# Evaluation of Tie-Breaking and Parameter Ordering for the IPO Family of Algorithms Used in Covering Array Generation

Kristoffer Kleine[1], Ilias Kotsireas[2], and Dimitris E. Simos[1(✉)]

[1] SBA Research, 1040 Vienna, Austria
{kkleine,dsimos}@sba-research.org
[2] Wilfrid Laurier University, Waterloo, ON, Canada
ikotsire@wlu.ca

**Abstract.** The IPO (In-Parameter-Order) family of algorithms is a popular set of greedy methods for the construction of covering arrays. Aspects such as tie-breaking behavior or parameter ordering can have major impact on the quality of the resulting arrays but have so far not been studied in a systematic manner. In this paper, we survey and present a general framework for the IPO family of algorithms (i.e. IPOG, IPOG-F and IPOG-F2) and present ways to instantiate these abstract components. Then, we evaluate the performance of these variations on a large set of instances, in an extensive experimental setting in terms of covering array sizes.

**Keywords:** Covering arrays · IPO family · Tie-breaks
Parameter ordering · Experiments

## 1 Introduction

A covering array (CA) is a mathematical object defined by four positive integers and denoted as $CA(N; t, k, v)$. It is a $N \times k$ matrix where $N$ is the number of rows, $k$ the number of columns (often referred to as parameters), $t$ the size of interactions that are covered and $v$ is the size of the alphabet. A covering array is defined by its *t-covering property*: for any $t$-selection of columns, all $v^t$ $t$-tuples between the selected columns occur at least once in the array. $t$ is called the strength of the CA. A mixed-level covering array (MCA) is a generalization of a CA where each column $i$ has its own alphabet size $v_i$. An MCA is denoted as $MCA(N; t, k, (v_1, \ldots, v_k))$. The tuple $(t, k, (v_1, \ldots, v_k))$ $(t \leq k)$ is referred to as the configuration of an $MCA$.

The general problem of constructing optimal covering arrays (i.e., in terms of minimal size $N$) is believed to be a hard combinatorial optimization problem and it has significant applications in software and hardware testing [8,9]. Moreover, it is tightly coupled with NP-hard problems [2]. As a result, there has been a lot

© Springer International Publishing AG, part of Springer Nature 2018
C. Iliopoulos et al. (Eds.): IWOCA 2018, LNCS 10979, pp. 189–200, 2018.
https://doi.org/10.1007/978-3-319-94667-2_16

of effort on developing and improving algorithmic approaches for covering array generation (c.f. Sect. 2).

The In-Parameter-Order (IPO) family of algorithms is a set of greedy algorithms for constructing covering arrays and their representatives have been shown to produce acceptable sized covering arrays [3]. There have been multiple efforts in the area of improving the IPO variants, in terms of reducing the size of generated covering arrays, but to the best of our knowledge, no systematic evaluation of these proposals exist. In this work, we aim to provide an overview of these optimization efforts and evaluate their effectiveness.

This paper is structured as follows: In Sect. 2 we discuss preliminaries and related work. Section 3 presents the IPO family of algorithms and existing and novel methods to parameterize it. Section 4 proposes an evaluation of the algorithms and discusses the results and finally, Sect. 5 concludes the paper.

## 2 Background

The In-Parameter-Order (IPO) strategy was first proposed in [15] as a greedy algorithm for covering array construction. It constructs a covering array one column at a time and each extension is divided into a horizontal and an optional vertical extension. We discuss the algorithm in greater detail in Sect. 3.

While the original algorithm was limited to arrays of strength 2 (pair-wise), subsequent works have generalized the algorithm to allow the generation of higher strength arrays [13] (IPOG) as well as integrated constraint handling in [17,18]. In [5], the authors propose variants of the IPO strategy, namely IPOG-F, IPOG-F2, by extending the search space in the horizontal extension. In [14], the IPOG-D variant is presented which includes a recursive construction method aimed at reducing the number of combinations to be enumerated.

Many works have been dedicated to improving parts of the IPO algorithms in order to minimize covering arrays sizes. In [4] a graph-coloring scheme integrated into the vertical extension is proposed to reduce the resulting array sizes. [16] modify IPOG with additional optimizations aimed at reducing *don't-care* values in order to minimize the number of rows. [6,7] discuss and evaluate the impact of tie-breaking on the generated arrays and propose a new tie-breaker which reduces the generated array sizes.

Related to the aim of this paper is the works of [1] for "one-test-at-a-time" greedy construction algorithms. There, the authors propose a general framework of such algorithms and provide concrete ways to instantiate it. In their analysis they evaluate the most important parameters which contribute to smaller covering array sizes.

## 3 The IPO Family of Algorithms

The IPO family of algorithms is a set of algorithms for constructing covering arrays and has seen use in many applications in practice [12]. The most important representatives are IPOG [13], IPOG-F [5] and IPOG-F2 [5]. They are greedy

algorithms able to generate (mixed-level) covering arrays for any configuration. They all follow the same basic schema (c.f. Algorithm 1) and start out by building an initial covering array of strength $t$ by constructing the cross-product of the first $t$ columns.

**Algorithm 1.** IPO Algorithm

```
procedure IPO(configuration)
    configuration ← Sort-Criterion(configuration)
    Array ← cross-product of first t columns
    for i ← t, ..., k do
        HorizontalExtension(i)
        if there are uncovered tuples then
            VerticalExtension(i)
        end if
    end for
end procedure
```

Then, the array is extended one column at a time in a two-dimensional fashion. First, a new column is added to the array by a horizontal extension. This step will assign values to the new column in a greedy manner with the objective to maximize coverage gain. Algorithm 2 shows the procedure for `IPOG`. Here, rows are considered from top to bottom and in each one the value with the maximum coverage gain (in terms of previously uncovered $t$-tuples) is selected. If multiple values provide maximum gain then the tie must be broken with a predefined strategy. This will be discussed in further detail in Sect. 3.1.

`IPOG-F` and `IPOG-F2` extend the search space in the horizontal extension by also optimizing the order in which rows are extended. For this purpose, unassigned rows are additionally scanned in each iteration to find the best position and value to assign. While `IPOG-F` considers the actual coverage to decide on the best value, `IPOG-F2` tries to estimate the amount of covered tuples. This is an optimization which allows for a faster implementation compared to `IPOG-F`, but it sacrifices accuracy. The estimation is achieved by keeping an estimated value of covered tuples in an array for each row and value. When a value $a$ is chosen in row $r$, the estimator is incremented by the number of shared tuples between row $s$ and all other unassigned rows $s$.

The horizontal extension either finishes when all tuples have been covered or each entry in the new column has been assigned a value. In the latter case, there are still tuples left which have not yet been covered and they are added in the vertical extension step. This step grows the array by adding more rows which include missing tuples.

In the vertical extension all remaining uncovered tuples are added to the array to ensure that the first $i$ columns form a covering array. Tuples can either be added by appending a new row to the array that contains the tuple or by finding an already existing row which can fit the tuple. The latter case is possible

**Algorithm 2.** Horizontal Extension (IPOG)

```
procedure HorizontalExtension(i)
    for row ← 0, ... ,Array.rows do
        v ← select value with highest coverage gain
        if multiple candidate values then
            break tie
        end if
        Array[row][i] ← v
        if all tuples are covered then return
        end if
    end for
end procedure
```

since don't-care values can occur in the array and may be overwritten by a tuple without destroying the $t$-covering property.

### 3.1 Tie-Breakers

During the horizontal extension, tie-breaking may be necessary in the case that two or more values for a row in the new column provide the same, maximum coverage gain, i.e. cover the most new $t$-tuples. We will refer to these values as candidates. In the following we will give an overview to possible tie-breaking strategies.

**Random Tie-Breaker.** The simplest approach is to choose one value out of all candidates at random. This can be implemented efficiently but it will introduce non-determinism to algorithm and the generated covering arrays will possibly differ on subsequent runs of the algorithm. This tie-breaker is oblivious to the previous history of the extension.

**Deterministically-Seeded Random Tie-Breaker.** This is a variant of the `Random` tie-breaker. Here, ties are still broken randomly with the help of a pseudorandom generator, but the generator is seeded with a constant at the beginning which results in a deterministic behaviour of the algorithm.

**Lexicographic Tie-Breaker.** This tie-breaker will always prefer the (lexicographically) smallest candidate if multiple are available. This can of course introduce a bias towards smaller values in the new column.

**Cyclic Tie-Breaker.** This tie-breaker builds upon the `Lexicographic` tie-breaker, but maintains the last chosen value and starts the search from this one instead of the first. The aim is to remove bias towards smaller values, however the last chosen value is more likely to be picked again in the next iteration.

**Cyclic-Next Tie-Breaker.** This tie-breaker works exactly as the `Cyclic` tie-breaker, but will start from the next value following the last chosen value. This tie-breaker was first proposed in [6,7].

**Value-Balanced Tie-Breaker.** This tie-breaker keeps track of how many times a value has been used so far in the extended column. In an optimal situation, each value for the new column occurs exactly the same amount of times and the aim of this tie-breaker is to mimic this behaviour by balancing the occurrences of these values. Values are preferred when they so far have occured less frequently than other candidate values.

**$\alpha$-balanced Tie-Breaker.** This tie-breaker builds upon the value-balanced tie-breaker by not only considering the balance of values in the new column, but the balance of lower-strength tuples involving the new parameter. This is based on the notion of $\alpha$-balance which was introduced by [10] and functions as a tie-breaker in the following way: first, the number of would-be-covered $t-1$ tuples are compared for each candidate. If there still is a tie, the next lower strength is tried and so on. If at $t=1$ there still exists a tie, then the smallest value will be preferred.

### 3.2 Tuple Enumeration Order

In the vertical extension, uncovered tuples are added one-by-one to the array. So far it has not been studied, if different enumeration orders of these tuples have any impact on the resulting arrays. We propose besides the common lexicographically-ascending (tuples of small lexicographical order first) order, the reverse, i.e. from lexicographically largest to smallest. Furthermore, switching between the orderings every other vertical extension could prove beneficial to achieve smaller arrays.

### 3.3 Parameter Ordering

One simple option to influence the covering array generation is the order in which columns are extended. Since the covering property is not affected by column permutations one can permute the configuration before starting the generation and apply the reverse permutation afterwards. Note that this is only useful for mixed-level covering arrays. Informal consensus is that the `IPO` strategy generates smaller arrays when columns are sorted by decreasing alphabet size, but, to the best of our knowledge, this has so far not been subject to an experimental evaluation.

While the number of column permutations in general is too large in practice, we propose to investigate the following:

**Ascending** Sort columns with increasing alphabet size from smallest to largest
**Descending** Sort columns with decreasing alphabet size from largest to smallest

**Alternating** Intersperse large and small columns and switch between large and small columns from one extension to the next. We propose two variants. The first starts with the smallest, followed by the largest and thirdly the second-smallest, etc. The second starts with the largest, followed by the smallest and so on.

## 4 Evaluation

### 4.1 Setup

To evaluate the different algorithm configurations we chose a set of (M)CA instances based upon the benchmarks used in [1] to study the behaviour of greedy, one-test-at-a-time MCA generation algorithms. The instances are summarized in Fig. 1a.

| Instances |
|---|
| $10^4$ |
| $3^{40}$ |
| $3^4$ |
| $6^4$ |
| $3^4, 4^5$ |
| $6^6, 5^5, 3^4$ |
| $7^8, 2^{20}$ |
| $5^1, 3^8, 2^2$ |
| $5^{10}, 2^{10}$ |
| $8^2, 7^2, 6^2, 5^2$ |
| $10^1, 9^1, 8^1, 7^1, 6^1, 5^1, 4^1, 3^1, 2^1, 1^1$ |

(a) Set of benchmark (M)CA instances

| Tie Breakers | Tuple Orders | Parameter Orders |
|---|---|---|
| Alpha-Balanced | Alternating | Alternating-large |
| Cyclic | Ascending | Alternating-small |
| Cyclic-next | Descending | Ascending |
| Deterministic | | Descending |
| Lexicographic | | |
| Random | | |
| Value-Balanced | | |

(b) Configuration options for `IPO`

**Fig. 1.** Benchmark setup

We implemented all IPO variants in our own implementation described further in [11]. The particular algorithm as well as the tie-breaker, tuple order and parameter ordering are selectable via a configuration option at runtime. This results in 63 distinct algorithm configurations for CA generation and 252 distinct configurations for MCA generation. The configuration options are summarized in Fig. 1b.

Each algorithm configuration was used to generate (M)CAs for the selected benchmark instances for strengths between 2 and 4. The experiments were conducted on the Graham cluster of the Shared Hierarchical Academic Research Computing Network (SHARCNET). Configurations involving the `Random` tie-breaker were repeated 10 times. Due to space limitations, we only discuss selected and aggregated results, but we provide the full data set as well as visualizations

on a dedicated website[1] for the interested reader. There we also provide further measurements into the test generation time differences observed between the tested configurations.

### 4.2 Results

In order to meaningfully compare different configurations options across instances we first normalized the computed covering array sizes to a relative measure representing the deviation of the mean. We computed the mean for each instance and based on the result computed the relative improvement or degradation for each individual run. This value shows how much better or worse one configuration performs in comparison to the other ones. The results are summarized in Table 1 and are visualized in Figs. 2a and b.

**Table 1.** Relative improvement for different configurations compared to the mean

| | IPOG | IPOG-F | IPOG-F2 |
|---|---|---|---|
| **Tie Breaker** | | | |
| Alpha-balanced | 1.0128 ±0.0759 | 0.9632 ±0.0619 | 1.0572 ±0.0662 |
| Cyclic | 1.0175 ±0.1394 | 0.9461 ±0.0678 | 1.0379 ±0.0867 |
| Cyclic-next | 0.9721 ±0.0858 | 0.9403 ±0.0799 | 1.0296 ±0.1012 |
| Deterministic | 0.9920 ±0.0429 | 0.9560 ±0.0539 | 1.0500 ±0.0664 |
| Lexicographic | 1.0140 ±0.0764 | 0.9651 ±0.0687 | 1.0580 ±0.0673 |
| Random | 0.9933 ±0.0465 | 0.9548 ±0.0545 | 1.0497 ±0.0673 |
| Value-balanced | 0.9951 ±0.0630 | 0.9590 ±0.0563 | 1.0549 ±0.0645 |
| **Tuple Order** | | | |
| Alternating | 0.9950 ±0.0656 | 0.9533 ±0.0580 | 1.0454 ±0.0681 |
| Ascending | 0.9987 ±0.0663 | 0.9582 ±0.0626 | 1.0584 ±0.0794 |
| Descending | 0.9944 ±0.0630 | 0.9530 ±0.0561 | 1.0432 ±0.0644 |
| **Parameter Order** | | | |
| Alternating-large | 0.9962 ±0.0309 | 0.9529 ±0.0268 | 1.0503 ±0.0510 |
| Alternating-small | 1.0082 ±0.0374 | 0.9667 ±0.0336 | 1.0763 ±0.0525 |
| Ascending | 1.0285 ±0.0443 | 1.0006 ±0.0440 | 1.0870 ±0.0519 |
| Descending | 0.9463 ±0.0791 | 0.8910 ±0.0442 | 0.9952 ±0.0793 |

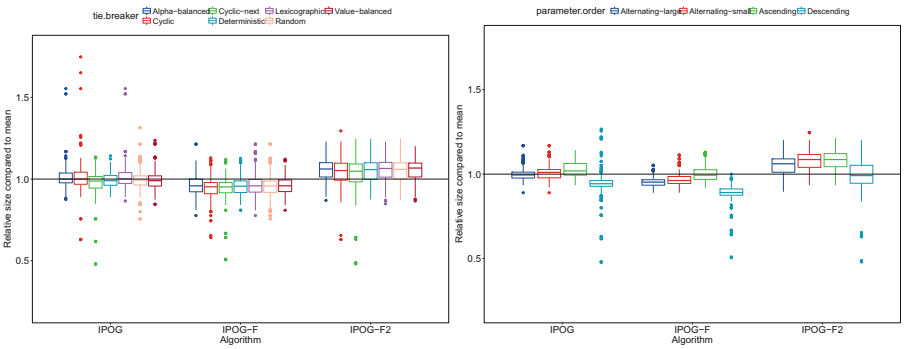
In general, `IPOG-F` produces the smallest arrays, followed by `IPOG` and `IPOG-F2`. Comparing the results for the different tie-breakers, no one choice seems to impact array sizes significantly, however, the `Cyclic-next` tie-breaker overall yields the best results. It, together with the `Cyclic` tie-breaker is able to generate some arrays with up to 50% less rows. However, the `Cyclic` tie-breaker exhibits extreme results in the other direction and in corner-cases with array

[1] https://matris.sba-research.org/data/iwoca2018.

sizes exceeding 50% larger than the mean are produced. This is also the case for the `Alpha-balanced` and `Lexicographic` tie-breaker.

Judging from the results in Table 1, the order in which tuples are enumerated does not seem to affect the resulting covering array size in any significant way.

The largest impact can be attributed to the sorting order of columns. Sorting in descending order of alphabet size leads to significantly smaller covering arrays, especially in the case of `IPOG-F`. Alternating between large and small columns has some impact and is better than sorting columns in ascending order.



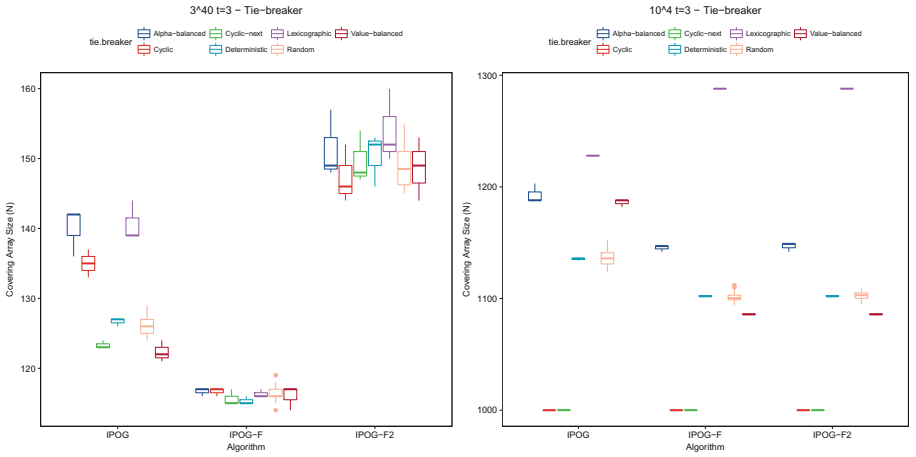

(a) Tie-breakers (b) Parameter orders

**Fig. 2.** Relative improvement compared to the mean

*Selected benchmark results.* Aside from the general performance, for specific instances the various configuration options can have differing impact. In the following, we discuss some results for selected instances. In order to meaningfully analyze the results we have grouped the results by both the algorithm (i.e., `IPOG`, `IPOG-F` or `IPOG-F2`) and one of either tie-breaker, tuple-order or parameter-order. Inside each group we have computed the mean and the standard deviation. The results show absolute values instead of relative difference.

**$3^{40}$ ($t = 3$)** The results of this experiment are summarized in Table 2 and the generated covering array sizes per tie-breaker are visualized in Fig. 3a. `IPOG-F` produces the smallest arrays and shows very low variance when comparing different tie-breakers. In contrast, the results for `IPOG` are much more dependent on the tie-breaker. Here, the best results are obtained with the `Value-balanced` tie-breaker which produces arrays 16% smaller than when using the `Alpha-balanced` tie-breaker. `IPOG-F2` shows no significantly differing behavior with different tie-breakers. Furthermore, the order in which tuples are enumerated have no major impact.

**$10^4$ ($t = 3$)** The results for this experiment can be found in Table 2 and a comparison of the tie breakers can be found in Fig. 3b. Here, the configurations which use either the `Cyclic` or `Resuming` tie-breakers manage to generate an




(a) $3^{40}$ $(t = 3)$ (b) $10^4$ $(t = 3)$

**Fig. 3.** Results for different tie-breakers

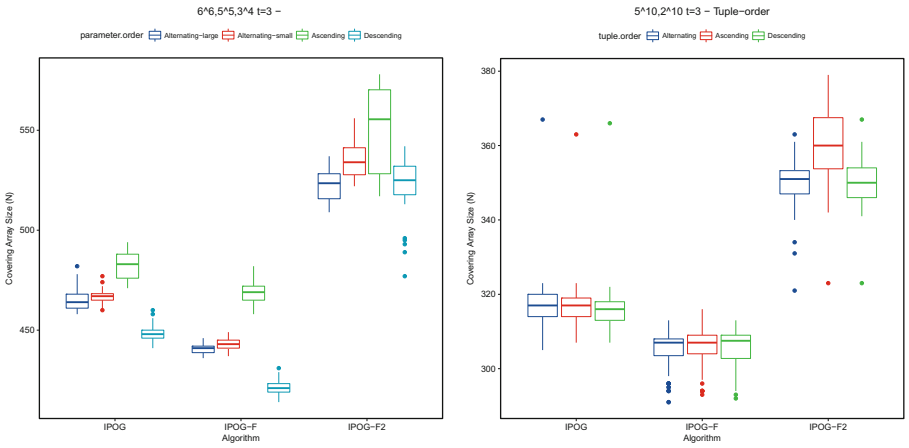
**Table 2.** Results for CA experiments

| | $3^{40}$ $t = 3$ | | | $10^4$ $t = 3$ | | |
|---|---|---|---|---|---|---|
| | IPOG | IPOG-F | IPOG-F2 | IPOG | IPOG-F | IPOG-F2 |
| Tie Breaker | | | | | | |
| Alpha-balanced | 140.0 ±3.5 | 116.7 ±0.6 | 151.3 ±4.9 | 1193.0 ±8.7 | 1145.3 ±2.9 | 1146.7 ±4.0 |
| Cyclic | 135.0 ±2.0 | 116.7 ±0.6 | 147.3 ±4.2 | 1000.0 ±0.0 | 1000.0 ±0.0 | 1000.0 ±0.0 |
| Cyclic-next | 123.3 ±0.6 | 115.7 ±1.2 | 149.7 ±3.8 | 1000.0 ±0.0 | 1000.0 ±0.0 | 1000.0 ±0.0 |
| Deterministic | 126.7 ±0.6 | 115.3 ±0.6 | 150.3 ±3.8 | 1135.3 ±1.2 | 1102.3 ±0.6 | 1102.3 ±0.6 |
| Lexicographic | 140.7 ±2.9 | 116.3 ±0.6 | 154.0 ±5.3 | 1228.0 ±0.0 | 1288.0 ±0.0 | 1288.0 ±0.0 |
| Random | 125.9 ±1.3 | 116.4 ±1.0 | 148.9 ±2.7 | 1136.2 ±7.0 | 1101.4 ±4.3 | 1102.2 ±3.7 |
| Value-balanced | 122.3 ±1.5 | 116.0 ±1.7 | 148.7 ±4.5 | 1186.0 ±3.5 | 1085.7 ±0.6 | 1085.7 ±0.6 |
| Tuple Order | | | | | | |
| Alternating | 127.6 ±5.7 | 116.2 ±0.9 | 147.4 ±2.3 | 1130.0 ±58.2 | 1102.4 ±62.1 | 1103.7 ±62.1 |
| Ascending | 128.7 ±6.4 | 116.5 ±1.0 | 152.6 ±3.0 | 1131.9 ±59.2 | 1102.1 ±61.9 | 1102.1 ±61.9 |
| Descending | 127.6 ±4.8 | 116.1 ±1.0 | 148.3 ±2.5 | 1132.6 ±58.2 | 1102.1 ±62.0 | 1102.6 ±62.2 |

orthogonal array (since the size is equal to $v^t$) for the three algorithms. Interestingly, the `Lexicographic` tie-breaker, although similar to the other two, performs the worst in all cases with almost 30% larger array sizes. As before, in this case the tuple order has no real impact.

### 4.3 $6^6 5^5 3^4$ $(t = 3)$

For this instance (see Table 3), there is no large variance when comparing different tie-breakers. `IPOG-F` produces the smallest arrays, while `IPOG-F2` produces the largest. Here, the parameter order has a measurable impact and ordering the parameters by descending size can improve array sizes by up to 5% in this case. These results are visualized in Fig. 4a.




(a) $6^6 5^5 3^4$ $(t = 3)$ (b) $5^{10} 2^{10}$ $(t = 3)$

**Fig. 4.** Results for different parameter orders (left) and tuple orders (right)

**Table 3.** Results for MCA experiments

| | $6^6 5^5 3^4 t = 3$ | | | $5^{10} 2^{10} t = 3$ | | |
|---|---|---|---|---|---|---|
| | IPOG | IPOG-F | IPOG-F2 | IPOG | IPOG-F | IPOG-F2 |
| Tie Breaker | | | | | | |
| Alpha-balanced | 470.5 ±13.6 | 441.9 ±17.1 | 533.8 ±18.2 | 316.0 ±5.2 | 305.5 ±5.4 | 353.5 ±7.8 |
| Cyclic | 465.8 ±14.1 | 443.9 ±16.3 | 532.3 ±16.4 | 331.1 ±20.7 | 308.2 ±1.6 | 344.2 ±14.0 |
| Cyclic-next | 464.7 ±12.3 | 444.0 ±18.1 | 529.0 ±23.8 | 316.3 ±4.4 | 307.6 ±3.9 | 352.0 ±11.8 |
| Deterministic | 465.2 ±13.9 | 442.8 ±20.0 | 532.1 ±15.8 | 316.2 ±3.5 | 304.1 ±6.0 | 355.0 ±7.7 |
| Lexicographic | 471.6 ±9.8 | 442.3 ±17.2 | 525.5 ±26.1 | 316.9 ±4.0 | 306.2 ±7.0 | 355.0 ±8.3 |
| Random | 465.2 ±13.2 | 443.5 ±17.3 | 533.8 ±16.9 | 315.4 ±4.1 | 304.6 ±5.5 | 353.8 ±8.5 |
| Value-balanced | 462.9 ±14.2 | 443.7 ±19.3 | 530.8 ±16.1 | 314.6 ±4.3 | 304.8 ±5.7 | 352.5 ±6.6 |
| Tuple Order | | | | | | |
| Alternating | 466.2 ±14.0 | 442.5 ±17.9 | 527.6 ±18.7 | 316.6 ±7.9 | 304.7 ±5.6 | 349.8 ±6.6 |
| Ascending | 466.3 ±14.8 | 444.6 ±19.1 | 542.5 ±20.2 | 317.0 ±7.0 | 305.4 ±5.5 | 359.9 ±10.2 |
| Descending | 464.9 ±10.3 | 443.0 ±14.9 | 527.7 ±8.0 | 316.0 ±7.4 | 305.3 ±5.3 | 349.7 ±6.1 |
| Parameter Order | | | | | | |
| Alternating-large | 465.1 ±5.1 | 440.4 ±2.6 | 522.5 ±8.0 | 317.8 ±2.4 | 307.2 ±2.4 | 348.3 ±3.7 |
| Alternating-small | 467.0 ±3.4 | 442.9 ±3.0 | 534.7 ±7.9 | 317.7 ±2.2 | 308.1 ±2.2 | 353.2 ±5.1 |
| Ascending | 482.8 ±6.3 | 468.8 ±5.4 | 550.8 ±20.7 | 317.6 ±2.6 | 307.5 ±2.5 | 360.4 ±9.3 |
| Descending | 448.4 ±4.1 | 421.4 ±3.6 | 522.4 ±13.9 | 313.2 ±13.8 | 297.7 ±5.4 | 350.7 ±11.4 |

**$5^{10}2^{10}(t = 3)$** The results for this instance are described in Table 3 and a comparison of different tuple orders is visualized in Fig. 4b. The tuple order seems to only make a difference for `IPOG-F2`, where both the `Alternating` and `Descending` order outperform the `Ascending` order. This is also the case in instance $6^6 5^5 3^4$ $(t = 3)$.

## 5 Conclusion

In this paper we have studied the impact of tie-breaking, parameter ordering and tuple enumeration order in the `IPO` family of algorithms. We have compared their effectiveness in terms of their ability to reduce covering array sizes in a large evaluation. In summary, `IPOG-F` overall manages to produce the smallest arrays compared to `IPOG` and `IPOG-F2`. Furthermore, the choice of tie-breaker seems to not matter a great deal when averaging over all instances, but the right choice can have large impact on selected instances. In the case of MCA generation, we measured the largest reduction in array size when ordering columns by decreasing alphabet size, with up to 12% reduction in size compared to the mean.

**Acknowledgments.** The research presented in this paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grants 851205 (Security Protocol Interaction Testing in Practice - SPLIT) and 865248 (SECuring Web technologies with combinatorial Interaction Testing - SecWIT).

Part of this research has also been carried out in the context of the Austrian COMET K1 program which is publicly funded by the Austrian Research Promotion Agency (FFG) and the Vienna Business Agency (WAW).

This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET) and Compute/Calcul Canada.

## References

1. Bryce, R.C., Colbourn, C.J., Cohen, M.B.: A framework of greedy methods for constructing interaction test suites. In: Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, pp. 146–155. ACM (2005)
2. Cheng, C.T.: The test suite generation problem: optimal instances and their implications. Discrete Appl. Math. **155**(15), 1943–1957 (2007)
3. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constructing test suites for interaction testing. In: Proceedings of the 25th International Conference on Software Engineering, ICSE 2003, pp. 38–48. IEEE Computer Society (2003)
4. Duan, F., Lei, Y., Yu, L., Kacker, R.N., Kuhn, D.R.: Improving IPOG's vertical growth based on a graph coloring scheme. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–8 (2015)
5. Forbes, M., Lawrence, J., Lei, Y., Kacker, R.N., Kuhn, D.R.: Refining the in-parameter-order strategy for constructing covering arrays. J. Res. Nat. Inst. Stan. Technol. **113**(5), 287 (2008)
6. Gao, S.W., Lv, J.H., Du, B.L., Colbourn, C.J., Ma, S.L.: Balancing frequencies and fault detection in the in-parameter-order algorithm. J. Comput. Sci. Technol. **30**(5), 957–968 (2015)
7. Gao, S., Lv, J., Du, B., Jiang, Y., Ma, S.: General optimization strategies for refining the in-parameter-order algorithm. In: 2014 14th International Conference on Quality Software (QSIC), pp. 21–26. IEEE (2014)
8. Hartman, A.: Software and hardware testing using combinatorial covering suites. In: Golumbic, M., Hartman, I.A. (eds.) Graph Theory, Combinatorics and Algorithms, Operations Research/Computer Science Interfaces Series, vol. 34, pp. 237–266. Springer, Heidelberg (2005)

9. Hartman, A., Raskin, L.: Problems and algorithms for covering arrays. Discrete Math. **284**(1–3), 149–156 (2004)
10. Kampel, L., Simos, D.E.: Set-based algorithms for combinatorial test set generation. In: Wotawa, F., Nica, M., Kushik, N. (eds.) ICTSS 2016. LNCS, vol. 9976, pp. 231–240. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47443-4_16
11. Kleine, K., Simos, D.E.: An efficient design and implementation of the in-parameter-order algorithm. Math. Comput. Sci. **12**(1), 51–67 (2018)
12. Kuhn, D., Kacker, R., Lei, Y.: Introduction to Combinatorial Testing. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis, London (2013)
13. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG: a general strategy for T-way software testing. In: 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2007, pp. 549–556. IEEE (2007)
14. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. Softw. Test. Verification Reliab. **18**(3), 125–148 (2008)
15. Lei, Y., Tai, K.C.: In-parameter-order: a test generation strategy for pairwise testing. In: Proceedings of Third IEEE International High-Assurance Systems Engineering Symposium, pp. 254–261. IEEE (1998)
16. Younis, M.I., Zamli, K.Z.: MIPOG-an efficient t-way minimization strategy for combinatorial testing. Int. J. Comput. Theory Eng. **3**(3), 388 (2011)
17. Yu, L., Duan, F., Lei, Y., Kacker, R.N., Kuhn, D.R.: Constraint handling in combinatorial test generation using forbidden tuples. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–9 (2015)
18. Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R.N., Kuhn, D.R.: An efficient algorithm for constraint handling in combinatorial test generation. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, pp. 242–251 (2013)