individually is problematic, and one wants general techniques to address the construction of arrays for t-restrictions. One general technique, explored in many of these contexts, is a recursive method using column replacement via hash families (e.g., [6]). But these techniques rely on knowing solutions for few columns to produce solutions for many.

Simple greedy or random algorithms produce solutions, but they cannot be expected to minimize the number of rows. We propose a general technique here to "post-optimize" an array, reducing its number of rows. We demonstrate that the reduction obtained is worthwhile, and sometimes dramatic.

2 Post-optimization

In [14], an heuristic method for reducing the number of rows in a covering array is developed. It relies on the fact that certain entries of the array may not be needed to ensure coverage. Such entries can be changed arbitrarily, with the result that other entries that were previously required, are no longer needed. The method exploits this to produce entire rows that are not needed. These can be deleted, improving the size of the array, and the process can be repeated. For covering arrays, the method has surprising success, and therefore we wish to apply the technique more generally. Here we develop it for general t-restriction problems.

2.1 Necessity Analysis

Consider an $N \times k$ array A on symbol set $\Delta \cup \{\star\}$ that meets the t-restriction $(\mathcal{P}_1, \ldots, \mathcal{P}_\tau)$. Evidently if A contains a row that consists entirely of $\star$ symbols, this row is not used to meet any of the requirements, and can be removed. The primary objective of our method is repeatedly to produce such an all-$\star$ row for removal. To do this, we consider the necessity of each entry.

When one of the demands is met for columns $(x_1, \ldots, x_t)$ for a single row of the array, the t entries in these columns in this row are *strictly necessary* to meet the demand. One might hope that all entries of the array not determined to be necessary in this way can be changed to $\star$, since they are not "needed". However once one is changed to $\star$, further entries may now become necessary. Indeed determining the maximum number of entries that can be simultaneously changed to $\star$ is NP-hard [14].

We therefore adopt a more useful notion of necessity. Let ρ be a permutation of $\{1, \ldots, N\}$, the row indices. For each demand and each tuple of t columns, there is a *first* row (under ρ) in which this demand is met; the entries in the t columns of this row are *necessary*. A single scan of the array now suffices to determine all necessary entries. All others are *unnecessary*, and all can be changed to $\star$ while ensuring that all demands are still met.

In determining necessity in this way, every demand must be checked in each t-tuple of columns. This can often be accomplished by considering only a subset of the t-tuples, as follows: Let π be a permutation of $\{1, \ldots, t\}$. If there are two demands $\mathcal{P}_a$ and $\mathcal{P}_b$ so that $\mathcal{P}_b = \{(\nu_{\pi_1}, \ldots, \nu_{\pi_t}) : (\nu_1, \ldots, \nu_t) \in \mathcal{P}_a\}$, then we can either (1) not check demand $\mathcal{P}_b$ if demand $\mathcal{P}_a$ is checked, or (2) not check the t-tuple $(x_{\pi_1}, \ldots, x_{\pi_t})$ of columns if $(x_1, \ldots, x_t)$ is checked. In practice this reduces the effort to determine necessity for most cases of interest.

2.2 Generic Post-optimization

We may be very lucky, and find that after marking unnecessary entries, we have an entire row of $\star$ entries. But this should not be expected. Following ideas from the special case of covering arrays [14], we employ two observations. Let A be an $N \times k$ array that satisfies the t-restriction $(\mathcal{P}_1, \ldots, \mathcal{P}_\tau)$. First, reordering

the rows of A results in an array that still satisfies the t-restriction (provided, of course, that the specifications of the row alphabets are permuted in the same manner as are the rows). And secondly, an entry of $\star$ in row r and column c can be replaced by any symbol in $\Sigma_r \cap \Delta_c$, and the resulting array still satisfies the t-restriction.

input : t-restriction with demands $\mathcal{P}_1, \ldots, \mathcal{P}_\tau$,
$N \times k$ array A satisfying the t-restriction,
$ITERATION\_LIMIT$ – number of iterations to be performed,
$LOCAL\_LIMIT$ – number of iterations allowed with no row removal
output: $M \times k$ array C satisfying the t-restriction with $M \leq N$

$\rho \longleftarrow$ identity $C \longleftarrow A$ $noImprovementCounter \longleftarrow 0$ $maxUnnecessaryElements \longleftarrow 0$ **for** $i \leftarrow 1$ $ITERATION\_LIMIT$ **do**
Locate necessary and unnecessary entries in C using row order ρ Change all unnecessary entries to $\star$ $currentMax \longleftarrow$ maximum number of $\star$s in a row of C **if** $currentMax > maxUnnecessaryElements$ **then**
$maxUnnecessaryElements \longleftarrow currentMax$
$noImprovementCounter \longleftarrow 0$
else
add 1 to $noImprovementCounter$
endif
if C *contains any rows consisting entirely of* $\star$*s* **then**
Remove all such rows from C, adjusting N and
ρ $maxUnnecessaryElements \longleftarrow 0$ Nominate a row of array C and adjust ρ to make this row the last
endif
for *every* $\star$ *at position* (r, c) *in* C *with* $r \neq \rho(N)$ **do**
if $C(\rho(N), c) \in \Delta_c \cap \Sigma_r$ **then**
$C(r, c) \longleftarrow C(\rho(N), c)$
else
$C(r, c) \longleftarrow$ random value in $\Delta_c \cap \Sigma_r$
endif
endfor
if $noImprovementCounter \geq LOCAL\_LIMIT$ **then**
Choose permutation ρ of $\{1, \ldots, N\}$ at random
$noImprovementCounter \longleftarrow 0$
else
Choose ρ at random, without changing $\rho(N)$
endif
endfor

Algorithm 1. A generic post-optimization algorithm for k-restriction problems

These form the basis of a remarkably simple algorithm for post-optimization. We repeatedly change $\star$ entries to entries in Δ, implicitly reorder the rows of the array, mark unnecessary entries, and delete any rows that now contain only $\star$. A more precise version is shown in Figure 1.

Because progress occurs when a row is eliminated, a worthwhile intermediate goal is to attempt to make a row with as many $\star$ entries as we can. However, row reordering could result in a row with many $\star$ entries having none in the next iteration. Therefore the algorithm nominates one row, retained at the end of the row order, in which it repeatedly attempts to increase the number of $\star$s. By so doing, the method may become trapped in a local optimum, where no further $\star$ entries are formed in the nominated row. For this reason, a means to escape such local optima by requiring progress is included; when progress has apparently stalled, a complete row reordering is done, resulting in a new row becoming nominated.

We report computational results for Algorithm 1 in §4. Prior to doing so, we examine an interesting variant of the method.

3 Post-optimization with Conflict Graphs

In Algorithm 1, every entry is deemed to be either necessary or unnecessary. Consider the first 3×5 array on $\Delta = \{0, 1, 2\}$ in Figure 1; this array is a $\{1, 2\}$-SHF . Every entry is strictly necessary. But the entry in the (3,4) position can nonetheless be changed, from 1 to 0, forming a second array that also satisfies the demand.

2 1 1 0 1	2 1 1 0 1
0 2 1 1 0	0 2 1 1 0
1 1 2 1 0	1 1 2 **0** 0

Fig. 1. $\{1, 2\}$-SHFs

Once changed, there is a possibility that an unnecessary entry appears, and progress can be made. Next we explore transformations that permit the replacement of entries while still satisfying the t-restriction, for a certain type of t-restriction.

3.1 Conflict Graphs

A demand $\mathcal{P}_\ell$ is *totally symmetric* if, for every permutation π of the symbols in Δ, $(\pi(\delta_1), \ldots, \pi(\delta_t)) \in \mathcal{P}_\ell$ if and only if $(\delta_1, \ldots, \delta_t) \in \mathcal{P}_\ell$. We consider now only those t-restrictions with totally symmetric demands $\mathcal{P}_1, \ldots, \mathcal{P}_\tau$. When the demands are all totally symmetric, the symbols within any row can be permuted arbitrarily while still satisfying the t-restriction. A new row produced in this manner handles neither more nor fewer of the demands. We are interested in modifying the row to handle all of the demands that it currently does, but possibly to handle more. To do this, we develop conflict graphs, focussing on SHFs. Roughly speaking, edges indicate a requirement for columns to contain different symbols; we make this precise now.

Let A be an $N \times k$ array, a $\{w_1, \ldots, w_s\}$-SHF with $t = \sum_{i=1}^{s} w_i$. Let $z_j = \sum_{i=1}^{j} w_i$. Then the demand to be satisfied is $\{(\delta_1, \ldots, \delta_t) \in \Delta^t : \delta_a \neq \delta_b \text{ or } z_j < a, b \leq z_{j+1} \text{ for } j \in \{0, \ldots, s-1\}\}$. We construct a collection of graphs $G_1, \ldots, G_N$, one for each row, as follows. Each G_i contains k vertices, representing the column indices $\{1, \ldots, k\}$. As before, for every t-tuple of columns, we determine the *first* row in which the demand is met. Suppose that the demand is first met for columns $(x_1, \ldots, x_t)$ in row r. For row r to continue to meet this demand, it must be the case that the symbols $(\sigma_1, \ldots, \sigma_t)$ satisfy $\sigma_a \neq \sigma_b$ except possibly when $z_j < a, b \leq z_{j+1}$ for $j \in \{0, \ldots, s-1\}$. To represent this, we place an edge in G_r between vertices x_a and x_b for all $1 \leq a, b \leq t$, *except* when $z_j < a, b \leq z_{j+1}$ for $j \in \{0, \ldots, s-1\}$. Once all t-tuples of columns are processed in this way, the graphs $G_1, \ldots, G_N$ are the *conflict graphs* of array A for this demand. When the t-restriction consists of multiple (separating) demands, each can be processed in the same way, possibly adding further edges to the conflict graphs; this results in conflict graphs for the entire t-restriction. To connect with our earlier discussion, when vertex c is isolated (is incident on no edges) in G_r, this is precisely the same as saying that the (r, c) entry of A is unnecessary.

Interpret row r of A as a vertex colouring of G_r in $|\Sigma_r|$ colours, where the entry a_{rc} is treated as a colour of vertex c in G_r. This is a proper colouring. More importantly, suppose that we form *any* proper colouring of G_r; this can be interpreted as a row – and this row must meet all of the demands met for the first by the original row. When the new colouring is not simply a permutation of the original one, the new row may meet more demands than does the original! Each conflict graph can be assigned a new proper colouring independently, producing a new array of the same size satisfying the t-restriction. Thus, even when no unnecessary entries arise, we can transform the array – and perhaps form unnecessary entries. Extending the post-optimization process to incorporate these recolouring transformations, while still nominating a row in which to maximize the number of $\star$ entries, opens a further avenue to seek improvements. Before pursuing this further, we consider a small extension.

As developed thus far, conflict graphs are suitable for SHFs with multiple separation requirements, and therefore for PHFs and DHFs as well. For strengthening and partitioning hash families, however, we encounter a difficulty. It is not the case that any recolouring of the conflict graphs will serve. Indeed in these situations, the demand requires not only that a certain separation be accomplished, but also that not too many symbols (colours) are used in the separation; just recolouring the conflict graph properly does not ensure the latter. PaHFs admit an easy modification to the conflict graphs. When a demand is met in columns $(x_1, \ldots, x_t)$ in row r, this demand can only be met if vertices x_i and x_j receive different colours when $a_{rx_i} \neq a_{rx_j}$, *and receive the same colour when* $a_{rx_i} = a_{rx_j}$. Then in forming the conflict graphs, whenever we find that x_i and x_j must receive the same colour in G_r, we identify (coalesce) x_i and x_j into a single vertex, ensuring that they receive the same colour. (This can be effectively implemented using the disjoint set forest method [9].) Any recolouring of the (coalesced) conflict graphs continues to meet all demands.

One could also accommodate more general strengthening requirements in this approach, by adding vertices to the conflict graph to ensure that when a demand is met, not too many different colours are assigned to the corresponding columns. However, this appears to increase the size of the conflict graphs exponentially, so we do not pursue it here. Instead we focus on SHFs and PaHFs.

3.2 Recolouring Conflict Graphs

Vertex colouring is NP-complete [11]. However, our interest is in finding a colouring of a graph G_r in $|\Sigma_r|$ colours, when G_r is known to be $|\Sigma_r|$-colourable. Simply permuting the colours changes nothing. We want a *non-trivial* recolouring, a proper colouring that is not simply a permutation of the original. Unfortunately, deciding the existence of a non-trivial recolouring is also NP-complete. To see this, one can use the fact that deciding whether a 3-SAT formula has a second satisfying assignment, given one satisfying assignment, is NP-complete [19]. Then using the well-known reduction from 3-SAT to vertex colouring [11], one finds that deciding the existence of a non-trivial vertex recolouring is also NP-complete.

With this complexity in mind, we do not make a concerted effort to find non-trivial recolourings for the conflict graphs. Rather we use a simple greedy approach. For each G_r, collapse multiple edges (if present) and sort the vertices in nonincreasing order by degree, breaking ties at random. Now process the vertices in this order, assigning each in turn a colour chosen at random from those not already assigned to one of its neighbours. If none is available for some vertex, no colouring is produced. We repeat this process until either a colouring is produced, or a limit on the number of attempts is reached. When a colouring is found, its colours are interpreted as symbols to replace row r. When no colouring is found, the row is left unchanged.

Adding this recolouring method to the generic post-optimization strategy produces a variant, *recolouring post-optimization*.

4 Computational Results

A specialization of generic post-optimization has been surprisingly successful at improving covering arrays [13,14]. Here we focus on applications to hash families, but remind the reader that there is a wide variety of t-restriction problems in which the methods could be employed. We always treat homogeneous, uniform hash families with N rows, k columns, v symbols, and a restriction of strength t. `C++` implementations of both generic and recolouring post-optimization were tested using an 8-core Intel Xeon processor clocked at 2.66GHz with 4MB of cache, bus speed 1.33GHz, and 16GB of memory. Testing proceeds by first generating an array one row at a time, choosing rows at random, until all demands are met. Post-optimization is then applied to improve the solution, if possible. Except when noted, post-optimization was executed for one minute on a single core.

Perfect hash families have been very extensively studied (for example, [18] and references therein). For strengths $t \in \{5, 6\}$, $v = t$, and $k \leq 25$, generic post-optimization of random arrays rarely produces arrays that are competitive with the best known sizes; this should be expected, given the computational effort invested [4,18]. What is surprising is that recolouring post-optimization not only recreates many of the best known results, but constructs a 10×10 PHF with $t = v = 5$, shown in Figure 2. This improves on the previous best known 13×10 and 11×9 PHFs [17].

We expect the most useful applications to arise for t-restriction problems that have not been extensively researched before. In producing a $\{w_1, \ldots, w_s\}$-SHF, one could naturally use a PHF of strength $t = \sum_{i=1}^{s} w_i$, provided that the number of symbols is at least t. We therefore compare cases for $\{2, 2, 1\}$-, $\{4, 1\}$-, and $\{3, 2\}$-SHFs, with the best known results for PHFs [17]. Table 1 gives a selection of results from generic post-optimization. Columns headed by 'G' are from generic post-optimization, those headed 'I' are sizes of the initial random array, and the one headed 'B' is the best known result from [17].

Naturally as v is decreased, the number of rows generally increases, as one would expect. Because of the randomness of post-optimization, it can happen that a solution with fewer rows is found even when k is increased of v is decreased;

Table 1. Generic post-optimization for SHFs

	$\{1,1,1,1,1\}$			$\{2,2,1\}$						$\{4,1\}$				$\{3,2\}$			
$v \rightarrow$	5	5	5	5	5	4	4	3	3	5	4	3	2	5	4	3	2
$k \downarrow$	B	G	I	G	I	G	I	G	I	G	G	G	G	G	G	G	G
5	1	1	9	1	7	6	18	15	80	3	3	3	5	4	5	6	10
6	3	3	28	3	24	7	40	15	99	3	3	5	6	5	5	10	15
7	6	6	49	6	24	7	58	28	212	3	4	5	7	7	7	11	15
8	8	8	73	8	42	12	75	28	236	4	6	7	8	7	9	13	22
9	11	12	107	13	45	22	98	28	371	4	6	9	9	7	11	16	27
10	13	15	109	13	41	22	92	28	417	5	7	9	10	8	11	18	31
11	16	20	125	17	55	33	118	75	338	7	7	12	20	12	14	21	35
12	21	25	141	19	59	36	112	84	385	7	9	13	21	11	16	23	39
13	26	30	138	22	71	43	127	98	502	8	9	14	13	12	16	25	43
14	32	36	166	25	71	47	136	108	454	8	10	17	22	12	17	26	45
15	35	40	165	27	71	50	135	126	453	10	12	18	23	14	18	29	49
16	39	45	181	28	74	54	147	138	542	9	12	18	24	15	20	30	52
17	44	50	185	32	79	57	146	150	513	10	13	19	25	16	20	33	55
18	49	56	200	33	77	62	156	164	469	11	14	22	24	16	22	33	59
19	53	61	217	35	78	67	158	177	526	11	14	22	26	17	23	34	62
20	57	64	263	37	78	71	163	188	442	11	15	24	27	18	23	35	64
21	61	70	244	39	95	75	151	204	471	12	15	25	27	18	25	38	66
22	64	75	254	42	77	82	168	220	484	13	16	25	29	19	25	39	69
23	68	81	244	44	84	83	179	233	618	12	17	27	28	20	27	40	73
24	71	84	294	44	98	86	213	233	606	13	17	27	28	21	27	41	77
25	74	90	248	50	88	92	187	247	670	14	18	28	31	22	29	43	78

2	3	4	3	0	4	1	0	2	1
0	1	1	3	4	2	2	4	3	0
1	4	2	0	3	0	3	4	2	1
1	1	2	3	3	4	0	4	2	0
2	4	4	1	1	2	0	3	0	3
1	1	4	3	4	0	0	2	2	3
2	1	0	3	2	0	4	1	4	3
4	3	0	2	4	1	3	1	2	0
4	2	0	3	0	3	2	1	4	1
2	4	1	4	3	2	3	0	0	1

4	4	1	4	0	3	2	4	5	1	0
5	0	5	1	3	2	0	2	2	3	4
5	4	3	1	2	3	3	0	2	3	0
3	1	4	0	4	0	2	5	3	5	1
4	1	0	0	4	3	5	2	5	1	3
3	0	0	2	2	5	2	2	4	1	1
0	1	5	4	0	1	0	5	3	3	2
5	2	1	4	0	3	3	5	2	0	4
5	2	3	5	0	0	1	4	3	5	1

Fig. 2. A 10×10 PHF with $t = v = 5$, and a 9×11 $(6, 2)$-DHF with $v = 6$

for example, a 13×13 $\{4, 1\}$-SHF with $v = 3$ was found, having fewer rows than the 21×12 solution with $v = 3$ and the 14×13 solution with $v = 4$. Here one could treat the 13×13 solution as having $v = 4$, or delete a column to obtain a 13×12 solution with $v = 3$. We have not recorded such implications in the tabulation, so as to focus on the results of post-optimization. Recolouring post-optimization can often improve these results, sometimes substantially: The 247×25 $\{2, 2, 1\}$-SHF with $v = 3$ improves to a 213×25 solution, a 13% reduction in the number of rows.

Restrictions with more than one demand can also be treated. Suppose, for example, that we want an array that is $\{4, 1\}$- and $\{3, 2\}$-separating with $k = 25$ and $v = 5$. Rather than using the 74×25 PHF, we could combine the 14×25 $\{4, 1\}$-SHF and the 22×25 $\{3, 2\}$-SHF to produce a 36×25 solution. Better yet, generic post-optimization using the two demands simultaneously yields a 22×25 solution in one minute. Similarly, a 49×25 array that is $\{2, 2, 1\}$-, $\{4, 1\}$-, and $\{3, 2\}$-SHF with $v = 5$ was found, which unexpectedly has fewer rows than the 50×25 $\{2, 2, 1\}$-SHF in Table 1.

Distributing hash families impose a number of separation demands simultaneously. Table 2 show results for $(6, s)$-DHFs with $s \in \{2, 3, 6\}$. The case when $s = 6$ is the PHF, and the known result from [17] is reported. The remaining values are from post-optimization. When $v = 6$, both generic and recolouring post-optimization typically produce solutions with fewer rows than the PHF. Notably, recolouring post-optimization often yields a much smaller result than does generic post-optimization, supporting the belief that conflict graph recolouring can avoid many of the local optima encountered in the generic method. Because these cases have strength $t = 6$, when $k = 20$ we are examining 27,907,200 6-tuples of columns to check demands. Hence one might expect that our standard one minute limit on computation time does not permit many iterations! Indeed permitting five minutes rather than one improves the 116×20 $(6, 3)$-DHF with $v = 6$ to an 88×20 solution.

Recolouring post-optimization also improves a 15×11 $(6, 2)$-DHF with $v = 6$ to a 9×11 solution, shown in Figure 2, and it improves a 36×20 $(6, 2)$-DHF with $v = 6$ to a 24×20 solution. In the interests of conserving space, we do not provide a complete list.

Table 2. Generic and recolouring post-optimization on (t, s)-DHFs

	Known	Generic									Recolouring			
	$(6,6)$	$(6,2)$					$(6,3)$				$(6,3)$			
$k \downarrow v \rightarrow$	6	6	5	4	3	2	6	5	4	3	6	5	4	3
6	1	1	7	12	15	31	10	17	31	90	1	16	28	89
7	4	8	8	11	20	42	11	17	39	113	4	16	37	101
8	8	7	15	18	27	50	17	25	54	177	10	22	48	175
9	13	9	13	20	32	57	16	31	64	224	16	29	63	223
10	18	11	16	24	39	72	21	38	83	291	20	37	81	278
11	24	15	19	26	43	84	26	46	99	352	25	44	95	340
12	27	15	21	30	49	96	34	94	150	563	30	53	113	403
13	39	19	24	32	55	110	37	66	140	516	36	61	133	473
14	53	20	26	39	62	123	44	81	162	590	40	72	150	539
15	64	25	29	43	69	136	55	93	185	676	47	83	173	615
16	77	26	32	45	72	151	63	113	211	765	52	90	190	685
17	86	29	35	47	81	164	71	112	236	853	64	101	214	769
18	94	30	39	53	89	182	87	130	273	1021	70	112	233	832
19	106	32	46	58	94	196	101	148	300	1021	74	127	256	914
20	120	36	53	63	104	213	116	158	321	1121	75	136	282	984

Table 3. Post-optimization on (t, s)-PaHFs. Columns labelled R are recolouring, the rest generic.

	$(4,2)$-				$(5,2)$-				$(5,4)$-			$(6,5)$-		$(7,6)$-	
$k \downarrow v \rightarrow$	4	3	2	2R	5	4	3	2	5	5R	4	6	5	7	6
5	10	10	10	10	15	15	15	15	10	10	10				
6	12	12	10	10	15	24	15	16	16	15	18	15	15		
7	12	12	11	11	21	20	21	20	24	23	25	23	31	21	21
8	14	11	12	11	29	29	29	28	29	28	21	41	55	38	45
9	14	15	15	11	36	35	35	35	33	33	46	53	80	66	98
10	17	11	16	11	43	40	39	39	43	42	56	74	100	112	162
11	17	16	17	17	47	45	44	44	49	48	70	93	196	220	271
12	19	18	18	17	50	49	49	46	61	55	81	122	187	313	388
13	22	21	19	19	54	54	53	52	66	64	91	152	242	573	580
14	22	21	21	20	56	57	56	54	73	70	106	177	283	909	1047
15	23	22	21	20	61	62	61	58	80	76	118	209	345		
16	24	24	23	21	68	63	62	61							
17	26	25	23	22	69	70	68	64							
18	26	25	24	23	75	73	70	69							
19	27	26	25	24	88	80	75	72							
20	30	28	25	25	92	81	80	74							
21	29	29	26	26	129	103	85	80							
22	29	30	27	27	157	127	99	82							
23	32	30	28	28	119	126	96	83							
24	34	33	29	28	208	150	105	86							
25	35	33	30	29	258	159	123	89							

Table 3 reports on the results of post-optimization on randomly generated PaHFs; there is no known result with which to compare. Examining the $(4, 2)$- and $(5, 2)$-PaHFs, it is striking that allowing more symbols often yields a larger number of rows; yet it is clear that one can simply not use the extra symbols, and so an array on fewer symbols remains a solution on more. The behaviour is an artifact of the random selection process for the initial array. Indeed when more symbols are provided, the chance increases that, while columns are separated, too many symbols are used to do so. This could be overcome by using a better greedy method to make the initial array, for example the methods in [7]. This does not occur in every case examined. For $(6, 5)$- and $(7, 6)$-PaHFs, better results are obtained with more symbols permitted. In these cases, in a separation $\binom{t}{2} - 1$ pairs must have different values, but only one pair requires the same. Hence selecting rows uniformly at random yields a better initial solution in these cases.

Recolouring post-optimization (in the two columns marked R in Table 3) yields improvements beyond those obtained by generic post-optimization. Coalescing vertices in the conflict graphs appears to have lessened the benefit of recolouring; nonetheless it is striking that improvements remain possible.

5 Conclusion

Arrays for t-restrictions permeate many different applications. General tools to construct them include greedy methods and random methods, but both appear to yield arrays with an unnecessarily large number of rows. Naturally more sophisticated methods can typically be devised for a specific t-restriction, but requires careful analysis of the specifics of the restriction. Therefore we have developed a general technique, focussing first on unnecessary entries and then on changeable entries, to eliminate rows repeatedly. Even with modest investments of computation time, and even starting with poor input arrays, these post-optimization methods yield useful arrays. We anticipate that their main value is in improving solutions found by methods other than simple random techniques, as has been the case with covering arrays. However, the real strength of the methods is their ability to deal with arbitrary t-restriction problems. Applications beyond the realms of hash families and covering arrays appear well worth further research.

Acknowledgements. Thanks to Daniel Horsley and Violet Syrotiuk for helpful discussions about this research.

References

1. Ahlswede, R., Deppe, C., Lebedev, V.: Threshold and Majority Group Testing. In: Aydinian, H., Cicalese, F., Deppe, C. (eds.) Ahlswede Festschrift. LNCS, vol. 7777, pp. 488–508. Springer, Heidelberg (2013)
2. Ahlswede, R., Wegener, I.: Search Problems. Wiley Interscience (1987)
3. Alon, N., Moshkovitz, D., Safra, S.: Algorithmic construction of sets for k-restrictions. ACM Transactions on Algorithms 2, 153–177 (2006)

4. Colbourn, C.J.: Constructing perfect hash families using a greedy algorithm. In: Li, Y., Zhang, S., Ling, S., Wang, H., Xing, C., Niederreiter, H. (eds.) Coding and Cryptology, pp. 109–118. World Scientific, Singapore (2008)
5. Colbourn, C.J.: Distributing hash families and covering arrays. J. Combin. Inf. Syst. Sci. 34, 113–126 (2009)
6. Colbourn, C.J.: Covering arrays and hash families, Information Security and Related Combinatorics. In: NATO Peace and Information Security, pp. 99–136. IOS Press (2011)
7. Colbourn, C.J., Horsley, D., Syrotiuk, V.R.: Strengthening hash families and compressive sensing. Journal of Discrete Algorithms 16, 170–186 (2012)
8. Colbourn, C.J., Zhou, J.: Improving two recursive constructions for covering arrays. Journal of Statistical Theory and Practice 6, 30–47 (2012)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
10. Du, D.-Z., Hwang, F.K.: Combinatorial group testing and its applications, 2nd edn. World Scientific Publishing Co. Inc., River Edge (2000)
11. Karp, R.M., Miller, R.E., Thatcher, J.W.: Reducibility among combinatorial problems. Journal of Symbolic Logic 40(4), 618–619 (1975)
12. Liu, L., Shen, H.: Explicit constructions of separating hash families from algebraic curves over finite fields. Designs, Codes and Cryptography 41, 221–233 (2006)
13. Nayeri, P., Colbourn, C.J., Konjevod, G.: Randomized Postoptimization of Covering Arrays. In: Fiala, J., Kratochvíl, J., Miller, M. (eds.) IWOCA 2009. LNCS, vol. 5874, pp. 408–419. Springer, Heidelberg (2009)
14. Nayeri, P., Colbourn, C.J., Konjevod, G.: Randomized postoptimization of covering arrays. European Journal of Combinatorics 34, 91–103 (2013)
15. Stinson, D.R., Van Trung, T., Wei, R.: Secure frameproof codes, key distribution patterns, group testing algorithms and related structures. J. Statist. Plann. Infer. 86, 595–617 (2000)
16. Stinson, D.R., Zaverucha, G.M.: Some improved bounds for secure frameproof codes and related separating hash families. IEEE Transactions on Information Theory, 2508–2514 (2008)
17. Walker II, R.A.: Phftables, `http://www.phftables.com` (accessed March 10, 2012)
18. Walker II, R.A., Colbourn, C.J.: Perfect hash families: Constructions and existence. Journal of Mathematical Cryptology 1, 125–150 (2007)
19. Yato, T., Seta, T.: Complexity and completeness of finding another solution and its application to puzzles. IEICE - Transactions on Fundamentals of Electronics, Communications and Computer Sciences E86-A(5), 1052–1060 (2003)