TU WIEN Informatics

Exakte Methoden zur Generierung von Covering Arrays

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Logic and Computation

eingereicht von

Irene Hiess, BSc
Matrikelnummer 01326056

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Dimitris E. Simos
Mitwirkung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl

Wien, 29. Jänner 2024

Irene Hiess	Dimitris E. Simos

TU WIEN Informatics

Exact Methods for Covering Array Generation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Logic and Computation

by

Irene Hiess, BSc
Registration Number 01326056

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Dimitris E. Simos
Assistance: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl

Vienna, 29th January, 2024

Irene Hiess	Dimitris E. Simos

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Irene Hiess, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Jänner 2024

Irene Hiess

Danksagung

Zuallererst möchte ich mich bei meiner Familie und meinem Hund Frankie für ihre Unterstützung bedanken. Außerdem danke ich der MATRIS Forschungsgruppe, besonders Ludwig, Dimitris and Michael. Mein Dank geht an Ludwig für die Einladung in die Gruppe und gute Ratschläge für Paper schreiben und Inhalte. Ich danke Dimitris dafür, dass er mich in die Gruppe aufgenommen hat, für die Betreuung meiner Arbeit und für sein Feedback. Zu guter Letzt danke ich Michael für nützliche Diskussionen und die Ergebnisse seiner SIPO und FIPOG Algorithmen, die er mir für den Vergleich mit dem IPO-MAXSAT Algorithmus zur Verfügung gestellt hat. Ohne sie würde diese Arbeit nicht in dieser Form und Qualität existieren.

Acknowledgements

First of all I want to thank my family and my dog Frankie for their support. Additionally I thank the whole MATRIS research group, especially Ludwig, Dimitris and Michael. My thanks go to Ludwig for inviting me into the group and for advice regarding paper writing and content. I thank Dimitris for allowing me to join the group, for supervising my thesis and for his feedback. Finally, I am thankful for useful discussions with Michael and for him providing experimental results of the SIPO and FIPOG algorithms he developed, for comparison with the IPO-MAXSAT algorithm presented in this thesis. Without them, this thesis would not exist in this form and quality.

Kurzfassung

Covering Arrays sind interessante kombinatorische Objekte, die Bekanntheit erlangt haben durch ihre Anwendbarkeit für kombinatorisches Testen, einem Zweig von Software Testen. Die Generierung von Covering Arrays wird normalerweise als Optimierungsproblem betrachtet, wo es gewünscht ist, ein Covering Array mit einer minimalen Anzahl von Zeilen zu finden. Durch Fortschritte in den vergangenen Jahren wurden exakte Methoden sehr effizient und anpassungsfähig, das heißt sie können bei zahlreichen Problemen angewandt werden. Für Covering Array Generierung mittels exakter Methoden existieren mehrere Kodierungen. Die Anwendung von exakten Methoden für einzelne Teile eines Covering Array Generierungsalgorithmus dagegen wurde nur selten untersucht. Der Zweck dieser Arbeit ist eine Untersuchung der Möglichkeit, Algorithmen zur Erzeugung von Covering Arrays mit exakten Methoden zu verbessern. Im Zuge dieser Arbeit wurden zwei neue Algorithmen entwickelt, die exakte Methoden, genauer gesagt SAT solving, pseudo-Boolean constraint solving und MaxSAT solving, anwenden, um Teilprobleme von Covering Array Generierungsalgorithmen zu lösen. Der vorgestellte ClassifyBalancedCAs Algorithmus kann Covering Arrays klassifizieren, also alle nicht-äquivalenten Covering Arrays einer gegebenen Größe zählen, und diese Menge von Covering Arrays generieren. In einigen Fällen ist der ClassifyBalancedCAs Algorithmus schneller als alle existierenden Algorithmen zur Klassifizierung von Covering Arrays, besonders wenn eine Optimierung namens balancebasiertes Pruning verwendet wird. Zusätzlich wurde der IPO-MAXSAT Algorithmus entwickelt, bei dem MaxSAT verwendet wird, um optimale Lösungen für Teilprobleme der In-Parameter-Order (IPO) Strategie zu finden. Dadurch, dass optimale Lösungen für Teilprobleme verwendet werden, können die Fähigkeiten und Limitierungen der IPO Strategie untersucht werden. Experimente zeigen, dass die Verwendung von optimalen Lösungen für Teilprobleme die Qualität der produzierten Covering Arrays erhöht. Sie zeigen aber auch, dass die generierten Covering Arrays dennoch nicht optimal sind. Die präsentierten Algorithmen demonstrieren, dass es vorteilhaft sein kann, existierende Covering Array Algorithmen mit exakten Methoden zu erweitern.

Abstract

Covering arrays are interesting combinatorial objects that gained much popularity due to their application in combinatorial testing, which is a branch of software testing. The generation of covering arrays is usually treated as an optimization problem, where it is desired to find a covering array with a minimal number of rows. Over the past years exact methods have evolved to be efficient and highly adaptive, meaning they can be applied to many different problems. While several encodings for covering array generation with some exact methods exist, the application of exact methods for only part of a covering array generation algorithm has rarely been studied. The purpose of this thesis is examining the possibility of enhancing covering array generation algorithms with exact methods. In the course of this thesis two new algorithms were developed applying exact methods, in particular SAT solving, pseudo-Boolean constraint solving and MaxSAT solving, to subproblems occurring in algorithms for covering array generation. The proposed algorithm ClassifyBalancedCAs is capable of covering array classification, that is counting all non-equivalent covering arrays of a given size, and exhaustive generation of all such covering arrays. In several cases the ClassifyBalancedCAs algorithm is faster than all existing covering array classification algorithms, especially when an optimization called balance-based pruning is used. Additionally, the IPO-MAXSAT algorithm was developed, where MaxSAT is used to find optimal solutions for subproblems occurring in the greedy In-Parameter-Order (IPO) strategy. Using optimal solutions for subproblems allows to investigate the capabilities and limitations of the IPO strategy. Experiments show that better solutions for subproblems lead to higher quality of the generated covering arrays, however, using optimal solutions for subproblems is not sufficient to generate optimal covering arrays. The presented algorithms show that enhancing covering array generation algorithms with exact methods can be beneficial.

Publications arisen from this Thesis

The following scientific publications have arisen from the work conducted as part of this Master thesis.

1. Irene Hiess, Ludwig Kampel, Michael Wagner and Dimitris E. Simos, "*IPO-MAXSAT: Combining the in-parameter-order strategy for covering array generation with maxsat solving (extended abstract)*", in **SoCS 2022: Proceedings of the International Symposium on Combinatorial Search**, vol. 15, no. 1, pages 288–290, 2022.

2. Irene Hiess, Ludwig Kampel, Michael Wagner and Dimitris E. Simos, "*IPO-MAXSAT: The In-Parameter-Order Strategy combined with MaxSAT solving for Covering Array Generation*", in **2022 24th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)**, pages 71-79, 2022.

Contents

CHAPTER 1

Introduction

A covering array (CA) is a combinatorial array with values from a finite alphabet, where for every selection of a given number of columns all possible tuples have to occur at least once in some row. As a generalization of orthogonal arrays, which are thoroughly described in [HSS99], CAs have been studied both with regards to their combinatorial aspects (see [Col04] for a survey), and for their applications in practice. The requirements on tuple occurrences make CAs a useful tool in software testing. More specifically, combinatorial testing (CT) is a black box testing technique, where input space coverage is achieved by deriving test sets from CAs. When deriving a test set from a CA, every row of the CA is converted to a test and every column of the array corresponds to a parameter of the tested system. Using such a test set ensures due to the way CAs are defined that all interactions of up to a given number of parameters are tested. Empirical studies have shown that most software bugs depend on interactions of only a small number of parameters [HWKK20], meaning with a proper testing oracle these bugs can be found with CT. While exhaustive testing is infeasible in most cases, CT is a viable alternative that allows thorough testing with a smaller number of test cases. For an introduction to CT, see [KKL13].

The generation of CAs is usually considered an optimization problem, where for given parameters defining the properties of a CA the goal is to find a CA with a minimal or small number of rows. Finding a small CA is especially useful in the case of CT where additional rows lead to additional tests and therefore more testing effort. However, the generation of optimal CAs is a challenging problem with complexity yet unknown [KS19]. Many different algorithms for CA generation exist: heuristic algorithms with a focus on execution speed [LKK+07, CDFP97, BC09], metaheuristic algorithms [WKS21, AZL12, TJAG16] with a focus on minimizing the number of rows of the generated CAs as well as exact algorithms [HPSS06, IMTJ18, KMN+20] giving an exact answer to the question whether a CA is optimal. Each of these techniques has different strengths and weaknesses. Heuristics allow fast generation of CAs but the arrays generated with heuristics are usually far from

optimal. Metaheuristics are slower than pure heuristic algorithms and might require more intermediate storage but they often allow to drastically reduce the number of rows when compared to heuristic algorithms. However, although they produce good CAs and in many cases optimal CAs, they do not provide any guarantees or information on solution quality. A solution delivered by a metaheuristic can be optimal or just a local optimum. Exact algorithms, whether they are based on an existing exact method or are a problem-specific algorithm, are able to generate optimal CAs and prove optimality of the generated CA. Although the applicability of exact algorithms in practice is limited due to scalability issues, they are quite interesting from a theoretical point of view. Not only can they prove the existence and non-existence of specific CA instances, they can also be used for exhaustive generation and classification of CAs.

The aim of this thesis is the generation of optimal or close to optimal CAs, which is addressed using exact methods. An advantage of exact methods is the ability to solve problems without developing a problem-specific algorithm. Only an encoding is required, then an existing solver can be used for generating a solution to the problem. The flexibility of exact methods has led to the development of optimized solvers, which are very efficient and can be used instead of problem-specific algorithms that need to be developed and optimized separately for every domain. Exact methods have also been applied to the problem of CA generation. CAs have for example been constructed with the exact methods CSP and SAT [HPSS06, BMTI10]. The problem of CA generation was encoded as constraint set or propositional formula and CAs were derived from the solutions found by the corresponding solver. For several instances, the application of exact methods allowed to draw conclusions on the optimal size of a CA. However, a major downside of encoding the complete problem of CA generation for a given size is scalability. In an exponential search space, iterating over all feasible solutions potentially takes exponential time. Applying exact methods only to subproblems of an algorithm can reduce the search space of the individual calls to the considered exact method solver and therefore might increase efficiency. The application of exact methods for only part of a CA generation algorithm has rarely been studied. In fact, to the best of the authors knowledge, the only work pertaining CA generation making use of an exact method for generating array parts fulfilling the coverage criteria of CAs is [LMZ16]. This thesis attempts to mitigate this shortcoming by exploring the applicability of exact methods to subproblems of existing CA generation algorithms. Two new algorithms based on existing CA generation algorithms, using an exact method (SAT, pseudo-Boolean constraints or MaxSAT) for occurring subproblems, were developed and implemented. The ClassifyBalancedCAs algorithm uses an exact method for efficiently finding all feasible solutions in an algorithm where exact solving is required, while the IPO-MAXSAT algorithm replaces a heuristic solution in a CA generation strategy with an optimal solution generated via an exact method, allowing conclusions on the abilities of the considered strategy regarding solution quality under optimal circumstances.

The first proposed algorithm, ClassifyBalancedCAs, is a CA classification algorithm making use of column extension and backtracking. Algorithms of this type have been

previously proposed e.g. in [IMTJ18] and [KMN+20]. The number of CAs of a given size is counted by exhaustively generating those CAs. In addition to using SAT or pseudo-Boolean constraint solving for exhaustive generation of solutions to a subproblem (generation of columns), ClassifyBalancedCAs is the first algorithm capable of classifying a variant of CAs called *balanced covering arrays*. The ClassifyBalancedCAs algorithm is for several instances the fastest currently existing CA classification algorithm. Using a technique called balance-based pruning, the runtime can be reduced further for some special cases.

The second algorithm developed in the course of this thesis is called IPO-MAXSAT. It combines the In-Parameter-Order (IPO) strategy described in [LKK+07] with MaxSAT solving to receive optimal solutions for the occurring subproblems called *horizontal extension* and *vertical extension*. With a greedy algorithm for the occurring subproblems, IPO is an excellent strategy for fast generation of CAs that is also implemented in tools that are used in practice for CT, see [WKS+20] and [YLKK13]. Using MaxSAT for the subproblems leads to the generation of CAs with fewer rows, and an increased runtime when compared to other IPO algorithms. While the IPO-MAXSAT algorithm will not be as useful in practice due to the drastically increased runtime, the usage of optimal solutions for subproblems allows to experiment with different kinds and degrees of solution optimality and investigate the performance of the IPO strategy with these optimal solutions. Experiments show that higher quality of subproblem solutions leads to higher quality of the generated CAs. However, in general no optimal CAs are generated with IPO-MAXSAT.

The remainder of this thesis is structured as follows: In Chapter 2 the required concepts are introduced. Definitions for CAs and related notions are given, together with explanations of used concepts, in particular symmetry breaking and exact methods. In Chapter 3 existing work in the area of CA generation is discussed, with a focus on row and column extension algorithms, i.e. algorithms starting with an empty or partial array and extending it with additional rows or columns until a CA of the desired size is constructed. Additionally, CA generation algorithms based on exact methods are summarized. Afterwards, in Chapters 4 and 5 the contribution of this thesis is given. In Chapter 4 the ClassifyBalancedCAs algorithm is presented, which is an exact column extension algorithm capable of CA classification. Additionally, with an experimental evaluation the ClassifyBalancedCAs algorithm is compared to existing CA classification algorithms. In Chapter 5 the algorithm IPO-MAXSAT is described, which enhances the existing heuristic algorithm IPO with the exact method MaxSAT. While the developed algorithm is not competitive to IPO with regard to runtime, replacing an intermediate step with an exact method allows to draw conclusions on the capabilities and limitations of the IPO algorithm, as discussed in Section 5.4. Finally, Chapter 6 concludes the thesis with a summary of the developed algorithms and the lessons learned, together with possible future work.

CHAPTER 2

Preliminaries

In this chapter, preliminaries and notations required in the remainder of this thesis are introduced. First, since the focus of this thesis is on CA generation, definitions for CAs and related concepts will be given. Afterwards, the exact methods applied in this thesis will be briefly introduced.

2.1 Covering array definitions

A CA is defined analogously to [CD07]:

Definition 1. *A* covering array (CA) *denoted as* $CA_\lambda(N; t, k, v)$ *is an* $N \times k$ *array over an arbitrary alphabet* Σ_v *with* v *symbols, where every* t*-tuple over* Σ_v *occurs at least* λ *times as a row in every selection of* t *columns. The parameter* t *is called the* strength *of the covering array,* k *is the number of columns of the array, and* v *is the number of symbols in the array. When* $\lambda = 1$*, the subscript can be omitted in the notation. The number of rows* N*, also called* size*, can be omitted when inessential in the context.*

Without loss of generality, the alphabet $\Sigma_v = \{0, \ldots, v-1\}$ will be used throughout this thesis. The parameter v is also called *alphabet size*, since it defines the size of the used alphabet. To simplify discussion about concepts of CAs, several additional existing notions commonly used in the literature are introduced here.

Definition 2. *A selection of* t *columns with corresponding values is called* t-way interaction*. It is denoted as a set* $\{(c_1, u_1), \ldots, (c_t, u_t)\}$*, where* $1 \leq c_1 < \ldots < c_t \leq k$ *and* $u_i \in \Sigma_v$ *for* $i = 1, \ldots, t$*.*

In this thesis, the following notation is used to describe sets of t-way interactions:

$$A = \begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 2 & 2 \\ 0 & 2 & 2 & 2 & 2 & 0 & 1 \\ 1 & 0 & 1 & 1 & 2 & 0 & 2 \\ 1 & 1 & 0 & 2 & 2 & 2 & 0 \\ 1 & 1 & 2 & 0 & 0 & 1 & 1 \\ 1 & 2 & 1 & 2 & 1 & 1 & 0 \\ 2 & 0 & 2 & 2 & 0 & 2 & 2 \\ 2 & 1 & 2 & 1 & 1 & 0 & 0 \\ 2 & 2 & 0 & 0 & 2 & 1 & 2 \\ 2 & 2 & 1 & 1 & 0 & 2 & 1 \end{array}$$

Figure 2.1: A covering array $\mathsf{CA}(12; 2, 7, 3)$.

Definition 3. *The set of all t-way interactions on columns $1, \ldots, k$ is denoted as $\mathbb{T}_{v,k,t}$. This is the set*

$$\mathbb{T}_{v,k,t} = \{\{(c_1, u_1), \ldots, (c_t, u_t)\} | 1 \leq c_1 < \ldots < c_t \leq k, u_i \in \Sigma_v \text{ for } i = 1, \ldots, t\}\}$$

The size of the set $\mathbb{T}_{v,k,t}$ is $v^t\binom{k}{t}$ because there are $\binom{k}{t}$ possible selections of t out of k columns and v^t possible selections of values for every column selection.

Definition 4. *A row $\vec{r} = (a_1, \ldots, a_k)$ of an $N \times k$ array A is said to* cover *a t-way interaction $\tau = \{(c_1, u_1), \ldots, (c_t, u_t)\}$ if the values in the row $\vec{r}$ match the values of τ in the respective columns, i.e. $a_{c_i} = u_i$ for $i = 1, \ldots, t$. An array A is said to* cover *a t-way interaction τ if A has a row that covers τ. In this thesis, for an array A the index set of the rows that cover an interaction τ is denoted as $A\restriction_\tau$.*

During the construction of a CA it might be that not all array entries are assigned at the same time. In this case, some entries of an array A have a value assigned, while other array entries have not been assigned a value. Such array entries without value are called unassigned *or* star-values*. Then the definition of $A\restriction_\tau$ can be extended to contain the indices of all rows that* can cover *τ. These are the indices of the rows that match τ on all positions that do not contain a star-value.*

Using these definitions, a CA is an array where all t-way interactions are covered. An example of a $\mathsf{CA}(12; 2, 7, 3)$ is given in Figure 2.1. In every selection of $t = 2$ columns, every tuple over the alphabet $\Sigma_3 = \{0, 1, 2\}$ occurs at least once as a row. For example the 2-way interaction $\{(2, 2), (4, 2)\}$ is covered in the fourth and eighth row, therefore the set of row indices where $\{(2, 2), (4, 2)\}$ is covered is the set $A\restriction_{\{(2,2),(4,2)\}} = \{4, 8\}$

Existence of a $\mathsf{CA}(t, k, v)$ is trivial, since the $v^k \times k$ array containing all v^k tuples over the alphabet $\Sigma_v = \{0, \ldots, v-1\}$, those are all elements of $\{0, \ldots, v-1\}^k$, is also a $\mathsf{CA}(t, k, v)$.

However, finding a $\mathsf{CA}(t,k,v)$ with a small or even minimal number of rows is a difficult problem that is actively researched and of practical interest in CT.

Definition 5. *The* covering array number (CAN) *denoted as* $\mathsf{CAN}(t,k,v)$ *for given* (t,k,v) *is the minimum number of rows* N *where a* $\mathsf{CA}(N;t,k,v)$ *exists. The CAN problem is the problem of finding* $CAN(t,k,v)$ *for given parameters* (t,k,v)*. A* $CA(N;t,k,v)$ *is called* optimal *if* $N = CAN(t,k,v)$*, i.e. if it has a minimum number of rows.*

An online table of the currently best known upper bounds for CAN, together with the source of the bound, is maintained at [Col].

In addition to minimizing the number of rows N for given (t,k,v), it is also possible to maximize the number of columns k for given $(N;t,v)$. Therefore, a dual bound to CAN is the maximal number of columns k of a $\mathsf{CA}(N;t,k,v)$ for given parameters N, t and v. This bound is denoted as $\mathsf{CAK}(N;t,v)$ and can be defined as follows:

Definition 6. $\mathsf{CAK}(N;t,v)$ *is defined as the maximal number of columns* k *where a* $\mathsf{CA}(N;t,k,v)$ *exists, i.e.*

$$\mathsf{CAK}(N;t,v) = \max\{k : \exists \mathsf{CA}(N;t,k,v)\}$$

CAK and CAN are simply two different viewpoints. As the following equations show, CAK values can be determined from CAN values and vice versa.

$$\mathsf{CAK}(N;t,v) = \max\{k : \mathsf{CAN}(t,k,v) \leq N\}$$
$$\mathsf{CAN}(t,k,v) = \min\{N : \mathsf{CAK}(N;t,v) \geq k\}$$

In the literature mostly the CAN bound is used. This is also related to the application of CT, where a function with k parameters is tested using a CA with k columns, i.e. the number of columns is an input parameter to CA generation. Since the number of columns is often fixed, it is natural to use the CAN bound to discuss the limits of CA generation for a specific instance.

Two important generalizations of CAs that are also widely used in the practice of CT are the *mixed covering array (MCA)* and the *constrained CA*. An MCA allows a more flexible specification of the alphabet size than CAs. While the alphabet size v of a CA is the same for every column, when defining an MCA a list of alphabet sizes $v_1, \ldots, v_k$ is given, where for every column $c \in \{1, \ldots, k\}$ the alphabet size v_c of column c is defined separately. In the context of CT, this is required when testing systems where not every parameter has the same number of possible values. Constrained CAs on the other hand allow to impose constraints on the rows occurring in the array, i.e. the occurrence of specific value combinations might be forbidden. Such constraints can be specified as lists of forbidden parameter interactions (as m-way interactions for some $m \leq k$) or as a formula. Since constraints might prevent some t-way interactions from occurring in the array at all, the definition of coverage is repaired to only enforce coverage of t-way interactions not

forbidden by the given constraints. For CT, constrained CAs allow to exclude forbidden input parameter combinations from the generated tests. Constrained MCAs are also often referred to as constrained CAs. In this thesis, MCAs and constrained CAs are not considered.

CAs have the property that every t-way interaction appears *at least* once in the array. A related notion is the packing array (PA), defined in [SM02, SM04]. This is an array with a definition similar to CAs, with the difference that every t-way interaction appears *at most* once in the array. A formal definition looks as follows:

Definition 7. *A* packing array (PA) *denoted as* $PA_\lambda(N; t, k, v)$ *is an* $N \times k$ *array over an arbitrary alphabet* Σ_v *with* v *symbols where every* t*-tuple over* Σ_v *occurs at most* λ *times as a row in every selection of* t *columns. The parameter* t *is also called the* strength *of the packing array. When* $\lambda = 1$*, the subscript can be omitted in the notation. The number of rows* N*, also called* size*, can be omitted when inessential in the context.*

While for CAs the minimum number of rows (CAN) for given parameters t, k, v is of interest, for PAs there is a maximum number of rows where a PA for given parameters t, k, v exists. PAs exists for every parameter combination t, k, v, since an array with zero rows is a PA. A trivial upper bound on the number of rows of a $\mathsf{PA}_\lambda(N; t, k, v)$ is $N \leq \lambda v^t$ because each of the v^t possible tuples in a selection of t columns can appear at most λ times. However, the exact upper bound is more interesting:

Definition 8. *The* packing array number (PAN) *denoted as* $\mathsf{PAN}(t, k, v)$ *for given* (t, k, v) *is the maximum number of rows* N *where a* $\mathsf{PA}(N; t, k, v)$ *exists.*

Intersections of CAs and PAs have been described and examined in [KHKS23] under the name *balanced CAs*. This variant of CAs is the focus of the next subsection.

2.1.1 Balance for covering arrays

The notion of *balanced CAs* as defined in [KHKS23] is used.

Definition 9. *For vectors* $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_t), \boldsymbol{y} = (y_1, \ldots, y_t) \in \mathbb{N}^t$ *a* $(\boldsymbol{\lambda}, \boldsymbol{y})$-balanced covering array *denoted as* $\mathsf{CA}^{\boldsymbol{y}}_{\boldsymbol{\lambda}}(N; t, k, v)$ *is an* $N \times k$ *array over an arbitrary alphabet* Σ_v *with* v *symbols, where for* $1 \leq j \leq t$ *every* j*-tuple over* Σ_v *occurs at least* λ_j *and at most* y_j *times as a row in every selection of* j *columns.*

The set of all $(\boldsymbol{\lambda}, \boldsymbol{y})$*-balanced CAs for given parameters* N, t, k *and* v *is denoted as* $\mathbb{CA}^{\boldsymbol{y}}_{\boldsymbol{\lambda}}(N; t, k, v)$.

The definition of balanced CAs allows to impose restrictions on the number of occurrences of j-way interactions of size $j \leq t$.

It is easy to see that when $\boldsymbol{\lambda} \geq \boldsymbol{\lambda}'$ and $\boldsymbol{y} \leq \boldsymbol{y}'$, then $\mathbb{CA}^{\boldsymbol{y}}_{\boldsymbol{\lambda}}(N; t, k, v) \subseteq \mathbb{CA}^{\boldsymbol{y}'}_{\boldsymbol{\lambda}'}(N; t, k, v)$. This follows from $\boldsymbol{\lambda}$ imposing an *at least* bound on the number of tuple occurrences,

while $\boldsymbol{y}$ imposes an *at most* bound. With a partial ordering on balance vectors defined as $(\boldsymbol{\lambda}, \boldsymbol{y}) \preccurlyeq (\boldsymbol{\lambda}', \boldsymbol{y}')$ if and only if $\boldsymbol{\lambda} \geq \boldsymbol{\lambda}'$ and $\boldsymbol{y} \leq \boldsymbol{y}'$, for $\boldsymbol{\lambda}, \boldsymbol{\lambda}', \boldsymbol{y}, \boldsymbol{y}' \in \mathbb{N}^t$, balance vectors $(\boldsymbol{\lambda}, \boldsymbol{y})$ are called *stronger* than balance vectors $(\boldsymbol{\lambda}', \boldsymbol{y}')$, if $(\boldsymbol{\lambda}, \boldsymbol{y}) \preccurlyeq (\boldsymbol{\lambda}', \boldsymbol{y}')$.

With stronger balance vectors, the differences between the number of occurrences of different tuples are smaller and the array is considered to be more balanced.

In [KHKS23] several bounds are described for *balance vectors* $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_t)$ and $\boldsymbol{y} = (y_1, \ldots, y_t)$ where no solutions exist, or where stronger balance vectors $(\boldsymbol{\lambda}', \boldsymbol{y}')$ exist with $\mathbb{CA}^{\boldsymbol{y}}_{\boldsymbol{\lambda}}(N; t, k, v) = \mathbb{CA}^{\boldsymbol{y}'}_{\boldsymbol{\lambda}'}(N; t, k, v)$.

In particular, when the inequalities

$$\lambda_i \leq \left\lfloor \frac{N}{v^i} \right\rfloor, \text{ for } 1 \leq i \leq t \text{ or} \tag{2.1}$$

$$y_i \geq \left\lceil \frac{N}{v^i} \right\rceil, \text{ for } 1 \leq i \leq t \tag{2.2}$$

are not satisfied, no balanced CA $\mathsf{CA}^{\boldsymbol{y}}_{\boldsymbol{\lambda}}(N; t, k, v)$ can exist. When on the other hand any of the following inequalities are violated, it is possible to choose *stronger* balance vectors, without reducing the search space and without losing solutions:

$$\lambda_i \geq v \cdot \lambda_{i+1}, \text{ for } 1 \leq i < t, \tag{2.3}$$

$$y_i \leq v \cdot y_{i+1}, \text{ for } 1 \leq i < t, \tag{2.4}$$

$$y_{i+1} \leq y_i - (v-1)\lambda_{i+1}, \text{ for } 1 \leq i < t, \tag{2.5}$$

$$\lambda_{i+1} \geq \lambda_i - (v-1)y_{i+1}, \text{ for } 1 \leq i < t, \tag{2.6}$$

$$\lambda_i \geq N - (v^i - 1)y_i, \text{ for } 1 \leq i \leq t, \tag{2.7}$$

$$y_i \leq N - (v^i - 1)\lambda_i, \text{ for } 1 \leq i \leq t. \tag{2.8}$$

Remark 1. As is stated in [KHKS23, Remark 1], when the inequalities above are not violated but equality holds, then some balance constraints are implied by others:

- Equation (2.3): If $\lambda_i = v \cdot \lambda_{i+1}$ and every $(i+1)$-way interaction is enforced to occur at least λ_{i+1} times, then it is guaranteed that every i-way interaction occurs at least λ_i times.
- Equation (2.4): If $y_i = v \cdot y_{i+1}$ and every $(i+1)$-way interaction is enforced to occur at most y_{i+1} times, then the at most y_i constraints are implied and can be omitted.
- Equation (2.5): If $\lambda_i = N - (v^i - 1)y_i$ and every i-way interaction occurs at most y_i times, then it is implied that every i-way interaction occurs at least λ_i times.
- Equation (2.6): If $y_i = N - (v^i - 1)\lambda_i$ and every i-way interaction occurs at least λ_i times, then all i-way interactions also occur at most y_i times.

$$A = \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{matrix}, \qquad B = \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}$$

Figure 2.2: Two equivalent covering arrays $\mathsf{CA}(5; 2, 4, 2)$.

- Equation (2.7): If $\boldsymbol{y}_{i+1} = \boldsymbol{y}_i - (v-1)\lambda_{i+1}$ and there are constraints ensuring all $(i+1)$-way interactions appear at least λ_{i+1} times and all i-way interactions appear at most $\boldsymbol{y}_i$ times, then it is not necessary to enforce that all $(i+1)$-way interactions appear at most $\boldsymbol{y}_{i+1}$ times, since this is implied by the other constraints.
- Equation (2.8): Similarly, if $\lambda_{i+1} = \lambda_i - (v-1)\boldsymbol{y}_{i+1}$, all i-way interactions occur at most λ_i times and all $(i+1)$-way interactions occur at most $\boldsymbol{y}_{i+1}$ times, then it follows that all $(i+1)$-way interactions appear at least λ_{i+1} times.

Additionally, when searching for $(\boldsymbol{\lambda}, \boldsymbol{y})$-balanced CAs with at least k columns the following bounds for $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_t)$ and $\boldsymbol{y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_t)$ vectors are given:

$$\lambda_i \geq \mathsf{CAN}(t-i, k-i, v), \tag{2.9}$$
$$\boldsymbol{y}_i \leq \mathsf{PAN}(t-i, k-i, v). \tag{2.10}$$

It was shown in [KHKS23] that every $\mathsf{CA}^{\boldsymbol{y}}_{\boldsymbol{\lambda}}(N; t, k, v)$ fulfills the inequalities (2.9) and 2.10, therefore those inequalities can potentially be used for search space reduction without omitting any of the desired solutions, i.e. CAs with N rows of strength t with alphabet size v and at least k columns. In this thesis, search space reduction using this method is referred to as *balance-based pruning*.

2.1.2 Covering array equivalences and symmetry breaking

Certain actions on CAs are guaranteed to again result in a CA. Such actions are row permutations, column permutations and symbol permutations within a column, see e.g. [CKRSP10]. If a CA B can be produced from a CA A via such operations, then A and B are called *equivalent*[1]. In [KHKS23] it was argued that for balanced CAs the same equivalences hold as for CAs. An example of two equivalent CAs is given in Figure 2.2, where B can be constructed from A by permuting the symbols in the first and fourth column with the symbol permutation $\sigma(0) = 1$ and $\sigma(1) = 0$. Additionally, the first two rows are swapped with the last three rows. This amounts to sorting the rows lexicographically after applying the symbol permutations.

The maximal size of the equivalence class consisting of row permutations, column permutations and symbol permutations within a column on an array of size $N \times k$

[1]The term *isomorphic* is also used in the literature.

over an alphabet of size v is $N!k!(v!)^k$. This is because there are $N!$ possible row permutations, $k!$ possible column permutations and for each of the k columns there are $v!$ possible symbol permutations. Since row, column and symbol permutations can be arbitrarily combined, this yields a maximal size of $N!k!(v!)^k$. While the equivalence class is not always of maximal size (because some combinations of equivalence actions yield automorphisms), the often huge number of equivalent solutions can cause immense overhead for a search algorithm. Therefore, for an algorithm performing exact search it is of great importance to apply *symmetry breaking* to reduce the search space, meaning the number of considered equivalent solutions is reduced via additional constraints or by adapting the used algorithm to skip some equivalent solutions.

Incomplete symmetry breaking removes some equivalent solutions but not necessarily all. However, it is usually computationally cheaper than complete symmetry breaking, where only one solution of each equivalence class is accepted.

Methods to break *some* symmetries

For the symmetries known for CAs it is possible to individually break each kind of symmetry. An early work where symmetry breaking is applied to CAs is [HPSS06]. Row and column symmetries are broken by imposing a lexicographic ordering on rows, as well as columns. It is easy to see that with such constraints neither row permutations alone, nor column permutations alone, will generate any new CA that is permitted by the constraints. Additionally, permutations of symbols within a column are broken by setting the first symbol of every row to '0', and by enforcing that the remaining symbols are ordered increasingly by their number of occurrences in the column. For example, a '1' occurring 10 times in a column and a '2' occurring 5 times in the same column would not be allowed with this constraint. With the mentioned constraints, all symmetries arising from column permutations, all symmetries arising from row permutations and all symmetries arising from symbol permutations within a column are broken. However, as stated in [FFH+02] for general matrix models where row and column permutations lead to equivalent solutions, breaking individual symmetries is not sufficient to break all symmetries. Breaking individual symmetries is also not sufficient for CAs, since symmetries arising from combinations of different kinds of equivalence actions still exist. The example given in Figure 2.2 displays two equivalent CAs, both with lexicographically ordered rows and columns and with the first row containing only the symbol '0'.

Enforcing a lexicographic ordering on rows and columns is a straightforward way for symmetry breaking in matrix models with row and column permutations as symmetry actions that has been applied to CAs [HPSS06, IMTJ18, YZ06] and also investigated for general matrix models with this kind of symmetry [FFH+02, KNW10].

Symbol permutations within a column are a less common symmetry than row and column permutations in other problems than CA generation and have therefore not been researched that extensively. As mentioned above, one option for breaking the symmetry of symbol permutations within a column is an ordering of the symbols in every column

with regard to their occurrence count in that column. This method has been applied in [HPSS06], as well as in [BMTI10]. In [YZ08] a different approach is used for breaking symmetries arising from symbol permutations: The authors presented a search-based algorithm where the values are assigned one-by-one. This allows to use the least number heuristic (LNH), where for the assignment of a value $u > 0$ in a column c, the value $u - 1$ has to occur already in column c. In [TJIM16] a weaker kind of symbol symmetry breaking is used. For the first v rows of every column a symbol u can only be assigned to row i if $u < i$. For example, the first row can only be assigned the symbol '0'. The second row is only allowed to be assigned the symbols '1' or '0'.

An important concern when combining different symmetry breaking methods is compatibility, as was also mentioned in [HPSS06]. For example, assume a lexicographically increasing row ordering is combined with a symbol symmetry breaking method of setting the first row of every column to '1'. While both symmetry breaking methods would be fine individually, when combined they forbid all solutions. For an array with $(1, \ldots, 1)$ as first row, no lexicographically larger row containing the symbol '0' in the first column exists, and therefore the array cannot be a CA.

Methods to break *all* symmetries

Although the above methods allow to drastically reduce the search space, in some cases it is desired to accept only one solution of each equivalence class. This requires a criterion for deciding which unique object of each equivalence class is accepted. For example, an ordering can be defined on the considered objects and the accepted objects can be restricted to the minimal element of each equivalence class with regard to the defined ordering. Accepting only the lexicographic leader of each equivalence class is also called the lex-leader method. Constraints enforcing lex-leadership of the considered objects in their equivalence class can then be used to break all symmetries, see e.g. [CGLR96]. In the case of CAs, the array can be linearized. A linearization consists of an arbitrary ordering of the array entries, such that the array can be written as a vector. The linearized arrays, i.e. the vectors resulting from the linearization, can then be lexicographically ordered and, for each equivalence class, the lexicographically minimal array with regard to the used linearization is the desired solution. A schematic of the column-wise linearization of a 4×3 array looks as follows:

$$\begin{pmatrix} x_1 & x_5 & x_9 \\ x_2 & x_6 & x_{10} \\ x_3 & x_7 & x_{11} \\ x_4 & x_8 & x_{12} \end{pmatrix} \quad \mapsto \quad (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}).$$

In this thesis, only a column-wise linearization is used. Other well-known linearizations are a row-wise linearization and snake lex, described in [GMRD09].

When one constraint is created per array reachable via any combination of row, column or symbol permutations within a column, then all symmetries are broken. For example,

the array

$$\begin{pmatrix} x_2 & x_6 & x_{10} \\ x_1 & x_5 & x_9 \\ x_3 & x_7 & x_{11} \\ x_4 & x_8 & x_{12} \end{pmatrix}$$

resulting from swapping the first two columns would be excluded by the constraint

$$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}) \leq_{lex} (x_2, x_1, x_3, x_4, x_6, x_5, x_7, x_8, x_{10}, x_9, x_{11}, x_{12})$$

to break this symmetry. Lex-leader symmetry breaking via one constraint per equivalent solution uses $N!k!(v!)^k - 1$ constraints in the case of CAs.

A computational method with lower complexity to break symmetries induced by row and column permutations is given in [KNW10], and the method is generalized in [YH11] for other kinds of symmetries. The method is based on a lexicographic ordering of the considered arrays as well. A *feasibility check* is used, where partial arrays constructed during a search are already checked for minimality. If a partial array cannot lead to a lexicographically minimal array, then it is discarded and its extensions are not explored by the search algorithm. Unlike the method described above, instead of considering all $N!k!(v!)^k - 1$ arrays that are reachable via symmetry actions, only the arrays for all but one kind of symmetry are generated. For example, all $k!(v!)^k - 1$ arrays reachable from a given array A via any combinations of column permutations and permutations of symbols within columns are generated. To check whether an equivalent array that is lexicographically smaller than A exists, the rows of the generated $k!(v!)^k - 1$ arrays are sorted and compared with A. If any such array is lexicographically smaller than A, then it is known that A is not a lex-leader. If, on the other hand, no lexicographically smaller array than A is found, then A is indeed lexicographically minimal. This method has lower complexity than exhaustively generating all arrays in the equivalence class of A. In the case of CAs, the enumeration of all equivalent solutions would require generation and comparison of all $k!(v!)^k N!$ equivalent arrays. However, the complexity of the described method is only $O(k!(v!)^k kN \log N)$ if a sorting algorithm with complexity $O(kN \log N)$ is used for sorting the rows. The factor k in the complexity of the sorting algorithm is the complexity of comparing two rows of an array with k columns.

To reduce the number of feasibility checks required, computationally cheap symmetry breaking methods only removing some symmetries can be combined with the powerful but expensive feasibility check, as long as consistency of the used methods is maintained.

2.2 Exact methods

Exact methods like SAT solving have gained much popularity in recent decades. With exact methods, it is not necessary to develop a custom algorithm and implement it to

solve a problem. Instead, an encoding can be used. In the case of SAT solving this refers to an encoding of the considered problem as a propositional formula in conjunctive normal form (CNF). Once an encoding is established, a solver for an exact method can be used to find a solution to the encoded problem, and after back-translation also a solution to the original problem. Due to the flexibility and applicability of exact methods, the development of solvers is an active research field where highly optimized solvers exist for different formalisms. An example displaying the advances of the field is the well-known SAT competition[2], which takes place every year and encourages both developers of SAT solvers to develop and submit more advanced solvers as well as users of SAT solving to submit hard benchmarks for a more challenging competition.

In this thesis the applicability of exact methods to CA generation is examined, more specifically the applicability to subproblems of CA generation algorithms. The exact methods applied in this thesis are SAT solving, (linear) pseudo-Boolean constraint solving and MaxSAT solving. A more thorough introduction to these topics can be found in [BHvMW21], here only the most important notions and notations used in this thesis are explained.

2.2.1 Notions related to SAT solving

SAT solvers usually accept as input a propositional formula in conjunctive normal form (CNF), which is a specific format of a formula. For every propositional formula there exists an equivalent formula in CNF, therefore only formulas in CNF are considered here. Such a formula consists of the following parts:

- *Propositional variables*: Propositional variables are denoted as single letters in this thesis, possibly with an index, e.g. $x, x_i, h_j, \ldots$
- *Literals*: Propositional variables and propositional variables negated with the symbol '$\neg$', e.g. $\neg x$, are literals.
- *Clauses*: A disjunction of a finite number of literals is a clause. A disjunction of literals is depicted using the symbols '$\vee$' or '$\bigvee$'. Examples for clauses are $x_3 \vee \neg r_j$ and $\bigvee_{i=0}^{3} x_i$, where the latter is a short notation for $x_0 \vee x_1 \vee x_2 \vee x_3$. Clauses can also be empty or consist of only one literal, e.g. x_i.

A *formula in CNF* is then a conjunction of clauses, where a conjunction is either depicted with the conjunction symbols '$\wedge$' or '$\bigwedge$', or as a set of clauses. Examples for propositional

[2] See http://www.satcompetition.org/ for more information about the SAT competition.

formulas in CNF are

$$C_1 \wedge C_2 \wedge C_3,$$
$$\bigwedge_{i=1}^{3} C_i,$$
$$\{C_1, C_2, C_3\} \text{ and}$$
$$\{C_i | i = 1, \ldots, 3\},$$

where C_1, C_2 and C_3 are clauses. All of these examples denote the same formula, using different notations.

The above defines the syntax of formulas in propositional logic, as used in this thesis. In the next step, the semantic of such formulas is described. A vital part of defining the semantics of formulas are truth assignments: A *truth assignment* for a formula in propositional logic (in CNF) is a function assigning a truth value to every propositional variable contained in the considered formula. A truth assignment is then with simple rules extended to an *evaluation function* assigning a truth value to the formula. Let $\mathcal{A} : \mathcal{V} \to \{0, 1\}$ be a truth assignment assigning a value to every propositional variable, where $\mathcal{V}$ denotes the set of propositional variables. Then an evaluation function $\mathcal{I}$ is defined as follows:

- $\mathcal{I}(P) = \mathcal{A}(P)$ for propositional variables P.
- $\mathcal{I}(\neg P) = 1 - \mathcal{A}(P)$ for propositional variables P.
- $\mathcal{I}(C) = 1$ for a clause C if C contains a literal ℓ with $\mathcal{I}(\ell) = 1$, otherwise $\mathcal{I}(C) = 0$.
- $\mathcal{I}(F) = 1$ for a formula F in CNF if for every clause C in F $\mathcal{I}(C) = 1$, otherwise $\mathcal{I}(F) = 0$.

Note that the above definition only defines truth values for formulas in CNF. While the given evaluation function can be easily be extended for arbitrary formulas, such an extension is not required for the contents of this thesis. A truth value of 1 is also called *true*, while a truth value of 0 is called *false*. A formula F is called *satisfiable* if there exists a truth assignment such that F evaluates to 1 or *true* under this assignment. An assignment where F evaluates to *true* is also called a *satisfying assignment*, a *model* or a *solution* to F. A *SAT solver* is a tool determining for a given propositional formula in CNF whether it is satisfiable. The output of a SAT solver is either a satisfying assignment for the given formula or that no such assignment exists, meaning the formula is *unsatisfiable*.

2.2.2 A brief introduction to linear pseudo-Boolean constraints

Linear pseudo-Boolean constraints are of the form

$$\sum_{i=1}^{n} c_i L_i \geq m,$$
$$\sum_{i=1}^{n} c_i L_i \leq m, \text{ or}$$
$$\sum_{i=1}^{n} c_i L_i = m,$$

where for $i = 1, \ldots, n$ the L_i are literals as defined above for CNF formulas in propositional logic, i.e. propositional variables or negated propositional variables, and the c_i and m are integers. Non-linear pseudo-Boolean constraints exist, but in this work only linear pseudo-Boolean constraints are considered.

A truth assignment $\mathcal{A} : \mathcal{V} \to \{0, 1\}$ satisfies a pseudo-Boolean constraint $\sum_{i=1}^{n} c_i L_i \geq m$ if the inequality $\sum_{i=1}^{n} c_i \mathcal{I}(L_i) \geq m$ is satisfied, where $\mathcal{I}(P) = \mathcal{A}(P)$ and $\mathcal{I}(\neg P) = 1 - \mathcal{A}(P)$ for propositional variables P. An evaluation of the other types of pseudo-Boolean constraints (with $\leq$ and $=$) is defined analogously.

A set of pseudo-Boolean constraints S is satisfied by a truth assignment $\mathcal{A}$ if every pseudo-Boolean constraint in S is satisfied by $\mathcal{A}$. Similar to a SAT solver, a pseudo-Boolean constraint solver takes as input a set of pseudo-Boolean constraints S and outputs a satisfying truth assignment for S if one exists and otherwise that no satisfying assignment exists.

Every SAT problem can easily be specified as a pseudo-Boolean constraint formulation. A propositional formula in CNF $F = \{C_i | i = 1, \ldots, n\}$ consisting of n clauses of the form $C_i = \bigvee_{j=0}^{n_i} L_{i,j}$ for $i = 1, \ldots, n$ is satisfied by a truth assignment $\mathcal{A}$ iff the set of pseudo-Boolean constraints $\{\sum_{j=1}^{n_i} L_{i,j} \geq 1 | i = 1, \ldots, n\}$ is satisfied by $\mathcal{A}$.

2.2.3 MaxSAT

MaxSAT is a problem related to the SAT problem, however, instead of being a decision problem MaxSAT is an optimization problem. Propositional variables, literals and clauses are defined as for propositional logic formulas in CNF. In this thesis a type of MaxSAT called *weighted partial* MaxSAT is considered. A weighted partial MaxSAT formula consists of two types of clauses: hard clauses and soft clauses. Hard clauses are normal clauses, as required for a propositional formula in CNF. Soft clauses are clauses associated with a weight $w \in \mathbb{N}$. In this thesis, soft clauses are denoted as (w, C), where $w \in \mathbb{N}$ is called the weight of the clause and C is a clause.

A MaxSAT solver given a MaxSAT formula F consisting of hard and soft clauses outputs a truth assignment $\mathcal{A}$, such that all hard clauses evaluate to *true* under $\mathcal{A}$ and the weight of violated soft clauses, i.e. soft clauses that evaluate to *false* under $\mathcal{A}$, is minimal. If

the formula consisting only of the hard clauses of F is not satisfiable, then no such truth assignment exists. In this case the MaxSAT solver returns as answer that F is unsatisfiable. A truth assignment satisfying all hard clauses and minimizing the weight of violated soft clauses is called *optimal solution* to the MaxSAT formula F. When optimality is clear from the context, optimal might be omitted and the optimal solution is simply called a *solution* to F.

CHAPTER 3

Existing Approaches and Algorithms for CA Generation

In addition to CAs being interesting mathematical structures, the application of CAs in combinatorial testing has led to the development of various CA generation algorithms, see [TJIMAG19] for a survey. In this chapter, a short introduction to the most related CA generation techniques shall be given.

3.1 Extension algorithms

Algorithms that step-by-step extend an empty or small array to a CA of the desired size are categorized as extension algorithms in this thesis. In most cases, such extension steps consist of the addition of either rows or columns.

3.1.1 Row extension algorithms

As stated in Definition 1, a CA is a two-dimensional array, where every t-way interaction of the columns occurs in some row of the array. Since the defining condition is based on an *existence* property, it is always possible to build a CA by appending sufficiently many different rows to an (initially possibly empty) array. Row extension algorithms are based on this property. They add rows to the processed array, until a CA is found. Several algorithms with different strategies for selecting the new rows exist. A popular subcategory of row extension algorithms are one-test-at-a-time (OTAT) algorithms, where exactly one row is added in each iteration.

One of the earliest CA generation algorithms, Automatic Efficient Test Generator (AETG) [CDFP97], follows the OTAT strategy. This algorithm generates M candidate rows at every extension step, where M is an algorithm parameter. Candidate rows are produced by first selecting a (parameter,value)-pair that occurs in a maximum number of t-way

interactions not covered in the array generated so far. Afterwards, in a random parameter order, for every parameter a value is chosen such that a maximum number of additional t-way interactions, i.e. t-way interactions not covered in the current array, is covered. Once M candidate rows are produced, the candidate row covering the largest number of-additional t-way interactions is selected and added to the generated array. This continues until the array becomes a CA, which is guaranteed to happen because at every iteration at least one new t-way interaction is covered.

A similar approach is used by the tool Test Case Generator (TCG). Here, the number of generated candidate rows per step is not a parameter, but based on the maximum alphabet size of the CA or MCA to be generated. Further, when an MCA is generated, the columns to be included in the final array are sorted by alphabet cardinality such that columns with a larger alphabet are assigned a value first. However, the row added to the array is chosen the same way as by AETG: The candidate row that covers the largest number of t-way interactions not covered in the array generated so far is used for the array extension.

Another important row extension algorithm is the Deterministic Density Algorithm (DDA) algorithm [BC09]. When generating a new row, for every column and for every (column,value)-pair that is considered for assignment a *density* is defined, based on the number of t-way interactions that might be covered when assigning the respective column or value. In a first step, column densities are calculated and compared to select the column that is assigned a value next. The column with maximum density is selected, and value densities are only calculated for this column. The value with the highest density in the considered column is then selected and the generated row is assigned the selected value in the position of the selected column. Once all values of a row are assigned, the row is added to the array and a new row is generated using the same method. Rows are generated and added to the array until all t-way interactions are covered and the generated array is a CA.

The algorithms mentioned above use heuristics to generate candidate rows covering a large number of t-way interactions. The algorithm proposed in [YZK+10] uses linked lists storing the t-way interactions that need to be covered to generate candidate rows for extension. It constructs at every extension step a candidate row from the linked lists covering a maximal number of t-way interactions not covered in the array to be extended. The maximal number of t-way interactions that can be covered with a new row is stored as a weight. At every step the algorithm attempts to find a row covering that number of t-way interactions. If no such row is found, the weight is decreased by one. The algorithm terminates once all t-way interactions are covered. Although optimal rows are selected for extension from the linked lists, the algorithm does not always outperform the row extension algorithm AETG with regard to size of the generated CAs in the presented evaluation.

3.1.2 Column extension algorithms

Although less popular than row extension algorithms, there have also been attempts of greedy column extension algorithms. Such an algorithm uses a fixed number N of rows, and the first t columns are initialized to contain all v^t t-tuples over Σ_v. If there are more than v^t rows, the remaining rows can be assigned arbitrary values, depending on the algorithm. This yields a CA with t columns. Additional columns are added to the CA in a way that the extended array is also a CA. Adding a column to a CA introduces new t-way interactions to be covered. After addition of the new column, all t-way interactions have to be covered. Depending on the CA to be extended, there might not exist a column that is suitable for the extension, meaning there might not exist a column where all required t-way interactions are covered after the extension step. In that case, the generated CA is returned. The problem with this approach is that it is not possible to guarantee the generation of a $\mathsf{CA}(t, k, v)$ for given parameters t, k, v. The algorithm might return a CA with less than k columns when no further column extension is possible. This is problematic for CT, where a CA with one column per parameter of the system to be tested is required. Nevertheless, some examples of greedy column extension algorithm are given in this subsection.

The IFS algorithm [KS16] starts with calculating a *store*, containing all columns that can possibly be added to the generated CA. Afterwards, the CA is extended greedily by selecting suitable columns from the store. To reduce the number of possible columns stored in the store and to provide a guiding heuristic for the greedy algorithm, the concept of α-*balance* was introduced, which corresponds to $(\boldsymbol{\lambda}, \boldsymbol{y})$-balance with the lower bound $\boldsymbol{\lambda} = \alpha$ and no upper bound $\boldsymbol{y}$, this can for example be achieved with $\boldsymbol{y} = (N, \ldots, N)$. The IFS algorithm served as inspiration for the ClassifyBalancedCAsalgorithm presented in Chapter 4 of this thesis.

Additionally, there exists a greedy column extension algorithm based on pseudo-Boolean constraints [LMZ16]. Again, a version of balance is employed to serve as guiding heuristic when choosing an appropriate column. A pseudo-Boolean constraint solver is used to generate columns for extension. Since the algorithm is based on an exact method, it will be reviewed again in subsection 3.2.

CA classification via column extension with backtracking

While greedy column extension algorithms are not of much practical or theoretical interest, an introduction of backtracking makes column extension algorithms suitable for CA classification and exhaustive CA generation.

First classification results have been presented in [CKRSP10]. The used algorithm computes all possible initializations of the first t columns, and all possible column extensions of every intermediate array. Symmetry breaking is applied to count only one CA of every equivalence class.

This algorithm has later been adopted and refined in [TJIM16]. Again, all possible

initializations of the first t columns are computed. Rows and columns are ordered lexicographically, this allows to iterate over all possible column extensions by taking the value of the previous column as base value and incrementing the value of the new column by one. To count only non-equivalent CAs, for every column extension leading to a CA complete symmetry breaking is performed via a minimality check, checking for lexicographic leadership of the generated array. One problem with the approach for generating columns for extension is that many unnecessary candidate columns are tried when a column extension cannot lead to a CA due to a value assigned in an early row. Therefore, in [IMTJ18] the algorithm is adapted to assign one cell of a column at a time. Additionally, the minimality check is improved to be more efficient.

A similar but slightly different approach was followed in [KMN+20]. Again, a column extension algorithm for CA classification is presented, but both column extension and the minimality check are mapped to other problems and solved by external programs. Column extension is mapped to a covering problem and the minimality check uses *nauty*, a tool able to determine graph isomorphisms. Classification results are only presented for CAs of strength two ($t = 2$).

3.1.3 IPO: A combination of column and row extension

An important and widely used strategy for CA generation is the In-Parameter-Order (IPO) strategy, introduced in [LT98] for strength $t = 2$ and generalized for higher strength in [LKK+07]. The IPO strategy uses a combination of column and row extension to extend an array one column at a time, while ensuring that for every number of columns k' with $t \leq k' \leq k$ a CA is generated. The strategy consists of three phases: *array initialization*, *horizontal extension* (column extension) and *vertical extension* (row extension). At the beginning, the generated array is initialized as $v^t \times t$ array containing all v^t t-tuples over Σ_v as rows. This array is a CA with t columns. Afterwards, horizontal and vertical extensions are executed until the desired array size is reached. In horizontal extension, the current array is extended with an additional column. That column is desired to cover a large number of t-way interactions. If the extended array is again a CA, the next horizontal extension is executed. If, on the other hand, not all t-way interactions are covered after horizontal extension and the array is not a CA anymore, then it is repaired with a vertical extension step to become a CA again: Every missing t-way interaction is added to the generated array, either by adding new rows or by assigning values to star-values, where star-values are array entries that have not been assigned a value. If a new row r is added to the array for a t-way interaction τ, then all columns of r involved in τ are assigned the required values to cover τ. The other columns of r remain star-values and can sometimes be used in vertical extension to cover required t-way interactions without having to add additional rows to the generated array. After vertical extension all missing t-way interactions are covered, the generated array is again a CA and the strategy continues with the next horizontal extension, if the desired CA size is not reached yet. Repairing coverage of the constructed arrays with vertical extensions (row extensions) allows to generate CAs with an arbitrary number of columns, although

the strategy is based on column extension. Combining column and row extension allows to generate compact arrays at a decent execution speed. The IPO strategy has been extensively researched and used in several other algorithms, adapting horizontal and/or vertical extension.

In [LT98], where the IPO strategy was initially proposed, two algorithms for horizontal extension are given. One algorithm attempts to find a column, such that the number of rows that need to be added in vertical extension is minimized. Due to the exponential time complexity of this algorithm, a second horizontal extension algorithm is proposed that is greedy and, with slight adaptions, also used in state-of-the-art IPO algorithms. This algorithm iterates over all rows from top of the array to bottom. For every row of the new column the value covering the largest number of t-way interactions not covered in the current array is selected and assigned. For vertical extension, a greedy algorithm is used, covering one missing t-way interaction at a time. For every missing interaction, the algorithm first iterates over the array and attempts to find star-values that can be used to cover the interaction. If this is not possible, then a new row is added to the array with values assigned only in positions required to cover the currently considered missing t-way interaction. For strength $t = 2$ the proposed greedy algorithm for vertical extension produces optimal results. In [FLL$^+$08] the IPOG-F algorithm is proposed with an adapted approach for horizontal extension. Instead of iterating over the rows in a fixed order, for every assignment the (row,value)-pair maximizing the number of covered t-way interactions is selected and assigned. The authors of [KS18] developed the FIPOG algorithm, which is a fast version of the IPO algorithm, by optimizing data structures and various algorithmic details. Several state-of-the-art CA generation tools are based on algorithms implementing the IPO strategy, for example [YLKK13] with IPOG-F and [WKS$^+$20] with FIPOG.

Various adaptations of the IPO algorithm generating smaller CAs than the basic greedy algorithm have been proposed. In [DLY$^+$15] the problem of vertical extension is reduced to a graph coloring problem and solved with a graph coloring algorithm. This potentially reduces the number of rows that are added during vertical extension, and therefore also the size of the resulting CA. In [YZ11] the MIPOG algorithm is proposed, where star-values are integrated in horizontal extension instead of vertical extension. During horizontal extension, for every row an assignment maximizing the number of covered t-way interactions is selected, where values can be assigned to the value in the new column of the considered row, as well as to all star-values in the considered row. Including star-values in horizontal extension allows to make more informed choices during horizontal extension and therefore generate smaller CAs. Additionally, star-values can be ignored in vertical extension because they are already considered in horizontal extension. Finally, the SIPO algorithm proposed in [WKS21] generates improved solutions for horizontal extension by applying the metaheuristic search method Simulated Annealing. Starting from an initial horizontal extension solution, Simulated Annealing is used to optimize the values of the newly added column and star-values, where the optimization goal is to cover as many t-way interactions as possible, as well as to keep star-values in the array. The

latter objective is added to prevent the algorithm from assigning values to star-values that do not need to be assigned. Applying a metaheuristic after every horizontal extension step reduces the size of the generated CAs, at the cost of increasing the runtime of the algorithm.

3.2 Approaches based on exact methods

An early work on covering array generation with CSP and SAT solving was presented in [HPSS06]. A CA is mapped to two matrices of variables: A matrix corresponding directly to the values of the CA and another matrix encoding for every row and selection of t columns the covered t-way interaction. Three CSP formulations using only one matrix or both are given. Further, a SAT formulation using both matrices is described. The formulations presented in this thesis and other existing SAT formulations for CAs are closely related to the SAT formulation given in [HPSS06], therefore it is repeated here: The constraints below are defined for all $1 \leq i \leq N$, $1 \leq j \leq k$, $1 \leq j' \leq \binom{k}{t}$, $x, x' \in \{0, \ldots, v-1\}$ and $y, y' \in \{0, \ldots, v^t - 1\}$.

$$\bigvee_x m_{ijx} \tag{3.1}$$

$$\neg m_{ijx} \vee \neg m_{ijx'}, \text{ for } x < x' \tag{3.2}$$

$$\bigvee_y a_{ij'y} \tag{3.3}$$

$$\neg a_{ij'y} \vee \neg a_{ij'y'}, \text{ for } y < y' \tag{3.4}$$

$$\neg a_{ij'y} \vee m_{ijx} \text{ for appropriate } i, j', y, j, x \text{ to channel the values of } a_{ij'y} \text{ and } m_{ijx}, \tag{3.5}$$

$$\bigvee_{j'} a_{ij'y} \tag{3.6}$$

The intended meaning of the variables m_{ijx} is that the array entry in the i-th row and j-th column of the generated CA is assigned the value x, if the variable is set to *true*. Thereby, the clauses (3.1) and (3.2) ensure that every array entry is assigned exactly one value. The variables $a_{ij'y}$ correspond to the j'-th selection of t columns of the generated array, and y determines the value of the entries in this column selection and the i-th row. Phrasing the meaning differently, if a variable $a_{ij'y}$ is set to *true*, then the t-way interaction on columns given by j' and with parameter values given by y is covered in the i-th row. Again, clauses (3.3) and (3.4) ensure a unique assignment of values to every row and column selection. Since these two sets of variables can contradict, it is necessary to channel them with constraints as in (3.5). The clauses are generated such that the appropriate array values given by m_{ijx} are enforced for every variable $a_{ij'y}$ that is set to *true* in a model. Finally, clauses (3.6) ensure coverage of all t-way interactions. It is stated in [HPSS06] that clauses (3.1,3.3,3.4) can be omitted without changing the CAs derivable from solutions to the formula because every solution to the formula where the clauses are omitted can be transformed to a solution to the original formula.

In [HPSS06] only local search SAT solvers are used to solve the generated SAT formulas. Local search SAT solvers only attempt to find a satisfying assignment, instead of exploring the full search space until existence or non-existence of a satisfying assignment is proven. Since only part of the search space is explored, symmetry breaking constraints are less important than for exact search and might even slow down the search, therefore no symmetry breaking constraints are included in the given SAT formula. Symmetry breaking for CA generation is only discussed for CSP solving in [HPSS06]. For the CSP formulations lexicographic ordering of rows and columns is enforced, and symbols within a column are selected such that they are ordered by their number of occurrences, i.e. for symbols $x, y \in \{1, \ldots, v-1\}$ with $x \leq y$ the symbol x has to occur at least as often in the respective column as the symbol y. Additionally, the first row of every generated CA is only allowed to contain the symbol 0. The value 0 is excluded from the occurrence count restriction because setting one row to a fixed value is easier to enforce and propagate than a constraint depending on occurrence counts.

The SAT formulation given in [HPSS06] is extended with symmetry breaking constraints and solved with complete SAT solvers in [BMTI10]. For symmetry breaking, the constraints described for the CSP formulation in [HPSS06] are added to the SAT formula. However, a translation of the symmetry breaking constraints to propositional logic is not provided by the authors. Additionally, the applicability of the order encoding for CA generation is explored. The order encoding does not use variables m_{ijx}, where a value x is assigned to position (i, j) of a matrix. Instead, a variable m'_{ijx} denotes that the value at position (i, j) is *at most* x. To ensure validity of assignments, it is necessary to add the following constraints to the formula for every matrix position (i, j) and every value $1 \leq x \leq v - 1$:

$$m'_{ij(v-1)}$$
$$\neg m'_{ij(x-1)} \vee m'_{ijx}$$

The first constraint ensures that every array entry is assigned at most the value $v - 1$, while the second constraint says when a variable $m'_{ij(x-1)}$, denoting that the array entry at position (i, j) is assigned a value that is at most $x - 1$, is set to *true*, then the variable m'_{ijx} denoting that the entry has at most the value x needs to be set to *true* as well. When constraints concern a range of elements, the order encoding is more efficient than the one-hot encoding, where every value is encoded separately as for the model given in [HPSS06]. Three models are given that differ in the usage of the order encoding: The first model corresponds to the model from [HPSS06], extended with symmetry breaking constraints. The second model uses an order encoding to encode the values represented by both variable groups mentioned above. Finally, the third model is called mixed encoding, it uses order encoding for the variables m_{ijx} and the one-hot encoding for the variables $a_{ij'y}$ as in [HPSS06]. In the comparison, the model using only order encoding and the mixed encoding performed better than the original encoding from [HPSS06], i.e. the SAT solver required less time to solve the generated formulas. There was no major performance difference between the two models using order encoding.

In [LMZ16] a greedy column extension algorithm is presented that uses pseudo-Boolean constraint solving to generate columns suitable for extension. The algorithm is greedy, so at every extension step only one solution is needed and symmetry breaking is not required. The first $t-1$ columns are generated randomly. Afterwards, constraints are used to ensure coverage of all t-way interactions and, as a heuristic to increase the number of columns of the resulting array, 1-balance of the generated columns is maximized, i.e. the symbols of a newly generated column all occur with about the same cardinality. Only encoding one column of a CA allows for a simplified formula when compared to a formula encoding a full CA as done in [HPSS06] and [BMTI10]. Again, due to the relevance of the given encoding to this work, the encoding is repeated here and put into context with the encoding from [HPSS06]. Assume an array A with k columns is extended with an additional column. The column for extension can then be generated as follows using pseudo-Boolean constraints: Similar to the variables m_{ijx} used in [HPSS06] and described above, variables $P_{i,u}$ are used for $1 \leq i \leq N$ and $u \in \Sigma_v$ to define the values of array entries. If a variable $P_{i,u}$ is set to *true* in a model, then the entry in the i-th column of the newly generated column is set to the value u. Only one column is added at a time, therefore no variable index denoting the corresponding column is necessary. In [HPSS06] additional variables $a_{ij'y}$ were required to encode where every t-way interaction is covered. Such variables are not required when only one column is added at a time. First of all, for every array entry it is ensured that exactly one value is assigned with constraints

$$\sum_{u \in \Sigma_v} P_{i,u} = 1$$

for every row $i \in \{1, \ldots, N\}$. In [HPSS06], this task was fulfilled by the clauses given in (3.1) and (3.2). To create constraints ensuring coverage, the notion of p-sets is defined: For a $(t-1)$-way interaction τ, a p-set is the set of indices of the rows where τ is covered. Using the notation introduced in this thesis, for a given array A with k columns and alphabet size v, a p-set is a set $A\restriction_{\tau_{t-1}}$ for arbitrary $\tau_{t-1} \in \mathbb{T}_{v,k,t-1}$. Coverage is then enforced by constraints

$$\sum_{i \in A\restriction_{\tau_{t-1}}} P_{i,u} \geq 1,$$

for every $\tau_{t-1} \in \mathbb{T}_{v,k,t-1}$ and every value $u \in \Sigma_v$. These constraints fulfill the same purpose as the clauses from [HPSS06] given in Equations (3.3)-(3.5), however, a reduced number of clauses is required. This is because in every row only one value can be set and therefore coverage of a t-way interaction $\tau = \{(c_1, u_1), \ldots, (c_t, u_t)\}$ with $c_t = k+1$ in a row r only depends on the assignment of the variable P_{r,u_t}, while in [HPSS06] all the Boolean variables $m_{rc_1u_1}, \ldots, m_{rc_tu_t}$ have to be set to *true* for τ to be covered in row r. This demonstrates how the complexity of the generated formulation is reduced by restricting the scope of the formulation to a column instead of a whole array. The two constraint types given above already ensure correctness of the generated columns. However, to provide a guiding heuristic for the algorithm, additional *balance constraints*

are used:

$$\left\lfloor \frac{N}{v} \right\rfloor \leq \sum_{1 \leq i \leq N} P_{i,u} \leq \left\lceil \frac{N}{v} \right\rceil,$$

for every $u \in \Sigma_v$. These constraints ensure for every generated column that all symbols appear almost the same amount of time, up to a difference of 1. Using the notation of [KHKS23], the algorithm generates $(\boldsymbol{\lambda}, \boldsymbol{y})$-balanced CAs with $\lambda_1 = \left\lfloor \frac{N}{v} \right\rfloor$ and $\boldsymbol{y}_1 = \left\lceil \frac{N}{v} \right\rceil$ and no balance defined in other positions of $\boldsymbol{\lambda}, \boldsymbol{y}$. Since the algorithm is greedy, generation of a CA with a certain number of columns for given parameters is not guaranteed, even when such a CA exists.

An exact algorithm for CA generation called EXACT is presented in [YZ06] and [YZ08]. This algorithm does not employ a solver for an exact method. Instead, an application specific exact algorithm using branching, propagation and backtracking is proposed. For generating a $\mathsf{CA}(N; t, k, v)$ the EXACT algorithm starts with an $N \times k$ array, where the values of array entries are not specified except for the $v^t \times t$ subarray in the upper left corner of the array. The values of this subarray are set such that all v^t t-tuples over $\{0, \ldots, v-1\}$ are contained as a row. One value of the array is assigned at a time, either via propagation when an array entry is completely determined by constraints, or by branching when no propagation is possible. If a conflict occurs, the algorithm backtracks by deleting some array assignment. Coverage constraints ensure that every generated array is a CA and partial symmetry breaking constraints are employed to reduce the number of solutions to be visited. In particular, row and column lexicographic order is enforced, in a way that it does not conflict with the values already assigned in the $v^t \times t$ subarray in the upper left corner of the array. Symmetries arising from symbol permutations within a column are broken with the least number heuristic (LNH): A new value $u > 0$ can only be assigned in a column c if column c already contains the value $u - 1$. This reduces symmetries by limiting the number of options when choosing a value in some partially assigned column c. The EXACT algorithm terminates as soon as some solution of the desired size is generated. No classification or exhaustive search are performed. In addition to symmetry breaking, a kind of balance can be used to reduce the number of solutions: for interactions of size up to a user-defined parameter SL, the difference between the occurrence count of SL-way interactions on a specific column selection is at most 1. This condition can be expressed using $(\boldsymbol{\lambda}, \boldsymbol{y})$-balance with the constraint that $\boldsymbol{y}_i - \lambda_i \leq 1$ for $1 \leq i \leq SL$. However, while providing a speedup through search space reduction in some cases, depending on the value SL, such a constraint might forbid all solutions of a given size.

The approaches for exact methods described above (except for [LMZ16]) start with a fixed CA size and determine whether a CA fulfilling the given conditions exist. Finding an optimal CA then requires several executions of the used method, until for given t, k, v existence of a $\mathsf{CA}(N; t, k, v)$ is shown, together with non-existence of a $\mathsf{CA}(N-1; t, k, v)$. In this case, the computed $\mathsf{CA}(N; t, k, v)$ is optimal. There are also approaches integrating the task of finding an optimal CA into the used exact method.

In [AIMTJ13] a MaxSAT formulation for CA generation is developed, where minimizing the number of rows is encoded as optimization objective in MaxSAT. For this purpose, a row usage flag u_r is introduced for every row $r = 1, \ldots, N$, where N is an upper bound on the number of rows of the generated array. Constraints ensure that no values are assigned in rows where the flag u_r is assigned *false*. When minimizing the number of violated singleton clauses $\neg u_r$ for $r = 1, \ldots, N$, the number of rows of the resulting array is optimal.

Another approach for integrating optimization of array size with an exact method is incremental SAT solving, as done in [YKA+15]. Incremental SAT solving is a technique for fast solving of several related SAT formulas. More specifically, when a formula F' is generated from a formula F via addition of new clauses or temporary assumptions, then solving F' incrementally after solving F can be more efficient than solving F and F' completely independent. In [YKA+15] a propositional formula for generation of (constrained) CAs is presented, where the number of array rows allowed by the formulation can be reduced with incremental SAT solving. Using incremental solving, optimal CAs can be computed more efficiently than with the separate solver calls required by most of the approaches described previously.

In addition to generation of CAs, exact methods like SAT solving, SMT solving and CSP solving have also been employed for generation of constrained CAs, in particular for checking whether a generated (partial) array violates the defined constraints, see for example [YBA+16] or [YLN+13]. Especially when using an exact method for CA generation, solvers for exact methods are a natural way to verify validity of the generated arrays with regards to constraints. One example is [YKA+15], already mentioned above, where incremental SAT solving is combined with a SAT formulation for generation of constrained CAs to find optimal constrained CAs. In [AMO+22] the MaxSAT formulation from [AIMTJ13] for unconstrained CAs is adapted and extended to support the generation of optimal constrained CAs. Additionally, incomplete MaxSAT solvers are used with the developed MaxSAT formulation for faster generation of not necessarily optimal CAs. As mentioned before, in this thesis only CAs without constraints are considered.

CHAPTER 4

A Column Extension Algorithm for CA Classification

In this section the ClassifyBalancedCAs algorithm is described, a new column extension algorithm with backtracking. This type of algorithm and related algorithms were also described in Section 3.1.2. The algorithm given in this section has a similar structure as the CA classification algorithm presented in [IMTJ18], however, the ClassifyBalancedCAs algorithm employs a different strategy for building suitable columns, in particular an exact solver, and it incorporates the notion of balance, which allows for additional optimizations and generation of balanced CAs.

4.1 Structure of the ClassifyBalancedCAs Algorithm

The described algorithm generates for given balance vectors $(\boldsymbol{\lambda}, \boldsymbol{y})$, number of rows N, strength t and alphabet size v all non-equivalent $(\boldsymbol{\lambda}, \boldsymbol{y})$-balanced $\mathsf{CA}^{\boldsymbol{y}}_{\boldsymbol{\lambda}}(N; t, k, v)$ for all $t \leq k \leq \mathsf{CAK}^{\boldsymbol{y}}_{\boldsymbol{\lambda}}(N; t, v)$. When no balance vectors $(\boldsymbol{\lambda}, \boldsymbol{y})$ are given, then the set of all non-equivalent $\mathsf{CA}(N; t, k, v)$ is generated for all $t \leq k \leq \mathsf{CAK}(N; t, v)$.

The ClassifyBalancedCAs algorithm starts with an empty matrix and uses column extension with backtracking to generate CAs. Columns are generated using an exact method. While many different exact methods are possible for this purpose, formulations for SAT solvers and pseudo-Boolean constraint solvers are provided in this thesis. Incomplete symmetry breaking based on lexicographic ordering of columns, rows and symbols within a column is included in the presented formulations for column generation. Additionally, complete symmetry breaking is ensured with an additional feasibility check of partial arrays, enforcing all generated arrays to be lexicographically minimal in their equivalence class.

The formulations for column generation using the exact methods SAT solving or pseudo-Boolean constraint solving are given in Section 4.2, and a description of the complete symmetry breaking technique is given in Section 4.3.

A pseudocode of ClassifyBalancedCAs is available in Algorithm 1. A given array A is recursively extended with all columns fulfilling the coverage, balance and symmetry breaking requirements, such that all feasible extensions of A are explored. When no array is given as parameter, the algorithm starts with an array with N rows and 0 columns, i.e. an $\{0,\ldots,v-1\}^{N\times 0}$ array. In the parameter list of Algorithm 1 this empty array is denoted as $A=\emptyset$, where $\emptyset$ is the default value for A. To store the generated CAs, the algorithm keeps a set called *foundCAs* that is initialized to an empty set in line 1. Immediately afterwards, the candidate columns for extension are discovered: A SAT or pseudo-Boolean formula Φ encoding the conditions for candidate columns is generated by the function GenerateFormula$(A,t,v,\boldsymbol{\lambda},\boldsymbol{y})$ in line 2. The formula is generated in a way such that all columns derivable from solutions to Φ fulfill the coverage and, if required, balance constraints, when added to the array A. Additionally, the formula contains partial symmetry breaking constraints enforcing a lexicographic ordering of rows, columns and an ordering of symbols within a column, such that all symbol permutations produce a lexicographically larger column. A detailed description of the generated formula is given in Section 4.2. In a next step, the formula Φ is given to a SAT or pseudo-Boolean constraints solver and all columns derivable from solutions to Φ are stored in a list called *store*, see line 3. Once all the candidate columns are computed, it is time to extend the array A. Using a loop, every column *col* in *store* is considered and used for extension of A (line 4). Since the columns derived from solutions to Φ may lead to equivalent solutions, i.e. solutions that should be removed by symmetry breaking, for every column *col* a feasibility check of A when extended with *col* is executed in the function Admissible$([A,col])$ in line 5 as described in Section 4.3. If it turns out that an extension column is not admissible according to the applied symmetry breaking, then the column is ignored. If, on the other hand, Admissible$([A,col])$ confirms that *col* is a valid extension column, then further actions are taken for the array A extended with *col*: if $[A,col]$ has at least t columns, then it is already a CA and added to the set *foundCAs* for reporting, see lines 6-8. Afterwards, using a recursive call to ClassifyBalancedCAs$([A,col],N,t,v,\boldsymbol{\lambda},\boldsymbol{y})$ all extensions of $[A,col]$ to a lexicographically minimal CA are explored and the reported CAs are again added to the set *foundCAs* in line 9. Finally, once all columns have been explored, the algorithm returns the arrays stored in *foundCAs* (line 12).

Example 1. *In Figure 4.1 a search tree of the ClassifyBalancedCAs algorithm for the instance $N=5$, $t=2$, $v=2$ is given. At every step of the search tree, depicted with a rectangle, a set of candidate columns for extension is calculated and stored in* store. *As mentioned above, candidate columns are all columns fulfilling the coverage and balance requirements, together with some basic symmetry breaking. For every candidate column the algorithm first checks lex-leadership of the extended array with a call to the* Admissible *function. If the column does not lead to a lexicographically minimal array, it is discarded. In the figure, such cases are indicated with a dashed ellipse, connected*

Algorithm 1 ClassifyBalancedCAs($A = \emptyset, N, t, v, \boldsymbol{\lambda}, \boldsymbol{y}$)

```
1: foundCAs ← ∅
2: Φ ← GenerateFormula(A, N, t, v, λ, y)
3: store ← GetAllSolutions(Φ)              ▷ initialize Store, call SAT or PB solver
4: for col ∈ store do
5:     if Admissible([A, col]) then        ▷ Call the feasibility check given in Section 4.3
6:         if numColums([A, col]) ≥ t then
7:             foundCAs ← foundCAs ∪ [A, col]          ▷ Report found CA
8:         end if
9:         foundCAs ← foundCAs ∪ ClassifyBalancedCAs([A, col], N, t, v, λ, y)
10:    end if
11: end for
12: return foundCAs
```

with the respective candidate column via an arrow. If the column is admissible, then the extensions to A, extended with the candidate column, are explored using a recursive call to Algorithm 1 and an arrow points from the candidate column to the box corresponding to the array extended with the candidate column. The levels of the depicted search tree correspond to the size of the array. For example, at level $k = 2$ the array A is of size 2 and it will be extended with a candidate column to a $\mathsf{CA}(5, 2, 3, 2)$ *with $k = 3$ columns. The search tree is traversed via depth-first search, i.e. when considering a candidate column for extension of an array A, the subtree arising from extension of A with the candidate column is completely examined before considering the next candidate column for extension to A. In the given example the* store *at level $k = 4$ is empty, this means no candidate columns exist and the algorithm backtracks to level $k = 3$, where the next candidate column is examined. No* $\mathsf{CA}(5; 2, k, 2)$ *with $k = 5$ columns exists, however, the search tree shows that there is one such CA with $k = 4$ columns, two CAs with $k = 3$ columns and one CA with $k = 2$ columns.*

4.2 Exact methods for generation of feasible columns

In this section the connection between the proposed ClassifyBalancedCAs algorithm and exact methods is described: To compute candidate columns for column extension first a propositional logic formula in CNF or a set of pseudo-Boolean constraints is generated, where the formula in propositional logic is called Φ_{SAT} and the pseudo-Boolean constraint formulation is called Φ_{PB} in this section. In a second step, an exact solver is used to compute all models of the generated problem formulation and the candidate columns are derived from the computed models. In the following, a detailed description of the generated formulations is given. In the remainder of this section, the array considered for extension is denoted as A, with strength t, number of columns k, number of rows N and the alphabet Σ_v. Recall that a pseudo-Boolean constraint of the form $\Sigma_{i=1}^{n} x_i \geq 1$ for Boolean variables x_i is equivalent to a propositional clause $\bigvee_{i=1}^{n} x_i$. In the following,

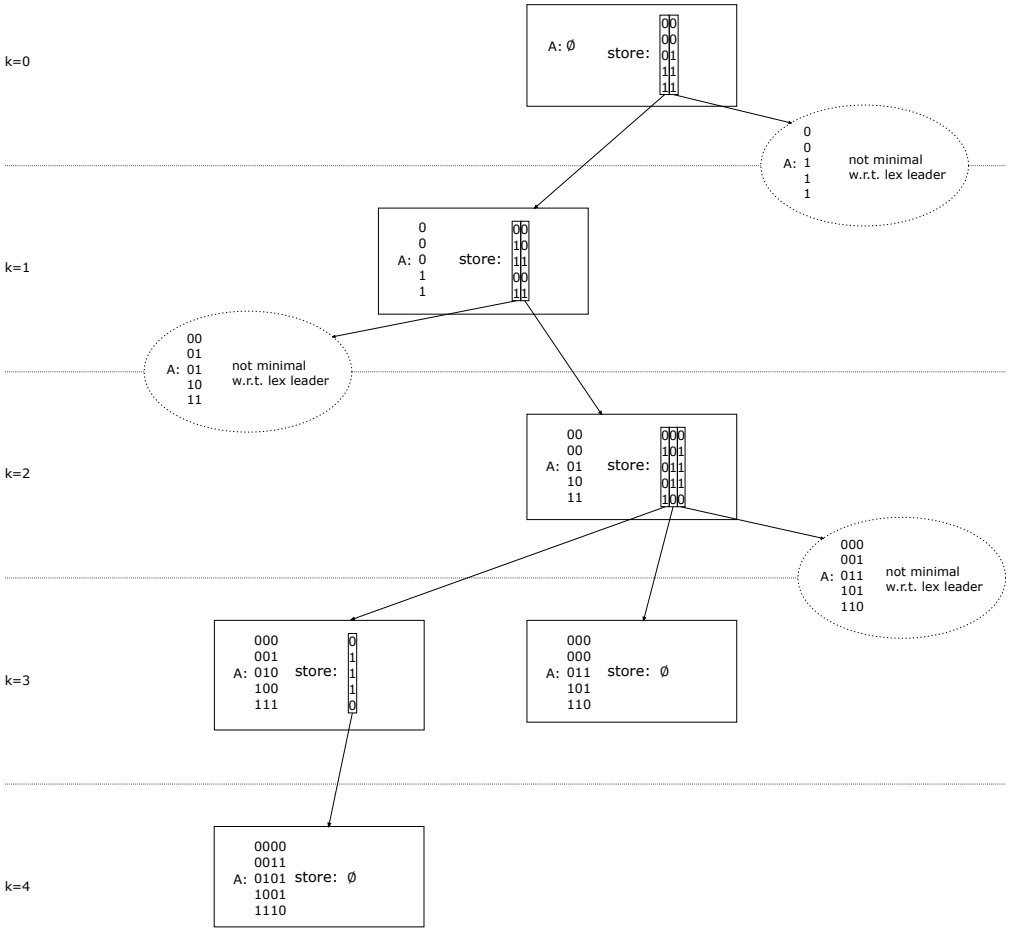


Figure 4.1: Search tree of the ClassifyBalancedCAs algorithm when generating all non-equivalent $\mathsf{CA}(5;2,k,2)$ for $2 \leq k \leq \mathsf{CAK}(5;2,k,2)$.

for every constraint type a pseudo-Boolean formulation will be given, together with an equivalent formulation in propositional logic. For constraints of the form $\Sigma_{i=1}^{n} x_i \geq 1$ no further explanation of the translation to propositional logic will be provided. In Section 3.2 several existing problem formulations for CA generation were given. While there are some differences, most problem formulations can be divided into the following types of constraints:

- Validity of array assignments
- Coverage constraints
- Symmetry breaking
  - Lexicographic ordering of columns
  - Lexicographic ordering of rows

– Symmetry breaking for symbol permutations within a column

Symmetry breaking constraints are only present in problem formulations for exact search.

The same constraint types are considered in this work. In addition to the listed constraint types also *balance constraints* are used, to enable generation of balanced CAs by the ClassifyBalancedCAs algorithm.

4.2.1 Variables and constraints for validity of array assignments

Column extension consists of extending a given array A with an additional column $(u_1, \ldots, u_N)^T \in \{0, \ldots, v-1\}^N$. An example for the extension of an array with a column of unknown values is given in Equation 4.1:

$$\begin{array}{cccc} 0 & 0 & 0 & u_1 \\ 0 & 0 & 0 & u_2 \\ 0 & 0 & 1 & u_3 \\ 0 & 1 & 0 & u_4 \\ 0 & 1 & 1 & u_5 \\ 1 & 0 & 0 & u_6 \\ 1 & 0 & 1 & u_7 \\ 1 & 1 & 0 & u_8 \\ 1 & 1 & 1 & u_9 \\ 1 & 1 & 1 & u_{10} \end{array} . \tag{4.1}$$

In the developed SAT and pseudo-Boolean formulations the set of Boolean variables

$$\{x_{r,\ell} | r \in \{1, \ldots, N\}, \ell \in \Sigma_v\}$$

is used to derive a column for extension. More specifically, given a truth assignment $\mathcal{A}$, the r-th row is assigned a value ℓ if $\mathcal{A}(x_{r,\ell}) = 1$, i.e. the variable $x_{r,\ell}$ corresponding to the r-th row and the value ℓ evaluates to *true* under the given truth assignment. To ensure that the derived array entries are well-defined, constraints are required. For example, it is not possible that one array entry is assigned two different values. Additionally, each array entry needs to be assigned some value. This means for every row r and every model of Φ_{SAT} and Φ_{PB}, exactly one variable in the set $\{x_{r,\ell} | \ell \in \Sigma_v\}$ is assigned *true*.

In Φ_{PB}, this can be easily realized with pseudo-Boolean constraints

$$\sum_{\ell=0}^{v-1} x_{r,\ell} = 1, \qquad \text{for every row } r = 1, \ldots, N.$$

In Φ_{SAT}, encoding an *exactly-one* constraint is not that straightforward. First, the *exactly-one* constraint is divided into an *at-least-one* and an *at-most-one* constraint. The

clauses

$$\bigvee_{\ell=0}^{v-1} x_{r,\ell}, \qquad \text{for every row } r = 1, \ldots, N$$

ensure that every row of the new column is assigned at least one value. An *at-most-one* constraint could be realized with a naive encoding as follows:

$$\neg x_{r,i} \vee \neg x_{r,j}, \qquad \text{for every row } r = 1, \ldots, N \text{ and all values } 0 \leq i < j \leq v-1.$$

However, this encoding would introduce $O(Nv^2)$ clauses. Instead, the sequential counter encoding from [Sin05] is used, which introduces auxiliary variables y_i for $i = 0, \ldots, v-2$ but only requires $N(3v-4)$ clauses. Adding the following clauses to Φ_{SAT}, it is guaranteed that every array entry is assigned at most one value:

$$\begin{aligned} \neg x_{r,i} \vee y_i, &\qquad \text{for every } i = 0, \ldots, v-2, \\ \neg y_{i-1} \vee y_i, &\qquad \text{for every } i = 1, \ldots, v-2, \\ \neg x_{r,i} \vee \neg y_{i-1}, &\qquad \text{for every } i = 1, \ldots, v-1. \end{aligned}$$

Remark 2. In the binary case ($v = 2$) it is possible to only use solution variables $\{x_r | r \in \{1, \ldots, N\}\}$, where the value of the new column in the r-th row is set to 1 if x_r is assigned *true* in the corresponding model, while the entry is set to 0 if x_r evaluates to *false* in the corresponding model. In this case, the *at-most-one* constraints can be omitted since no Boolean variable can be assigned *true* and *false* at the same time. SAT and pseudo-Boolean constraint formulations for binary CAs can be derived from the given formulations by omitting the *at-most-one* constraints and replacing in the remaining constraints the variables $x_{r,1}$ with variables x_r, and all variables $x_{r,0}$ with the literals $\neg x_r$ for every row $r = 1, \ldots, N$. It is also possible to omit the *at-least-one* constraints if a solver is used that only returns complete models. For a solver capable of returning partial models, these constraints are required.

4.2.2 Coverage constraints

When a new column $(u_1, \ldots, u_N)^T$ is added to an array A, then all t-way interactions τ containing the new column need to be covered. Since a t-way interaction can be depicted as a $(t-1)$-way interaction extended with a *(column,value)*-combination, the t-way interactions to be covered are exactly the t-way interactions occurring in the set $\{\tau_{t-1} \cup \{(k+1, \ell)\} | \tau_{t-1} \in \mathbb{T}_{v,k,t-1}, \ell \in \Sigma_v\}$, where $\mathbb{T}_{v,k,t-1}$ denotes the set of $(t-1)$-way interactions on the first k columns of an array with alphabet size v. A t-way interaction $\tau_{t-1} \cup \{(k+1, \ell)\}$ is covered iff the value ℓ occurs in some row r of the new column, that is $u_r = \ell$, where τ_{t-1} is covered in the r-th row of A. Coverage can be ensured by Φ_{PB} with *at-least-one* constraints for every combination of a $(t-1)$-way interaction with a value ℓ:

$$\sum_{r \in A\restriction_{\tau_{t-1}}} x_{r,\ell} \geq 1, \qquad \text{for every } \tau_{t-1} \in \mathbb{T}_{v,k,t-1} \text{ and all } \ell \in \Sigma_v,$$

τ_2	ℓ	coverage constraints	τ_2	ℓ	coverage constraints
$\{(1,0),(2,0)\}$	1	$x_1 + x_2 + x_3 \geq 1$	$\{(1,0),(2,0)\}$	0	$\neg x_1 + \neg x_2 + \neg x_3 \geq 1$
$\{(1,0),(2,1)\}$	1	$x_4 + x_5 \geq 1$	$\{(1,0),(2,1)\}$	0	$\neg x_4 + \neg x_5 \geq 1$
$\{(1,1),(2,0)\}$	1	$x_6 + x_7 \geq 1$	$\{(1,1),(2,0)\}$	0	$\neg x_6 + \neg x_7 \geq 1$
$\{(1,1),(2,1)\}$	1	$x_8 + x_9 + x_{10} \geq 1$	$\{(1,1),(2,1)\}$	0	$\neg x_8 + \neg x_9 + \neg x_{10} \geq 1$
$\{(1,0),(3,0)\}$	1	$x_1 + x_2 + x_4 \geq 1$	$\{(1,0),(3,0)\}$	0	$\neg x_1 + \neg x_2 + \neg x_4 \geq 1$
$\{(1,0),(3,1)\}$	1	$x_3 + x_5 \geq 1$	$\{(1,0),(3,1)\}$	0	$\neg x_3 + \neg x_5 \geq 1$
$\{(1,1),(3,0)\}$	1	$x_6 + x_8 \geq 1$	$\{(1,1),(3,0)\}$	0	$\neg x_6 + \neg x_8 \geq 1$
$\{(1,1),(3,1)\}$	1	$x_7 + x_9 + x_{10} \geq 1$	$\{(1,1),(3,1)\}$	0	$\neg x_7 + \neg x_9 + \neg x_{10} \geq 1$
$\{(2,0),(3,0)\}$	1	$x_1 + x_2 + x_6 \geq 1$	$\{(2,0),(3,0)\}$	0	$\neg x_1 + \neg x_2 + \neg x_6 \geq 1$
$\{(2,0),(3,1)\}$	1	$x_3 + x_7 \geq 1$	$\{(2,0),(3,1)\}$	0	$\neg x_3 + \neg x_7 \geq 1$
$\{(2,1),(3,0)\}$	1	$x_4 + x_8 \geq 1$	$\{(2,1),(3,0)\}$	0	$\neg x_4 + \neg x_8 \geq 1$
$\{(2,1),(3,1)\}$	1	$x_5 + x_9 + x_{10} \geq 1$	$\{(2,1),(3,1)\}$	0	$\neg x_5 + \neg x_9 + \neg x_{10} \geq 1$

Table 4.1: pseudo-Boolean constraints to ensure coverage when extending the array from Equation (4.1) with a fourth column to a $\mathsf{CA}(10;3,4,2)$.

where $A\restriction_{\tau_{t-1}}$ is the set of all rows of A that cover the interaction τ_{t-1}. Similarly, in Φ_{SAT} coverage is ensured with clauses

$$\bigvee_{r \in A\restriction_{\tau_{t-1}}} x_{r,\ell}, \qquad \text{for every } \tau_{t-1} \in \mathbb{T}_{v,k,t-1} \text{ and all } \ell \in \Sigma_v,$$

Coverage clauses only exist if the array A that is extended has at least $t-1$ columns. Otherwise, if the extended array has less than t columns, no t-way interactions exist on the extended array. If A has at least $t-1$ columns, then the number of coverage clauses or pseudo-Boolean constraints is $v^t\binom{k}{t-1}$, since $|\mathbb{T}_{v,k,t-1}| = v^{t-1}\binom{k}{t-1}$ and $|\Sigma_v| = v$.

Example 2. *The pseudo-Boolean constraints in Table 4.1 enforce coverage of a new column when extending the array given in Equation* (4.1) *with a fourth column to a* $\mathsf{CA}(10;3,4,2)$. *Similarly, in a SAT formulation the clauses in Table 4.2 can be used to enforce coverage, as the clauses in Table 4.2 are equivalent to the pseudo-Boolean constraints in Table 4.1. Each constraint or clause ensures coverage of the 3-way interaction* $\tau_2 \cup \{(4,\ell)\}$, *where* τ_2 *and* ℓ *are given in the respective columns. To give an example, coverage of the 3-way interaction* $\{(1,0),(2,0),(4,0)\}$ *is ensured with the pseudo-Boolean constraint* $\neg x_1 + \neg x_2 + \neg x_3 \geq 1$ *or the propositional clause* $\neg x_1 \vee \neg x_2 \vee \neg x_3$, *because the 2-way interaction* $\{(1,0),(2,0)\}$ *occurs in rows 1, 2 and 3 and the given constraint and clause ensure that the extension column contains a 0 in one of these rows. Since the alphabet of the considered array is binary, the literals* $\neg x_r$ *are used for* $r = 1,\ldots,N$ *instead of* $x_{r,0}$.

4.2.3 Balance constraints

The ClassifyBalancedCAs algorithm is capable of generating balanced CAs. For this purpose, additional constraints are required. To recapitulate, an array is $(\boldsymbol{\lambda}, \boldsymbol{y})$-balanced

τ_2	ℓ	coverage clauses	τ_2	ℓ	coverage clauses
$\{(1,0),(2,0)\}$	1	$x_1 \vee x_2 \vee x_3$	$\{(1,0),(2,0)\}$	0	$\neg x_1 \vee \neg x_2 \vee \neg x_3$
$\{(1,0),(2,1)\}$	1	$x_4 \vee x_5$	$\{(1,0),(2,1)\}$	0	$\neg x_4 \vee \neg x_5$
$\{(1,1),(2,0)\}$	1	$x_6 \vee x_7$	$\{(1,1),(2,0)\}$	0	$\neg x_6 \vee \neg x_7$
$\{(1,1),(2,1)\}$	1	$x_8 \vee x_9 \vee x_{10}$	$\{(1,1),(2,1)\}$	0	$\neg x_8 \vee \neg x_9 \vee \neg x_{10}$
$\{(1,0),(3,0)\}$	1	$x_1 \vee x_2 \vee x_4$	$\{(1,0),(3,0)\}$	0	$\neg x_1 \vee \neg x_2 \vee \neg x_4$
$\{(1,0),(3,1)\}$	1	$x_3 \vee x_5$	$\{(1,0),(3,1)\}$	0	$\neg x_3 \vee \neg x_5$
$\{(1,1),(3,0)\}$	1	$x_6 \vee x_8$	$\{(1,1),(3,0)\}$	0	$\neg x_6 \vee \neg x_8$
$\{(1,1),(3,1)\}$	1	$x_7 \vee x_9 \vee x_{10}$	$\{(1,1),(3,1)\}$	0	$\neg x_7 \vee \neg x_9 \vee \neg x_{10}$
$\{(2,0),(3,0)\}$	1	$x_1 \vee x_2 \vee x_6$	$\{(2,0),(3,0)\}$	0	$\neg x_1 \vee \neg x_2 \vee \neg x_6$
$\{(2,0),(3,1)\}$	1	$x_3 \vee x_7$	$\{(2,0),(3,1)\}$	0	$\neg x_3 \vee \neg x_7$
$\{(2,1),(3,0)\}$	1	$x_4 \vee x_8$	$\{(2,1),(3,0)\}$	0	$\neg x_4 \vee \neg x_8$
$\{(2,1),(3,1)\}$	1	$x_5 \vee x_9 \vee x_{10}$	$\{(2,1),(3,1)\}$	0	$\neg x_5 \vee \neg x_9 \vee \neg x_{10}$

Table 4.2: Coverage clauses for extending the array from Equation (4.1) with a fourth column to a $\mathsf{CA}(10;3,4,2)$.

with $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_t)$ and $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_t)$ if every i-way interaction occurs at least λ_i and at most γ_i times in the array for $i = 1, \ldots, t$.

Using pseudo-Boolean constraints, it is quite simple to model balance in Φ_{PB} with the following constraints:

$$\sum_{r \in A|_{\tau_{i-1}}} x_{r,\ell} \geq \lambda_i, \qquad \text{for } 1 \leq i \leq t, \text{ all } \tau_{i-1} \in \mathbb{T}_{v,k,i-1} \text{ and all } \ell \in \Sigma_v, \tag{4.2}$$

$$\sum_{r \in A|_{\tau_{i-1}}} x_{r,\ell} \leq \gamma_i, \qquad \text{for } 1 \leq i \leq t, \text{ all } \tau_{i-1} \in \mathbb{T}_{v,k,i-1} \text{ and all } \ell \in \Sigma_v. \tag{4.3}$$

Similar to coverage constraints, each constraint ensures balance of the i-way interaction $\tau_{i-1} \cup \{(k+1,\ell)\}$, where the constraints from (4.2) enforce the lower bound λ_i on occurrences of $\tau_{i-1} \cup \{(k+1,\ell)\}$, and the constraints given in (4.3) enforce the upper bound γ_i. Note that balance constraints for i-way interactions are only generated when the array A has at least $i-1$ columns, i.e. $k \geq i-1$, otherwise the set $\mathbb{T}_{v,k,i-1}$ is empty. Another noteworthy case is $i = 1$: if $i = 1$, then the set $\mathbb{T}_{v,k,0}$ only contains the empty interaction $\emptyset$, which is covered by every row. Therefore, balance of 1-way interactions is correctly enforced with constraints

$$\sum_{r=1}^{N} x_{r,\ell} \geq \lambda_1, \qquad \text{for all } \ell \in \Sigma_v,$$

$$\sum_{r=1}^{N} x_{r,\ell} \leq \gamma_1, \qquad \text{for all } \ell \in \Sigma_v.$$

Modeling balance constraints in Φ_{SAT} is not that straightforward. Similar to the clauses for validity of array assignments in section 4.2.1, the sequential counter encoding from [Sin05] is used to encode the balance constraints in propositional logic in CNF.

A constraint that at most n out of m variables $y_1, \ldots, y_m$ are allowed to be set to *true* is realized with clauses

$$\bigwedge_{i=1}^{m-1} \neg y_i \vee s_{i,1},$$
$$\bigwedge_{j=2}^{n} \neg s_{1,j},$$
$$\bigwedge_{i=2}^{m-1} \bigwedge_{j=1}^{n} \neg s_{i-1,j} \vee s_{i,j},$$
$$\bigwedge_{i=2}^{m-1} \bigwedge_{j=2}^{n} \neg y_i \vee \neg s_{i-1,j-1} \vee s_{i,j} \text{ and}$$
$$\bigwedge_{i=2}^{m} \neg y_i \vee \neg s_{i-1,n},$$

where the variables $s_{i,j}$ for $i \in \{1, \ldots, m-1\}, j \in \{1, \ldots, n\}$ are new auxiliary variables. For every such constraint $(m-1)n$ additional variables are introduced. The constraint that at least n out of m variables $y_1, \ldots, y_m$ have to be set to *true* in every model corresponds to a constraint that at most $m-n$ out of m variables $y_1, \ldots, y_m$ are allowed to be set to *false*, therefore such a constraint can be realized with clauses

$$\bigwedge_{i=1}^{m-1} y_i \vee s_{i,1},$$
$$\bigwedge_{j=2}^{m-n} \neg s_{1,j},$$
$$\bigwedge_{i=2}^{m-1} \bigwedge_{j=1}^{m-n} \neg s_{i-1,j} \vee s_{i,j},$$
$$\bigwedge_{i=2}^{m-1} \bigwedge_{j=2}^{m-n} y_i \vee \neg s_{i-1,j-1} \vee s_{i,j} \text{ and}$$
$$\bigwedge_{i=2}^{m} y_i \vee \neg s_{i-1,m-n},$$

where the variables $s_{i,j}$ for $i \in \{1, \ldots, n-m-1\}, j \in \{1, \ldots, n\}$ are again new auxiliary variables.

Such a translation is applied to every at-most and at-least constraint defined in (4.2) and (4.3). This translation yields a high number of additional variables and clauses, however, the search space reduction gained from balance constraints in many cases outweighs the additional complexity of the generated formula, as is also demonstrated by the experimental evaluation presented in Section 4.4.

As shown in [KHKS23, Remark 1] and repeated in Remark 1 in Section 2.1.1 in this thesis, under some conditions certain balance constraints are redundant. Since it is not necessary to enforce redundant constraints, the generated pseudo-Boolean or SAT formulation can be simplified by omitting those redundant constraints.

Example 3. *Continuing the example from before, when the* $((4,2,1),(6,3,2))$*-balanced array- with* $N = 10$ *rows given in Equation* (4.1) *is extended with an additional column in a way that the balance is maintained, then first of all it is determined which balance constraints are redundant. With* $\boldsymbol{\lambda} = (4,2,1)$ *and* $\boldsymbol{y} = (6,3,2)$*, the constraints*

- $\lambda_1 = 4$ *because of* $\lambda_1 = 2 \cdot \lambda_2$,
- $\lambda_2 = 2$ *because of* $\lambda_2 = 2 \cdot \lambda_3$,
- $\lambda_3 = 1$ *because this is already enforced by coverage constraints,*
- $y_1 = 6$ *because of* $y_1 = N - \lambda_1$ *and*
- $y_3 = 2$ *because of* $y_3 = y_2 - \lambda_3$

are redundant for the given reasons and can therefore be omitted from the formulations Φ_{PB} *and* Φ_{SAT}*. The only remaining balance constraint is* $y_2 = 3$*, which provides an upper bound on the number of times a two-way interaction is allowed to appear in the array. In* Φ_{PB}*, the constraint* $y_2 = 3$ *results in the pseudo-Boolean constraints listed in Table 4.3. For every constraint, the corresponding* τ_{i-1}*-way interaction is given, together with the value* ℓ *that is used in the new column to extend* τ_{i-1} *to the* i*-way interaction* $\tau_{i-1} \cup \{(k+1,\ell)\}$ *whose balance is enforced by the constraint. Using the sequential counter encoding described above, the balance constraints can be translated to a propositional logic formula in CNF. To give an example for this translation, applying the sequential counter encoding to the pseudo-Boolean constraint* $x_1 + x_2 + x_3 + x_4 + x_5 \leq 3$ *results in the clauses*

$$\neg x_1 \vee s_{1,1}, \neg x_2 \vee s_{2,1}, \neg x_3 \vee s_{3,1}, \neg x_4 \vee s_{4,1}, \neg s_{1,2}, \neg s_{1,3}, \neg s_{1,1} \vee s_{2,1}, \neg s_{1,2} \vee s_{2,2},$$
$$\neg s_{1,3} \vee s_{2,3}, \neg s_{2,1} \vee s_{3,1}, \neg s_{2,2} \vee s_{3,2}, \neg s_{2,3} \vee s_{3,3}, \neg s_{3,1} \vee s_{4,1}, \neg s_{3,2} \vee s_{4,2}, \neg s_{3,3} \vee s_{4,3},$$
$$\neg x_2 \vee \neg s_{1,1} \vee s_{2,2}, x_2 \vee \neg s_{1,2} \vee s_{2,3}, \neg x_3 \vee \neg s_{2,1} \vee s_{3,2}, x_3 \vee \neg s_{2,2} \vee s_{3,3},$$
$$\neg x_4 \vee \neg s_{3,1} \vee s_{4,2}, x_4 \vee \neg s_{3,2} \vee s_{4,3}, \neg x_2 \vee \neg s_{1,3}, \neg x_3 \vee \neg s_{2,3}, \neg x_4 \vee \neg s_{3,3}, \neg x_5 \vee \neg s_{4,3}.$$

To receive a SAT formulation for balance constraints, the same translation is applied to all pseudo-Boolean balance constraints. However, due to the complexity of this encoding no complete example is given here.

τ_1	ℓ	balance constraints
$\{(1,0)\}$	1	$x_1 + x_2 + x_3 + x_4 + x_5 \leq 3$
$\{(1,0)\}$	0	$\neg x_1 + \neg x_2 + \neg x_3 + \neg x_4 + \neg x_5 \leq 3$
$\{(1,1)\}$	1	$x_6 + x_7 + x_8 + x_9 + x_{10} \leq 3$
$\{(1,1)\}$	0	$\neg x_6 + \neg x_7 + \neg x_8 + \neg x_9 + \neg x_{10} \leq 3$
$\{(2,0)\}$	1	$x_1 + x_2 + x_3 + x_6 + x_7 \leq 3$
$\{(2,0)\}$	0	$\neg x_1 + \neg x_2 + \neg x_3 + \neg x_6 + \neg x_7 \leq 3$
$\{(2,1)\}$	1	$x_4 + x_5 + x_8 + x_9 + x_{10} \leq 3$
$\{(2,1)\}$	0	$\neg x_4 + \neg x_5 + \neg x_8 + \neg x_9 + \neg x_{10} \leq 3$
$\{(3,0)\}$	1	$x_1 + x_2 + x_4 + x_6 + x_8 \leq 3$
$\{(3,0)\}$	0	$\neg x_1 + \neg x_2 + \neg x_4 + \neg x_6 + \neg x_8 \leq 3$
$\{(3,1)\}$	1	$x_3 + x_5 + x_7 + x_9 + x_{10} \leq 3$
$\{(3,1)\}$	0	$\neg x_3 + \neg x_5 + \neg x_7 + \neg x_9 + \neg x_{10} \leq 3$

Table 4.3: Balance constraints for extending the array from Equation (4.1) with a fourth column to a $((4,2,1),(6,3,2))$-balanced $\mathsf{CA}(10;3,4,2)$.

4.2.4 Symmetry breaking: row permutations

To prevent visiting two arrays that are identical up to row permutations, a lexicographic ordering of rows is enforced. In Φ_{PB} this ordering is enforced with constraints

$$\neg x_{r,\ell} + \sum_{j=\ell}^{v-1} x_{r+1,j} \geq 1, \quad \text{for all pairs of identical rows } r \text{ and } r+1, \text{ and } \ell \in \Sigma_v \setminus \{0\},$$

while in Φ_{SAT} the lexicographic ordering is ensured with clauses

$$\neg x_{r,\ell} \vee \bigvee_{j=\ell}^{v-1} x_{r+1,j}, \quad \text{for all pairs of identical rows } r \text{ and } r+1, \text{ and } \ell \in \Sigma_v \setminus \{0\}.$$

If two consecutive rows are not identical, then they are already in lexicographic order, since the same constraint was enforced during generation of earlier columns. Similarly, if there are identical rows, they will always be consecutive. If such consecutive identical rows with indices r and $r+1$ exist, then the constraints ensure that the value u_{r+1} assigned to row $r+1$ is greater or equal to the value u_r assigned to row r. The constraints can be read as follows: Either a value $x_{r,\ell}$ is not set to *true*, or there is a value $j \geq \ell$ with $x_{r+1,j}$ assigned to *true*.

Example 4. *The array given in* (4.1) *has two consecutive pairs of identical rows, those are rows* $(1,2)$ *and rows* $(9,10)$*. Using the following two pseudo-Boolean constraints, lexicographic ordering of the rows after extending the array with a fourth column is ensured.*

$$\neg x_1 + x_2 \geq 1,$$
$$\neg x_9 + x_{10} \geq 1.$$

The first constraint is satisfied by an interpretation if either $\neg x_1$ *is assigned* true*, which means the new column contains the value* 0 *in the first row, or if* x_2 *is assigned* true*, which means the second row of the new column contains the value* 1*. The relationship* $u_1 \leq u_2$ *is obviously satisfied for all values* $u_1, u_2 \in \{0, 1\}$ *satisfying* $u_1 = 0$ *or* $u_2 = 1$*. Similarly, the propositional clauses*

$$\neg x_1 \vee x_2,$$
$$\neg x_9 \vee x_{10}$$

can be added to Φ_{SAT} *to guarantee a lexicographic ordering of rows.*

4.2.5 Symmetry breaking: column permutations

Similar to the lexicographic ordering of rows, a lexicographic ordering on columns is enforced to prevent the ClassifyBalancedCAs algorithm from visiting solutions that are identical up to column permutations. To enforce the ordering, constraints ensure that the new column $(u_1, \ldots, u_N)^T$ is lexicographically larger than the most recently added column, that is column k of A, denoted with $(y_1, \ldots, y_N)^T$ here to define these constraints. Note that $(y_1, \ldots, y_N)^T$ already has assigned values $y_i \in \Sigma_v$ for $i = 1, \ldots, N$ when the constraints are created. The pseudo-Boolean constraints

$$\sum_{j=y_r}^{v-1} x_{r,j} + \sum_{r'=1}^{r-1} \sum_{j=y_{r'}+1}^{v-1} x_{r',j} \geq 1, \qquad \text{for all } r = 1, \ldots, N \text{ with } y_r > 0$$

ensure a lexicographic ordering of columns when added to Φ_{PB}. Column $(u_1, \ldots, u_N)^T$ is lexicographically larger than column $(y_1, \ldots, y_N)^T$ if at the first position with $u_i \neq y_i$, we have $u_i > y_i$. An equivalent condition can be formulated as follows: For every $r = 1, \ldots, N$, there is either $y_r \leq u_r$ or there is some row $r' < r$ with $y_{r'} < u_{r'}$. The first part of the constraints, $\sum_{j=y_r}^{v-1} x_{r,j}$ is greater than or equal to one if $y_r \leq u_r$ in the solution corresponding to the considered assignment. The second part of the constraints, $\sum_{r'=1}^{r-1} \sum_{j=y_{r'}+1}^{v-1} x_{r',j}$ is greater than or equal to one if there is a row $r' < r$ with $y_{r'} < u_{r'}$ in the corresponding solution. It is easy to see that the sum of the two terms is greater than or equal to one if the condition for lexicographic ordering stated above is met. For rows with $y_r = 0$ it is not necessary to add a constraint, because every possible value of u_r will satisfy $0 \leq u_r$. The following clauses that are equivalent to the constraints above are added to Φ_{SAT} to enforce a lexicographic ordering of columns:

$$\bigvee_{j=y_r}^{v-1} x_{r,j} \vee \bigvee_{r'=1}^{r-1} \bigvee_{j=y_{r'}+1}^{v-1} x_{r',j}, \qquad \text{for all } r = 1, \ldots, N \text{ with } y_r > 0.$$

Enforcing lexicographic ordering of columns and rows is a common method of symmetry breaking and is also referred to as double-lex constraints in the literature [KNW10].

Example 5. *For the CA given in Equation* (4.1)*, the extension column* $(u_1, \ldots, u_N)^T$ *needs to be lexicographically larger than* $(0,0,1,0,1,0,1,0,1,1)^T$*, which is the third column of the array, i.e. the column that was most recently added. This is achieved with the pseudo-Boolean constraints*

$$\begin{aligned} r = 3\text{:}\ & x_3 + x_1 + x_2 \geq 1, \\ r = 5\text{:}\ & x_5 + x_1 + x_2 + x_4 \geq 1, \\ r = 7\text{:}\ & x_7 + x_1 + x_2 + x_4 + x_6 \geq 1, \\ r = 9\text{:}\ & x_9 + x_1 + x_2 + x_4 + x_6 + x_8 \geq 1 \text{ and} \\ r = 10\text{:}\ & x_{10} + x_1 + x_2 + x_4 + x_6 + x_8 \geq 1, \end{aligned}$$

where constraints are only added for the rows 3, 5, 7, 9 and 10 with a value greater than 0 in the third column. Since the considered array has a binary alphabet, every clause ensures there is either a '1' in the considered row of $(u_1, \ldots, u_N)^T$*, or there is a '1' in a row higher up where the third column contains a '0'. The equivalent clauses in propositional logic are*

$$\begin{aligned} r = 3\text{:}\ & x_3 \vee x_1 \vee x_2, \\ r = 5\text{:}\ & x_5 \vee x_1 \vee x_2 \vee x_4, \\ r = 7\text{:}\ & x_7 \vee x_1 \vee x_2 \vee x_4 \vee x_6, \\ r = 9\text{:}\ & x_9 \vee x_1 \vee x_2 \vee x_4 \vee x_6 \vee x_8 \text{ and} \\ r = 10\text{:}\ & x_{10} \vee x_1 \vee x_2 \vee x_4 \vee x_6 \vee x_8. \end{aligned}$$

4.2.6 Symmetry breaking: symbol permutations

The remaining kind of equivalence action of CAs that requires symmetry breaking constraints are symbol permutations. Different methods to break symmetries induced by symbol permutations exist. Here, symmetries induced by symbol permutations are broken via a lexicographic ordering on the first appearances of every symbol within a column, meaning a symbol $u_r > 0$ is only allowed to occur at row r, if the symbol $u_r - 1$ occurs in a row with a lower index, that is $u_{r'} = u_r - 1$ for some row $r' < r$. To give an example, in the column $(0,0,1,0,1,2,2)^T$ the first appearance of '0' is in the first row, the first appearance of '1' is in the third row and the first appearance of '2' is in the sixth row. The first appearance of '0' occurs before the first appearance of '1', which again occurs before the first appearance of '2'. If now a symbol permutation swapping '1' and '2' is applied, this results in the column $(0,0,2,0,2,1,1)^T$, where the first appearance of '2' occurs in the third row, which is before the first appearance of '1' in the sixth row and therefore violates the described constraint. This symmetry breaking condition implies that the first row of every column is set to '0', since the symbol '0' has to occur before every other symbol. This method of symmetry breaking for symbol permutations is similar to the LNH heuristic used for CA generation in [YZ08]. The algorithm described there assigns entries of an array one-by-one. According to the LNH heuristic, a cell can only be assigned a value u_r if the value $u_r - 1$ is already assigned to some cell in the

same column. If a fixed assignment order is used, filling each column from the top, the LNH heuristic is equivalent to the symbol permutation symmetry breaking used here. Additionally, this method only allows symbol permutations yielding a lexicographically minimal array with regard to a column-wise linearization for complete symmetry breaking. The pseudo-Boolean constraints

$$x_{1,0} \geq 1 \text{ and}$$
$$\neg x_{r,\ell} + \sum_{r'=2}^{r-1} x_{r',\ell-1} \geq 1 \text{ for all rows } r \in \{2,\ldots,N\}, \text{ and all } \ell \in \{2,\ldots,v-1\}$$

break symmetries according to the described method when added to Φ_{PB}. The first constraint $x_{1,0} \geq 1$ simply sets the value in the first row to '0', while the constraint $\neg x_{r,\ell} + \sum_{r'=2}^{r-1} x_{r',\ell-1} \geq 1$ ensures for a value $\ell \geq 2$ occurring in a row $r \geq 2$ that the value $\ell - 1$ occurs in some row $r' < r$. It is not necessary to add such constraints for the value $\ell = 1$ because the value '0' always occurs in the first row and therefore before the first occurrence of '1'. The equivalent SAT formulation for Φ_{SAT} consists of the clauses

$$x_{1,0} \text{ and}$$
$$\left(\neg x_{r,\ell} \vee \bigvee_{r'=2}^{r-1} x_{r',\ell-1}\right) \text{ for all rows } r \in \{2,\ldots,N\}, \text{ and all } \ell \in \{2,\ldots,v-1\}.$$

4.2.7 Blocking clauses: finding all columns suitable for extension

SAT and pseudo-Boolean solvers in general only return one model for a given formula or set of constraints, or they determine that no model exists. For the proposed ClassifyBalancedCAs algorithm performing exhaustive search, it is at every extension step required to receive all columns that are suitable for extension of the given array A, that means all models of the constructed formula or constraint set. For this purpose, a method called *blocking clauses* [McM02] is used: The solver is executed to solve the formula or constraint set and returns one solution. Using an additional clause or constraint, the formula or constraint set is extended such that the reported solution is forbidden. Consider for example the column $(u_1,\ldots,u_{10})^T = (0,0,1,1,0,1,0,0,1,1)^T$, which is one possible solution for extension of the array given in Equation (4.1). A constraint

$$x_1 + x_2 + \neg x_3 + \neg x_4 + x_5 + \neg x_6 + x_7 + x_8 + \neg x_9 + \neg x_{10} \geq 1$$

invalidating the model corresponding to the solution column $(0,0,1,1,0,1,0,0,1,1)^T$ can be added to Φ_{PB}. Afterwards the solver is again executed with the modified formula and the process is repeated until no further solutions exist. Similarly, the clause

$$x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4 \vee x_5 \vee \neg x_6 \vee x_7 \vee x_8 \vee \neg x_9 \vee \neg x_{10}$$

can be added to Φ_{SAT} to cause the SAT solver to deliver a different solution in the next iteration.

Since this approach requires many calls to a SAT or pseudo-Boolean constraint solver, incremental solving is used, which is a technique implemented in most modern SAT and pseudo-Boolean constraint solvers that improves the solving efficiency when many similar formulas are solved iteratively.

4.3 Complete symmetry breaking with a lex-leader feasibility check

The symmetry breaking constraints included in the SAT and pseudo-Boolean formulations do not forbid all equivalent solutions. To enable classification, but also to further reduce the search space, an additional feasibility check is executed to verify whether a partial array is minimal in its equivalence class and therefore might have extensions that are also minimal in their equivalence classes. A feasibility check or minimality check has previously been applied to CAs in [IMTJ18]. While there are small differences, the basic structure of the algorithm is the same. For a given array A, all arrays equivalent to A up to a combination of column permutations and symbol permutations within a column are constructed. For every such array the rows are then sorted and the resulting array is lexicographically compared to A, using a column-wise linearization. If A is lexicographically minimal, then it is lexicographically smaller than every such constructed array. The feasibility check outputs *true* and the search continues with all extensions of A. If A is not minimal, then some constructed array will be lexicographically smaller than A. The feasibility check outputs *false* and the array A is discarded from the search and not further extended. A description of the feasibility check can be found in Algorithm 2. A permuted array P is constructed that is empty at first (line 2). Additionally, a set *remaining* is constructed, containing all columns of A that do not occur in P. Since P is initially empty, the set *remaining* contains all columns of A at this point (line 2). Afterwards, a recursive function CHECK-MINIMALITY(A, P,*remaining*) is used to step-by-step construct all possible arrays P from columns of A, permuted with all possible symbol permutations (line 3). The function CHECK-MINIMALITY(A, P,*remaining*) receives as parameters the array A that is checked for lex-leadership, that is minimality in its equivalence class, together with an array P, that can be extended to an array equivalent to A by adding the columns of the set *remaining* to P. At every recursion level, P is extended with one column in every possible way: To every column in *remaining*, every possible symbol permutation σ is applied, and P is extended with the resulting column, yielding an array P' (line 7), whose rows are sorted in the next step (line 8). The thereby constructed array is the lexicographically minimal array that can be constructed from the chosen columns and symbol permutations. Every such constructed array P' is compared to the first $|P'|$ columns of A, where $|P'|$ is the number of columns of P'. Because a column-wise linearization is used, if P' is lexicographically smaller than A, then every extension of P' to an array equivalent to A will be lexicographically smaller than A. Therefore, A is not minimal and the algorithm returns *false* (line 10). If, on the other hand, P' is lexicographically larger than A, then the selected column and symbol permutation of P' will not yield any array that is lexicographically smaller than

A and this branch can be ignored. If P' is equal to the first $|P'|$ columns of A, then further exploration is necessary to decide whether A is minimal. In a recursive call CHECK-MINIMALITY(A,$remaining \backslash \{col\}, P'$) further extensions to P' are explored (line 12). If minimality of A can be refuted in this call, then *false* is returned. Otherwise, the algorithm continues with the remaining extensions to the array P that was given as parameter. Finally, once all column and symbol permutations have been explored and minimality of A was not refuted, the algorithm returns *true* in line 18 to confirm minimality of A.

Algorithm 2 ADMISSIBLE(A)

```
1:  P ← empty column list                          ▷ Initialize P as empty list of columns
2:  remaining ← set of all columns of A            ▷ Initialize set of all columns
3:  return CHECK-MINIMALITY(A, P,remaining)

4:  function CHECK-MINIMALITY(A, P, remaining)
5:      for all col in remaining do
6:          for all σ symbol permutation on Σ_v do
7:              P' ← [P, σ(col)]
8:              P' ← SortRows(P')
9:              if P' <_lex A[1, ..., |P'|] then          ▷ A is not minimal
10:                 return False
11:             else if P' = A[1, ..., |P'|] then
12:                 if CHECK-MINIMALITY(A,remaining\{col}, P') = false then
13:                     return false
14:                 end if
15:             end if
16:         end for
17:     end for
18:     return true
19: end function
```

In order to optimize the algorithm, several improvements were made that are not depicted in Algorithm 2 for the sake of simplicity. First of all, no columns or rows are actually copied during the search. Instead, index lists are used, together with an array of size v called *permutation* storing the symbol permutation of the current recursion level. To give some examples, the set *remaining* initially consists of an index list $\{1, \ldots, k\}$, that is reduced during the search, as columns are removed from the set, for example to $\{3, 5\}$, when P contains all columns of A, except for the third and fifth column. Ordering of *remaining* does not matter. Additionally, an index list *rows* is used, storing the row permutation leading to the sorted array P'. The list *rows* is initialized as $[1, \ldots, N]$ and the indices are permuted later on. For an array with 6 rows a permuted index list might be $[2, 1, 3, 6, 5, 4]$. A symbol permutation of a column of an array with alphabet size $v = 3$ could be $[2, 1, 0]$. In this case, a symbol '0' in A is interpreted as symbol '2' in P, a '1' in

A corresponds to a '1' in P and a symbol '2' in A corresponds to a '0' in P. Altogether, when a column $(u_1, \ldots, u_N)$ of A is added to P using the row permutation defined by *rows* and the symbol permutation defined by *permutation*, then the r-th row of the new column of P contains the symbol $permutation[u_{rows[r]}]$.

Secondly, the sorting step is optimized by reusing information from the previous recursion level. At every recursion level, a sorted array P with $|P|$ columns is given or, more precisely, the information that is required to further extend the array P, for example the row ordering *rows*. The array P is then extended with an additional column to an array P' and the rows are sorted. However, the rows of P' are sorted lexicographically, meaning if two rows differ on the first $|P|$ columns, then their ordering is completely determined by the first $|P|$ columns. Only the ordering of rows that are equal on their first $|P|$ positions may change from adding one additional column to P. This means when the row ordering of P is known, together with the groups of rows that are equal in P, then the ordering of P stored in *rows* can be reused and only each group of rows equal in P needs to be sorted according to the value of the new column added to P'. Using this optimization, the complexity of row comparison during the sorting of P' is reduced from $O(|P'|)$ to $O(1)$, because only the values in the newly added column are compared. Additionally, not one section of size N is sorted, but the sorting areas are reduced to several sections of sizes $N_1, \ldots, N_m$ for some $m \leq N$, with $N_1 + \ldots + N_m = N$. This optimization also affects the comparison of P' to the first $|P'|$ columns of A. Sorting P' will not change anything in the first $|P| = |P'| - 1$ columns of P'. Because P is only extended recursively when P matches the first $|P|$ columns of A, for comparison of P' with A it is sufficient to compare the newest column of P' with the column $|P'|$ of A, which again reduces the effort of array comparison from $O(N|P|)$ to $O(N)$.

The applied sorting algorithms are quicksort and a modified version of selection sort. While quicksort generally has a lower time complexity than selection sort, for alphabet $v > 2$ the modified implementation of selection sort performed faster and is therefore used for these alphabet sizes.

Example 6. *The feasibility check described in this section is demonstrated using the array*

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}, \tag{4.4}$$

which also occurs in the example search tree of ClassifyBalancedCAs in Figure 4.1 and is not a lexicographically minimal array in its equivalence class. Figure 4.2 displays a possible search tree of the feasibility check when checking for lexicographic minimality of A. In the figure also the mentioned optimizations are included, i.e. using index lists instead of storing columns and dividing the columns to be sorted into sections of several rows. For every column, the sections of A are depicted in the upper left corner of the figure. The sections are chosen such that in every subarray consisting of the rows in one

section and all columns to the left of the partitioned column, all rows are identical. For example, column 2 of A is divided into two sections, where the subarray of the first section only contains (0) *as row, while the subarray of the second section only contains* (1) *as row. The third column is divided into four sections, where the corresponding four subarrays contain the rows* $(0,0)$*,* $(0,1)$*,* $(1,0)$ *and* $(1,1)$*, respectively. After initializing* remaining *with* $[1,2,3]$ *because A has three columns and the row permutation* rows *with* $[1,2,3,4,5]$ *because A has five rows, the search can begin. In the depicted example search tree, the search begins with taking column 2 from* remaining *and negating it. The considered array is binary, therefore the only symbol permutations are identity, that is* $\sigma(0) = 0$ *and* $\sigma(1) = 1$*, and negation, that is* $\sigma(0) = 1$ *and* $\sigma(1) = 0$*. The array* P' *resulting from extension of the empty array P with the negated column 2 of A is then sorted. This results in the column* $(0,0,1,1,1)^T$*, which is compared to the first column of A, which is* $(0,0,0,1,1)^T$ *and therefore lexicographically smaller than* P'*. Because* P' *can not lead to an array lexicographically smaller than A, the branch is discarded and the empty array P is instead extended with the identity column 2 of A. After sorting, the extended array* P' *is* $(0,0,0,1,1)^T$*, which is equal to the first column of A, and recursively* P' *is further extended. In the next step, the array* $(0,0,0,1,1)^T$ *constructed from column 2 of A with the row permutation* $(1,2,4,3,5)$ *is further extended with column 1 of A, which is also permuted with* rows$=(1,2,4,3,5)$*. Every section of the second column of A is sorted individually. This results in the new column* $(0,0,1,0,1)^T$*, which is equal to the second column of A. Only column 3 is still in* remaining*. In the next step, the permuted array is extended with column 3 of A and again sorted sectionwise. Since the result is equal to A, no refutation for minimality of A was found and a different column permutation is tried at the previous recursion level. After appending column 3 of A negated and sorting sectionwise, the resulting array* P' *is lexicographically smaller than A and* false *is returned by the feasibility check, because A is not minimal.*

4.4 Experimental evaluation of ClassifyBalancedCAs

The classification results determined with the ClassifyBalancedCAs algorithm were described in [KHKS23] and are available online at [MAT], however, due to the large amount of data and because the classification results are not the focus of this thesis they are not repeated here. Instead, this section focuses on evaluating the performance of the algorithm. For this purpose, first the runtime of ClassifyBalancedCAs is compared to other algorithms for CA classification, in particular the algorithms described in [TJIM16], [IMTJ18] and [KMN+20]. Afterwards, two versions of ClassifyBalancedCAs using different exact methods (SAT or pseudo-Boolean constraints) are compared. As described in the previous section, the ClassifyBalancedCAs algorithm applies either a SAT solver or a pseudo-Boolean constraint solver. For the experiments either the SAT solver MiniSat 2.2.0 [ES04] was used, or clasp 3.3.6 [GKS12] was used as a pseudo-Boolean constraint solver. The ClassifyBalancedCAs algorithm was implemented in C++ and all experiments with the algorithm were conducted on a server with an AMD EPYC 7502P processor with 32 cores at 2.5 GHz base clock and 3.35 GHz boost clock and 128GB of

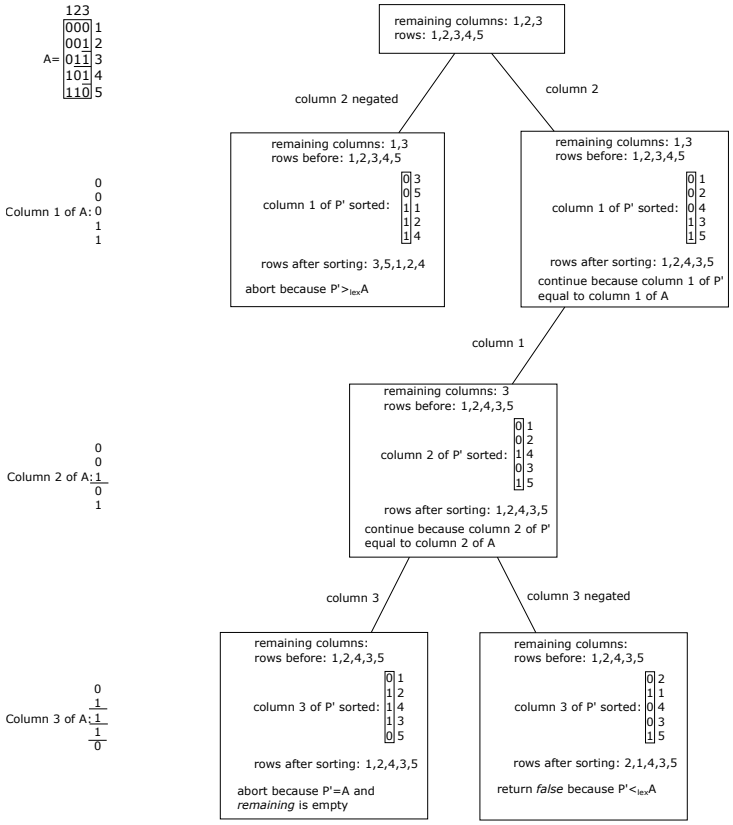


Figure 4.2: Search tree of the feasibility check when checking for lexicographic minimality of the array A given in Equation (4.4).

RAM. The runtimes of other algorithms for comparison were taken from the respective papers. Since the algorithms for comparison were executed on different machines, the comparability of the given times is limited, especially when the runtime difference is small.

A comparison of ClassifyBalancedCAs with the algorithms from [TJIM16] and [IMTJ18] is given in Table 4.4. The columns headed by 'N', 't' and 'v' provide values for the respective parameters of a $\mathsf{CA}(N;t,k,v)$, while the column headed by 'k' provides the maximum k where a $\mathsf{CA}(N;t,k,v)$ exists for the given N, t and v, i.e. the column provides $\mathsf{CAK}(N;t,v)$. Except for the last column ('PB with pruning time (s)'), the runtimes in the table are given for the classification of *all* CAs with the corresponding number of rows N, strength t and alphabet size v, i.e. a runtime corresponds to the classification of all arrays $\mathsf{CA}(N;t,i,v)$ for i from t to $\mathsf{CAK}(N;t,v)$. Three variants of ClassifyBalancedCAs are provided for comparison:

- The column headed with 'SAT time (s)' displays the runtime of the ClassifyBalancedCAs variant generating formulas in propositional logic and using MiniSAT

2.2.0 as SAT solver.

- The column headed with 'PB time (s)' displays the runtime of the ClassifyBalancedCAs variant generating pseudo-Boolean constraints and using clasp 3.3.6 for solving the generated problems.

- Lastly, to display the advantage of balance-based pruning, the runtimes of an ClassifyBalancedCAs variant applying this pruning method is given in the column headed with 'PB with pruning time (s)'. The column headed by 'λ for pruning' displays the balance vector λ that was used to restrict the search space. The vector λ was chosen according to [KHKS23, Lemma 1] (repeated in Equation (2.9) in this thesis), such that no CAs with k columns are removed from the search space, where k is the value given in the column headed by 'k'. While the variant with balance-based pruning generates less information than the other algorithms, as there is no guarantee that all CAs with less than k columns are classified, it can provide a major runtime advantage when one is only interested in CAs with a larger number of columns.

As mentioned, the times for the compared algorithms other than ClassifyBalancedCAs were copied from the respective papers. Since the times were not always given in seconds, a conversion into seconds is provided where necessary. The runtimes from the considered ClassifyBalancedCAs variants are always given in seconds and rounded down, i.e. a runtime of 0 means the program terminated in less than one second. For some instances no runtime was given in [TJIM16], those instances are marked with a '-' in the corresponding table cells. Similarly, for some instances no result was computed with variants of the ClassifyBalancedCAs algorithm and the corresponding table cells are marked with '-'. For instances with a runtime longer than 5 seconds, the time of the fastest algorithm is depicted in bold. For smaller runtimes a meaningful comparison is not possible. It can be seen that for all instances where the best time is marked, the ClassifyBalancedCAs algorithm was faster than the algorithms from [TJIM16] and [IMTJ18] with a speedup factor larger than 100 on the three instances $(N = 20, t = 2, v = 4)$, $(N = 25, t = 2, v = 5)$ and $(N = 36, t = 2, v = 6)$.

For the instances $(N = 14, t = 2, v = 3)$ and $(N = 38, t = 2, v = 6)$ no result was computed with the pseudo-Boolean variant of the ClassifyBalancedCAs algorithm. While the proposed algorithm would most likely be faster than the algorithm given in [IMTJ18], a comparison was not possible due to the extensive runtime. The runtimes given in [IMTJ18] for these instances are 4720,168 hours and 7404,128 hours, which correspond to 196 days in the first case and 308 days for the second instance.

The column headed with 'PB with pruning time (s)' was considered separately since, as mentioned above, this variant only considers a reduced search space. The times given in this column are marked bold when balance-based pruning provides a considerable speedup when compared to the algorithm variant without pruning. Especially for the instance $(N = 33, t = 3, v = 3)$ balance-based pruning provides a huge advantage by

reducing the runtime from 290 seconds to less than one second. However, balance-based pruning does not always help. First of all, it can only be applied for $t > 2$ and $N > v^t$. Second, in some cases the runtime reduction is not noteworthy. See for example the instance $(N = 16, t = 3, v = 2)$ where the runtime is only reduced from 90000 to 87604. In this case the classification without balance constraints using a SAT solver is faster than using pseudo-Boolean constraints with balance-based pruning.

Table 4.4: Runtime comparison of different variants of ClassifyBalancedCAs with the classification algorithms from [TJIM16] and [IMTJ18].

						ClassifyBalancedCAs			
N	t	v	k	time from [TJIM16]	time from [IMTJ18]	SAT time (s)	PB time (s)	λ for pruning	PB with pruning time (s)
4	2	2	3	85 μs $<$ 1 s	0.002 s	0	0		
5	2	2	4	197 μs $<$ 1 s	0.003 s	0	0		
6	2	2	10	0.025 s	0.034 s	0	0		
7	2	2	15	0.373 s	0.57 s	1	0		
8	2	2	35	5.51 h = 19836 s	$>$ 12 h	**2754**	2764		
8	3	2	4	666 μs $<$ 1 s	0.002 s	0	0		
10	3	2	5	0.012 s	0.005 s	0	0	(5,2,1)	0
12	3	2	11	0.851 s	1.088 s	0	0	(6,2,1)	0
15	3	2	12	1.39 h = 5004 s	0.443 h = 1594 s	**228**	384	(7,2,1)	**70**
16	3	2	14	1052.65 h = 3789540 s	130.573 h = 470062 s	**55929**	90000	(7,2,1)	**87604**
16	4	2	5	0.158 s	0.189 s	0	0		
21	4	2	6	694.6 s	0.346 s	0	0	(10,5,2,1)	0
24	4	2	12	53.39 h = 192204 s	72.846 s	**7**	9	(12,6,2,1)	**1**
9	2	3	4	0.006 s	0.011 s	0	0		
11	2	3	5	1.78 s	0.028 s	0	0		
12	2	3	7	252.38 s	1.52 s	0	1		
13	2	3	9	4.14 h = 14904 s	0.368 h = 1324 s	353	**301**		
14	2	3	10	-	4720.168 h = 16992604 s	-	-		
27	3	3	4	-	0.156 s	0	0		
33	3	3	6	-	1.453 h = 5230 s	317	**290**	(11,3,1)	**0**
16	2	4	5	629.66 s	2.216 s	0	0		
19	2	4	6	-	0.545 h = 1962 s	**71**	**71**		
20	2	4	6	-	1645.534 h = 5923922 s	**28642**	56758		
25	2	5	6	-	0.383 h = 1378 s	10	**6**		
36	2	6	3	-	56.981 h = 205131 s	2052	**1234**		
37	2	6	4	-	100.768 h = 362764 s	467755	**265063**		
38	2	6	4	-	7404.128 h = 26654860 s	-	-		

In Table 4.6 the performance of ClassifyBalancedCAs, both with a SAT solver and with a pseudo-Boolean constraint solver, is compared with the runtime of the algorithm described in [KMN+20]. There are no columns for balance-based pruning, because balance-based pruning is only applicable for $t > 2$ and [KMN+20] only considers instances with $t = 2$. As above, the runtime of the algorithm from [KMN+20] was copied from the paper and the times in the table are the times required to classify all CAs with the given number of rows N, strength t and alphabet size v. In [KMN+20] only the times per extension step are given, i.e. the time to produce all CAs with $k + 1$ columns when all CAs with k columns are given, therefore the time depicted in Table 4.6 is the sum of the times listed in [KMN+20] for an instance. An example is given in Table 4.5. The total time required to generate all CAs for the given parameters is the sum of the times given there, i.e. the time to generate all CAs with $(N = 38, t = 2, v = 6)$ is $156h + 1074s + 143.7s = 156h$.

As in Table 4.4, in Table 4.6 the runtime of the fastest algorithm per instance is marked bold, if the runtime is longer than 5 seconds. In this table we see that there is no clear winner. While ClassifyBalancedCAs with a pseudo-Boolean solver is faster on two instances, the algorithm from [KMN+20] outperforms ClassifyBalancedCAs on

Table 4.5: Excerpt of the data from Table 1 in [KMN+20] to show how runtimes are described there.

N	t	v	k	No. of CAs	time from [KMN+20]
38	2	6	2	3	-
38	2	6	3	30491	156.0 h
38	2	6	4	8865	1074.0 s
38	2	6	5	0	143.7 s

four instances and there are several more instances where ClassifyBalancedCAs did not terminate within a reasonable timeframe. An interesting observation is that the algorithm from [KMN+20] was faster on the instance $(N = 19, t = 2, v = 4)$, however, ClassifyBalancedCAs was faster on the larger instance $(N = 20, t = 2, v = 4)$. This suggests that ClassifyBalancedCAs scales better with regard to the number of rows N. The runtime of the instances $(N = 27, t = 2, v = 5)$ and $(N = 28, t = 2, v = 5)$ supports this observation. While the algorithm from [KMN+20] outperforms ClassifyBalancedCAs on both instances, the ratio

$$\frac{\text{(time of ClassifyBalancedCAs algorithm)}}{\text{(time of algorithm from [KMN+20])}}$$

decreases for higher N: for $N = 27$ the ratio is $483/62 = 7.79$, while for $N = 28$ it is only $957672/245914 = 3.89$. However, the proposed ClassifyBalancedCAs algorithm seems to scale worse with regard to the alphabet size v. Although it outperforms the algorithm from [KMN+20] on some instances for $v = 3$ and $v = 4$, for higher alphabet the algorithm from [KMN+20] is always faster than the algorithm presented in this thesis.

To compare the performance of the ClassifyBalancedCAs variants with a SAT solver and with a pseudo-Boolean constraint solver, Figures 4.3, 4.4 and 4.5 depict the runtime of both ClassifyBalancedCAs versions (SAT and pseudo-Boolean constraints (PB)) for different balance vectors, i.e. different amounts of search space reduction. Additionally, the black line headed '# backtracks' depicts the number of backtracks, which corresponds to the total number of CAs during the search and the size of the search space. In all three Figures we can make similar observations. With appropriate scaling, the runtime of the ClassifyBalancedCAs variant with pseudo-Boolean constraints directly corresponds to the size of the search space. The variant using a SAT solver on the other hand suffers from a lot of overhead from enforcing the balance constraints, which can be seen as high fluctuation of the blue line in the figures. When balance vectors require many balance constraints to be enforced, the variant using SAT solving and the sequential counter encoding for balance constraints is much slower than the variant using pseudo-Boolean constraints, where balance constraints can be handled naturally. However, when balance constraints can be skipped due to redundancy, as shown in [KHKS23, Remark 1] and also described in Remark 1 of this thesis, the runtime of the variant with a SAT solver is similar to the runtime of the variant using a pseudo-Boolean constraint solver, and even performs better in several cases.

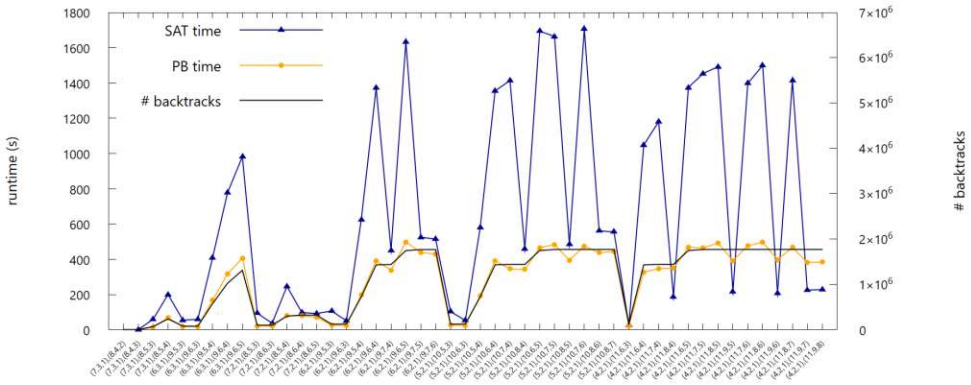


Figure 4.3: Runtime of the ClassifyBalancedCAs algorithm when classifying the instance $\mathsf{CA}^{\boldsymbol{y}}_{\lambda}(15;3,k,2)$ for different balance vectors. Times are given both for the ClassifyBalancedCAs variant with the SAT solver MiniSAT and for the variant with clasp as pseudo-Boolean constraints solver.

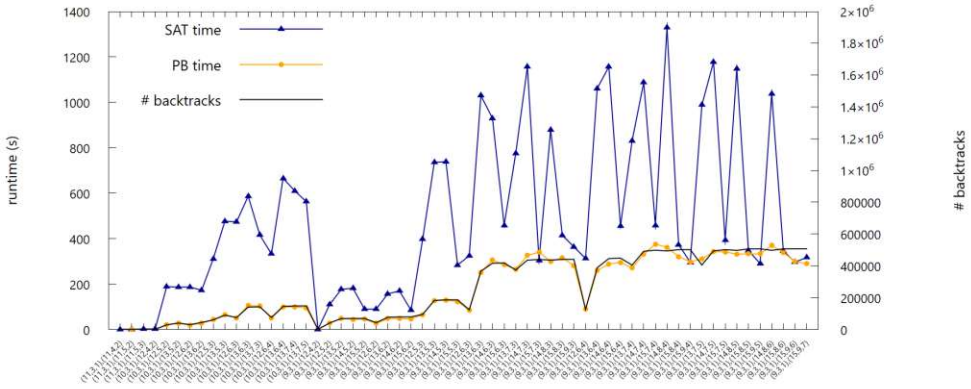


Figure 4.4: Runtime of the ClassifyBalancedCAs algorithm when classifying the instance $\mathsf{CA}^{\boldsymbol{y}}_{\lambda}(33;3,k,3)$ for different balance vectors. Times are given both for the ClassifyBalancedCAs variant with the SAT solver MiniSAT and for the variant with clasp as pseudo-Boolean constraints solver.

Table 4.6: Runtime comparison of ClassifyBalancedCAs where a SAT solver or a pseudo-Boolean solver is used for column generation with the classification algorithm from [KMN+20].

					ClassifyBalancedCAs	
N	t	v	k	time from [KMN+20]	SAT time (s)	PB time (s)
10	2	3	4	<1 s	0	0
11	2	3	5	<1 s	0	0
12	2	3	7	<1 s	0	1
13	2	3	9	1703 s	353	**301**
14	2	3	19	1065 days	-	-
17	2	4	5	<1 s	0	0
18	2	4	5	<1 s	0	1
19	2	4	6	**41 s**	71	71
20	2	4	6	45,33 h = 163201 s	**28642**	56758
26	2	5	6	<1 s	5	3
27	2	5	6	**62 s**	655	483
28	2	5	6	**68,3 h = 245914 s**	-	957672
37	2	6	4	**6 h = 21600 s**	467755	265063
38	2	6	4	156 h	-	-
39	2	6	5	101 d	-	-

To conclude this section, the ClassifyBalancedCAs algorithm is a new classification algorithm that is able to compete with and partly outperform state-of-the-art CA classification algorithms following a column extension strategy. One weakness of the developed algorithm is scalability with regards to the alphabet size v. Table 4.4 showed some examples where balanced-based pruning can be applied. We saw that balance-based pruning can sometimes provide a huge runtime improvement, however, it can only be applied for some cases and even when it can be applied it not always brings much of an improvement. When balance constraints are enforced, it is advantageous to use a pseudo-Boolean constraint solver, where balance constraints can be handled naturally, instead of a SAT solver where an encoding is required. At least with the sequential counter encoding used in this thesis the performance with balance constraints and SAT solving was worse than when using the tool clasp as pseudo-Boolean constraint solver.

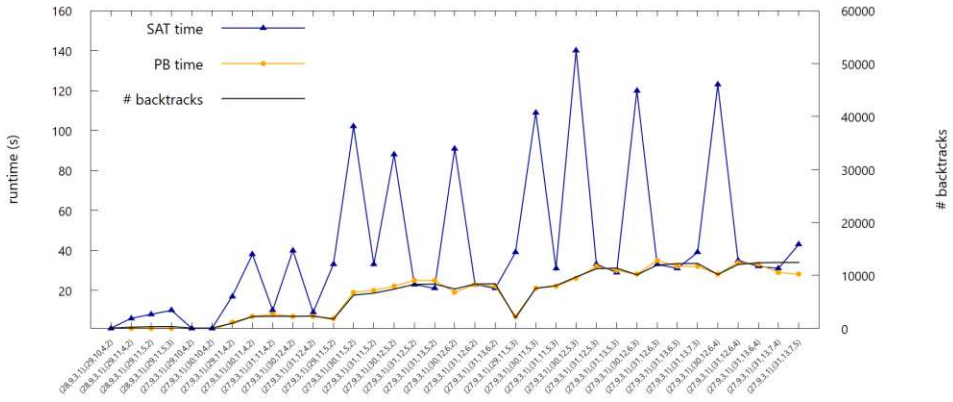


Figure 4.5: Runtime of the ClassifyBalancedCAs algorithm when classifying the instance $\mathsf{CA}^{\boldsymbol{\gamma}}_{\boldsymbol{\lambda}}(85; 4, k, 3)$ for different balance vectors. Times are given both for the ClassifyBalancedCAs variant with the SAT solver MiniSAT and for the variant with clasp as pseudo-Boolean constraints solver.

CHAPTER 5

An Adaption of the IPO Algorithm using MaxSAT

The IPO-MAXSAT algorithm presented in this section is based on the popular IPO strategy that was also mentioned in Section 3.1.3. In general, the IPO strategy consists of three parts: *Array initialization*, *horizontal extension* and *vertical extension*.

Array initialization makes use of the fact that every t-way interaction occurs at least once in a CA. For the first t columns, a $v^t \times t$ array having as rows all v^t possible t-tuples over the alphabet Σ_v is generated. Such an array is part of every CA, up to equivalence.

In the step called *horizontal extension* the array is then extended with additional columns, with the aim of reaching k columns for an input parameter k. The column for extension is usually generated heuristically such that the array resulting from extension with the new column covers a large number of t-way interactions, but not necessarily all t-way interactions.

If all t-way interactions of the current array are covered, then the IPO strategy simply continues with horizontal extension to add more columns to the array. However, if some t-way interactions are missing, the current array is not a CA anymore. This is fixed in the step called *vertical extension*. Extending the array in vertical direction, rows are added to the array until all t-way interactions are covered. Again, this usually happens via a heuristic or greedy algorithm. When adding new rows to the array, only row entries that are required to cover the missing t-way interactions are assigned a value. Other entries of the newly added rows remain unassigned (star-values). These star-values might then in later vertical extension steps be used to cover additional t-way interactions without introducing new rows.

The extension steps of the IPO strategy are displayed schematically in Figure 5.1. For a binary CA with star-values (marked in red) first a horizontal extension step is executed,

a	b	c	d	e
0	0	0	0	h_1
0	1	1	1	h_2
1	0	1	0	h_3
1	1	0	1	h_4
s_1	0	s_2	1	h_5
s_3	1	s_4	0	h_6
v_1	v_2	v_3	v_4	v_5
v_6	v_7	v_8	v_9	v_{10}

Figure 5.1: Example for extension steps of the IPO strategy with horizontal extension marked blue, vertical extension marked green and star-values surrounded with a red border.

adding a new column (marked in blue) with values $(h_1, \ldots, h_6)^T$. Afterwards a vertical extension step is executed to ensure coverage of all t-way interactions. In this step t-way interactions are covered, either by assigning values to existing star-values or by adding new rows to the array (marked in green).

When implemented using greedy algorithms, the IPO strategy can be an excellent strategy for CA generation, yielding arrays with a decently small number of rows without taking too much time. In this thesis, the IPO strategy was implemented using a different approach: Instead of getting small, non-optimal solutions very fast, a MaxSAT solver is used to compute optimal solutions for every extension step, both horizontal and vertical extension. While this worsens the runtime, smaller arrays are generated than with heuristic extension strategies. To apply a MaxSAT solver, before every extension step a weighted partial MaxSAT formula is generated and solved by the MaxSAT solver. The solution found by the solver is then used to derive a new column or row for extension. Figure 5.2 displays an example for the problem translation of horizontal extension (column extension) to a weighted partial MaxSAT formula and the derivation of a column from the model found by the MaxSAT solver.

In the following, first the MaxSAT formulation for horizontal extension is described, and afterwards the MaxSAT formulation for vertical extension. Finally, several variants of the IPO-MAXSAT algorithm are introduced and an experimental evaluation is presented.

5.1 Horizontal extension via a MaxSAT formulation

In horizontal extension an array is extended with an additional column with the goal of covering as many t-way interactions as possible. Additionally, although they are usually not considered in horizontal extension, in the formulation presented here existing star-values are taken into consideration and can be assigned if this helps to increase coverage. As secondary optimization goal, if two solutions achieve the same amount of

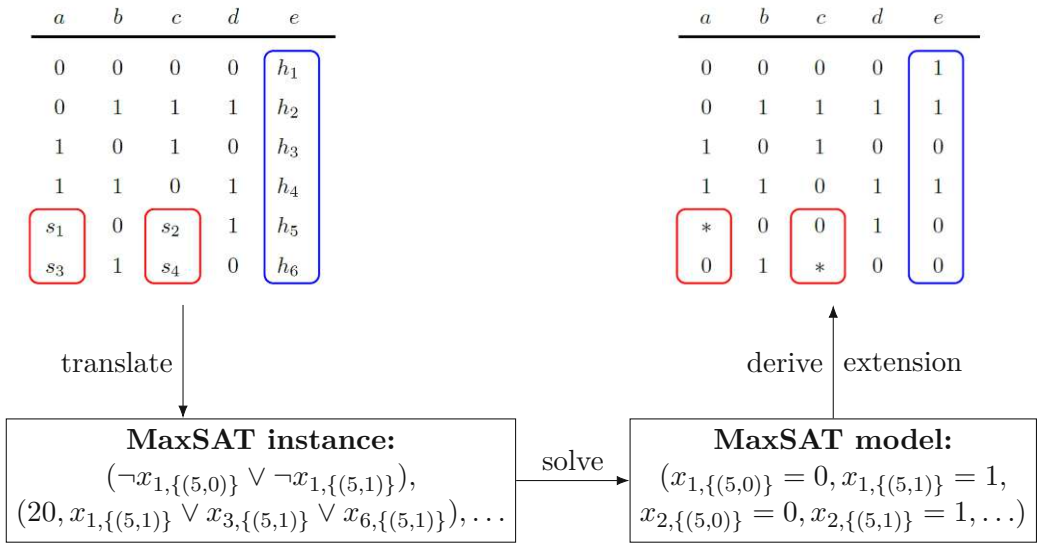


Figure 5.2: Schematics of the column extension (horizontal extension) performed by the IPO-MAXSAT algorithm.

coverage, then a solution leaving more values unassigned is preferred. In other words, a secondary goal is that the resulting array contains a maximal number of star-values that can be used for optimization of t-way coverage in later extension steps.

The formula generated for horizontal extension, called Ψ_{hor} in the following, consists of hard clauses ensuring validity of the generated models, meaning that an array assignment can be derived from every solution of Ψ_{hor}. The optimization goals are realized via weighted soft clauses in Ψ_{hor}, using a higher weight for the primary optimization goal of maximizing coverage and a lower weight for the secondary optimization goal of maximizing the amount of star-values in the resulting array. Every optimal solution of Ψ_{hor} then corresponds to an optimal array extension.

5.1.1 Variables

For the definition of variables and constraints, the set C_r^* denotes for $r = 1, \ldots, N$ the set of column indices where row r contains a star-value. Additionally, l denotes the index of the column that is added to the array next.

The solution, that is the array extension, is then derived from *single value variables* $x_{r,\{(c,u)\}}$ for $r = 1, \ldots, N$, every $c \in C_r^* \cup \{l\}$ and every $u \in \Sigma_v$, where an array entry at row r and column c is assigned a value u if the variable $x_{r,\{(c,u)\}}$ is assigned to *true* in the corresponding model of Ψ_{hor}. Note that $\{(c, u)\}$ is also a 1-way interaction. If for an array entry in a row r and column c all variables $x_{r,\{(c,u)\}}$ for $u \in \Sigma_v$ are assigned *false*, then the array entry remains unassigned and becomes a star-value. In addition to the single value variables, Ψ_{hor} also contains *value group variables* $x_{r,\tau}$

defining several values in a row at once. Such variables are defined for $r = 1, \ldots, N$ and $\tau = \{(c_1, u_1), \ldots, (c_{i-1}, u_{i-1}), (l, u_i)\}$, where τ is an i-way interaction with $2 \leq i \leq t$, $c_j \in C_r^*, u_j \in \Sigma_v$ for $j = 1, \ldots, i-1$ and $u_i \in \Sigma_v$.

Similar to single value variables, value group variables define the values of several array entries as follows: If a variable $x_{r,\{(c_1,u_1),\ldots,(c_{i-1},u_{i-1}),(l,u_i)\}}$ is assigned *true* in a model, then for $j = 1, \ldots, i-1$ the array entry at row r in column c_j is assigned the value u_j in the corresponding solution. Additionally, column l of row r is assigned the value u_i. Hard constraints are used to prevent a conflict between the assignments of single value variables and value group variables.

5.1.2 Hard constraints

Hard constraints ensure that every model of Ψ_{hor} describes a valid array assignment. For this purpose, the consistency of single value variables and value group variables is enforced by hard clauses

$$\neg x_{r,\tau} \vee x_{r,\{(c,u)\}}$$

in Ψ_{hor} for every variable $x_{r,\tau}$, where τ is an i-way interaction with $2 \leq i \leq t$ and $(c, u) \in \tau$. Additionally, hard clauses

$$\neg x_{r,\{(c,u_1)\}} \vee \neg x_{r,\{(c,u_2)\}}$$

ensure for every row $r = 1, \ldots, N$ and every column $c \in C_r^* \cup \{l\}$ that no two values $u_1, u_2 \in \Sigma_v$ are assigned to the same array cell. To avoid duplicate clauses (due to commutativity) clauses are only added for values $u_1, u_2 \in \Sigma_v$ with $u_1 < u_2$. The number of clauses created by this encoding of at most one value constraints grows quadratic in v. Although encodings with a better asymptotic growth exist, in this case a quadratic growth is acceptable because only small values of v are used. Since star-values, which are unassigned array entries, are desired, there are no clauses enforcing that every array cell takes on some value.

5.1.3 Soft constraints

Soft constraints are not required for models of Ψ_{hor} to be valid array assignments. However, they are required to find good array assignments meeting the optimization goals of horizontal extension.

The secondary maximization goal is maximization of the number of star-values. The number of star-values is maximal if the number of assigned array cells is minimal, therefore the weighted soft clauses

$$(1, \neg x_{r,\{(c,u)\}})$$

for $r = 1, \ldots, N$, $c \in C_r^* \cup \{l\}$ and $u \in \Sigma_v$ ensure that the number of star-values is maximal. A clause weight of 1 gives this optimization goal a low priority. To ensure

that coverage maximization takes precedence over star-value optimization, the number of star-value maximization clauses w is used as weight for the coverage clauses described next.

The primary and more important optimization goal is maximization of the number of covered t-way interactions. The horizontal extension of IPO is only executed when the current array is a CA, meaning that all t-way interactions in the array that is extended are already covered. Only t-way interactions of the form $\tau = \{(c_1, u_1), \ldots, (c_{t-1}, u_{t-1}), (l, u_l)\}$ containing the newly added l-th column have to be considered. Recall that for an array A with star-values and a t-way interaction τ the set $A\restriction_\tau$ denotes the index set of all rows of A where τ is covered or can be covered by assigning star-values. More specifically, $A\restriction_\tau$ contains the indices of all rows $\vec{r} = (r_1, \ldots, r_{l-1})$ of A where for every position (c, u) of τ either $r_c = u$ or r_c is a star-value. A does not have an l-th column at this point, therefore all values in the l-th column of A are considered to be star-values. Further, let $\tau\restriction_C$ denote the interaction τ restricted to columns occurring in the column set C, formally written $\tau\restriction_C = \{(c, u) \mid (c, u) \in \tau \wedge c \in C\}$. Using these notations, coverage of a t-way interaction τ is added to Ψ_{hor} as optimization goal with a clause

$$(w, \bigvee_{r \in A\restriction_\tau} x_{r,(\tau\restriction_{C_r^* \cup \{l\}})}),$$

where $w = Nv + \sum_{r=1}^{N} v|C_r^*|$ is the number of clauses for star-value maximization. Since the clauses for star-value maximization have weight 1, w corresponds to the sum of the weight of the star-value maximization clauses. While a lower weight would be sufficient (not covering an interaction τ can yield at most t additional star-values and therefore t non-violated star-value maximization clauses), the chosen weight w certainly ensures that coverage takes precedence over star-value maximization. Similar to the coverage clauses presented in Section 4.2.2, every coverage clause corresponds to one t-way interaction τ and is satisfied by a model if the induced array assignment covers the corresponding t-way interaction τ. Since there are $v^t\binom{l-1}{t-1}$ t-way interactions that need to be covered when an l-th column is added, there are $v^t\binom{l-1}{t-1}$ soft clauses for coverage maximization in the MaxSAT formulation Ψ_{hor}.

5.1.4 Number of variables and clauses

To give an overview on the size of the formula Ψ_{hor}, the numbers of variables are depicted in Table 5.1 and the numbers of clauses per constraint type are listed in Table 5.2. The total number of variables and clauses in Ψ_{hor} can be calculated by summing the values in the respective tables.

5.2 Vertical extension via a MaxSAT formulation

The coverage clauses used in horizontal extension are only soft clauses, therefore coverage of all t-way interactions is not guaranteed. Vertical extension is executed to ensure

Table 5.1: Numbers of variables in Ψ_{hor} per variable type.

Variable group	Number of variables
Single value variables $x_{r,\{(c,u)\}}$	$Nv + \sum_{r=1}^{N} \lvert C_r^* \rvert v$
Value group variables $(x_{r,\tau})$	$\sum_{r=1}^{N} \sum_{i=2}^{t} \binom{\lvert C_r^* \rvert}{i-1} v^i$

Table 5.2: Numbers of hard (h) and soft (s) clauses in Ψ_{hor} per constraint type.

Constraints	Number of clauses
Consistency of variable groups (h)	$\sum_{r=1}^{N} \sum_{i=2}^{t} \binom{\lvert C_r^* \rvert}{i-1} v^i i$
At most one value per array entry (h)	$O(\sum_{r=1}^{N} (1 + \lvert C_r^* \rvert) v^2)$
Star-value maximization (s)	$\sum_{r=1}^{N} (1 + \lvert C_r^* \rvert) v$
Coverage clauses (s)	$v^t \binom{l-1}{t-1}$

coverage of the remaining t-way interactions. This is done by adding new rows or by assigning values to star-value positions. In this section, a MaxSAT formulation Ψ_{vert} is described that can be used to derive optimal vertical extensions for an array A, to extend A to a CA. Optimality in this case means a minimal number of added rows and again, as secondary goal, maximization of the number of star-values in the extended array.

In the following, let M be the set of t-way interactions on columns of A that are not covered in A, let R be the set of row indices of all rows considered for vertical extension, those are the row indices $\{r \mid \exists \tau \in M : r \in A\restriction_\tau\}$ together with the indices of all newly added rows, and let C be the set of all columns occurring in an interaction in M, that is $C = \{c \mid \exists \tau \in M, u \in \Sigma_v : (c,u) \in \tau\}$. The developed MaxSAT formulation for vertical extension requires an upper bound on the number of newly added rows. One option is to use the trivial upper bound $|M|$, that can be derived from creating a new row for each missing interaction in M and adding those rows to A. For the vertical extension formulation Ψ_{vert} only the subarray comprised of rows in R and columns in C is considered, because other parts of A cannot be used to cover additional t-way interactions from M and are therefore not relevant for vertical extension.

The problem of vertical extension is quite similar to the problem of generating a CA, with the difference that there is a reduced set of interactions to be covered, and there are preassigned values. In both cases, the goal is to cover all required t-way interactions. It is possible to adapt SAT or MaxSAT formulations for CA generation as presented in [AIMTJ13] and [YKA+15] to derive a MaxSAT formulation for vertical extension. In [AIMTJ13] a MaxSAT formulation for *optimal* CA generation is described, where hard clauses ensure that the generated model corresponds to a CA, and soft clauses are used to minimize the number of rows of the array. In [YKA+15] optimality of the generated CAs is ensured with incremental SAT solving, meaning instead of optimization via MaxSAT, several consecutive SAT calls are used to generate an optimal CA. For adapting the formulations to vertical extension all coverage clauses relating to t-way interactions already covered in the array are omitted. Additionally, since only a subset

of the generated array needs to be assigned new values, variables corresponding to already assigned areas of the generated array can be omitted or assigned a fixed value, depending on the existing assignments. Another difference to existing formulations is the maximization of star-value occurrences.

5.2.1 Variables

The MaxSAT formulation Ψ_{vert} uses three kinds of variables. First, *value variables* $x_{r,c,u}$ for rows $r \in R$, columns $c \in C$ and values $u \in \Sigma_v$ define assignments of single values. In an array extension derived from a model of Ψ_{vert}, the array entry at row r and column c is assigned value u if $x_{r,c,u}$ is set to *true* in the corresponding assignment. Second, *coverage flags* $c_{r,\tau}$ are set to *true* in a model if the t-way interaction $\tau \in M$ is covered in a row $r \in R$. These coverage flags fulfill a similar function to the value group variables used in Ψ_{hor}, however, they are only defined for t-way interactions $\tau \in M$. Finally, *row usage flags* y_r indicate for a newly added row $r \in R$ whether it is used for vertical extension. If a flag y_r is *false* in a model of Ψ_{vert}, then no values are assigned in row r and the row r is not added to A in this vertical extension step. Such row flags have previously been used with MaxSAT in [AIMTJ13] to minimize the number of rows of the generated CAs.

5.2.2 Hard clauses

As in Ψ_{hor}, hard clauses ensure that every model of Ψ_{vert} corresponds to a valid solution for vertical extension. This means, every array entry is assigned at most one value, the assignments of different variable types are consistent with each other and all missing t-way interactions are covered when the array is extended with the derived extension.

That every array entry is assigned at most one value can be achieved with clauses

$$\neg x_{r,c,u_1} \vee \neg x_{r,c,u_2},$$

for every row $r \in R$, column $c \in C$ and $u_1, u_2 \in \{0, \ldots, v-1\}$ with $u_1 < u_2$. The condition $u_1 < u_2$ is added to avoid duplicate clauses. Again, every array entry is assigned at most one value but there is no constraint to assign at least one value to an array entries. Array entries that are not assigned any value are allowed and even desired, as these result in star-values.

The second hard constraint, that is quite similar to consistency of the two variable types in Ψ_{hor}, ensures consistency of value variables with coverage flags. If a coverage flag $x_{r,\tau}$ is set to *true*, then the value variables corresponding to row r and the entries of the t-way interaction τ need to be set to *true* as well in every model of Ψ_{vert}. For this purpose, clauses

$$\neg c_{r,\tau} \vee x_{r,c_i,u_i}, \tag{5.1}$$

are added to Ψ_{vert} for every row $r \in R$, every t-way interaction $\tau \in M$ with $\tau = \{(c_1, u_1), \ldots, (c_{t-1}, u_{t-1}), (c_t, u_t)\}$, $c_t = l$ and $i = 1, \ldots, t$.

Consistency of row and coverage flags is ensured with clauses

$$\neg c_{r,\tau} \vee y_r,$$

for every newly added row $r \in R$ and every t-way interaction $\tau \in M$. These clauses simply ensure that if a row r is used to cover a t-way interaction, then the row flag y_r is also set to *true*, such that the row r is added to A in vertical extension. This link of coverage and row flags was also presented in [YKA+15].

The array A considered for extension already has assigned values. While it is possible to propagate these values during formula generation and simplify the formula Ψ_{vert} accordingly, an easier solution is to add a singleton clause

$$x_{r,c,u}$$

to Ψ_{vert} for every row $r \in R$ and column $c \in C$, where A already has the value u assigned. Using these singleton clauses, consistency of the developed solution for vertical extension with the existing values of A is guaranteed.

In Ψ_{hor}, coverage was realized via soft clauses. In Ψ_{vert}, coverage is not optional and therefore realized with hard clauses

$$\bigvee_{r \in R} c_{r,\tau}$$

for every missing interaction $\tau \in M$. Such a clause ensures that the t-way interaction τ is covered in at least one row after vertical extension.

In vertical extension, without symmetry breaking there may exist several equivalent solutions. For example, if only n out of the $|M|$ rows considered for extension are actually required, then $\binom{|M|}{n}$ options exist to select rows to be added to A. To break this kind of symmetry, clauses

$$y_{r-1} \vee \neg y_r,$$

are added to Ψ_{vert} for every new row r, where $r-1$ is also a newly added row. While these clauses are not required for receiving a correct solution, they can speed up the search by reducing the solution space of Ψ_{vert}. With these clauses, if n rows are to be added to A, those rows will always be the first n rows out of the $|M|$ rows considered for extension. This kind of symmetry breaking was also proposed in [YKA+15].

5.2.3 Soft clauses

Similar to Ψ_{hor}, star-values are maximized in Ψ_{vert} using soft clauses with weight 1. The clauses

$$(1, \neg x_{r,c,u}),$$

are added to Ψ_{vert} for every row $r \in R$, every column $c \in C$ and every value $u \in \Sigma_v$. Using the small weight of 1 again allows to define a primary optimization goal, that is minimization of the number of used rows, using a higher weight.

Minimization of the number of used rows is done via minimization of the number of row usage flags that are set to *true*. Using clauses

$$(w, \neg y_r)$$

with weight $w = |R| \cdot |C|$ for every newly added row r, the MaxSAT solver solving Ψ_{vert} strives to minimize the number of used rows. The weight $w = |R| \cdot |C|$ ensures that this optimization goal takes precedence over maximization of the number of star-values. There cannot be more than $|R| \cdot |C|$ violated star-value maximization clauses because there are at most $|R| \cdot |C|$ array positions that may be assigned a value and due to the constraints for unique assignments at every array position, no two clauses $(1, \neg x_{r,c,u_1})$, $(1, \neg x_{r,c,u_2})$ can be violated for the same array position $(r, c), r \in R, c \in C$.

5.2.4 Number of variables and clauses

The number of variables of Ψ_{vert} per variable group is given in Table 5.3 and the number of clauses in Ψ_{vert} is given in Table 5.4, again per clause purpose. Altogether, Ψ_{vert} contains $O(|R|(|C|v + |M|))$ variables and $O(|R|(|C|v^2 + |M|t))$ clauses. Exact numbers of the generated variables and clauses are not given, as these depend on the structure of the given array A.

Table 5.3: Numbers of variables in Ψ_{vert} per variable group.

Variable group	Number of variables
Value variables ($x_{r,c,u}$)	$\|R\| \cdot \|C\|v$
Coverage flags ($c_{r,\tau}$)	$\|R\| \cdot \|M\|$
Row flags (y_r)	$\|M\|$

Table 5.4: Numbers of hard (h) and soft (s) clauses in Ψ_{vert} per constraint purpose.

Constraints	# Clauses
Validity of assignments (h)	$O(\|R\| \cdot \|C\|v^2)$
Consistency of coverage flags and values (h)	$O(\|R\| \cdot \|M\|t)$
Consistency of row and coverage flags (h)	$O(\|R\| \cdot \|M\|)$
Consistency with existing values (h)	$O(\|R\| \cdot \|C\|)$
Coverage clauses (h)	$\|M\|$
Row order (h)	$\|M\| - 1$
Star-value maximization (s)	$\|R\| \cdot \|C\|v$
Row usage minimization (s)	$\|M\|$

5.3 IPO-MAXSAT variants

In the previous sections two MaxSAT formulations were presented, one for horizontal extension (Ψ_{hor}) and one for vertical extension (Ψ_{vert}). By adapting these formulations, different variants of IPO-MAXSAT can be created. In the presented formulations, existing star-values are considered and can be assigned in both horizontal and vertical extension. However, a vertical extension step is always preceded by a horizontal extension step. This means, when star-values are considered in horizontal extension, then all t-way interactions coverable via assignment of star-values are already covered during horizontal extension, and considering star-values in both horizontal and vertical extension will not improve performance of the algorithm. It is always sufficient to consider star-values in either horizontal extension, or vertical extension. In state-of-the-art greedy implementations of the IPO strategy, star-values are usually considered only in vertical extension. Based on the above observation, a few different IPO-MAXSAT variants are defined:

- IPO-MAXSAT$_{\text{h,v*}}$: Star-values are only considered in vertical extension and ignored during horizontal extension. The formula Ψ_{hor} for horizontal extension is adapted to this variant by setting $C_r^* = \emptyset$ for every row $r \in \{1, \ldots, N\}$ in the construction of Ψ_{hor}. The formula Ψ_{vert} remains the same as described above.
- IPO-MAXSAT$_{\text{h*,v}}$: In the second variant, star-values are only considered in horizontal extension and ignored during vertical extension. The formula Ψ_{hor} is created the way it is defined above, such that star-values are considered. To ignore star-values in vertical extension, the set R that is used for the construction of Ψ_{vert} is defined to contain only newly added rows, that means R is only allowed to contain rows $r > N$. Although an adaption of Ψ_{vert} is not strictly necessary, since it will not be possible to cover more t-way interactions during vertical extension using star-values, the adaption can be used to simplify the formula Ψ_{vert} and might improve the runtime of the MaxSAT solver.
- IPO-MAXSAT$_{\text{h*,vg}}$: The third variant differs from the other two insofar that the MaxSAT formulation Ψ_{vert} for vertical extension is not used at all. Instead, a simple greedy algorithm that is commonly used in state-of-the-art IPO algorithms is employed for vertical extension. For every t-way interaction τ that needs to be covered, the greedy algorithm iterates over the given array A until a row is found where τ can be covered by assigning star-values. If such a row is found, the star-values are assigned the values required to cover τ. If τ can not be covered in any existing row, then A is extended with an additional row that covers τ and only contains star-values in positions that are not required to cover τ. For horizontal extension the formulation Ψ_{hor} is used, with star-values considered in horizontal extension.

The names of the IPO-MAXSAT variants are chosen such that the $*$ denotes where star-values are considered, in horizontal extension (h) or vertical extension (v). Additionally,

the 'g' in the final variant IPO-MAXSAT$_{\text{h*,vg}}$ denotes that a greedy algorithm is used for vertical extension.

5.4 Experimental Evaluation of IPO-MAXSAT

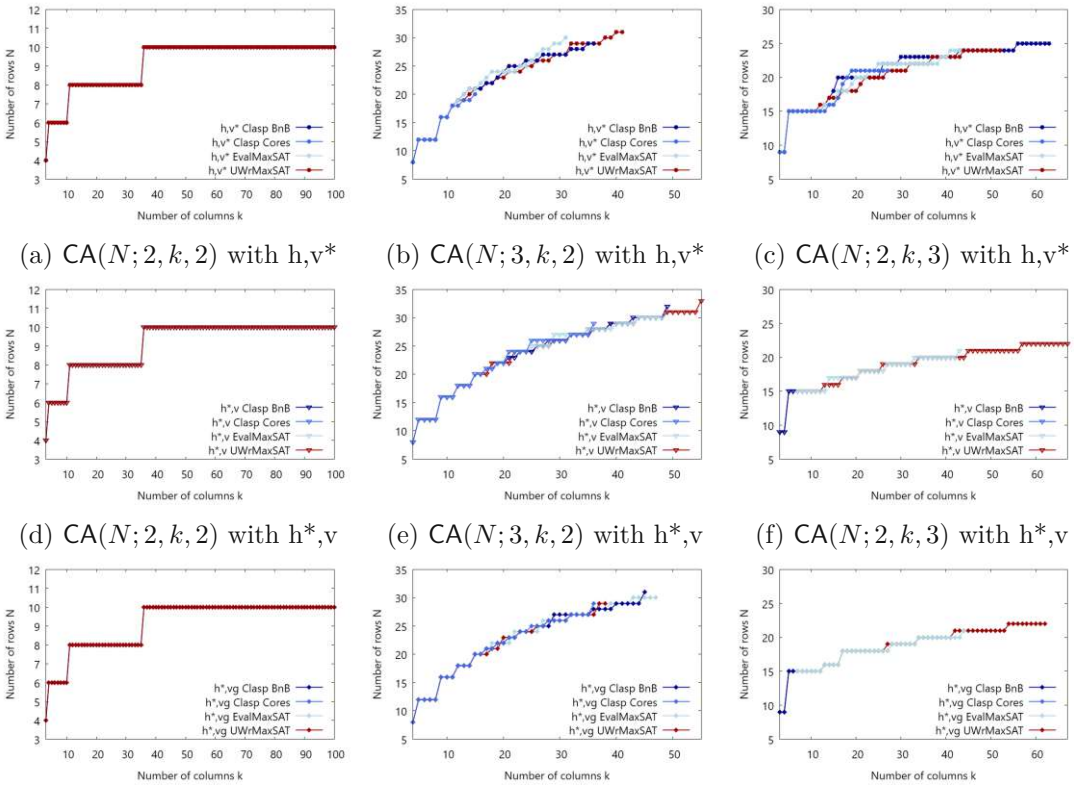
In this section an experimental evaluation and comparison of the three introduced IPO-MAXSAT variants is presented. For the evaluation four different MaxSAT solvers were used. First, for every IPO-MAXSAT variant the performance of the different MaxSAT solvers is compared. Afterwards, for every variant of IPO-MAXSAT the best-performing solver is selected and its performance is compared to state-of-the-art CA generation algorithms and bounds on CA sizes. All these experiments were carried out for three different CA instances: for generation of a $\mathsf{CA}(N; 2, k, 2)$, a $\mathsf{CA}(N; 3, k, 2)$ and a $\mathsf{CA}(N; 2, k, 3)$, with $k \leq 100$. The experiments were conducted on the same machine as the ClassifyBalancedCAs evaluation, that is a server with an AMD EPYC 7502P processor with 32 cores at 2.5 GHz base clock and 3.35 GHz boost clock and 128GB of RAM. For each computation a time limit of one hour was set. While the maximum k for which CA generation is attempted was set to 100, for several instances the algorithm reached a timeout and therefore already stopped at some $k < 100$.

5.4.1 Evaluation with different solvers

For the experiments with IPO-MAXSAT, four different solvers are used: Clasp 3.3.6 [GKS12], once with a branch-and-bound algorithm and once with an algorithm based on unsatisfiable cores. Further, the solvers EvalMaxSAT [Ave21] and UWrMaxSAT [Pio21] from the MaxSAT evaluation 2021[1] are used, which are both based on a core-guided OLL procedure. While Clasp is not a dedicated MaxSAT solver, as the experiments show it performed reasonably well when using the branch-and-bound algorithm option and even outperformed the other MaxSAT solvers on some instances.

Comparisons of the size, that is the number of rows, of the generated CAs for every IPO-MAXSAT variant and every used MaxSAT solver are displayed in Figure 5.3. Each subfigure displays the results of the four used solvers for one class of CAs and one IPO-MAXSAT variant. The subfigures in the same column display the results for one class of CAs, while the subfigures in the same row display the results for one variant of IPO-MAXSAT. To distinguish IPO-MAXSAT variants and solvers in the figures, each IPO-MAXSAT variant is depicted with a different symbol in the graphs, while the results of every solver have a unique color. The horizontal axis displays the number of generated columns, while the vertical axis displays the number of rows of the generated arrays. Each line in the graphs represents one IPO-MAXSAT execution for the generation of a $\mathsf{CA}(t, 100, v)$, where different values for t and v were used. Due to the structure of the IPO strategy, after every horizontal extension either the generated array is already a CA, or a vertical extension is executed ensuring coverage of all t-way interactions in the

[1] https://maxsat-evaluations.github.io/2021/



(a) $\mathsf{CA}(N; 2, k, 2)$ with h,v* (b) $\mathsf{CA}(N; 3, k, 2)$ with h,v* (c) $\mathsf{CA}(N; 2, k, 3)$ with h,v*

(d) $\mathsf{CA}(N; 2, k, 2)$ with h*,v (e) $\mathsf{CA}(N; 3, k, 2)$ with h*,v (f) $\mathsf{CA}(N; 2, k, 3)$ with h*,v

(g) $\mathsf{CA}(N; 2, k, 2)$ with h*,vg (h) $\mathsf{CA}(N; 3, k, 2)$ with h*,vg (i) $\mathsf{CA}(N; 2, k, 3)$ with h*,vg

Figure 5.3: Comparison of variants of IPO-MAXSAT using different MaxSAT solvers by means of three classes of CAs.

generated array. This means, when generating a $\mathsf{CA}(t, 100, v)$, then for every $2 \leq k \leq 100$ a $\mathsf{CA}(t, k, v)$ will be generated in an intermediate step. In the given graphs the size (number of rows N) of every such intermediate array is presented. The generation of a $\mathsf{CA}(t, 100, v)$ was terminated after one hour, therefore results are only given for some $k < 100$ when the considered algorithmic variant of IPO-MAXSAT did not terminate within one hour with the used MaxSAT solver.

As the figures show, the only class of CAs where a $\mathsf{CA}(t, 100, v)$ was generated is $\mathsf{CA}(2, k, 2)$ in the first column. Additionally, for this class of CAs all solvers and IPO-MAXSAT variants produce CAs of the same size. For the other classes of CAs, in the second and third column, there are small differences in the size of the generated CAs. However, as expected, there is no MaxSAT solver outperforming another with regard to size of the generated CAs. For the runtime on the other hand, there are differences based on the solver. These differences are visible when some line terminates at smaller k than others. In such a case, the execution corresponding to the line ending at a smaller k was slower and did not generate as many CAs within the given time limit. Especially the solver Clasp with the algorithm variant based on unsatisfiable cores did not perform too well.

For every other used MaxSAT solver, there is at least one instance where it outperformed the other MaxSAT solvers with regard to number of columns k generated within the time limit.

5.4.2 Comparison with the state of the art

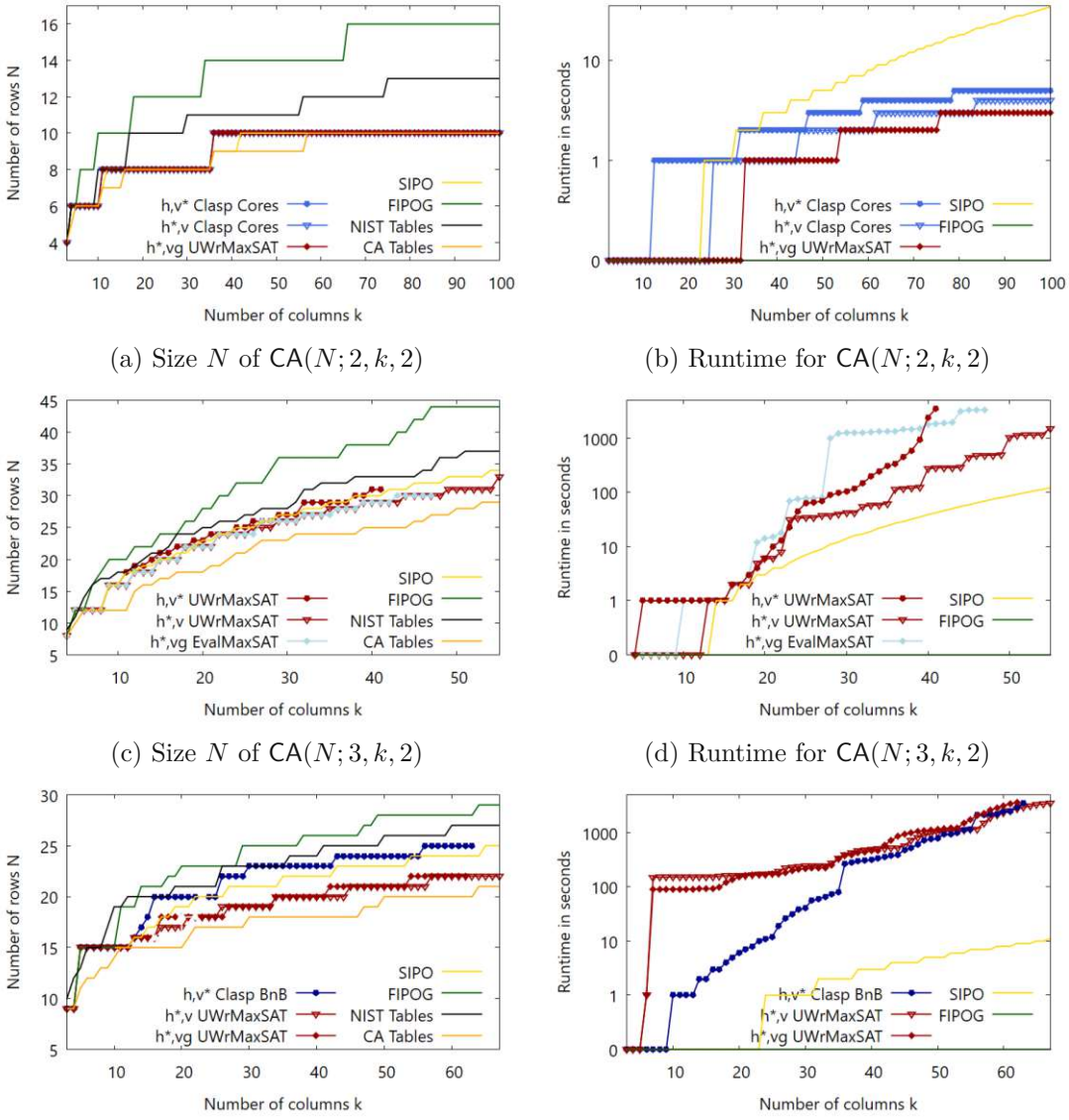
For a comparison with the state of the art for CA generation, for every IPO-MAXSAT variant the best solver (the one computing the largest number of columns) is selected per CA instance and the execution result is compared with state-of-the-art approaches and also with the best known upper bounds on CAN.

IPO-MAXSAT is compared against the following algorithms and bounds:

- **SIPO** is an IPO algorithm where Simulated Annealing is used to find better solutions for horizontal extension [WKS21]. Similar to IPO-MAXSAT, the SIPO algorithm uses improved solutions for intermediate extension steps, where improved means coverage of more t-way interactions and also occurrence of more star-values. However, Simulated Annealing is only applied to solutions for horizontal extension and the metaheuristic Simulated Annealing that is used by the SIPO algorithm does not necessarily find an optimal solution for horizontal extension. The results presented were achieved using the algorithmic parameters prooposed in [WKS21] and 10000 base iterations.
- **FIPOG** is a state-of-the-art IPO algorithm for CA generation using greedy approaches for both horizontal and vertical extension. The algorithm is described in [KS18].
- **NIST Tables** are a large online repository of CAs that is available under [Cov]. The CAs were generated with the IPOG-F algorithm proposed in [FLL+08].
- **CA Tables**: the currently best known upper bounds on CAN, collected from all currently known CA generation approaches. These bounds are available online under [Col].

In every row of Figure 5.4, for one class of CAs a comparison on the size of the generated CAs (left in Figure 5.4) and, for some approaches, runtime (right in Figure 5.4) is presented. No runtimes are given for NIST tables and CA tables, since these are online resources where no runtime is available. For CA tables a runtime also is not applicable, since some bounds were achieved via mathematical constructions or similar means.

First, the performance of the different variants of IPO-MAXSAT is compared. It can be seen that the variants IPO-MAXSAT$_{\text{h*,v}}$ and IPO-MAXSAT$_{\text{h*,vg}}$, where star-values are considered in horizontal extension, produce CAs with fewer than or the same number of rows as the variant IPO-MAXSAT$_{\text{h,v*}}$, where star-values are considered in vertical extension. While there is no difference for the generation of binary CAs of strength 2



(a) Size N of $\mathsf{CA}(N;2,k,2)$

(b) Runtime for $\mathsf{CA}(N;2,k,2)$

(c) Size N of $\mathsf{CA}(N;3,k,2)$

(d) Runtime for $\mathsf{CA}(N;3,k,2)$

(e) Size N of $\mathsf{CA}(N;2,k,3)$

(f) Runtime for $\mathsf{CA}(N;2,k,3)$

Figure 5.4: Comparison of variants of IPO-MAXSAT against state-of-the-art IPO algorithms and best known CAs.

($\mathsf{CA}(N;2,k,2)$), for $\mathsf{CA}(N;3,k,2)$ the gap between the two kinds of algorithms is already visible and even more magnified in the results for $\mathsf{CA}(N;2,k,3)$. This suggests that treating and assigning star-values during horizontal extension is beneficial for the size of the generated CAs. There is no significant difference in the size of the CAs generated by the two variants considering star-values in horizontal extension, IPO-MAXSAT$_{\text{h*,v}}$ and IPO-MAXSAT$_{\text{h*,vg}}$, although the variant IPO-MAXSAT$_{\text{h*,vg}}$ only uses a greedy

algorithm for vertical extension instead of an optimal solution like the proposed MaxSAT approach. This shows that, at least for the considered instances, finding a good solution for horizontal extension is much more important than the quality of the vertical extension solution. It can also be seen that all IPO-MAXSAT variants produce equally sized CAs for strength two with a binary alphabet ($\mathsf{CA}(2, k, 2)$). This instance is a special case, because for strength two and binary alphabet the CAN value $\mathsf{CAN}(2, k, 2)$ is known for $k \in \mathbb{N}$. The IPO-MAXSAT algorithm produced CAs with an optimal number of rows for k with $\mathsf{CAN}(2, k, 2)$ even, and a CA with one more row than necessary for $\mathsf{CAN}(2, k, 2)$ odd. This phenomenon might be caused by two 2-way interactions in the same selection of two columns missing after a horizontal extension step. One way to counteract this might be to minimize the number of missing t-way interactions sharing the same selection of columns. However, this additional optimization objective is not included in the current work.

For the runtime of the different variants, it is difficult to make conclusions due to the small number of data points and because the runtimes of every instance were collected from a single execution. However, it is visible that there is no major difference in runtime for the generation of binary CAs of strength two, where the CAs generated by different variants are of the same size. For the other two instances, $\mathsf{CA}(N; 3, k, 2)$ and $\mathsf{CA}(N; 2, k, 3)$, it can be seen that the runtime of IPO-MAXSAT$_{\text{h,v*}}$ increases steeper than the runtime of the variants where star-values are considered in horizontal extension. This might be due to the increased number of rows of the generated CAs and the therefore increased complexity of the generated MaxSAT formulations.

When comparing the sizes of the CAs generated by IPO-MAXSAT with CAs generated by the state-of-the-art approaches selected for comparison, it can be seen that even the worst variant of IPO-MAXSAT is consistently at least as good and in most cases even better than the greedy FIPOG algorithm and the IPOG-F algorithm, which is displayed as 'NIST Tables' in the graphics. The SIPO algorithm, which also considers star-values in horizontal extension, is better than the IPO-MAXSAT variant IPO-MAXSAT$_{\text{h,v*}}$ considering star-values only in vertical extension and worse than the IPO-MAXSAT variants considering star-values in horizontal extension. This is because SIPO only uses a heuristic solution instead of the exact solutions used by IPO-MAXSAT. Only for the instance $\mathsf{CAN}(2, k, 2)$ SIPO outperforms all IPO-MAXSAT variants in some cases and finds optimal CAs where IPO-MAXSAT does not. A comparison with the best currently known CAN bounds, depicted as 'CA Tables', shows that the CAs produced by IPO-MAXSAT are optimal for the instance $\mathsf{CAN}(2, k, 2)$ when the number of required rows is even and for a few small CAs of the other classes. However, in general IPO-MAXSAT does not produce optimal CAs. This means CA generation might be further improved with better optimization criteria, or a larger area of optimization. It is notable that while IPO-MAXSAT cannot compete with 'CA Tables', when comparing the difference between CA sizes generated by greedy algorithms, IPO-MAXSAT and the bounds given in 'CA Tables', as k increases the difference between the greedy algorithms and IPO-MAXSAT grows faster than the difference between IPO-MAXSAT and 'CA Tables'. This shows

that the advantage of IPO-MAXSAT with regards to optimality of the generated CAs grows with instance size. Unfortunately the runtime scalability of IPO-MAXSAT does not permit generating large CAs in reasonable time.

For the runtime results, the ranking of the compared approaches is opposite to the size ranking of generated CAs. The greedy FIPOG algorithm that produces the largest arrays has a runtime below one second. Neither the IPO-MAXSAT variants nor the SIPO algorithm can compete with FIPOG with regards to execution speed. The SIPO algorithm that produces slightly larger arrays than IPO-MAXSAT uses a considerable computation time but is faster than IPO-MAXSAT, except for the trivial instance $\mathsf{CAN}(2, k, 2)$, where the SIPO algorithm would be faster without decrease in quality if a smaller number of base iterations was used. This shows that investing more resources in finding better solutions for the IPO extension steps yields higher quality solutions, meaning smaller CAs.

To conclude, while IPO-MAXSAT does not generate new optimal CAs and is too slow to be efficiently used in practice, it provides insight in the capabilities and limitations of the popular IPO strategy and will hopefully enrich the research on better IPO algorithms. Additionally, when CAs smaller than those provided by heuristic algorithms are desired, the runtime of CA generation is not a concern and no better method for generation of the required CA instance is publicly available, the IPO-MAXSAT algorithm is a viable option.

CHAPTER 6

Conclusion and Future Work

In this thesis two new CA generation algorithms making use of exact methods were presented.

The algorithm ClassifyBalancedCAs is a classification algorithm for CAs based on column extension. The algorithm consists of two major parts: First, an exact method (SAT or pseudo-Boolean constraint solving) is used to generate columns suitable for extension. Second, a feasibility check is used to discard column extension candidates not meeting the symmetry breaking requirements. While similar classification algorithms have been used before, the application of SAT solving and pseudo-Boolean constraint solving for column generation for this type of algorithm is novel. Additionally, a faster feasibility check for complete symmetry breaking was proposed in this thesis. As the results in Section 4.4 show, for several instances the ClassifyBalancedCAs algorithm is faster than existing classification algorithms. In addition to CA classification, the ClassifyBalancedCAs algorithm is capable of classification of *balanced CAs* as introduced in [KHKS23]. When using the information from [KHKS23, Lemma 1] (repeated in this thesis in Equation (2.9)) for balance-based pruning, the proposed algorithm gains further speedup, also for classification of CAs.

The IPO-MAXSAT algorithm on the other hand is based on the IPO strategy, which starts with a small initial array and step-by-step extends it to the desired size. MaxSAT solving is applied to find optimal solutions to the occurring subproblems called horizontal extension (i.e. extension with an additional column) and vertical extension (i.e. extension with additional rows), where optimal for horizontal extension means that a maximal number of t-way interactions is covered when the solution column is combined with the current array and only required values are assigned, since unassigned values provide potential for optimizations in later extension steps. For vertical extension an array extension is optimal if it adds a minimal number of rows to the considered array. As secondary objective, again the number of assigned values is minimized. Different variants of the IPO-MAXSAT algorithm are proposed and compared in this thesis. Two major

outcomes are that handling of unassigned values during horizontal extension reduces the size of the generated CAs, and that for the considered instances an exact algorithm for vertical extension, i.e. using the developed MaxSAT formulation and an exact MaxSAT solver, does not improve the performance of IPO-MAXSAT when compared to a simple greedy algorithm, neither in terms of time nor in terms of size of the generated CAs.

Each algorithm uses exact methods for a different purpose: The ClassifyBalancedCAs algorithm applies SAT solving or pseudo-Boolean constraint solving to find all solutions to the subproblem of generating a new column that can be used for extending the considered array to a CA. Finding all solutions is required by classification algorithms. Employing an exact method does not improve the algorithm result but improves the runtime by making use of optimized solvers that exist for the exact methods in use. The IPO strategy used by the IPO-MAXSAT algorithm on the other hand is usually implemented as heuristic algorithm where at every extension step a good solution is wanted, but in general there is no need for the extension solution to be optimal. The IPO-MAXSAT algorithm uses an optimal solution instead. This improves the size of the generated CAs, however, there is a trade-off with runtime, since exact MaxSAT solving is considerably slower than the greedy algorithm that is usually employed for this subproblem in state-of-the-art CA generation algorithms based on the IPO strategy. While the IPO-MAXSAT algorithm is slower than existing IPO algorithms and does not generate CAs smaller than the currently best known CA sizes, using an exact solution for a subproblem allows to explore the capabilities of the strategy and might support the search for new strategies to find good solutions for the considered subproblem.

This thesis showed the possibility of enhancing CA generation algorithms with exact methods, however, there is still much future work to explore in this field. Other CA generation algorithms can be enhanced, and there are more exact methods that can be utilized, for example integer linear programming, CSP or satisfiability modulo theories. In addition, the algorithms presented in this thesis might be extended and/or improved. Both algorithms might be improved with different encodings for the used exact methods. While they perform reasonably well for CA instances with a binary alphabet, the performance rapidly decreases for higher alphabet sizes, especially for the IPO-MAXSAT algorithm. Better encodings might help to alleviate this problem. Additionally, the algorithms can be extended to support more variants of CAs. A straightforward extension would be the support of MCAs, where each column can have a different alphabet size. When using arrays as test sets for CT, such that array columns correspond to input parameters of the system under test, this is a necessary extension. After all, not every software parameter has the same input domain size. Another important structure that is commonly used in practice are constrained CAs. Of particular interest for the topic of this thesis is the specification of such constraints as a formula. Since exact solving is included in both algorithms presented here, it would be possible to include a formula for constraints in the encodings generated by the algorithms and therefore support constrained CAs. However, such support also comes with new challenges, especially for the ClassifyBalancedCAs algorithm, where symmetry breaking is used extensively. With constraints defined on

specific columns much of the symmetry of CAs is lost. To the best of the authors knowledge complete symmetry breaking within constrained CAs has not been studied yet. Further, while the ClassifyBalancedCAs algorithm supports the generation and classification of balanced CAs, such support is not yet available for the IPO-MAXSAT algorithm. This is mostly due to the difficulty of specifying a balance vector when the array is growing also in vertical direction. The IPO-MAXSAT algorithm always starts with an initial array with t columns and v^t rows when generating a $\mathsf{CA}(t, k, v)$. This reduces the size of the subproblems occurring during early horizontal extension. However, an array with v^t rows will not satisfy a balance vector with $\lambda_1 > v^{t-1}$. On the contrary, starting with a larger initial array fulfilling the imposed balance constraints will remove the earlier mentioned advantage of starting with small arrays when using the IPO strategy. Additionally, it is difficult to choose a meaningful balance vector when the number of rows of the generated CA is not known beforehand. Finding ways to include balance in the IPO-MAXSAT algorithm and other algorithms implementing the IPO strategy will be subject to future work.

List of Figures

List of Tables

List of Algorithms

Acronyms

AETG Automatic Efficient Test Generator. 19, 20

CA covering array. 1–3, 5–14, 19–35, 41, 43, 46–50, 52, 55, 56, 59–61, 65–73

CAN covering array number. 7, 8, 67, 69

CNF conjunctive normal form. 14–16, 31, 36, 38

CT combinatorial testing. 1, 3, 7, 8, 21, 72

DDA Deterministic Density Algorithm. 20

IPO In-Parameter-Order. 3, 22, 23, 55, 56, 59, 65, 67, 70–73, 75

LNH least number heuristic. 12, 27, 41, 42

MCA mixed covering array. 7, 8, 20, 72

OTAT one-test-at-a-time. 19

PA packing array. 8

PAN packing array number. 8

TCG Test Case Generator. 20

Bibliography

[AIMTJ13] Carlos Ansótegui, Idelfonso Izquierdo, Felip Manyà, and José Torres Jiménez. A Max-SAT-based approach to constructing optimal covering arrays. In *Artificial Intelligence Research and Development: Proceedings of the 16th International Conference of the Catalan Association for Artificial Intelligence*, pages 51–59. IOS Press, 2013.

[AMO+22] Carlos Ansótegui, Felip Manyà, Jesus Ojeda, Josep M. Salvia, and Eduard Torres. Incomplete MaxSAT approaches for combinatorial testing. *Journal of Heuristics*, 28(4):377–431, 2022.

[Ave21] Florent Avellaneda. A short description of the solver EvalMaxSAT. *MaxSAT Evaluation 2021*, pages 10–11, 2021.

[AZL12] Bestoun S. Ahmed, Kamal Z. Zamli, and Chee Peng Lim. Application of particle swarm optimization to uniform and variable strength covering array construction. *Applied Soft Computing*, 12(4):1330 – 1347, 2012.

[BC09] Renée C. Bryce and Charles J. Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability*, 19(1):37–53, 2009.

[BHvMW21] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of satisfiability*, volume 336. IOS press, 2nd edition, 2021.

[BMTI10] Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 112–126, 2010.

[CD07] Charles J. Colbourn and Jeffrey H. Dinitz. *Handbook of Combinatorial Designs*. Discrete Mathematics and Its Applications. Taylor & Francis Group, CRC Press, Boca Raton, Fla., 2nd edition, 2007.

[CDFP97] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[CGLR96] James M Crawford, Matthew L Ginsberg, Eugene M Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.

[CKRSP10] C.J. Colbourn, G. Kéri, P.P. Rivas Soriano, and J.-C. Schlage-Puchta. Covering and radius-covering arrays: Constructions and classification. *Discrete Applied Mathematics*, 158(11):1158–1180, 2010.

[Col] Charles 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 2024-01-28.

[Col04] Charles J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche*, LIX(I-II):125–172, 2004.

[Cov] Covering Arrays Team, National Institute of Standards and Technology (NIST). Covering Arrays generated by IPOG-F. Available at `https://math.nist.gov/coveringarrays/ipof/ipof-results.html`, Accessed on 2024-01-28.

[DLY+15] Feng Duan, Yu Lei, Linbin Yu, Raghu N. Kacker, and D. Richard Kuhn. 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)*, pages 1–8, 2015.

[ES04] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.

[FFH+02] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming - CP 2002*, pages 462–477, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[FLL+08] Michael Forbes, Jim Lawrence, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 113(5):287–297, 2008.

[GKS12] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.

[GMRD09] Andrew Grayland, Ian Miguel, and Colva M. Roney-Dougal. Snake lex: An alternative to double lex. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 391–399, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[HPSS06] Brahim Hnich, Steven D. Prestwich, Evgeny Selensky, and Barbara M. Smith. Constraint models for the covering test problem. *Constraints*, 11:199–219, 2006.

[HSS99] A Samad Hedayat, Neil James Alexander Sloane, and John Stufken. *Orthogonal arrays: theory and applications*. Springer Series in Statistics. Springer Science & Business Media, 1st edition, 1999.

[HWKK20] Linghuan Hu, W. Eric Wong, D. Richard Kuhn, and Raghu N. Kacker. How does combinatorial testing perform in the real world: an empirical study. *Empirical Software Engineering*, 25(4):2661–2693, 2020.

[IMTJ18] Idelfonso Izquierdo-Marquez and Jose Torres-Jimenez. New optimal covering arrays using an orderly algorithm. *Discrete Mathematics, Algorithms and Applications*, 10(01):1850011, 2018.

[KHKS23] Ludwig Kampel, Irene Hiess, Ilias S. Kotsireas, and Dimitris E. Simos. Balanced covering arrays: A classification of covering arrays and packing arrays via exact methods. *Journal of Combinatorial Designs*, 31(4):205–261, 2023.

[KKL13] D.R. Kuhn, R.N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis, 2013.

[KMN+20] Janne I. Kokkala, Karen Meagher, Reza Naserasr, Kari J. Nurmela, Patric R. J. Östergård, and Brett Stevens. On the structure of small strength-2 covering arrays. *Journal of Combinatorial Designs*, 28(1):5–24, 2020.

[KNW10] George Katsirelos, Nina Narodytska, and Toby Walsh. On the complexity and completeness of static constraints for breaking row and column symmetry. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, pages 305–320, 2010.

[KS16] Ludwig Kampel and Dimitris E. Simos. Set-based algorithms for combinatorial test set generation. In Franz Wotawa, Mihai Nica, and Natalia Kushik, editors, *Testing Software and Systems*, pages 231–240, 2016.

[KS18] Kristoffer Kleine and Dimitris E. Simos. An efficient design and implementation of the in-parameter-order algorithm. *Mathematics in Computer Science*, 12(1):51–67, 2018.

[KS19] Ludwig Kampel and Dimitris E. Simos. A survey on the state of the art of complexity problems for covering arrays. *Theoretical Computer Science*, 800:107 – 124, 2019.

[LKK+07] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 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'07)*, pages 549–556, 2007.

[LMZ16] Hai Liu, Feifei Ma, and Jian Zhang. Generating covering arrays with pseudo-boolean constraint solving and balancing heuristic. In Richard Booth and Min-Ling Zhang, editors, *PRICAI 2016: Trends in Artificial Intelligence*, pages 262–270, 2016.

[LT98] Yu Lei and 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)*, pages 254–261, 1998.

[MAT] MATRIS. Classification of balanced covering arrays. Available online `https://srd.sba-research.org/data/balca`, Accessed on 2024-01-28.

[McM02] Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, pages 250–264, 2002.

[Pio21] Marek Piotrów. Uwrmaxsat in maxsat evaluation 2021. *MaxSAT Evaluation 2021*, pages 17–18, 2021.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, 2005.

[SM02] Brett Stevens and Eric Mendelsohn. Packing arrays and packing designs. *Designs, Codes and Cryptography*, 27(1):165–176, 2002.

[SM04] Brett Stevens and Eric Mendelsohn. Packing arrays. *Theoretical Computer Science*, 321(1):125–148, 2004.

[TJAG16] Jose Torres-Jimenez and Himer Avila-George. Search-based software engineering to construct binary test-suites. In Jezreel Mejia, Mirna Munoz, Álvaro Rocha, and Jose Calvo-Manzano, editors, *Trends and Applications in Software Engineering*, pages 201–212, Cham, 2016. Springer International Publishing.

[TJIM16] Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez. Construction of non-isomorphic covering arrays. *Discrete Mathematics, Algorithms and Applications*, 08(02):1650033, 2016.

[TJIMAG19] Jose Torres-Jimenez, Idelfonso Izquierdo-Marquez, and Himer Avila-George. Methods to construct uniform covering arrays. *IEEE Access*, 7:42774–42797, 2019.

[WKS+20] M. Wagner, K. Kleine, D. E. Simos, R. Kuhn, and 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)*, pages 191–200, 2020.

[WKS21] Michael Wagner, Ludwig Kampel, and Dimitris E. Simos. Heuristically enhanced ipo algorithms for covering array generation. In *Combinatorial Algorithms*, pages 571–586. Springer International Publishing, 2021.

[YBA+16] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. Greedy combinatorial test case generation using unsatisfiable cores. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 614–624, 2016.

[YH11] Justin Yip and Pascal Van Hentenryck. Symmetry breaking via lexleader feasibility checkers. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 687–692, 2011.

[YKA+15] Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, Yutaka Oiwa, and Armin Biere. Optimization of combinatorial testing by incremental SAT solving. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[YLKK13] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn. ACTS: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375, 2013.

[YLN+13] Linbin Yu, Yu Lei, Mehra Nourozborazjany, Raghu N. Kacker, and D. Richard Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 242–251, 2013.

[YZ06] Jun Yan and Jian Zhang. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 385–394, 2006.

[YZ08] Jun Yan and Jian Zhang. A backtracking search tool for constructing combinatorial test suites. *Journal of Systems and Software*, 81(10):1681–1693, 2008. Selected papers from the 30th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, September 7–21, 2006.

[YZ11] Mohammed I Younis and Kamal Z Zamli. Mipog-an efficient t-way minimization strategy for combinatorial testing. *International Journal of Computer Theory and Engineering*, 3(3):388–397, 2011.

[YZK+10] Mohammed I. Younis, Kamal Z. Zamli, Mohammad F.J. Klaib, Zainal Hisham Che Soh, Syahrul Abdullah, and Nor Isa. Assessing irps as an efficient pairwise test data generation strategy. *International Journal of Advanced Intelligence Paradigms*, 2(1):90–104, 2010.