Applied Mathematics and Computation 421 (2022) 126952

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc

In-Parameter-Order strategies for covering perfect hash families

Michael Wagner [a], Charles J. Colbourn [b], Dimitris E. Simos [a,*]

[a] SBA Research, MATRIS, Floragasse 7, Wien 1040, Austria
[b] School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, Arizona, USA

ARTICLE INFO

Article history:
Received 14 August 2021
Revised 11 January 2022
Accepted 13 January 2022
Available online 28 January 2022

Keywords:
Covering array
Covering perfect hash families
In-Parameter-Order
Algorithms
Permutation vector

ABSTRACT

Combinatorial testing makes it possible to test large systems effectively while maintaining certain coverage guarantees. At the same time, the construction of optimized covering arrays (CAs) with a large number of columns is a challenging task. Heuristic and Metaheuristic approaches often become inefficient when applied to large instances, as the computation of the quality for new moves or solutions during the search becomes too slow. Recently, the generation of covering perfect hash families (CPHFs) has led to vast improvements to the state of the art for many different instances of covering arrays. CPHFs can be considered a compact form of a specific family of covering arrays. Their compact representation makes it possible to apply heuristic methods for instances with a much larger number of columns. In this work, we adapt the ideas of the well-known In-Parameter-Order (IPO) strategy for covering array generation to efficiently construct CPHFs, and therefore implicitly covering arrays. We design a way to realize the concept of vertical extension steps in the context of CPHFs and discuss how a horizontal extension can be implemented in an efficient manner. Further, we develop a horizontal extension strategy for CPHFs with subspace restrictions that identifies candidate columns greedily based on conditional expectation. Then using a local optimization strategy, a candidate may be adjoined to the solution or may replace one of the existing columns. An extensive set of computational results yields many significant improvements on the sizes of the smallest known covering arrays.

1. Introduction

Covering Arrays (CAs) are a combinatorial design class of combinatorial designs that can be considered a generalization of orthogonal arrays. A (uniform) $\mathrm{CA}(N; t, k, v)$ is an $N \times k$ array in which each entry is from a v-ary alphabet and is defined by the property that for every possible selection of t columns, every t-tuple $\{0, 1, \ldots, v-1\}^t$ appears in at least one row of the sub-array. Whenever such a tuple appears in a row, we consider it *covered*, a CA can therefore also be defined as an array that covers all possible t-tuples in all t-selections of columns, further referred to as *column selections*.

The construction of CAs with a minimal number of rows is of particular interest, especially for practical applications. Given a strength t, number of columns k and alphabet of cardinality v, we consider the problem of generating a $\mathrm{CA}(N; t, k, v)$ as a *CA instance* and aim to construct a CA with the smallest number of rows possible. The smallest value of N for which

* Corresponding author.
E-mail addresses: mwagner@sba-research.org (M. Wagner), Charles.Colbourn@asu.edu (C.J. Colbourn), dsimos@sba-research.org (D.E. Simos).

https://doi.org/10.1016/j.amc.2022.126952
0096-3003/

$i = \{\beta^{(i)}\}$	$1 \times \beta_0^{(i)} + 1 \times \beta_1^{(i)} + 2 \times \beta_{t-1}^{(i)}$
$0 = 000$	$0 + 0 + 0 = 0$
$1 = 100$	$1 + 0 + 0 = 1$
$2 = 200$	$2 + 0 + 0 = 2$
$3 = 010$	$0 + 1 + 0 = 1$
$4 = 110$	$1 + 1 + 0 = 2$
$5 = 210$	$2 + 1 + 0 = 0$
$6 = 020$	$0 + 2 + 0 = 2$
$7 = 120$	$1 + 2 + 0 = 0$
$8 = 220$	$2 + 2 + 0 = 1$
$\vdots$	$\vdots$
$24 = 022$	$0 + 2 + 1 = 0$
$25 = 122$	$1 + 2 + 1 = 1$
$26 = 222$	$2 + 2 + 1 = 2$

Fig. 1. The permutation vector 112 is expanded for $t = q = 3$.

there exists a CA$(N; t, k, v)$ is called the *covering array number*, denoted by CAN$(t; k, v)$, and the respective CA is considered *optimal*. At the same time, since the construction of optimal CAs is a difficult optimization problem for the general case [1], the exact values for CAN only exist for a very limited number of instances, while only best known lower and upper bounds on CAN exist for the majority of CA instances. Online tables at [2] collect the smallest known upper bounds on CAN achieved by an explicit construction for CA instances with strengths $2 \le t \le 6$, alphabets $2 \le v \le 25$ with up to $k \le 10000$ columns.

One of the main application domains of CAs is *Combinatorial Testing* (CT), where CAs are used as the underlying mathematical structure representing combinatorial test sets. Each row in the CA represents one test case and each column corresponds to one parameter of the input model. The idea behind CT is based on empirical evidence that the vast majority of faults found in software or hardware systems is caused by interactions between a small number of parameters [3]. By applying CAs of strength t, we can ensure that all interactions of up to t parameters are tested. This method makes it possible to apply structured testing methods to systems that are by far too large to test exhaustively. For example, in past works, CT was successfully applied to find faults in systems with more than 2000 parameters [4]. At the same time, generating uniform CAs with such a large number of columns is not an easy feat. Since even state-of-the-art combinatorial test generation tools are unable to construct CAs with such a large number of columns, the authors had to apply a recursive doubling construction which recursively doubles the number of columns of seed CAs generated by In-Parameter-Order (IPO) algorithms. While this approach proved sufficient in this particular case, the generated CAs were far from optimal due to the large number of redundancies introduced by the recursive doubling construction. It is apparent that there is still a lack of as well as a demand for algorithms capable of generating small CAs for large CA instances effectively.

In this work we utilize a more compact form of certain families of CAs, so called *Covering Perfect Hash Families* (CPHFs), to design an IPO algorithm that can be effectively applied to generate CAs for large instances. Experiments with this algorithm confirm the effectiveness of this approach and improve the best known upper bounds for a large number of CA instances. Further, by devising an IPO algorithm for CPHFs with subspace restrictions, we were able to establish new upper bounds for strength $t = 3$ and $837 \le k \le 10000$ for alphabet $v = 4$ and $619 \le k \le 10000$ for $v = 5$. Our work is structured as follows. Section 2 provides a brief introduction to covering perfect hash families, while Section 3 summarizes related methods that construct CAs and CPHFs. In Section 4 we design an In-Parameter-Order algorithm for generating CPHFs, which we evaluate in detail in Section 5. Finally, we discuss our work on CPHFs with subspace restrictions in Section 6 and conclude the work in Section 7.

2. Permutation vectors and covering perfect hash families

Let q be a prime power and $\mathbb{F}_q$ a finite field of order q. A *permutation vector* can be denoted as a t-tuple $\mathbf{h} = (h_0, h_1, \ldots, h_{t-1})$ with entries h_i arising from $\mathbb{F}_q$, not all 0 and can be expanded into a vector of length q^t with symbols from $\mathbb{F}_q$ by forming the scalar product between $\mathbf{h}$ and the base q representation $(\beta_0^{(i)}, \beta_1^{(i)}, \ldots, \beta_{t-1}^{(i)})$ of every $i \in \{0, 1, \ldots, q^t - 1\}$. Each permutation vector can therefore be considered a compact representation of a vector of length q^t with symbols coming from $\mathbb{F}_q$. Every expanded vector has the property that each symbol of $\mathbb{F}_q$ appears in exactly q^{t-1} of its entries, while for the special case of $h_0 \neq 0$, the vector can be partitioned into q^{t-1} permutations of the q different symbols, hence the name *permutation vector*. Fig. 1 shows an example of such an expansion. The permutation vector (1,1,2) is expanded for $t = q = 3$

by forming the scalar product between the permutation vector (1,1,2) and the base three representation $(\beta_0^{(i)}, \beta_1^{(i)}, \beta_2^{(i)})$ of each $i \in \{0, 1, \ldots, 26\}$. The resulting vector of length $q^t = 27$ can be partitioned into 9 permutations of the symbols 0,1,2.

A set of t permutation vectors is *covering* if the sub-array with t columns and q^t rows obtained by expanding the t permutation vectors forms an orthogonal array (OA). The process of determining if a given t-tuple of permutation vectors is covering is called the *covering test*. The naive approach of expanding all permutation vectors and confirming whether the resulting array is an OA is quite inefficient. A better way is to utilize the fact that a t-tuple of permutation vectors is covering if and only if the t permutation vectors are linearly independent, see [5], which can easily and efficiently be confirmed with Gaussian elimination.

An $n \times k$ array of permutation vectors is called a *Covering Perfect Hash Family* (CPHF), denoted $\mathsf{CPHF}(n; k, q, t)$, if for every selection of t columns there exists at least one row in which the t-tuple of permutation vectors form a covering tuple. This property guarantees that the array obtained by expanding every permutation vector in the $\mathsf{CPHF}(n; k, q, t)$ is a $\mathsf{CA}(n \cdot q^t; t, k, q)$, since every column selection of the CA must now contain a sub-array that contains all possible t-tuples over the q symbols.

An example of a $\mathsf{CPHF}(2; 13, 3, 3)$ is shown in Fig. 3. It is an array with 2 rows and 13 columns, containing permutation vectors of length 3 with symbols arising from $\{0, 1, 2\}$ as entries. In each possible selection of 3 columns, one of the two 3-tuples of permutation vectors is covering.

Further, since the first element of any expanded permutation vector is always zero, the resulting CA contains n all-zero rows, $n - 1$ of which are redundant and can be deleted. The CPHF in Fig. 3 therefore represents a CA with $N = 2 \cdot 3^3 - 1 = 53$ rows.

Last, we can observe that multiplication of a permutation vector with any non-zero element of $\mathbb{F}_q$ creates a bijective mapping between symbols in the vector obtained by expanding the permutation vectors [6]. Since symbol permutation is considered an isomorphism in CAs, such permutation vectors can also be considered isomorphic. Therefore, it is sufficient to only regard permutation vectors where the first non-zero element is the identity, reducing the number of non-isomorphic permutation vectors to $\sum_{i=0}^{t-1} q^i = \frac{q^t-1}{q-1}$.

3. Related work

The generation of uniform CAs is a well-studied and active field of research. Many different construction methods, such as greedy algorithms [7], metaheuristics [8], recursive constructions [9] and exact methods [10] have been applied with excellent success, for a comprehensive survey about uniform CA generation methods see [11]. While exact methods always construct an optimal CAs and can therefore be used to determine the precise value of CAN, due to the combinatorial explosion of the size of the search space, they can only be applied to very small CA instances. Metaheuristic algorithms have been successfully applied to construct many of the currently best known CAs for instances with small strength and alphabets and a medium number of columns, but become too inefficient when large instances are concerned.

One way to generate slightly larger CAs effectively are greedy algorithms. We can distinguish between algorithms that construct a CA one test at a time, such as AETG [12] and the deterministic density algorithm (DDA) [13], and algorithms that grow an initial array in both dimensions, such as the In-Parameter-Order (IPO) family of algorithms. Due to their fast execution speed and their focus on column extensions which is very well-suited for CPHF generation, we will focus on IPO algorithms in this work.

The IPO strategy was first proposed in 1998 [7] and many different improvements and variations of the algorithm have been devised ever since. The algorithm, outlined in Algorithm 1, starts by constructing a CA with t columns and v^t rows.

Algorithm 1 IPOG Strategy.

```
Require: t, k, v
  A ← cross-product of the set of symbols for the first t columns
  for i ← t, ..., k do
      HorizontalExtension(i)
      if there are uncovered tuples then
          VerticalExtension(i)
      end if
  end for
```

Afterwards, in a procedure called *horizontal extension* new columns are added to the CA until the target number of columns is reached. In order to cover all missing t-tuples, the algorithm greedily selects values for all entries in the newly added column that maximize the number of newly covered tuples. If any uncovered tuples remain at the end of such a horizontal extension step, the algorithm performs a *vertical extension* in which all missing tuples are added to the array, adding rows if necessary.

Note that it is not necessary to assign values to all entries immediately. Such unassigned values are called star values and are deliberately left open in order to allow the algorithm to consider them in later extension steps. Classical IPO algorithms generally ignore star values during the horizontal extension, but attempt to first merge missing tuples into existing rows by

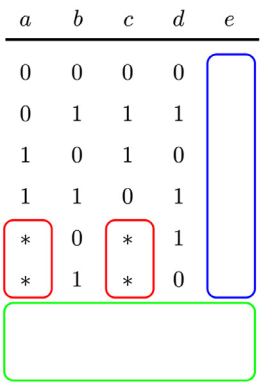

Fig. 2. Example of a CA generated by IPO.

121	010	012	102	120	110	122	111	112	100	011	101	001
112	102	122	100	101	110	010	121	111	120	011	001	012

Fig. 3. Example of a CPHF$(2; 13, 3, 3)$.

replacing star values with appropriate values. Fig. 2 shows an example of a CA$(6; 2, 4, 2)$ constructed by an IPO algorithm. If more than 4 columns are required, the algorithm appends new columns to the CA, depicted by the blue box to the right. The star values, denoted by $*$ in the red box, can be replaced during the vertical extension of later extension steps in order to merge missing tuples into existing rows. The algorithm only adds new rows to the CA (shown in green) if the missing tuple can not be merged into any row.

As mentioned in the introduction, even greedy algorithms are not always fast enough to construct CAs for very large instances within reasonable time. One approach to solve this issue is the application of recursive construction methods, which can be used effectively to generate larger CAs from one or more different *seed arrays*. While these methods can be an incredibly effective tool and can be applied to even the largest of instances, they also have several downsides. First, they are highly dependent on the number of rows of the CAs that are used as seed arrays. Second, the construction process usually introduces many redundancies in the CA, which leads to CAs with a relatively large number of rows.

Much better results for difficult CA instances can often be achieved by algorithms based on finite fields. Aside from cyclotomy [14], which generates $q \times q$ CAs with alphabet v and prime power q, $q \equiv 1 \pmod{v}$, different algorithms constructing CPHFs have recently been used to construct many of the best known CAs for higher alphabets. Their compact representation made it possible to apply backtracking algorithms [5] and different metaheuristics, such as [15] and [16], to instances larger than usual. For very large instances, greedy column extension algorithms for CPHFs [17] and affine composition methods [18] have been the most successful. One example of such column extension algorithms is the random column extension algorithm proposed in [17], which we briefly review here. The algorithm starts with an initial CPHF and appends new columns until no new CPHF can be found anymore. For each extension step, a certain number of random candidate columns are generated. If the array obtained by appending the column to the previous CPHF is again a CPHF, then the column is appended. Otherwise, the algorithm evaluates whether one of the existing columns is part of all uncovered column selections. In this case, said column can be replaced by the candidate, which provides some means to escape local optima.

In this work, we discuss how the In-Parameter-Order strategy can be applied effectively to generate CPHFs. In contrast to previous column extension algorithms, instead of generating candidates randomly, the algorithm carefully selects suitable values for the candidate columns. In addition, by introducing the concept of a vertical extension, we can move from the problem of finding a CPHF with a maximal number of columns for a given number of rows to the problem of generating a CPHF (and therefore a CA) with a given number of columns and minimal number of rows, which is much more common in practical applications.

4. An In-Parameter-Order strategy for covering perfect hash families

In order to design an In-Parameter-Order algorithm for covering perfect hash families, all individual steps of classical IPO algorithms, see Section 3 and Algorithm 1, need to be adjusted accordingly. In particular, it is necessary to generate an initial CPHF, extend it horizontally by appending a new column and selecting appropriate values for all entries in the newly added column and lastly cover all column selections that are still uncovered after the horizontal extension by means of vertical extension. In the following subsections, we present how those steps can be implemented effectively as part of the CPHFIPO algorithm, outlined in Algorithm 2, and discuss possible optimizations and their impact on the performance of the algorithm.

Algorithm 2 CPHFIPO.

Require: t, k, q
  $A \leftarrow$ Initial Array
  for $i \leftarrow t, \ldots, k$ **do**
    HorizontalExtension($i, A.rows$)
    if there are uncovered column selections **then**
      VerticalExtension()
    end if
  end for

4.1. Initial array

The first step of classical IPO algorithms is to generate an initial CA of strength t with $k = t$ columns. In classical IPO algorithms, this is done by constructing the cross-product of the set of symbols for the first t columns, i.e. an array containing all possible t-tuples on q symbols, see [7]. In CPHFIPO, to create an initial CPHF to serve as starting array for the algorithm, a t-tuple of permutation vectors has to be constructed that is covering. From the covering test, discussed in Section 2, we know that any non-singular matrix can be used to construct such a covering tuple. In this work, we simply select the identity matrix, the rows of which represent the t permutation vectors $\{e_i\}$, $i \in \{1, \ldots, t\}$. The initial array generated at the start of Algorithm 2 is therefore a CPHF with 1 row and t columns, where the t entries are the permutation vectors $\{e_i\}$.

4.2. Horizontal Extension

The main objective of the horizontal extension is to append a new column to the current array and (greedily) select values that maximize the number of covered tuples. For this purpose, all entries of the new column, $\mathbf{c} = (c_1, \ldots, c_n)^T$, need to be iterated, which can be done in order from top to bottom, such as during classical IPOG algorithms, but also different permutations of the row indices $1, \ldots, n$, such as random permutations, can be used.

The defining part of the horizontal extension of IPO algorithms is the *coverage gain* computation. In this procedure, all possible candidate values are iterated and for each candidate, the algorithm calculates the coverage gain, which is the number of tuples that are currently uncovered, but would be covered if the candidate is selected. Finally, the candidate with the highest coverage gain is selected. The horizontal extension of classical IPO algorithms, as well as the coverage gain computation is discussed in detail in [19].

When designing a horizontal extension for CPHFs, there are two main differences to consider. First, instead of covering any missing tuples, we need to consider whether all column selections contain at least one t-tuple of permutation vectors that is covering. This information can be stored in a bit-vector of length $\binom{k}{t-1}$, where each bit tracks the cover status of one column selection. Second, the number of candidate values is generally far larger for any strength $t > 2$ than for classical IPO algorithms, since all possible $\frac{q^t-1}{q-1}$ non-isomorphic permutation vectors need to be considered.

Our algorithm to extend a CPHF with i columns and n rows is outlined in Algorithm 3. First, we iterate all entries in the

Algorithm 3 Horizontal Extension in CPHFIPO.

procedure HorizontalExtension(i, n)
  for $row \leftarrow 1, \ldots, n$ (randomized) **do**
    $c_i \leftarrow CPHF[row][i]$
    Compute-Coverage-Gains(row, i) ▷ For all candidate perm. vec.
    $c_i \leftarrow$ select permutation vector p that maximizes coverage gain
    Mark newly covered column selection as covered
    if all column selections are covered **then**
      return
    end if
  end for
end procedure

newly added column $\mathbf{c}$ in random order. For each entry c_i, we consider all non-isomorphic permutation vectors as candidates and compute the coverage gain of each candidate. For this, we first check whether a column selection is still uncovered, as only previously uncovered column selections need to be considered. Afterwards, we compute whether the existing $(t-1)$-tuple of permutation vectors forms a covering tuple together with the candidate permutation vector. Finally, after coverage gains have been computed for all candidates, the permutation vector with maximum coverage gain is selected, while any ties are broken randomly. To improve the execution time of the coverage gain computation, we now discuss two possible optimizations.

Pre-computing covering tuples First, the computation of whether any t-tuple of permutation vectors is covering can be conducted as a pre-processing step for all possible t-way combinations of permutation vectors. While this idea has already been applied in previous works, we briefly want to discuss our implementation of this concept. Each permutation vector can be represented by an integer, using a bijective packing function, such as the function specified in Eq. 1, which packs every non-isomorphic permutation into an integer smaller than $2 \cdot q^{t-1}$. The integer 0 represents the vector where all elements h_i are 0, while the integer $2 \cdot q^{t-1} - 1$ represents the permutation vector where $h_0 = 1$ and all other elements h_i are $q - 1$. While this encoding contains some redundant integers which represent the set of isomorphic permutation vectors where $h_0 = 0$ as well as the all-zero vector, which by definition is no permutation vector, creating a tighter packing into integers up to $\frac{q^t-1}{q-1}$ would require a more sophisticated packing method which might introduce additional computational overhead. Since all powers of q up to q^t are pre-computed at compile time, packing a permutation vector only requires t additions and multiplications, which is why we accept this small memory overhead in this work. We plan to investigate the performance of different packing functions in future work.

We can use the distinct integer representations of the permutation vectors p_i to represent all possible t-way combinations of permutation vectors by another integer value, see Eq. 2. Before the main algorithm is executed, we can therefore compute for each possible combination of t permutation vectors if the resulting t-tuple is covering and store this information in a bit-vector of length $(2 \cdot q^{t-1})^t$. By keeping track of the integer representation of each permutation vector in the current CPHF, the coverage gain computation during the horizontal extension simplifies to using the packing function from Eq. 2 for the existing $(t-1)$ permutation vectors in conjunction with the integer representations of all possible candidates permutation vectors. The resulting integer value then serves as index to look up whether the corresponding tuple is covering. While this can significantly speed up the search, we want to note that this method of pre-computing covering tuples can be very memory intensive, therefore its use is generally limited to strengths $t \leq 3$ or $t = 4$ when the alphabet q is small.

$$\text{pack-vector}((h_0, h_1, \ldots, h_{t-1})) = \sum_{i=0}^{t-1} h_i \cdot q^{t-1-i} \tag{1}$$

$$\text{pack-tuple}((p_0, p_1, \ldots, p_{t-1})) = \sum_{i=0}^{t-1} p_i \cdot (2 \cdot q^{t-1})^{t-1-i} \tag{2}$$

Simultaneous coverage gain computation for pre-computed covering tuples Simultaneous coverage gain computation is an optimization that was introduced for classical IPO algorithms in [19]. Since the columns of the previous array remain constant throughout the entire horizontal extension step, a constant prefix can be computed for the corresponding $(t-1)$-tuple in each column selection. This can be applied to CPHFs by tracking the coverage gains for all candidate permutation vectors simultaneously in a vector of length $2 \cdot q^{t-1}$. We first iterate over all column selections and compute the prefix, which is the integer representation of the first $(t-1)$ permutation vectors. Afterwards, for each candidate permutation vector, we can compute the integer value of the entire t-tuple of permutation vectors, and therefore the index used for look-up in the bit-vector of pre-computed covering tuples, as the sum of the prefix and the candidate. If the respective tuple is covering, the candidate can be used as index to increment the respective element in the coverage gain vector. Thanks to this optimization, the prefix of each $(t-1)$-tuple has to be computed only once throughout the horizontal extension, instead of calculating it separately for each candidate. Due to the far larger number of candidates during the horizontal extension for CPHFs when compared to classical IPO algorithms, this optimization has a significant impact on the execution time of the algorithm. Algorithm 4 provides a detailed pseudocode of the coverage gain computation using this optimization.

Algorithm 4 Compute Coverage Gains with Precomputed Covering Tuples.

Require: t, List of precomputed covering tuples l
1: **procedure** COMPUTE-COVERAGE-GAINS(row, i)
2: $gains[1, \ldots, 2 \cdot q^{t-1}] \leftarrow 0$
3: **for all** column selections $\{j_1, \ldots, j_t\}$ **do**
4: **if** column selection is marked as covered in coverage-map **then**
5: Skip column selection
6: **end if**
7: $prefix \leftarrow pack((CPHF[row][j_1], \ldots, CPHF[row][j_{t-1}]))$
8: **for all** non-isomorphic permutation vectors with integer repr. p **do**
9: **if** $l[prefix + p]$ is covering **then**
10: Increment $gains[p]$
11: **end if**
12: **end for**
13: **end for**
14: **return** Index of $Max(gains)$
15: **end procedure**

Simultaneous coverage gain computation for Gaussian elimination While the previously discussed optimizations can not be applied as effectively to instances where pre-computing covering tuples is not feasible, there is still some merit to computing coverage gains simultaneously. First, given a $(t-1)$-tuple of permutation vectors, we can confirm if the permutation vectors are linearly dependent. If this is the case, it is impossible to form a covering t-tuple with this set of permutation vectors and it is therefore unnecessary to consider any candidates for this row and column selection. Further, by computing the coverage gains simultaneously, we can first execute Gaussian elimination for the constant $(t-1)$ permutation vectors and then simply confirm if each candidate permutation vector is linear independent to them. This reduces the steps necessary during Gaussian elimination when compared to restarting from scratch for every candidate. These concepts are outlined in Algorithm 5.

Algorithm 5 Compute Coverage Gains with Gaussian Elimination.

Require: t

```
1: procedure COMPUTE-COVERAGE-GAINS(row, i)
2:     gains[1, ..., 2 · q^(t−1)] ← 0
3:     for all column selections {j_1, ..., j_t} do
4:         if column selection is marked as covered in coverage-map then
5:             Skip column selection
6:         end if
7:         Do gaussian elimination for (CPHF[row][j_1], ..., CPHF[row][j_{t−1}])
8:         if number of columns with a pivot ≠ t − 1 then
9:             Skip column selection
10:        end if
11:        for all non-isomorphic permutation vectors pv do
12:            if pv can be a pivot for the missing column then
13:                p ← integer representation of pv
14:                Increment gains[p]
15:            end if
16:        end for
17:    end for
18:    return Index of Max(gains)
19: end procedure
```

4.3. Vertical Extension

In classical IPO algorithms, the objective of the vertical extension is to cover all missing tuples by merging them into existing rows and to add new rows if this is not possible. A missing tuple can be merged into an existing row if the values in the row at the respective positions either match or are star values. The tuple can then be added by simply replacing all star values with the corresponding values of the missing tuple. Therefore, merging tuples into existing rows is very straight forward in classical IPO algorithms.

When adapting the vertical extension of IPO algorithms to CPHFs this problem becomes a bit more complex. The first task of the vertical extension for CPHFs is to cover any missing column selections by replacing star values in order to generate covering tuples. In contrast to classical IPO algorithms, where a missing tuple can be covered in exactly one way, which is to have the specific set of values appear in one row in the respective columns of the column selection, when applied to CPHFs, this problem becomes ambiguous, as there exist many different combinations of permutation vectors that can form a covering tuple. In addition to determining if a row can be used to cover a missing column selection, it is therefore also necessary to consider which permutation vectors are well suited to form a covering tuple.

Our approach for the vertical extension of CPHFs is outlined in Algorithm 6. We iterate over all column selections that do not contain a covering tuple after the horizontal extension and cover them as follows. First, we search for a row that contains at least one star value in the positions of the uncovered column selection, as it is impossible to produce a covering tuple if all relevent entries already have a value assigned. Next, we determine if all non-star values are linearly independent, as no covering tuple can be formed otherwise. This can be done by performing Gaussian elimination for the existing permutation vectors. If the permutation vectors are linearly independant, i.e. if the number of permutation vectors that can serve as pivot for one of the t columns is $t-s$, where s is the number of star values in the tuple, then it is possible to form a covering tuple by setting the star values to the unit vectors $\{e_i\}$, in order to create permutation vectors as pivots for the remaining columns $\{i\}$. Fig. 4 shows an example of this approach. Assume there exists a row with the permutation vectors $a=[1,2,0]$ and $c=[1,2,1]$ in the positions of an uncovered column selection, as well as one entry b that is currently a star value. First, to confirm that a covering tuple can be formed we apply Gaussian elimination, which results in the first permutation vector a serving as pivot for the first column, while b is a pivot for the third column. This only leaves open the second column, for which a pivot permutation vector can be constructed by replacing the star value with the unit vector $b=[0,1,0]$.

Algorithm 6 Vertical Extension in CPHFIPO.

```
procedure VerticalExtension
    for all uncovered column selections cs do
        if ∃ row s.t. ≥ 1 entries in cs are star values then
            Do gaussian elimination
            if the existing permutation vectors are linearly independent then
                Replace star values with unit vectors e_i to add missing pivots
                Randomize unit vectors
            end if
        else
            Add new row to CPHF containing only star values
            Add a covering tuple in new row in cs
            Randomize covering tuple
        end if
    end for
end procedure
```

$$\begin{bmatrix}120\\ *\\ 121\end{bmatrix} = \begin{pmatrix}1&2&0\\ ?&?&?\\ 1&2&1\end{pmatrix} \xrightarrow{\text{Gauss}} \begin{pmatrix}1&2&0\\ ?&?&?\\ 0&0&1\end{pmatrix} \xrightarrow{\text{Cover}} \begin{pmatrix}1&2&0\\ 0&1&0\\ 0&0&1\end{pmatrix} \rightarrow \begin{bmatrix}120\\ 010\\ 121\end{bmatrix} \xrightarrow{\text{Randomize}} \begin{bmatrix}120\\ 111\\ 121\end{bmatrix}$$

Fig. 4. Example for vertical extension: First, Gaussian Elimination is used to determine missing pivot position i, which are then covered by the vector e_i. Afterwards, the new permutation vector is randomized to facilitate the search in later extension steps. In this example, this is done by multiplying it by 2 and adding the third permutation vector [121] to it.

While the resulting t-tuples of permutation vectors are covering, this approach has one major flaw. By always utilizing unit vectors e_i to form covering tuples, we introduce a lot of isomorphic permutation vectors into the CPHF, especially whenever new rows are added. Since a tuple can not be covering if it contains two isomorphic permutation vectors, this issue makes it difficult for the algorithm to cover missing column selections and therefore can significantly increase the number of rows of the constructed CPHFs. In order to solve this problem, we make use of the following two operations. First, as discussed in Section 2, multiplication of a permutation vector with any non-zero symbol of $\mathbb{F}_q$ creates an isomorphic permutation vector. Second, recall that a t-tuple of permutation vectors is covering if and only if all t permutation vectors are linearly independent.

Remark. Given t linearly independent vectors $\{a_i\}$, $i \in \{1, 2, \dots t\}$, if a multiple of any vector a_i is added to any of the other vectors from $\{a_i\}$, then the resulting set of t vectors is again linearly independent.

This fact allows us to add permutation vectors together within a covering tuple without loss of the covering property. Therefore, by multiplying each permutation vector of the tuple that used to be a star value by a randomly selected non-zero symbol and adding a multiple of one of the other permutation vectors to them, we can create new permutation vectors that are non-isomorphic to the initial ones, while still maintaining a covering tuple. Since the vertical extension is not critical in terms of run time, we can repeat this process a few times in order to obtain good randomization. In the example in Fig. 4, the permutation vector $b = [0, 1, 0]$ is first multiplied by 2, then the vector $c = [1, 2, 1]$ is added to it once, resulting in $b = [1, 1, 1]$, which is non-isomorphic to [0,1,0].

Last, if no row exists that can be used to cover a missing column selection, we need to add a new row at the bottom of the CPHF. This row contains star values in all positions aside from those corresponding to the missing column selection, which are initially set to the unit vectors e_i of the identity matrix, then randomized in the same manner as described above.

4.4. Further Optimizations

To further improve the performance of the algorithm, we implemented various optimizations. First, since the strength t is usually a small integer, the authors of [19] suggest to promote it to a compile-time constant. We also make use of this idea, as it allows the compiler to perform additional optimizations, such as using fixed-size arrays for permutation vectors as well as unrolling the loops used to pack permutation vectors as well as tuples of permutation vectors into integers. In addition, since CPHFs are only defined for uniform alphabets of cardinality q, where q is a prime power, q can also be considered constant. This affects for example the loop that iterates all candidate values. While having two separate compile-time constants can drastically increase the compilation time, making q a compile-time constant results in an additional speedup by approximately a factor of two, so we consider it a worthwhile trade-off for this work.

Interleaving vertical extension Since each row in the CPHF represents q^t rows in the resulting CA, it is of utmost importance to limit the number of times a new row has to be added. Therefore, the efficiency of the horizontal extension is even more important in CPHFIPO than during classical IPO algorithms. To maximize the information available to the horizontal

extension, we designed an approach that greedily assigns values to existing star values during the horizontal extension. Whenever the algorithm encounters an uncovered column selection where the current row contains a star value in at least one of the positions, it attempts to cover the tuple by selecting an appropriate value for each star value using the method described in Section 4.3. It is important to note that a set of values that is suitable to replace existing star values for one candidate permutation vector might not be effective for other candidates, which makes it necessary to compute and assign those replacement values separately for each candidate. Further, since our implementation of the vertical extension is not deterministic, in addition to the coverage gain, we need to keep track of which replacement values were used for each candidate. When selecting the best candidate, we set all star values to the stored replacement values selected during the evaluation step. While this optimization introduces some overhead in terms of run time, it can have a significant impact on the number of rows of the generated CPHFs. To reduce this overhead, we track which rows contain star values and only apply the interleaving vertical extension for rows that contain at least one.

Column replacement In [17], a random column extension algorithm for CPHFs is improved by allowing the algorithm to replace existing columns. This concept is based on the idea that if column selections remain uncovered at the end of a horizontal extension step, but all uncovered column selections contain one specific column c_o in addition to the newly added column c, then c_o can be replaced by the new column c. While the resulting CPHF has the same number of columns as before the horizontal extension, this approach enables the algorithm to eliminate *bad* columns and escape local minima. Of course this optimization can also be applied to the CPHFIPO algorithm, where we allow the algorithm to perform a set number of column replacements before proceeding with a vertical extension.

Retries Last, the optimal order in which entries in the newly added column should be assigned values is generally unknown. As mentioned in Section 4.2, we iterate the rows in a random order during the horizontal extension. In order to explore different possibilities, we introduce the concept of *retries*, which allows the algorithm to try a specified number of different random permutations of row indices, in order to explore other possibilities. A vertical extension is therefore only performed once the algorithm runs out of retries and either no column replacement is possible or the replacement limit is reached.

5. Algorithmic evaluation

In order to evaluate the efficiency of the CPHFIPO algorithm, we conducted two sets of experiments. First, we compared the run time in milliseconds as well as the number of rows of CAs generated by the CPHFIPO algorithm with the performance of the FIPOG algorithm [19] of the CA generation tool CAgen [20], which can be considered state of the art in terms of classical In-Parameter-Order algorithms [21]. In order to get a detailed comparison, we generated 10 CAs for each instance for four different alphabets, $v = q = 5, 9, 16, 25$, strengths $t = 2, 3$ and up to $k = 10000$ columns. Table 1 shows the run time as well as the minimum number of rows, with the average number of rows being provided in parenthesis whenever the minimum and average do not match. In addition, we provide the run time of the algorithms as well as the time required to pre-compute the list of covering tuples for CPHFIPO for each tested alphabet v and strength t. For this set of experiments, we limited the CPHFIPO algorithm to 10 column replacements per extension step and did not allow any retries.

Strength $t = 2$ can be considered a special case for which CPHFs with the maximal number of columns possible can easily be constructed for given any number of rows n. Nonetheless, we consider these instances for our experiments in order to verify whether a translation to CPHFs improves upon classical IPO algorithms even in this special case.

As expected, even for strength 2, CPHFIPO is already significantly faster than the FIPOG algorithm and always constructs optimal CPHFs. What is a bit surprising though is that the CAs generated by CPHFIPO are also smaller than those generated by FIPOG in all but one of the tested instances.

While the results for strength $t = 2$ already appear quite promising, the experiments for $t = 3$ really showcase the tremendous benefits of the CPHF approach. By using CPHFIPO, we managed to generate CAs with up to 10000 columns for $v = 5$ and 5000 columns for all other instances within the given time budget, which was limited to 1 day per instance. Given the same time limitations, FIPOG only managed to generate CAs with 2000 columns for $v = 5$ and only up to 200 columns for both $v = 16$ and $v = 25$. Further, the difference in the number of rows of the generated CAs between the two algorithms is astonishing. CPHFIPO performs especially well when instances with a large alphabet are concerned. For example, for $t = 3$, $v = 25$ and $k = 200$ columns it takes FIPOG more than 4 hours to generate a CA with 117011 rows, while CPHFIPO finds a CA with less than half the number of rows within just 2 seconds, excluding the 74 seconds required for pre-computing the covering tuples. In fact, CPHFIPO even manages to generate a CA with 5000 columns that still has significantly less rows than the FIPOG generated CA with only 200 columns. Recall that CPHFIPO is based on CPHFs, which only exist for alphabets where the cardinality v is a prime power, and can therefore only be applied to a limited set of instances. At the same time, our experiments suggest that, due to the significant reduction in the number rows and the fast execution times when compared to classical IPO algorithms, it is also worthwhile to make use of CPHFIPO to generate CAs with different alphabets by simply generating a CA for the next highest prime power and remapping all redundant symbols to valid values in a post-processing step.

In order to investigate how the algorithm performs against other state-of-the-art construction methods, we started the CPHFIPO algorithm for instances of strengths $3 - 6$ and all prime powers $3 \leq q \leq 25$ and compare the results with the currently best known upper bounds on CAN as listed in [2]. Since only best known upper bounds for CAs with up to $k = 10000$ columns are reported in [2], we terminate our algorithm once 10,000 columns are constructed. To allow the algorithm

Table 1
Benchmarks of the FIPOG algorithm of the tool CAGEN for different instances of strength $t = 2$ and $t = 3$ are compared to the performance of CPHFIPO, when limited to 10 column replacements and no retries. Entries in **bold** mark instances where an algorithm performed better in terms of the number of rows or the execution times respectively.

		$t = 2$				$t = 3$			
		FIPOG		CPHFIPO		FIPOG		CPHFIPO	
v	k	rows	time	rows	time	rows	time	rows	time
5	precomp				0				4
	25	54	5	**49**	**0**	429	43	**373**	**0**
	50	**68**	7	73	**0**	590	189	**497**	**2**
	100	81	21	**73**	**0**	762	1770	**621**	**20**
	200	89	49	**73**	**1**	933	16,200	**745**	**164**
	500	103	114	**97**	**9**	1163	255,324	**869 (881)**	**2583**
	1000	113	296	**97**	**24**	1336	2,232,303	**993 (1005)**	**20749**
	2000	125	1122	**121**	**99**	1509	19,581,442	**1117**	**168959**
	5000	139	7092	**121**	**612**	-	timeout	**1241 (1340)**	**2752354**
	10,000	149	29,164	**145**	**2539**	-	timeout	**1365 (1476)**	**21832935**
9	precomp				0				155
	25	184	6	**161**	**0**	2804	229	**1457**	**0**
	50	214	7	**161**	**0**	3656	1991	**2185**	**7**
	100	253	25	**161**	**0**	4566	19,116	**2185 (2694)**	**58**
	200	289	73	**241**	**1**	5509	171,408	**2913 (3058)**	**454**
	500	339	426	**241**	**5**	6814	2,939,136	**3641 (3932)**	**6312**
	1000	381	1553	**241**	**17**	7842	26,100,350	**4369**	**46664**
	2000	424	6040	**321**	**90**	-	timeout	**5097**	**347233**
	5000	482	37,144	**321**	**561**	-	timeout	**5825**	**5421992**
	10,000	527	155,120	**321**	**2262**	-	timeout	-	timeout
16	precomp				0				4847
	25	563	36	**511**	**0**	15,424	**1666**	**8191**	1
	50	660	72	**511**	**0**	20,362	18,304	**8191**	**19**
	100	770	147	**511**	**0**	25,536	186,513	**12286**	**190**
	200	884	619	**511**	**1**	30,858	3,426,846	**12286 (13514)**	**1061**
	500	1048	3656	**766**	**10**	-	timeout	**16381 (16790)**	**15876**
	1000	1172	14,164	**766**	**33**	-	timeout	**20476**	**118760**
	2000	1309	55,327	**766**	**118**	-	timeout	**20476 (21295)**	**918940**
	5000	1494	345,152	**1021**	**710**	-	timeout	**24571 (24980)**	**13900382**
	10,000	1639	1,380,350	**1021**	**2990**	-	timeout	-	timeout
25	precomp				0				74034
	25	1273	29	**625**	**0**	57,667	**8967**	**31249**	4
	50	1548	142	**1249**	**0**	76,600	98,960	**31249**	**27**
	100	1805	509	**1249**	**0**	96,521	1,189,066	**31249 (32811)**	**285**
	200	2092	1994	**1249**	**1**	117,011	15,043,387	**46873**	**2072**
	500	2499	13,005	**1249**	**8**	-	timeout	**46873 (60934)**	**32036**
	1000	2813	52,272	**1873**	**36**	-	timeout	**62497 (64059)**	**250590**
	2000	3138	212,995	**1873**	**141**	-	timeout	**78121**	**1995877**
	5000	3586	1,324,138	**1873**	**856**	-	timeout	**78121 (81245)**	**29509581**
	10,000	3924	5,094,138	**1873**	**3391**	-	timeout	-	timeout

to explore more candidate columns, we set a limit of $10000, 100, 10$ and 1 for the number of column replacements and retries per extension step for strengths $3, 4, 5$ and 6 respectively in this set of experiments. The results of our experiments are shown in Tables 2 - 5, where we present the number of columns of the CPHFs that CPHFIPO constructed for a given alphabet q and number of rows n. Whenever the CAs derived from the constructed CPHFs improve upon the previously best known upper bounds on CAN, the number of columns is provided in bold letters. Due to the complexity of the CPHF generation problem and the dependence of the execution time on the alphabet q and strength t, we decided not to set a limit on the number of columns during our experiments for strengths $t \geq 4$ and report any results where the algorithm has not terminated within our available time budget with an underline.

The CAs that correspond to the CPHFs constructed by CPHFIPO over the course of these experiments establish a total of 27068, 11635, 981 and 231 new upper bounds on CAN for strengths three, four, five and six respectively. In many cases CPHFIPO was able to construct CPHFs with a significantly larger number of columns than previous column extension methods [17], while improving bounds established by a variety of different algorithmic approaches, such as cyclotomy [14], recursive constructions [9,22], algorithms based on transformations [23] as well as other CPHF approaches such as [17,18,24] and [25]. While CPHFIPO does not improve any upper bounds for small instances of strength $t = 3$, the improvements to CA instances with a large number of columns are often significant. For example, for $q = 7$ and $n = 8$ rows, CPHFIPO constructed a CPHF with 10,000 columns, while previous extension methods only managed to find CPHFs with 6852 columns. The CA corresponding to the CPHF constructed by CPHFIPO therefore improves upon all best known upper bounds on CAN for $6852 < k \leq 10000$.

Table 2
Results of our experiments with CPHFIPO for all prime powers $3 \leq q \leq 25$ and strength $t = 3$. The table shows the number of columns that CPHFIPO constructed for a given number of rows and alphabet q. Entries in **bold** mark results that establish new upper bounds.

n \ q	3	4	5	7	8	9	11	13	16	17	19	23	25
1	4	6	6	8	10	8	9	12	11	13	14	16	16
2	13	16	21	28	34	37	44	54	66	69	84	102	113
3	22	33	46	81	96	120	167	214	298	335	394	527	603
4	42	71	112	217	283	369	547	**773**	**1186**	**1305**	**1667**	**2529**	**2978**
5	77	148	253	**589**	**841**	**1126**	**1869**	**2838**	**4915**	**5709**	**7551**	10,000	10,000
6	133	302	582	**1588**	**2409**	**3468**	**6389**	**10000**	10,000	10,000	10,000		
7	228	597	1284	**4113**	**6522**	**10000**	10,000						
8	388	**1138**	**2702**	**10000**	10,000								
9	657	**2115**	**5683**										
10	1065	**3969**	10,000										
11	1735	**7523**											
12	2824	10,000											
13	4716												
14	7687												
15	10,000												

Table 3
Results of our experiments with CPHFIPO for all prime powers $3 \leq q \leq 25$ and strength $t = 4$. The table shows the number of columns that CPHFIPO constructed for a given number of rows and alphabet q. Entries in **bold** mark results that establish new upper bounds while underlined entries signify instances where the algorithm has not yet terminated based on the limitations of 100 retries and column replacements per extension step.

n \ q	3	4	5	7	8	9	11	13	16	17	19	23	25
1	5	5	6	8	9	9	9	9	10	10	11	12	12
2	10	13	14	17	19	20	23	25	30	31	34	39	41
3	14	19	23	34	37	42	**51**	**62**	**75**	**80**	**89**	**107**	**121**
4	22	31	42	**64**	**74**	**87**	**114**	**143**	**186**	**203**	**233**	**302**	**344**
5	31	48	69	**117**	**149**	**175**	**239**	**313**	**451**	**495**	**587**	**658**	
6	44	**76**	115	**225**	**267**	**339**	**514**	**712**	**871**				
7	56	**108**	**184**	**389**	**491**	**672**	**1017**						
8	76	156	282	**653**	**935**	**1206**							
9	99	228	435	**1136**	**1407**								
10	129	333	631	**2065**									
11	170	438	1005	**3493**									
12	227	633	1591										
13	301	911	**2550**										
14	404	**1376**	**3886**										
15	478	2048											
16	647	2992											
17	869	4102											
18	1133												
19	**1494**												
20	**2026**												
21	**2694**												
22	**3582**												
23	**4382**												
24	**4712**												

The experiments for higher strengths highlight three important aspects of the performance of CPHFIPO. First, despite the fast execution speed of the algorithm, for higher strength, instances with a large number of columns become too time consuming for the algorithm. This can be attributed to the large number of candidates that need to be iterated when v and t become large. Remember that there exist $\frac{q^t-1}{q-1}$ non-isomorphic permutation vectors, so for instance, in the case of $q = 25$ and $t = 6$, which is the most difficult instance we tested, the algorithm needs to compute the coverage gain for more than 10 million candidates for each row. In these extreme cases, random column extension algorithms in combination with our proposed vertical extension or recursive methods should be better suited. At the same time, each row of such a CPHF$(n; k, 25, 6)$ corresponds to almost 250 million rows in the resulting CA. Therefore, even for a small number of columns, these arrays are already too large to be used in most practical applications. Second, with increasing strength, CPHFIPO is able to find improvements to the best known upper bounds for smaller and smaller instances. For $t = 3$ the smallest instance where new upper bounds were established had 589 columns, while the algorithm improved upper bounds for instances with as little as 51, 21 and even only 14 columns for strengths $t = 4, 5, 6$ respectively. Last, Table 3 nicely shows how effective our optimizations for pre-computed tuples are, in particular the shared prefix computation during the simultaneous coverage gain computation. Due to memory limitations we were only able to make use of that optimization for

Table 4
Results of our experiments with CPHFIPO for all prime powers $3 \leq q \leq 25$ and strength $t = 5$. The table shows the number of columns that CPHFIPO constructed for a given number of rows and alphabet q. Entries in **bold** mark results that establish new upper bounds while underlined entries signify instances where the algorithm has not yet terminated based on the limitations of 10 retries and column replacements per extension step.

n \ q	3	4	5	7	8	9	11	13	16	17	19	23	25
1	6	6	6	8	8	10	9	10	10	10	10	11	11
2	10	11	12	14	14	16	16	19	**21**	21	**23**	24	25
3	13	14	17	22	**24**	**26**	**30**	**33**	**39**	**41**	**45**	**51**	**54**
4	15	**21**	26	**34**	**37**	**43**	**51**	**62**	**73**	**80**	**89**	**107**	**113**
5	21	27	35	**52**	**60**	**70**	**89**	**111**	**145**	**136**			
6	24	**38**	**50**	**78**	**95**	**115**	**159**	**188**					
7	32	**50**	**69**	**118**	**156**	**187**							
8	40	61	94	**188**	**246**								
9	49	81	133	**277**									
10	60	**107**	187										
11	67	**143**	263										
12	82	190											
13	100	**252**											
14	125												
15	148												
16	172												
17	213												
18	257												
19	309												
20	371												

Table 5
Results of our experiments with CPHFIPO for all prime powers $3 \leq q \leq 25$ and strength $t = 6$. The table shows the number of columns that CPHFIPO constructed for a given number of rows and alphabet q. Entries in **bold** mark results that establish new upper bounds while underlined entries signify instances where the algorithm has not yet terminated based on the limitations of 1 retry and column replacement per extension step.

n \ q	3	4	5	7	8	9	11	13	16	17	19	23	25
1	7	7	7	8	9	10	9	9	10	10	10	11	11
2	12	11	11	13	**14**	14	15	16	18	17	18	20	20
3	12	13	15	**18**	**19**	**21**	**23**	**25**	**28**	**29**	**31**	**34**	**36**
4	14	17	20	**25**	**27**	**31**	**35**	**39**	**46**	**48**			
5	16	**21**	25	**34**	**39**	**44**	**48**	**56**					
6	19	**26**	32	**49**	**54**	**61**							
7	23	**33**	43	**67**	**72**								
8	27	40	56	**93**									
9	31	**50**	72										
10	36	**61**	88										
11	41	**77**	114										
12	49	89											
13	59	**107**											
14	65	129											
15	79												
16	92												
17	103												
18	114												
19	135												
20	147												
21	179												

instances with $q \leq 7$. Therefore we can observe a significant drop in the number of columns CPHFIPO managed to construct within the same time frame between the alphabets $q = 7$ and $q = 8$.

6. IPO strategies for CPHFs with subspace restrictions

Section 5 demonstrated that CPHFIPO is capable of generating small CAs effectively for many different CA instances. Nevertheless, every row added to a CPHF represents $v^t - 1$ rows in the respective CA, which can result in large jumps in the number of rows of CAs with a similar number of columns. For example, when considering the results from Table 2, if we were interested in constructing a CA of strength $t = 3$, $v = 11$ with $k = 550$ columns, a CPHF with $n = 5$ rows would be required, while a CPHF with only 3 fewer columns would require only $n = 4$ rows. This results in a CA with 73,201 rows

111	122	120	111	112	110	101	100	110	101	102	102	120
101	122	100	111	122	100	101	110	100	111	102	112	120
121	102	120	122	120	100	100	102	112	101	111	110	122
111	121	110	010	111	010	012	110	110	102	120	112	102

Fig. 5. Example for $\mathcal{S}_{m,u}$-restricted CPHFs: The first 13 columns of a $\mathcal{S}_{3,1}$-restricted CPHF$(4; 28, 4, 3)$ are depicted.

Table 6
Results of our experiments for SCPHFs for all prime powers $3 \le q \le 25$ and strength $t = 3$. The table shows the number of columns of constructed SCPHFs for a given number of rows and alphabet q. Entries in bold mark results that establish new upper bounds.

n \ q	3	4	5	7	8	9	11	13	16	17	19	23	25
2	10	15	22	31	36	41	50	62	77	84	91	112	120
3	22	34	49	84	105	125	173	222	299	334	392	519	592
4	37	70	117	227	280	356	533	**761**	**1168**	**1328**	**1651**	**2492**	**2976**
5	61	137	228	555	**792**	**1058**	**1802**	**2758**	4712	**5414**	**7326**	10000	10000
6	101	241	502	**1473**	**2234**	**3249**	**5585**	**9393**	10000	10000	10000		
7	156	467	1083	**3739**	**6086**	**8473**	10000						
8	248	880	2287	**8924**	10000	10000							
9	400	1652	4571	10000									
10	643	2870	8979										
11	1013	5183	10000										
12	1402	8892											
13	2071												
14	3167												
15	4871												
16	7002												

instead of only 58561, a massive difference. In order to obtain a more granular distribution of results and construct good CAs for a far larger range of columns, we use the concept of *subspace restrictions*.

In CAs, a row that is identical to any other row is *replicated* and can be removed from the CA. Subspace restrictions can be used to ensure the presence of such replicated rows by enforcing certain limitations on the elements of the permutation vectors. Let $\mathcal{F}_{t,p}$ be the set of all p-tuples of distinct entries from $\{0, \ldots, t-1\}$. This can be considered all different combinations of length p of the t different entries in a permutation vector. A subspace restriction of *dimension* p and *replication* r consists of an r-tuple $(x_1, \ldots x_r)$ of distinct entries from $\{1, \ldots, n\}$, which represent a set of r row indices, and an r-tuple $(U_1, \ldots, U_r)$, $U_i \in \mathcal{F}_{t,p}$, which can be considered a set of positions within a permutation vector. If we denote a CPHF$(n; k, q, t)$ as $A = (a_{ij})$ in which each entry a_{ij} is a permutation vector of length t, then we can address the ℓth element of this vector as $a_{ij\ell}$. Given a subspace restriction S defined by $(x_1, \ldots, x_r)$ and $(U_1, \ldots U_r)$ and denoting the element of U_a in position b as u_{ab}, then A *satisfies* the restriction S only if $a_{x_c.j.u_{c\ell}} = a_{x_d.j.u_{d\ell}}$ for all $1 \le j \le k$ and $1 \le \ell \le p$, $1 \le c, d \le r$. In short, the elements of the permutation vectors at the positions specified in U_i need to match for all permutation vectors in the rows $(x_1, \ldots, x_r)$, this has to be the case for all columns.

An example of a CPHF with subspace restrictions is given in Fig. 5, which satisfies two different subspace restrictions. The first restriction can be defined by the set of row indices (1,2,3), representing the first three rows of the CPHF, and the set of permutation vector indices $U_i = (0)$. Due to this restriction, in each column the first elements (h_0) of all permutation vectors have to be the same in the rows with index 1, 2 and 3, or more formally: $a_{1.j.0} = a_{2.j.0} = a_{3.j.0}$ for all $1 \le j \le k$. The second restriction can be defined by the set of row indices (1,2) and the set of permutation vector indices $U_i = (0, 2)$. This restriction enforces for each column that the permutation vectors in the first 2 rows not only match in h_0, but also in h_2, so $a_{1.j.0} = a_{2.j.0}$ and $a_{1.j.2} = a_{2.j.2}$ for all $1 \le j \le k$.

The reason this induces replicated rows in the resulting CA derives from the computation that expands permutation vectors. Whenever one or more positions in the permutation vector match, then the scalar product with a vector that is 0 in all non-matching positions will yield the same result. When forming the scalar product between the permutation vectors $\{h_0, h_1, \ldots h_{t-1}\}$ and the base q representations $(\beta_0^{(i)}, \beta_1^{(i)}, \ldots \beta_{t-1}^{(i)})$ of every $i \in \{0, 1, \ldots q^t - 1\}$, this is the case for exactly q^p multiplications. Since the subspace restriction is enforced on all columns, this creates a set q^p rows in the resulting CA which are replicated for each row in $(x_1, \ldots, x_r)$. Therefore, a subspace restriction of dimension p and replication r induces $(r-1) \cdot q^p$ replicated rows in the resulting CA that can be deleted.

Every CPHF has a subspace restriction of replication n and dimension 0, resulting in $(n-1)$ replicated rows. These are the previously discussed all-zero rows due to the first symbol of an expanded permutation vectors always being 0. Special cases of CPHFs are the *Sherwood Covering Perfect Hash Families* (SCPHFs), in which the first element of every permutation vector is 1. This corresponds to a subspace restriction with $(x_1, \ldots, x_n) = (1, 2, \ldots, n)$ and $U_i = (0)$ for $1 \le i \le n$, which induces $(n-1) \cdot q$ replicated rows in the CA. Table 6 depicts the results of our experiments for SCPHFs. Similar to the results for unrestricted CPHFs, the algorithm often improves upon previous SCPHF generation methods by constructing SCPHFs with more columns than random extension methods [17], metaheuristics [25] and even 3-stage methods [24]. It further improves upon best known upper bounds achieved by cyclotomy [14] and recursive constructions [9]. In total, our experiments with

SCPHFs achieve improvements to 23,283 upper bounds on CAN. As expected, applying CPHFIPO to unrestricted CPHFs constructed CPHFs with more columns than the SCPHF algorithm in all cases, given the same number of rows n. However, due to the smaller number of rows in the respective CAs, the results in Table 6 improve upon 14244 of the new upper bounds reported by CPHFIPO for strength three in Table 2.

Finally, in order to investigate a much broader range of subspace restrictions, we employed a variation on our earlier algorithm, which is outlined in Algorithm 7. The algorithm starts with a randomly constructed CPHF with n rows, typically

Algorithm 7 CPHFIPO for CPHFs with subspace restrictions.

Require: t, n, q, m, u, iteration limit *iterlim*
  $A \leftarrow$ Random $\mathcal{S}_{m,u}$-restricted CPHF
  $iterations \leftarrow 0$
  $k \leftarrow A.columns$
  while $n \geq m > u \geq 0$ or $m = u = 0$ **do**
    while $iterations < iterlim$ **do**
      c $\leftarrow$ empty new column
      for $row \leftarrow 1, \ldots, n$ (randomized) **do**
        $c_{row} \leftarrow$ permutation vector respecting $\mathcal{S}_{m,u}$ that maximizes coverage gain
      end for
      if all column selections are covered **then**
        Adjoin **c** to A
        $iterations \leftarrow 0$
        $k \leftarrow k + 1$
      else
        if all column selections not involving column **c**′ are covered **then**
          Replace column **c**′ by column **c**
        end if
      end if
    end while
    Decrement m and/or u
  end while

one that contains a maximal number of subspace restrictions. It appends new columns to the CPHF one at a time or replaces existing columns, as described in Section 4.4. During this horizontal extension, the values maximizing the number of newly covered column selections are selected, while respecting the constraints set by the subspace restrictions. In contrast to CPHFIPO, which performs a vertical extension if the horizontal extension meets a termination criterion, the algorithm for restricted CPHFs takes a different approach. Instead of adding a new row, the algorithm relaxes the subspace restrictions, which increases the number of available candidate permutation vectors and often permits the algorithm to find new suitable columns. In this work we examine more complex sets of subspace restrictions, described below.

Table 7
Results for $\mathcal{S}_{m,u}$-restricted CPHF$(6; k, 4, 3)$. The left side shows the number of columns of the generated CPHFs, while the right side shows the number of rows in the respective CA.

$n = 6$	Columns in CPHF					
	m					
$u \downarrow$	0	2	3	4	5	6
0	301	279	270	263	256	241
1		227	227	213	211	200
2			194	190	184	182
3				171	161	151
4					140	133
5						111
$n = 6$	Rows in CA					
	m					
$u \downarrow$	0	2	3	4	5	6
0	379	376	373	370	367	364
1		364	361	358	355	352
2			349	346	343	340
3				334	331	328
4					319	316
5						304

Table 8
$\mathcal{S}_{m,u}$-restricted CPHF$(n;k,4,3)$s that yield CA$(N;3,k,4)$s for $364 \le N \le 598$. **Bold** entries indicate best new bounds on the size of the covering array.

N	Entries k $(n_{m,u})$
364	231 $(7_{7,5})$ 241 $(6_{6,0})$
367	234 $(7_{6,5})$ 256 $(6_{5,0})$
370	263 $(6_{4,0})$
373	270 $(6_{3,0})$
376	268 $(7_{7,4})$ 279 $(6_{2,0})$
379	278 $(7_{6,4})$ 301 $(6_{0,0})$
382	293 $(7_{5,4})$
388	308 $(7_{7,3})$
391	322 $(7_{6,3})$
394	334 $(7_{5,3})$
397	**342** $(7_{4,3})$
400	338 $(8_{8,7})$ 347 $(7_{7,2})$
403	364 $(7_{6,2})$
406	378 $(7_{5,2})$
409	392 $(7_{4,2})$
412	392 $(7_{7,1})$ 394 $(8_{8,6})$
	402 $(7_{3,2})$
415	405 $(7_{6,1})$ 408 $(8_{7,6})$
418	427 $(7_{5,1})$
421	438 $(7_{4,1})$
424	452 $(7_{3,1})$ 457 $(8_{8,5})$
	467 $(7_{7,0})$
427	470 $(7_{2,1})$ 473 $(8_{7,5})$
	485 $(7_{6,0})$
430	490 $(8_{6,5})$ 500 $(7_{5,0})$
433	**522** $(7_{4,0})$
436	519 $(8_{8,4})$ **541** $(7_{3,0})$
439	541 $(8_{7,4})$ **560** $(7_{2,0})$
442	561 $(8_{6,4})$ 606 $(7_{0,0})$
445	580 $(8_{5,4})$
448	572 $(9_{9,8})$ 592 $(8_{8,3})$
451	616 $(8_{7,3})$
454	639 $(8_{6,3})$
457	660 $(8_{5,3})$
460	664 $(9_{9,7})$ 677 $(8_{8,2})$
	681 $(8_{4,3})$
463	691 $(9_{8,7})$ 694 $(8_{7,2})$
466	715 $(8_{6,2})$
469	747 $(8_{5,2})$
472	749 $(8_{8,1})$ 764 $(9_{9,6})$
	783 $(8_{4,2})$
475	774 $(8_{7,1})$ 800 $(9_{8,6})$
	816 $(8_{3,2})$
478	807 $(8_{6,1})$ **825** $(9_{7,6})$
N	Entries k $(n_{m,u})$
481	**837** $(8_{5,1})$
484	871 $(8_{4,1})$ 874 $(9_{9,5})$ **884** $(8_{8,0})$
487	897 $(9_{8,5})$ 907 $(8_{3,1})$ **909** $(8_{7,0})$
490	936 $(8_{2,1})$ 939 $(8_{6,0})$ **940** $(9_{7,5})$
493	959 $(9_{6,5})$ **978** $(8_{5,0})$
496	958 $(10_{10,9})$ 984 $(9_{9,4})$ **1001** $(8_{4,0})$
499	1019 $(9_{8,4})$ **1030** $(8_{3,0})$
502	1063 $(8_{2,0})$ **1064** $(9_{7,4})$
505	1085 $(9_{6,4})$ **1159** $(8_{0,0})$
508	1093 $(9_{9,3})$ 1097 $(10_{10,8})$ 1099 $(9_{5,4})$
511	1145 $(10_{9,8})$ 1149 $(9_{8,3})$
514	**1183** $(9_{7,3})$
517	**1197** $(9_{6,3})$
520	1199 $(9_{9,2})$ 1215 $(10_{10,7})$ **1235** $(9_{5,3})$
523	1246 $(9_{8,2})$ 1276 $(9_{4,3})$ **1291** $(10_{9,7})$
526	1305 $(9_{7,2})$ **1323** $(10_{8,7})$
529	**1352** $(9_{6,2})$
532	1366 $(10_{10,6})$ 1372 $(9_{9,1})$ **1392** $(9_{5,2})$
535	1430 $(9_{8,1})$ 1445 $(10_{9,6})$ **1490** $(9_{4,2})$
538	1482 $(9_{7,1})$ 1490 $(10_{8,6})$ **1529** $(9_{3,2})$
541	1504 $(9_{6,1})$ **1539** $(10_{7,6})$

(*continued on next page*)

Table 8 (*continued*)

N	Entries k ($n_{m,u}$)
544	1545 ($10_{10,5}$) 1551 ($9_{5,1}$) 1565 ($11_{11,10}$) **1652** ($9_{9,0}$)
547	1596 ($9_{4,1}$) 1599 ($10_{9,5}$) **1662** ($9_{8,0}$)
550	1645 ($9_{3,1}$) 1660 ($10_{8,5}$) **1694** ($9_{7,0}$)
553	1711 ($10_{7,5}$) 1723 ($9_{2,1}$) **1751** ($9_{6,0}$)
556	1740 ($11_{11,9}$) 1772 ($10_{10,4}$) 1797 ($9_{5,0}$) **1809** ($10_{6,5}$)
559	1759 ($11_{10,9}$) 1787 ($10_{9,4}$) **1868** ($9_{4,0}$)
562	1838 ($10_{8,4}$) **1934** ($9_{3,0}$)
565	1936 ($10_{7,4}$) **2014** ($9_{2,0}$)
568	1929 ($11_{11,8}$) 1936 ($10_{10,3}$) 1990 ($10_{6,4}$) **2117** ($9_{0,0}$)
571	2008 ($10_{9,3}$) 2023 ($11_{10,8}$) 2041 ($10_{5,4}$)
574	2041 ($10_{8,3}$) **2148** ($11_{9,8}$)
577	**2178** ($10_{7,3}$)
580	2168 ($10_{10,2}$) 2200 ($10_{6,3}$) **2222** ($11_{11,7}$)
583	2224 ($10_{9,2}$) 2250 ($11_{10,7}$) **2262** ($10_{5,3}$)
586	2303 ($10_{8,2}$) 2339 ($10_{4,3}$) **2370** ($11_{9,7}$)
589	2394 ($11_{8,7}$) **2404** ($10_{7,2}$)
592	2348 ($12_{12,11}$) 2412 ($10_{10,1}$) 2465 ($11_{11,6}$) **2475** ($10_{6,2}$)
595	2485 ($10_{9,1}$) 2533 ($11_{10,6}$) **2543** ($10_{5,2}$)
598	2548 ($10_{8,1}$) 2587 ($11_{9,6}$) **2619** ($10_{4,2}$)

Define $\mathcal{S}_{m,u}$ to be the set of subspace restrictions consisting of

1. $(x_1, \ldots, x_m) = (1, 2, \ldots, m)$, and $U_i = (0)$ for $1 \le i \le m$;
2. for $1 \le i \le u$,

$$(x_1, x_2) = \begin{cases} (1,2) \text{ and } U_i = (0,1) & \text{if} \quad i = 1 \\ (i-1, i+1) \text{ and } U_i = (0,2) & \text{if} \quad i \equiv 2, 3 \pmod 4 \\ (i-1, i+1) \text{ and } U_i = (0,1) & \text{if} \quad i \equiv 0, 1 \pmod 4, i > 1 \end{cases}$$

A CPHF$(n; k, q, t)$ with $t \ge 3$ may be $\mathcal{S}_{m,u}$-restricted provided that $m, u \ge 0$, $m \ge u + 1$ if $u > 0$, and $n \ge m$. For such restrictions, the reduction in the number of rows in the generated CA can be substantial. Elementary counting ensures that replicated rows can be removed to produce a CA with $n(q^t - 1) + 1 - \max(m - 1, 0)(q - 1) - uq(q - 1)$ rows.

Such a CPHF consists of one subspace restriction of dimension $p = 1$ and replication $r = m$, which forces the first elements in the permutation vectors to match in m different rows. In addition, this set of restrictions contains u separate subspace restrictions of dimension $p = 2$ and restriction $r = 2$, where in addition to the first element, either the second or the third element of the permutation vectors match in two rows. Recall that for strength $t = 3$ restrictions of dimension $p > 2$ do not make any sense, since they would yield a replicated row in the entire CPHF. Therefore, by layering subspace restrictions with $r = d = 2$ in the manner described, we obtain a compact packing for subspace restrictions for $t = 3$. The set of subspace restrictions satisfied by the CPHF depicted in Fig. 5 follow this design with one restriction of dimension $p = 1$ and replication $r = 3$ as well as one restriction with $p = d = 2$ and can therefore be described as $\mathcal{S}_{3,1}$.

Our algorithm starts with a $\mathcal{S}_{m=n,u=n-1}$-restricted CPHF and thereafter either reduces u or m, with $u < m$. To evaluate the performance of this method, we first examine the existence of $\mathcal{S}_{m,u}$-restricted CPHF$(6; k, 4, 3)$s for admissible choices of m and u. Table 7 shows the largest value of k for which such CPHFs were constructed by our method on the left side, while the right side displays the numbers of rows of the respective CA after all replicated rows are removed.

Imposing stronger restrictions typically reduces the number of columns in the CPHF, as expected. At the same time, however, it also reduces the number of rows in the CA that is generated. Different restrictions can yield the same number of rows in the CA: For example, an $\mathcal{S}_{6,0}$-restricted CPHF$(6; k, 4, 3)$, an $\mathcal{S}_{2,1}$-restricted CPHF$(6; k, 4, 3)$, and an $\mathcal{S}_{7,5}$-restricted CPHF$(7; k, 4, 3)$ all yield a CA$(364, 3, k, 4)$. In the results presented, we focus on selected $\mathcal{S}_{m,u}$-restricted CPHFs for which we find improved covering array numbers, noting that other choices of m and u can also prove useful in obtaining such improvements.

Our results for $\mathcal{S}_{m,u}$-restricted CPHFs are given in Tables 8 and 9 when $q = 4$ and Table 10 when $q = 5$. By employing different sets of subspace restrictions, we constructed CAs with a large diversity of numbers of rows. The results for both alphabet sizes yield significant improvements over the best previously known bounds.

For $q = 4$, the majority of best known CAs had been constructed with a greedy-metaheuristic 3-stage method [24] that first constructs a CPHF that may contain missing combinations using Simulated Annealing, then converts the CPHF to a CA while adding any missing tuples before finally applying a post-optimization method to reduce the number of rows of the generated CA. The results in Tables 8 and 9 improve every bound for $v = q = 4$ for $799 \le k \le 10000$ columns, improving upon previously best known upper bounds by up to 30 rows. Because many different choices of n, m, and u can lead to the

Table 9
$\mathcal{S}_{m,u}$-restricted CPHF$(n; k, 4, 3)$s that yield CA$(N; 3, k, 4)$s for $601 \le N \le 739$. **Bold** entries indicate best new bounds on the size of the covering array.

N	Entries k $(n_{m,u})$
601	2678 $(10_{7,1})$ 2709 $(10_{3,2})$ **2726** $(11_{8,6})$
604	2730 $(11_{11,5})$ 2744 $(12_{12,10})$ 2753 $(10_{6,1})$ 2825 $(11_{7,6})$ **2870** $(10_{10,0})$
607	2803 $(12_{11,10})$ 2811 $(10_{5,1})$ 2860 $(11_{10,5})$ **2921** $(10_{9,0})$
610	2938 $(11_{9,5})$ 2950 $(10_{4,1})$ **2975** $(10_{8,0})$
613	3025 $(10_{7,0})$ 3032 $(10_{3,1})$ **3057** $(11_{8,5})$
616	3036 $(12_{12,9})$ 3046 $(11_{11,4})$ 3058 $(10_{6,0})$ 3062 $(10_{2,1})$ **3126** $(11_{7,5})$
619	3111 $(12_{11,9})$ 3154 $(11_{10,4})$ 3164 $(10_{5,0})$ **3266** $(11_{6,5})$
622	3202 $(12_{10,9})$ 3219 $(11_{9,4})$ **3420** $(10_{4,0})$
625	3276 $(11_{8,4})$ **3494** $(10_{3,0})$
628	3246 $(12_{12,8})$ 3369 $(11_{11,3})$ 3379 $(11_{7,4})$ **3581** $(10_{2,0})$
631	3422 $(12_{11,8})$ 3431 $(11_{10,3})$ 3511 $(11_{6,4})$ **3820** $(10_{0,0})$
634	3531 $(12_{10,8})$ 3552 $(11_{9,3})$ 3677 $(11_{5,4})$
637	3646 $(12_{9,8})$ 3708 $(11_{8,3})$
640	3606 $(13_{13,12})$ 3643 $(12_{12,7})$ 3807 $(11_{7,3})$ **3846** $(11_{11,2})$
643	3769 $(12_{11,7})$ 3886 $(11_{6,3})$ **3909** $(11_{10,2})$
646	3876 $(12_{10,7})$ 4022 $(11_{9,2})$ **4084** $(11_{5,3})$
649	4070 $(12_{9,7})$ 4154 $(11_{8,2})$ **4227** $(11_{4,3})$
652	4004 $(13_{13,11})$ 4138 $(12_{12,6})$ 4244 $(12_{8,7})$ 4280 $(11_{7,2})$ **4345** $(11_{11,1})$
655	4164 $(12_{11,6})$ 4217 $(13_{12,11})$ 4354 $(11_{10,1})$ **4418** $(11_{6,2})$
658	4433 $(12_{10,6})$ 4448 $(11_{9,1})$ **4594** $(11_{5,2})$
661	4578 $(11_{8,1})$ 4673 $(12_{9,6})$ **4723** $(11_{4,2})$
664	4558 $(13_{13,10})$ 4693 $(12_{12,5})$ 4762 $(11_{7,1})$ 4845 $(12_{8,6})$ 4891 $(11_{3,2})$ **5183** $(11_{11,0})$
667	4758 $(13_{12,10})$ 4919 $(11_{6,1})$ 4922 $(12_{11,5})$ 5055 $(12_{7,6})$ **5185** $(11_{10,0})$
670	4840 $(13_{11,10})$ 5069 $(11_{5,1})$ 5100 $(12_{10,5})$ **5213** $(11_{9,0})$
673	5212 $(11_{4,1})$ 5233 $(12_{9,5})$ **5309** $(11_{8,0})$
676	5220 $(13_{13,9})$ 5394 $(11_{3,1})$ 5436 $(12_{8,5})$ 5491 $(11_{7,0})$ **5495** $(12_{12,4})$
679	5377 $(13_{12,9})$ 5533 $(12_{11,4})$ 5582 $(12_{7,5})$ 5613 $(11_{2,1})$ **5709** $(11_{6,0})$
682	5629 $(13_{11,9})$ 5679 $(12_{10,4})$ 5853 $(12_{6,5})$ **5933** $(11_{5,0})$
685	5819 $(13_{10,9})$ 5881 $(12_{9,4})$ **6158** $(11_{4,0})$
688	5581 $(14_{14,13})$ 5991 $(13_{13,8})$ 6097 $(12_{12,3})$ 6113 $(12_{8,4})$ **6335** $(11_{3,0})$
691	6132 $(12_{11,3})$ 6187 $(13_{12,8})$ 6326 $(12_{7,4})$ **6465** $(11_{2,0})$
694	6382 $(12_{10,3})$ 6450 $(13_{11,8})$ 6462 $(12_{6,4})$ **6865** $(11_{0,0})$
697	6632 $(12_{9,3})$ 6646 $(12_{5,4})$ 6663 $(13_{10,8})$
700	6472 $(14_{14,12})$ 6526 $(13_{13,7})$ 6822 $(12_{12,2})$ 6858 $(13_{9,8})$ **6898** $(12_{8,3})$
703	6514 $(14_{13,12})$ 6782 $(13_{12,7})$ 6971 $(12_{11,2})$ **7020** $(12_{7,3})$
706	6989 $(13_{11,7})$ 7221 $(12_{10,2})$ **7254** $(12_{6,3})$
709	7188 $(13_{10,7})$ 7438 $(12_{9,2})$ **7459** $(12_{5,3})$
712	7296 $(14_{14,11})$ 7425 $(12_{12,1})$ 7451 $(13_{13,6})$ 7481 $(13_{9,7})$ 7588 $(12_{8,2})$ **7684** $(12_{4,3})$
715	7349 $(14_{13,11})$ 7577 $(12_{11,1})$ 7691 $(13_{12,6})$ 7834 $(13_{8,7})$ **7911** $(12_{7,2})$
718	7608 $(14_{12,11})$ 7852 $(12_{10,1})$ 7963 $(13_{11,6})$ **8064** $(12_{6,2})$
721	8181 $(13_{10,6})$ 8221 $(12_{9,1})$ **8348** $(12_{5,2})$
724	8129 $(14_{14,10})$ 8151 $(13_{13,5})$ 8458 $(12_{8,1})$ 8509 $(13_{9,6})$ 8638 $(12_{4,2})$ **8892** $(12_{12,0})$
727	8491 $(14_{13,10})$ 8529 $(13_{12,5})$ 8608 $(12_{7,1})$ 8759 $(13_{8,6})$ 8982 $(12_{11,0})$ **8983** $(12_{3,2})$
730	8775 $(14_{12,10})$ 8787 $(13_{11,5})$ 8874 $(12_{6,1})$ 9141 $(13_{7,6})$ **9294** $(12_{10,0})$
733	9176 $(14_{11,10})$ 9176 $(13_{10,5})$ 9367 $(12_{5,1})$ **9552** $(12_{9,0})$
736	8499 $(15_{15,14})$ 8768 $(14_{14,9})$ 8942 $(13_{13,4})$ 9569 $(13_{9,5})$ 9620 $(12_{4,1})$ **9771** $(12_{8,0})$
739	9597 $(14_{13,9})$ **10000** $(12_{7,0})$

same number of rows in the generated covering array, in Tables 8 and 9, we sort the results by the number of rows in the covering array generated. Entries in each row of the form "k $(n_{m,u})$" indicate that an $\mathcal{S}_{m,u}$-restricted CPHF$(n; k, 4, 3)$ was found. By so doing, one can see directly which choices of n, m, and u yield the better results. Choices that yield best known covering arrays are shown in bold typeface in the table.

The results for $q = 5$ (Table 10) also yield improvements on the best known upper bounds on CAN for all instances with $619 \le k \le 10000$. Within the range of parameters, few different choices of n, m, and u lead to covering arrays with the same numbers of rows; moreover, we have not carried out computations for all different choices. Nevertheless, results are given in the same manner as in Tables 8 and 9: Here, entries in each row of the form "k $(n_{m,u})$" indicate that an $\mathcal{S}_{m,u}$-restricted CPHF$(n; k, 5, 3)$ was found. Most of the previously known upper bounds arose from a simulated annealing algorithm that constructs CPHFs with subspace restrictions [25], while some were from a recursive method [9]. Again, best new bounds on covering array numbers are shown in bold in Table 10. Despite the substantial improvements for CPHFs with subspace restrictions by applying horizontal extension, we do not expect that these are the best possible sizes for covering arrays. Indeed the success appears to stem from the focus on horizontal extension, permitting the relatively rapid selection of a most suitable column to add. Simulated annealing explores a larger search space; in principle, this enables it to find better solutions, but in practice this becomes prohibitively slow.

Table 10
Lower bounds on k for selected $\mathcal{S}_{m,u}$-restricted CPHF$(n; k, 5, 3)$s that yield CA$(N; 3, k, 5)$s. **Bold** entries improve on the size of the best known covering array.

N	k $(n_{m,u})$
525	108 $(5_{5,4})$
545	126 $(5_{5,3})$
549	131 $(5_{4,3})$
565	160 $(5_{5,2})$
569	165 $(5_{4,2})$
573	170 $(5_{3,2})$
585	177 $(5_{5,1})$
589	183 $(5_{4,1})$
593	187 $(5_{3,1})$
597	195 $(5_{2,1})$
605	228 $(5_{5,0})$
609	233 $(5_{4,0})$
613	236 $(5_{3,0})$
617	**244** $(5_{2,0})$
621	254 $(5_{0,0})$
625	209 $(6_{6,5})$
645	261 $(6_{6,4})$
649	267 $(6_{5,4})$
665	**306** $(6_{6,3})$
669	**315** $(6_{5,3})$
673	323 $(6_{4,3})$
685	**353** $(6_{6,2})$
689	**367** $(6_{5,2})$
693	**378** $(6_{4,2})$
697	**385** $(6_{3,2})$
705	**406** $(6_{6,1})$
709	**421** $(6_{5,1})$
713	**424** $(6_{4,1})$
717	**433** $(6_{3,1})$
721	449 $(6_{2,1})$
725	406 $(7_{7,6})$
	502 $(6_{6,0})$
729	**516** $(6_{5,0})$
733	**522** $(6_{4,0})$
737	**545** $(6_{3,0})$
741	**554** $(6_{2,0})$
745	478 $(7_{7,5})$
	578 $(6_{0,0})$
749	508 $(7_{6,5})$
765	581 $(7_{7,4})$
769	595 $(7_{6,4})$
773	**619** $(7_{5,4})$
785	**675** $(7_{7,3})$
789	**691** $(7_{6,3})$
793	**711** $(7_{5,3})$
797	**724** $(7_{4,3})$
805	**785** $(7_{7,2})$
809	**803** $(7_{6,2})$
813	**825** $(7_{5,2})$
817	**845** $(7_{4,2})$
821	**863** $(7_{3,2})$
N	Entries k $(n_{m,u})$
825	771 $(8_{8,7})$ **889** $(7_{7,1})$
829	**905** $(7_{6,1})$
833	**941** $(7_{5,1})$
837	**961** $(7_{4,1})$
841	**975** $(7_{3,1})$
845	914 $(8_{8,6})$ 991 $(7_{2,1})$
	1083 $(7_{7,0})$
849	932 $(8_{7,6})$ **1111** $(7_{6,0})$
853	**1129** $(7_{5,0})$
857	**1164** $(7_{4,0})$
861	**1181** $(7_{3,0})$
865	1065 $(8_{8,5})$ **1212** $(7_{2,0})$
869	1088 $(8_{7,5})$ **1291** $(7_{0,0})$

(*continued on next page*)

Table 10 (*continued*)

N	k ($n_{m,u}$)
873	1115 ($8_{6,5}$)
885	1240 ($8_{8,4}$)
889	1274 ($8_{7,4}$)
893	**1306** ($8_{6,4}$)
897	**1339** ($8_{5,4}$)
905	**1417** ($8_{8,3}$)
909	**1469** ($8_{7,3}$)
913	**1509** ($8_{6,3}$)
917	**1541** ($8_{5,3}$)
921	**1567** ($8_{4,3}$)
925	1389 ($9_{9,8}$) **1608** ($8_{8,2}$)
929	**1688** ($8_{7,2}$)
933	**1717** ($8_{6,2}$)
937	**1766** ($8_{5,2}$)
941	**1805** ($8_{4,2}$)
945	1641 ($9_{9,7}$) 1836 ($8_{8,1}$)
	1870 ($8_{3,2}$)
949	1674 ($9_{8,7}$) **1879** ($8_{7,1}$)
953	**1921** ($8_{6,1}$)
957	**1969** ($8_{5,1}$)
961	**2041** ($8_{4,1}$)
965	1908 ($9_{9,6}$) **2089** ($8_{3,1}$)
	2288 ($8_{8,0}$)
969	1925 ($9_{8,6}$) 2109 ($8_{2,1}$)
	2316 ($8_{7,0}$)
973	1995 ($9_{7,6}$) **2352** ($8_{6,0}$)
977	**2406** ($8_{5,0}$)
981	**2459** ($8_{4,0}$)
985	2233 ($9_{9,5}$) **2555** ($8_{3,0}$)
989	2271 ($9_{8,5}$) **2573** ($8_{2,0}$)
993	2330 ($9_{7,5}$) **2687** ($8_{0,0}$)
997	2363 ($9_{6,5}$)
1005	2554 ($9_{9,4}$)
1009	2641 ($9_{8,4}$)
1013	**2689** ($9_{7,4}$)
1017	**2711** ($9_{6,4}$)
1021	**2766** ($9_{5,4}$)
1025	2357 ($10_{10,9}$) **2778** ($9_{9,3}$)
N	Entries k ($n_{m,u}$)
1029	**2890** ($9_{8,3}$)
1033	**3111** ($9_{7,3}$)
1037	**3169** ($9_{6,3}$)
1041	**3216** ($9_{5,3}$)
1045	2725 ($10_{10,8}$) 3195 ($9_{9,2}$) **3240** ($9_{4,3}$)
1049	2823 ($10_{9,8}$) **3303** ($9_{8,2}$)
1053	**3534** ($9_{7,2}$)
1057	**3629** ($9_{6,2}$)
1061	**3657** ($9_{5,2}$)
1065	3230 ($10_{10,7}$) 3672 ($9_{4,2}$) **3734** ($9_{9,1}$)
1069	3269 ($10_{9,7}$) 3723 ($9_{3,2}$) **3780** ($9_{8,1}$)
1073	3384 ($10_{8,7}$) **3829** ($9_{7,1}$)
1077	**4037** ($9_{6,1}$)
1081	**4100** ($9_{5,1}$)
1085	3624 ($10_{10,6}$) 4121 ($9_{4,1}$) **4571** ($9_{9,0}$)
1089	3779 ($10_{9,6}$) 4208 ($9_{3,1}$) **4643** ($9_{8,0}$)
1093	3916 ($10_{8,6}$) 4263 ($9_{2,1}$) **4743** ($9_{7,0}$)
1097	4012 ($10_{7,6}$) **4754** ($9_{6,0}$)
1101	**4811** ($9_{5,0}$)
1105	4304 ($10_{10,5}$) **4871** ($9_{4,0}$)
1109	4389 ($10_{9,5}$) **5014** ($9_{3,0}$)
1113	4494 ($10_{8,5}$) **5110** ($9_{2,0}$)
1117	4580 ($10_{7,5}$) **5335** ($9_{0,0}$)
1121	4720 ($10_{6,5}$)
1125	5078 ($10_{10,4}$)
1129	5150 ($10_{9,4}$)
1133	5215 ($10_{8,4}$)
1137	**5356** ($10_{7,4}$)
1141	**5476** ($10_{6,4}$)
1145	5541 ($10_{10,3}$) **5598** ($10_{5,4}$)

(*continued on next page*)

Table 10 (*continued*)

N	k ($n_{m,u}$)
1149	**5680** ($10_{9,3}$)
1153	**5833** ($10_{8,3}$)
1157	**5949** ($10_{7,3}$)
1161	**6129** ($10_{6,3}$)
1165	6307 ($10_{10,2}$) **6326** ($10_{5,3}$)
1169	6496 ($10_{4,3}$) **6503** ($10_{9,2}$)
1173	**6693** ($10_{8,2}$)
1177	**6887** ($10_{7,2}$)
1181	**7084** ($10_{6,2}$)
1185	7058 ($10_{10,1}$) **7276** ($10_{5,2}$)
1189	7332 ($10_{9,1}$) **7470** ($10_{4,2}$)
1193	7562 ($10_{8,1}$) **7574** ($10_{3,2}$)
1197	**7709** ($10_{7,1}$)
1201	**7941** ($10_{6,1}$)
1205	**8979** ($10_{10,0}$)
1209	**9007** ($10_{9,0}$)
1213	**9180** ($10_{8,0}$)
1217	**9395** ($10_{7,0}$)
1221	**9517** ($10_{6,0}$)
1225	**9727** ($10_{5,0}$)
1229	**10000** ($10_{4,0}$)

7. Conclusion

Our work combines the popular In-Parameter-Order strategy with the very potent representation of a certain family of CAs, namely covering perfect hash families. We describe in detail how such an algorithm can be designed and optimized. Our experiments, which improve upon tens of thousands of best known upper bounds, showcase the tremendous potential of this algorithmic approach. In addition, applying similar methods to CPHFs with subspace restrictions yielded significant improvements to the best known upper bounds on CAN for the vast majority of instances of strength three and $v = 4, 5$.

At each stage of our methods, the current solution being extended is always a covering perfect hash family. Although we have reported on computations that start from scratch, the same horizontal extension techniques can be applied to any CPHF, no matter how it was initially produced. This opens the door to hybrid methods that combine the strengths of simulated annealing for smaller parameters, and of recursive and direct constructions, with the ability of the IPO methods to extend these for larger parameters. Viewed in this way, our methods complement, rather than replace, existing methods. However, the question of when to use each of the known methods in a hybrid approach remains to be determined, and surely depends on the total computation time available.

Acknowledgements

This research was carried out partly in the context of the Austrian COMET K1 program and publicly funded by the Austrian Research Promotion Agency (FFG) and the Vienna Business Agency (WAW). Research of the second author is funded by the U.S. National Science Foundation grants #1421058 and #1813729.

References

[1] L. Kampel, D.E. Simos, A survey on the state of the art of complexity problems for covering arrays, Theor Comput Sci 800 (2019) 107–124.
[2] C. J. Colbourn, Covering Array Tables for t=2,3,4,5,6, (Available at http://www.public.asu.edu/~ccolbou/src/tabby/catable.html, Accessed on 2021-08-03).
[3] D.R. Kuhn, R.N. Kacker, Y. Lei, et al., Practical combinatorial testing, NIST special Publication 800 (142) (2010) 142.
[4] D. Jarman, R. Smith, G. Gosney, L. Kampel, M. Leithner, D.E. Simos, R. Kacker, R. Kuhn, Applying combinatorial testing to large-scale data processing at adobe, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2019, pp. 190–193.
[5] G.B. Sherwood, S.S. Martirosyan, C.J. Colbourn, Covering arrays of higher strength from permutation vectors, J. Comb. Des. 14 (3) (2006) 202–213.
[6] J. Torres-Jimenez, I. Izquierdo-Marquez, Covering arrays of strength three from extended permutation vectors, Designs, Codes and Cryptography 86 (11) (2018) 2629–2643.
[7] Y. Lei, K.C. Tai, In-parameter-order: a test generation strategy for pairwise testing, in: Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231), 1998, pp. 254–261.
[8] J. Torres-Jimenez, H. Avila-George, Search-based software engineering to construct binary test-suites, in: J. Mejia, M. Munoz, A. Rocha, J. Calvo-Manzano (Eds.), Trends and Applications in Software Engineering, Springer International Publishing, 2016, pp. 201–212.
[9] M.B. Cohen, C.J. Colbourn, A.C.H. Ling, Constructing strength three covering arrays with augmented annealing, Discrete Math 308 (13) (2008) 2709–2722.
[10] B. Garn, D.E. Simos, Algebraic modelling of covering arrays, in: I.S. Kotsireas, E. Martínez-Moro (Eds.), Applications of Computer Algebra, Springer International Publishing, Cham, 2017, pp. 149–170.
[11] J. Torres-Jimenez, I. Izquierdo-Marquez, H. Avila-George, Methods to construct uniform covering arrays, IEEE Access 7 (2019) 42774–42797.
[12] L. Kampel, M. Leithner, D.E. Simos, Sliced AETG: a memory-efficient variant of the AETG covering array generation algorithm, Optimization Letters 14 (6) (2020) 1543–1556.
[13] R.C. Bryce, C.J. Colbourn, A density-based greedy algorithm for higher strength covering arrays, Softw. Test. Verif. Reliab. 19 (1) (2009) 37–53.
[14] C.J. Colbourn, Covering arrays from cyclotomy, Designs, Codes and Cryptography 55 (2) (2010) 201–219.

[15] J. Torres-Jimenez, I. Izquierdo-Marquez, A simulated annealing algorithm to construct covering perfect hash families, Mathematical Problems in Engineering 2018 (2018).
[16] R.A. Walker II, C.J. Colbourn, Tabu search for covering arrays using permutation vectors, J Stat Plan Inference 139 (1) (2009) 69–80.
[17] C.J. Colbourn, E. Lanus, K. Sarkar, Asymptotic and constructive methods for covering perfect hash families and covering arrays, Designs, Codes and Cryptography 86 (4) (2018) 907–937.
[18] C.J. Colbourn, E. Lanus, Subspace restrictions and affine composition for covering perfect hash families, The Art of Discrete and Applied Mathematics 1 (2) (2018) P2–03.
[19] K. Kleine, D.E. Simos, An Efficient Design and Implementation of the In-Parameter-order Algorithm, Mathematics in Computer Science (2017).
[20] M. Wagner, K. Kleine, D.E. Simos, R. Kuhn, R. Kacker, Cagen: A fast combinatorial test generation tool with support for constraints and higher-index arrays, in: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2020, pp. 191–200.
[21] A. Bombarda, E. Crippa, A. Gargantini, An environment for benchmarking combinatorial test suite generators, in: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2021, pp. 48–56.
[22] C.J. Colbourn, S.S. Martirosyan, T. Van Trung, R.A. Walker, Roux-type constructions for covering arrays of strengths three and four, Designs, Codes and Cryptography 41 (1) (2006) 33–57.
[23] C.J. Colbourn, G. Kéri, P.P. Rivas Soriano, J.C. Schlage-Puchta, Covering and radius-covering arrays: constructions and classification, Discrete Appl. Math. 158 (2010) 1158–1190.
[24] I. Izquierdo-Marquez, J. Torres-Jimenez, B. Acevedo-Juárez, H. Avila-George, A greedy-metaheuristic 3-stage approach to construct covering arrays, Inf Sci (Ny) 460 (2018) 172–189.
[25] J. Torres-Jimenez, I. Izquierdo-Marquez, Improved covering arrays using covering perfect hash families with groups of restricted entries, Appl Math Comput 369 (2020) 124826.