
INCOMPLETE MAXSAT APPROACHES FOR COMBINATORIAL TESTING

A PREPRINT

Carlos Ansótegui

Logic & Optimization Group (LOG)
University of Lleida, Lleida, Spain
carlos.ansotegui@udl.cat

Felip Manyà

Artificial Intelligence Research Institute (IIA, CSIC)
Campus UAB, 08193 Bellaterra, Spain
felip@iia.csic.es

Jesus Ojeda

Logic & Optimization Group (LOG)
University of Lleida, Lleida, Spain
jesus.ojedacontreras@udl.cat

Josep M. Salvia

Logic & Optimization Group (LOG)
University of Lleida, Lleida, Spain
josh.salvia@gmail.com

Eduard Torres

Logic & Optimization Group (LOG)
University of Lleida, Lleida, Spain
eduard.torres@udl.cat

ABSTRACT

We present a Satisfiability (SAT)-based approach for building Mixed Covering Arrays with Constraints of minimum length, referred to as the Covering Array Number problem. This problem is central in Combinatorial Testing for the detection of system failures. In particular, we show how to apply Maximum Satisfiability (MaxSAT) technology by describing efficient encodings for different classes of complete and incomplete MaxSAT solvers to compute optimal and suboptimal solutions, respectively. Similarly, we show how to solve through MaxSAT technology a closely related problem, the Tuple Number problem, which we extend to incorporate constraints. For this problem, we additionally provide a new MaxSAT-based incomplete algorithm. The extensive experimental evaluation we carry out on the available Mixed Covering Arrays with Constraints benchmarks and the comparison with state-of-the-art tools confirm the good performance of our approaches.

Keywords Combinatorial Testing · Maximum Satisfiability · Constraint Programming

1 Introduction

The Combinatorial Testing (CT) problem [42] addresses the question of how to efficiently verify the proper operation of a system, where a system can be a program, a circuit, a package that integrates several pieces of software, a GUI interface, a cloud application, etc. This problem requires to explore the parameter space of the system by iteratively testing different settings of the parameters to detect errors, bugs or faults. If we consider the system parameters as variables, a setting can be described as a *full* assignment to these parameters.

Exploring all the parameter space exhaustively, i.e., the set of all possible full assignments, is, in general, out of reach. Notice that if a system has a set of parameters P , the number of different full assignments is $\prod_{p \in P} g_p = \mathcal{O}(g^{|P|})$, where g_p is the cardinality of the domain of parameter p and g is the cardinality of the greatest domain.

The good news is that, in practice, there is no need to explore all the parameter space to detect errors, bugs or faults. We just need to *cover* a portion of the possible parameter combinations [32]. For example, most software errors (75%-80%) are caused by certain individual parameters or by the interaction of just two of them.

To cover that portion of parameter combinations exhaustively, Covering Arrays (CAs) play an important role in CT. Given a set of parameters P and a strength t , a Covering Array $CA(N; t, P)$ is a test suite of N tests that guarantee to cover all the possible interactions of t parameters (referred as t -tuples). Since executing a test in the system has a cost, we are interested in working with relatively small covering arrays. We refer to the minimum N for which a $CA(N; t, P)$ exists as the Covering Array Number, denoted by $CAN(t, P)$. In particular, we are interested in building an optimal CA, i.e., a covering array of length $CAN(t, P)$. Notice that it is guaranteed that the number of tests required to cover all t -way parameter combinations, for fixed t , grows logarithmically in the number of parameters [21], which indicates that optimal or near-optimal covering arrays can be used in practical terms. The computational challenge is to build optimal CAs in a reasonable time frame.

In this paper, we focus on *Mixed* Covering Arrays with *Constraints* (MCACs). The term *Mixed* refers to the possibility of having parameter domains of different sizes. The term *Constraints* refers to the existence of some parameter interactions that are not allowed in the system. These forbidden interactions are usually implicitly described by a set of constraints. The problem of computing an MCAC of minimum length, to which we refer in this paper as the Covering Array Number problem, is NP-hard [36].

There exist several greedy approaches that tackle the problem of building minimum MCACs, such as PICT [22], based on the OTAT framework [18], and ACTS [17], based on the IPOG algorithm [23]. One downside of these approaches is that they become more inefficient as the hardness of the set of forbidden interactions increases. Therefore, we are more interested in constraint programming approaches, which are better suited for handling constraints. For example, CALOT [47] is a tool for building MCACs based on Satisfiability (SAT) technology [16] that can handle constraints efficiently.

Within constraint programming techniques [43], SAT technology provides a highly competitive generic problem approach for solving decision problems. In particular, the decision problem to be solved is translated into a SAT instance (a propositional formula) and a SAT solver is used to determine whether there is a solution. In this paper, we will *review* in detail the CALOT tool, which essentially solves a sequence of SAT instances to compute an optimal MCAC. Each SAT instance in the sequence encodes the decision query of whether there exists an MCAC of a certain length. By iteratively bounding the length, the optimum can be determined.

Since the problem of computing minimum MCACs is, in essence, an optimization problem, we also consider its reformulation into the Maximum Satisfiability (MaxSAT) problem [16], which is an optimization version of the SAT problem.

We show empirically that MaxSAT approaches outperform ACTS and CALOT (the state-of-the-art) once the suitable MaxSAT encodings are used. We evaluate both complete or exact MaxSAT solvers (certify optimality) and incomplete MaxSAT solvers (provide suboptimal solutions). In particular, we show that while complete MaxSAT solvers perform similar to CALOT (substantially in contrast to previously reported experiments with MaxSAT solvers [47]), incomplete MaxSAT solvers obtain better suboptimal solutions and faster than ACTS and CALOT on many instances. This confirms the practical interest of incomplete MaxSAT approaches because, in real environments, we are mainly concerned with obtaining the best possible solution within a given budget.

Having confirmed the good performance of MaxSAT approaches for computing minimum MCACs, we explore another related problem, the Tuple Number (TN) Problem. Informally, the TN problem is to determine the minimum set of missing t -tuples in a test suite of N tests, or the maximum set of t -tuples that these N tests cover. In [19], this problem is studied in the context of Covering arrays with uniform domains and without constraints. In this paper, we explore (for the first time) the Mixed and with Constraints variants of the TN problem, assessing the performance of complete and incomplete MaxSAT approaches. Obviously, this problem is of interest when $N < CAN(t, P)$ ¹. We additionally present another incomplete approach based on MaxSAT technology to which we refer as MaxSAT Incremental Test Suite (Maxsat ITS), that *incrementally* builds the test suite with the help of a MaxSAT query that aims to maximize the coverage of allowed tuples at every step.

The Covering Array Number problem is concerned with reporting solutions with the least number of tests. From a practical point of view, whether we are satisfied with suboptimal solutions will depend on the cost of the tests. This cost basically includes the cost of generating the tests (computational resources) and the cost of testing the system. In particular, when the cost is too prohibitive in terms of our budget, and we are satisfied with covering a statistically significant portion of the tuples, we aim to solve (even suboptimally) the Tuple Number problem. Therefore, there exist real-world scenarios where all the approaches described in this paper are of practical interest.

¹For $N \geq CAN(t, P)$, the Tuple Number problem essentially corresponds to determine the number of allowed tuples in the corresponding MCAC problem.

The rest of the paper is structured as follows: section 2 introduces definitions on CAs, SAT/MaxSAT instances, constraints and SAT solvers. For computing MCACs of a given length, section 3 defines different SAT encodings and sections 4 and 5 describe techniques to make the SAT encodings more efficient. Section 6 introduces the incremental SAT algorithm CALOT for computing minimum MCACs. Subsequently, section 7 defines MaxSAT encodings and section 8 describes how to efficiently apply MaxSAT solvers. For the Tuple Number problem, section 9 defines a MaxSAT encoding and section 10 presents a new incomplete approach using MaxSAT solvers. To assess the impact of the presented approaches, section 11 reports on an extensive experimental investigation on the available MCAC benchmarks. Finally, section 12 concludes the paper.

2 Preliminaries

Definition 1. A System Under Test (SUT) model is a tuple $\langle P, \varphi \rangle$, where P is a finite set of variables p of finite domain, called SUT parameters, and φ is a set of constraints on P , called SUT constraints, that implicitly represents the parameterizations that the system accepts. We denote by $d(p)$ and g_p , respectively, the domain and the domain cardinality of p . For the sake of clarity, we will assume that the system accepts at least one parameterization.

In the following, we assume $S = \langle P, \varphi \rangle$ to be a SUT model. We will refer to P as S_P , and to φ as S_φ .

Definition 2. An assignment is a set of pairs (p, v) where p is a variable and v is a value of the domain of p . A test case for S is a full assignment A to the variables in S_P such that A entails S_φ (i.e. $A \models S_\varphi$). A parameter tuple of S is a subset $\pi \subseteq S_P$. A value tuple of S is a partial assignment to S_P ; in particular, we refer to a value tuple of length t as a t -tuple.

Definition 3. A t -tuple τ is forbidden if τ does not entail S_φ (i.e. $\tau \not\models S_\varphi$). Otherwise, it is allowed. We refer to the set of allowed t -tuples as $\mathcal{T}_a^{t,S} = \{\tau \mid \tau \models S_\varphi\}$, to the set of forbidden t -tuples as $\mathcal{T}_f^{t,S} = \{\tau \mid \tau \not\models S_\varphi\}$, and to the whole set of t -tuples in the SUT model S as $\mathcal{T}^{t,S} = \mathcal{T}_a \cup \mathcal{T}_f$.

When there is no ambiguity, we refer to $\mathcal{T}_a^{t,S}, \mathcal{T}_f^{t,S}, \mathcal{T}^{t,S}$ as $\mathcal{T}_a, \mathcal{T}_f, \mathcal{T}$, respectively.

Definition 4. A test case v covers a value tuple τ if both assign the same domain value to the variables in the value tuple, i.e., $v \models \tau$.

Definition 5. A Mixed Covering Array with Constraints (MCAC), denoted by $CA(N; t, S)$, is a set of N test cases for a SUT model S such that all t -tuples are at least covered by one test case. The term *Mixed* reflects that the domains of the parameters in S_P are allowed to have different cardinalities. The term *Constraints* reflects that S_φ is not empty.

Definition 6. The Covering Array Number, $CAN(t, S)$, is the minimum N for which there exists an MCAC $CA(N; t, S)$. An upper bound $ub^{CAN(t,S)}$ for $CAN(t, S)$ is an integer such that $ub^{CAN(t,S)} \geq CAN(t, S)$, and a lower bound $lb^{CAN(t,S)}$ is an integer such that $CAN(t, S) > lb^{CAN(t,S)}$.

When there is no ambiguity, we refer to $ub^{CAN(t,S)} (lb^{CAN(t,S)})$ as $ub (lb)$.

Definition 7. The Tuple Number, $T(N; t, S)$, is the maximum number of t -tuples that can be covered by a set of N tests for a SUT model S . An upper bound $ub^{T(N;t,S)}$ for $T(N; t, S)$ is an integer such that $ub^{T(N;t,S)} \geq T(N; t, S)$, and a lower bound $lb^{T(N;t,S)}$ is an integer such that $T(N; t, S) > lb^{T(N;t,S)}$.

When there is no ambiguity, we refer to $ub^{T(N;t,S)} (lb^{T(N;t,S)})$ as $ub (lb)$.

Definition 8. The MCAC problem is to find an MCAC of size N .

The Covering Array Number problem is to find an MCAC of size $CAN(t, S)$.

The Tuple Number problem is to find a test suite of size N that covers $T(N; t, S)$ t -tuples.

The MCAC problem is a decision problem. The Covering Array Number and the Tuple Number problems, to which we refer in short as the $CAN(t, S)$ and $T(N; t, S)$ problems, respectively, are optimization problems.

Definition 9. A literal is a propositional variable x or a negated propositional variable $\neg x$. A clause is a disjunction of literals. A formula in Conjunctive Normal Form (CNF) is a conjunction of clauses.

Definition 10. A weighted clause is a pair (c, w) , where c is a clause and w , its weight, is a natural number or infinity. A clause is hard if its weight is infinity (or no weight is given); otherwise, it is soft. A Weighted Partial MaxSAT instance is a multiset of weighted clauses.

Definition 11. A truth assignment for an instance ϕ is a mapping that assigns to each propositional variable in ϕ either 0 (False) or 1 (True). A truth assignment is *partial* if the mapping is not defined for all the propositional variables in ϕ .

Definition 12. A truth assignment I satisfies a literal x ($\neg x$) if I maps x to 1 (0); otherwise, it is falsified. A truth assignment I satisfies a clause if I satisfies at least one of its literals; otherwise, it is violated or falsified. The cost of a clause (c, w) under I is 0 if I satisfies the clause; otherwise, it is w . Given a partial truth assignment I , a literal or a clause is undefined if it is neither satisfied nor falsified. A clause c is a unit clause under I if c is not satisfied by I and contains exactly one undefined literal.

Definition 13. The cost of a formula ϕ under a truth assignment I , denoted by $cost(I, \phi)$, is the aggregated cost of all its clauses under I .

Definition 14. The Weighted Partial MaxSAT (WPMaXSAT) problem for an instance ϕ is to find an assignment in which the sum of weights of the falsified soft clauses is minimal (referred to as the optimal cost of ϕ) and all the hard clauses are satisfied. The Partial MaxSAT problem is the WPMaXSAT problem when all the soft clauses have the same weight. The MaxSAT problem is the Partial MaxSAT problem when there are no hard clauses. The SAT problem is the Partial MaxSAT problem when there are no soft clauses.

Definition 15. An instance of Weighted Partial MaxSAT, or any of its variants, is unsatisfiable if its optimal cost is ∞ . A SAT instance ϕ is satisfiable if there is a truth assignment I , called model, such that $cost(I, \phi) = 0$.

Definition 16. An unsatisfiable core is a subset of clauses of a SAT instance that is unsatisfiable.

Definition 17. Given a SAT instance ϕ and a partial truth assignment I , we refer as Unit Propagation, denoted by $UP(I, \phi)$, to the Boolean inference mechanism (propagator) defined as follows: Find a unit clause in ϕ under I , where l is the undefined literal. Then, propagate the unit clause, i.e. extend I with $x = 1$ ($x = 0$) if $l \equiv x$ ($l \equiv \neg x$) and repeat the process until a fixpoint is reached or a conflict is derived (i.e. a clause in ϕ is falsified by I).

We refer to $UP(I, \phi)$ simply as $UP(\phi)$ when I is empty.

Definition 18. Let A and B be SAT instances.

$A \models B$ denotes that A entails B , i.e. all assignments satisfying A also satisfy B .

It holds that $A \models B$ iff $A \wedge \neg B$ is unsatisfiable.

$A \vdash_{UP} B$ denotes that, for every clause $c \in B$, $UP(A \wedge \neg c)$ derives a conflict.

If $A \vdash_{UP} B$ then $A \models B$.

Definition 19. A *pseudo-Boolean* (PB) constraint is a Boolean function of the form $\sum_{i=1}^n q_i l_i \diamond k$, where k and the q_i are integer constants, l_i are literals, and $\diamond \in \{<, \leq, =, \geq, >\}$.

Definition 20. A Cardinality (Card) constraint is a PB constraint where all q_i are equal to 1.

Definition 21. An *At-Most-One* (AMO) constraint is a cardinality constraint of the form $\sum_{i=1}^n l_i \leq 1$.

Definition 22. An *At-Least-One* (ALO) constraint is a cardinality constraint of the form $\sum_{i=1}^n l_i \geq 1$.

Definition 23. An *Exactly-One* (EO) constraint is a cardinality constraint of the form $\sum_{i=1}^n l_i = 1$.

The interface of a modern SAT solver is presented in code fragment SATSolver. The input instance is added to the solver with functions `add_clause` and `add_retractable` (in case the clause can be retracted) (lines 5-7), which operate on a single clause, while functions `add` and `retract` operate on a set of clauses. The last two functions are overloaded to ease the usage of SAT solvers within MaxSAT solvers (lines 10-13 and 14-18). Variable `n_vars` indicates the number of variables of the input formula (line 1).

Function `solve` (lines 8-9) returns UNSAT (SAT) if the input formula is unsatisfiable (satisfiable) and sets variable `core` (`model`) to the corresponding unsatisfiable core (model). Function `assume` (line 4) allows to place an *assumption* on the truth value of a literal before function `solve` is called. Finally, modern SAT solvers also support an incremental solving mode, which allows to keep the learnt clauses across calls to the function `solve`.

3 The MCAC problem as SAT

In this section, we present the SAT encoding described in [47] to decide whether there exists a $CA(N; t, S)$ for a given SUT model $S = \langle P, \varphi \rangle$. It is similar to previous encodings described in [29, 30, 14, 40, 9].

In the following, we list the set of constraints that define the SAT encoding and describe the semantics of the propositional variables they refer to. To encode each constraint, we assume that AMO and EO cardinality constraints are translated into CNF through the regular encoding [4, 27] and the typical transformations [45] of propositional formulas into CNF are implicitly applied.

Code SATSolver: Members and functions interface

#Attributes

- 1 *n_vars* #number of variables of the formula loaded
 2 *core* #last core found
 3 *model* #last model found

#Methods

- 4 **function** *assume*(*x* : *literal*)
 | #Sets the literal *x* in the solver trail
 5 **function** *add_clause*(*c* : *clause*)
 | #Adds the clause *c* to the solver
 6 **function** *add_retractable*(*c* : *clause*)
 | #Adds the clause *c* to the retractable list of clauses of the solver
 7 **function** *retract_clause*(*c* : *clause*)
 | #Retracts the clause *c* from the solver's list of retractable clauses
 8 **function** *solve*()
 | #If formula is satisfiable, *status* ← *SAT*, *sat.model* is updated
 | #If formula is unsatisfiable, *status* ← *UNSAT*, *sat.core* is updated
 9 | **return** *status*
 10 **function** *add*(*ϕ* : *SAT formula*)
 11 | **foreach** *c_i* ∈ *ϕ* **do** *sat.add_clause*(*c_i*)
 12 **function** *retract*(*ϕ* : *SAT formula*)
 13 | **foreach** *c_i* ∈ *ϕ* **do** *sat.retract_clause*(*c_i*)
#Overloaded functions for SAT-based MaxSAT algorithms
 14 **function** *add*(*ϕ* : *Weighted Partial MaxSAT formula*)
 15 | **foreach** (*c_i*, *w_i*) ∈ *ϕ* **do**
 16 | | **if** *w_i* = ∞ **then** *sat.add_clause*(*c_i*) **else** *sat.add_retractable*(*c_i*)
 17 **function** *retract*(*ϕ* : *Weighted Partial MaxSAT formula*)
 18 | **foreach** (*c_i*, *w_i*) ∈ *ϕ* **do** **if** *w_i* ≠ ∞ **then** *sat.retract_clause*(*c_i*)
-

First, we define variables $x_{i,p,v}$ to be true iff test case i assigns value v to parameter p , and state that each parameter in each test case takes exactly one value as follows (where $[N] = \{1, \dots, N\}$):

$$\bigwedge_{i \in [N]} \bigwedge_{p \in P} \sum_{v \in d(p)} x_{i,p,v} = 1 \quad (X)$$

Second, as described in [41], in order to enforce the SUT constraints φ , for each test case i , we add the CNF formula that encodes the constraints of φ into SAT and substitute each appearance of the pair (p, v) in φ by the corresponding literal on propositional variable $x_{i,p,v}$ for each test case i .

$$\bigwedge_{i \in [N]} \text{CNF} \left(\varphi \left\{ \frac{\neg x_{i,p,v}}{p \neq v}, \frac{x_{i,p,v}}{p = v} \right\} \right) \quad (\text{SUTX})$$

Third, we introduce propositional variables c_τ^i and state that if they are true, then tuple τ must be covered at test i , by forcing the variables p in the test case to be assigned to the value specified in τ , as follows:

$$\bigwedge_{i \in [N]} \bigwedge_{\tau \in \mathcal{T}_a} \bigwedge_{(p,v) \in \tau} (c_\tau^i \rightarrow x_{i,p,v}) \quad (\text{CX})$$

Notice that only t -tuples that can be covered by a test case are encoded, i.e., $\tau \in \mathcal{T}_a$. In section 4, we discuss how to detect the t -tuples forbidden by the SUT constraints.

Finally, we state that every t -tuple $\tau \in \mathcal{T}_a$, must be covered at least by one test case, as follows:

$$\bigwedge_{\tau \in \mathcal{T}_a} \bigvee_{i \in [N]} c_\tau^i \quad (C)$$

Proposition 1. Let $Sat_{CX}^{N,t,S}$ be $X \wedge C \wedge CX \wedge SUTX$. $Sat_{CX}^{N,t,S}$ is satisfiable iff a $CA(N; t, S)$ exists.

Inspired by the incremental SAT approach in [47] (see section 6), we present another encoding where C and CX are replaced by CCX :

$$\bigwedge_{i \in [N]} \bigwedge_{\tau \in \mathcal{T}_a} \bigwedge_{(p,v) \in \tau} (c_\tau^i \rightarrow c_\tau^{i-1} \vee x_{i,p,v}) \quad (a) \text{ (CCX)}$$

$$\bigwedge_{\tau \in \mathcal{T}_a} c_\tau^N \quad (b)$$

$$\bigwedge_{\tau \in \mathcal{T}_a} (c_\tau^N \rightarrow \neg c_\tau^0) \quad (c)$$

Variables c_τ^i have now a different semantics, i.e., if they are true, τ is covered by test case i or by any lower test case j , where $1 \leq j \leq i$ (equation a). In order to guarantee that τ will be covered by some test, notice that we just need to force c_τ^N to be true and c_τ^0 to be false (variables c_τ^0 are additionally included in the encoding). This can be achieved by adding the unit clauses c_τ^N (equation b) and the implication $c_\tau^N \rightarrow \neg c_\tau^0$ (equation c) for every allowed tuple τ .

The seasoned reader may wonder why we do not simply replace equation (c) by $\bigwedge_{\tau \in \mathcal{T}_a} \neg c_\tau^0$. Indeed, this is possible. First, notice that UP on the conjunction of equations (b) and (c) will derive exactly the same. Second, for encoding some problems where it is not mandatory to cover all the tuples (see section 9), we have to erase equation (b) from CCX and also guarantee that if a tuple τ is not covered in an optimal solution, i.e., c_τ^N has to be False, then the related clauses in CCX have to be satisfied (these are hard clauses) and, if possible, to be trivially satisfied, i.e., without requiring search. Equation (c) eases this case for all the scenarios in section 9. Notice that, once c_τ^N is False, clauses in equation (c) are trivially satisfied and, by setting the remaining c_τ^i vars to True, clauses in equation (a) are also trivially satisfied.

Proposition 2. Let $Sat_{CCX}^{N,t,S}$ be $X \wedge CCX \wedge SUTX$. $Sat_{CCX}^{N,t,S}$ is satisfiable iff a $CA(N; t, S)$ exists.

Remark 1. There are some variations of equation (a) in CCX that can be beneficial when using some SAT solvers, as we will see in section 11.1. For example, we can use full implication instead of half implication in equation (a), i.e., $(c_\tau^i \leftrightarrow c_\tau^{i-1} \vee x_{i,p,v})$, or we can even use $(c_\tau^i \rightarrow c_\tau^{i-1} \vee x_{i,p,v}) \wedge (c_\tau^i \leftarrow c_\tau^{i-1})$. Also, we can consider full implication in equation (c) and, for some of the problems analyzed in section 11.1, we can even replace equation (c) by $\bigwedge_{\tau \in \mathcal{T}_a} \neg c_\tau^0$.

Example 1. As an example of SUT problem, we focus on the domain of autonomous driving. Table 1 shows the parameters and values, S_P , and the SUT constraints, S_φ :

$P \in S_P$	Abbrv.	Values
Luminosity	L	Day (dy), Night (ni)
Environment	E	Highway (hw), Urban (ur), Country (co)
Motor	M	Combustion (cb), Electric (el)
Sensor	S	Camera (ca), Radar (ra), Lidar (li)
S_φ		
$((L = ni) \wedge (E = co)) \rightarrow (S \neq ca)$		
$((E = hw) \vee (E = co)) \rightarrow (S \neq li)$		
$(M = el) \rightarrow (E = ur)$		

Table 1: Example of Autonomous Driving System Under Test.

We show how to build $Sat_{CX}^{N=10,t=2,S}$, where $N = 10$ is an upper bound ub for this SUT (see section 4 and Example 2).

To encode the X constraint, we add:

$$\begin{array}{ll}
 x_{1,L,dy} + x_{1,L,ni} = 1, & x_{1,E,hw} + x_{1,E,ur} + x_{1,E,co} = 1, \\
 x_{1,M,cb} + x_{1,M,el} = 1, & x_{1,S,ca} + x_{1,S,ra} + x_{1,S,li} = 1 \\
 & \vdots \\
 x_{10,L,dy} + x_{10,L,ni} = 1, & x_{10,E,hw} + x_{10,E,ur} + x_{10,E,co} = 1, \\
 x_{10,M,cb} + x_{10,M,el} = 1, & x_{10,S,ca} + x_{10,S,ra} + x_{10,S,li} = 1
 \end{array} \tag{Ex. X}$$

Next, for each test $(1, \dots, 10)$, we encode the SUT constraints SUTX:

$$\begin{array}{l}
 (x_{1,L,ni} \wedge x_{1,E,co}) \rightarrow \neg x_{1,S,ca} \\
 (x_{1,E,hw} \vee x_{1,E,co}) \rightarrow \neg x_{1,S,li} \\
 x_{1,M,el} \rightarrow x_{1,E,ur} \\
 \vdots \\
 (x_{10,L,ni} \wedge x_{10,E,co}) \rightarrow \neg x_{10,S,ca} \\
 (x_{10,E,hw} \vee x_{10,E,co}) \rightarrow \neg x_{10,S,li} \\
 x_{10,M,el} \rightarrow x_{10,E,ur}
 \end{array} \tag{Ex. SUTX}$$

Finally, the encoding of the CX and C constraints is shown below. We identify the set of allowed tuples, (\mathcal{T}_a) , as described in section 4. In particular, there are $|\mathcal{T}_a| = 33$ allowed tuples.

$$\begin{array}{lll}
 c_{\tau_1}^1 \rightarrow x_{1,L,dy}, & \dots, & c_{\tau_{33}}^1 \rightarrow x_{1,M,el} \\
 c_{\tau_1}^1 \rightarrow x_{1,E,hw}, & \dots, & c_{\tau_{33}}^1 \rightarrow x_{1,S,li} \\
 \vdots & \ddots & \vdots \\
 c_{\tau_1}^{10} \rightarrow x_{10,L,dy}, & \dots, & c_{\tau_{33}}^{10} \rightarrow x_{10,M,el} \\
 c_{\tau_1}^{10} \rightarrow x_{10,E,hw}, & \dots, & c_{\tau_{33}}^{10} \rightarrow x_{10,S,li}
 \end{array} \tag{Ex. CX}$$

$$(c_{\tau_1}^1 \vee c_{\tau_1}^2 \vee \dots \vee c_{\tau_1}^{10}), \quad \dots, \quad (c_{\tau_{33}}^1 \vee c_{\tau_{33}}^2 \vee \dots \vee c_{\tau_{33}}^{10}) \tag{Ex. C}$$

To build $\text{Sat}_{CCX}^{N=10,t=2,S}$, we encode the CCX constraint instead of the C and CX constraints:

$$\begin{array}{lll}
 c_{\tau_1}^1 \rightarrow c_{\tau_1}^0 \vee x_{1,L,dy}, & \dots, & c_{\tau_{33}}^1 \rightarrow c_{\tau_{33}}^0 \vee x_{1,M,el} \\
 c_{\tau_1}^1 \rightarrow c_{\tau_1}^0 \vee x_{1,E,hw}, & \dots, & c_{\tau_{33}}^1 \rightarrow c_{\tau_{33}}^0 \vee x_{1,S,li} \\
 \vdots & \ddots & \vdots
 \end{array} \tag{Ex. CCX a}$$

$$\begin{array}{lll}
 c_{\tau_1}^{10} \rightarrow c_{\tau_1}^9 \vee x_{10,L,dy}, & \dots, & c_{\tau_{33}}^{10} \rightarrow c_{\tau_{33}}^9 \vee x_{10,M,el} \\
 c_{\tau_1}^{10} \rightarrow c_{\tau_1}^9 \vee x_{10,E,hw}, & \dots, & c_{\tau_{33}}^{10} \rightarrow c_{\tau_{33}}^9 \vee x_{10,S,li}
 \end{array} \tag{Ex. CCX b}$$

$$\begin{array}{lll}
 c_{\tau_1}^{10} \rightarrow \neg c_{\tau_1}^0, & \dots, & c_{\tau_{33}}^{10} \rightarrow \neg c_{\tau_{33}}^0
 \end{array} \tag{Ex. CCX c}$$

Once we run a SAT solver on any of the previous SAT instances, if there exists a $CA(10; 2, S)$, it will return a satisfying truth assignment. To recover the particular $CA(10; 2, S)$ implicitly found by the solver, we just need to check the assignment to the $x_{i,p,v}$ variables. For example, if $x_{1,L,dy}$ is True then parameter Luminosity takes value day at test 1. Table 2 shows the result of this conversion.

	L	E	S	M
t_1	dy	hw	ca	cb
t_2	ni	hw	ra	cb
t_3	ni	ur	ca	el
t_4	dy	ur	ra	cb
t_5	dy	ur	li	cb
t_6	dy	co	ca	cb
t_7	ni	co	ra	cb
t_8	dy	ur	ra	el
t_9	ni	ur	li	cb
t_{10}	ni	ur	li	el

Table 2: $CA(10; 2, S)$ for the autonomous driving SUT.

4 Preprocessing for the MCAC problem

In the context of the Covering Array Number problem, we define an upper bound ub and a lower bound lb to be integers such that $ub \geq CAN(t, S) > lb$. When $ub = lb + 1$, we can stop the search and report ub as the minimum covering array number $CAN(t, S)$.

To get an initial value for ub , we can execute a greedy approach to obtain a suboptimal $CA(N; t, S)$ and set ub to N . For example, in the experiments, we use the tool ACTS [17] that supports Mixed Covering Arrays with Constraints. Moreover, a lower ub also implies a smaller initial encoding.

Additionally, by inspecting the solution, i.e., the test cases that certify the suboptimal $CA(N; t, S)$, we can compute which tuples are not covered, the set of *forbidden* tuples, since the suboptimal $CA(N; t, S)$ guarantees to cover all allowed t -tuples.

Furthermore, let r be the maximum number of allowed t -tuples associated to any parameter tuple of length t . Then, we can set $lb = r - 1$, since these r value tuples (mutually exclusive) need to be covered by different test cases.

Algorithm ForbiddenTuples: Detection of forbidden tuples.

```

1 Input: SUT model  $S$ , SAT solver  $sat$ 
2  $sat.add(Sat_{CX}^{N=1,t,S}[X, SUTX])$ 
3  $\mathcal{T}_f = \emptyset$ 
4 for  $\tau \in \mathcal{T}$  do
5     for  $(p, v) \in \tau$  do
6          $sat.assume(x_{1,p,v})$ 
7     if  $sat.solve() = UNSAT$  then  $\mathcal{T}_f \leftarrow \mathcal{T}_f \cup \tau$ 
8 return  $\mathcal{T}_f$ 

```

Using an approach like ACTS, not based on constraint programming techniques, has a drawback. It may not be efficient enough if testing the satisfiability of ϕ (the set of SUT constraints) is computationally hard. In this case, to detect the forbidden tuples, we can simply apply algorithm ForbiddenTuples. This algorithm tests, for every tuple τ (lines 4-7), if it is compatible with the SUT constraints (line 2) through a SAT query; if the solver results in unsatisfiability (line 7), the tuple is added to the set of forbidden tuples \mathcal{T}_f , which is ultimately returned by the algorithm (line 8).

For $t = 2$, which is already of practical importance [32], the experiments carried out in this paper show that this detection process is negligible runtime-wise.

Example 2. We show the result of algorithm ForbiddenTuples applied to the autonomous driving SUT presented in Example 1. It yields the following forbidden tuples \mathcal{T}_f for $t = 2$:

$$\mathcal{T}_f = \{(E = co, M = el), (E = hw, S = li), (E = hw, M = el), (E = co, S = li)\}$$

Then, the first parameter tuple with more allowed tuples, according to ForbiddenTuples, would be (E, S) . It has 7 allowed tuples, implying that $lb = 6$.

5 Symmetry Breaking for the MCAC problem

As [47], we fix the r t -tuples that conducted us to set the initial lb (see section 4) to test cases $\{1, \dots, r\}$. This helps us break row symmetries for the first r test cases. We will refer to this as fixed-tuple symmetry breaking.

There are other alternatives. We can impose row symmetry breaking constraints as [25]; since each row (test) represents a number in base 2, we can add constraints to order the tests in monotonic increasing order, from test 0 to test $N - 1$. We can also apply, as explained above, fixed-tuple symmetry breaking to the first r tuples (first partition) and apply row symmetry breaking constraints to the remaining $ub - lb + 1$ test cases (second partition). Furthermore, we can impose an order among the tuples in the first partition and the second partition, so that if two sets share the same value for the fixed tuple, then the one representing the lower number must be in the first partition.

Our experimental analysis shows that fixed-tuple symmetry breaking is superior to any other of the mentioned alternatives. For lack of space, we restricted all the experiments to this symmetry breaking approach.

Example 3. *We show how to apply symmetry breaking to the SUT in Example 1.*

Recall that (E, S) was the parameter tuple with the largest number of allowed tuples we selected. The set of allowed value tuples is: $\{\tau_1 = (E = hw, S = ca), \tau_2 = (E = hw, S = ra), \tau_3 = (E = ur, S = ca), \tau_4 = (E = ur, S = ra), \tau_5 = (E = ur, S = li), \tau_6 = (E = co, S = ca), \tau_7 = (E = co, S = ra)\}$.

To apply the fixed-tuple symmetry breaking variant, we just need to fix each allowed value tuple in a different test as shown below:

$$\begin{aligned}
 &x_{1,E,hw} \wedge x_{1,S,ca} \\
 &x_{2,E,hw} \wedge x_{2,S,ra} \\
 &x_{3,E,ur} \wedge x_{3,S,ca} \\
 &x_{4,E,ur} \wedge x_{4,S,ra} \\
 &x_{5,E,ur} \wedge x_{5,S,li} \\
 &x_{6,E,co} \wedge x_{6,S,ca} \\
 &x_{7,E,co} \wedge x_{7,S,ra}
 \end{aligned}
 \tag{Ex. SYM X}$$

6 Solving the $CAN(t, S)$ problem with Incremental SAT

In this section, we present the CALOT algorithm, which is an incremental SAT approach for computing optimal covering arrays with SUT constraints described by [47]. The input to the algorithm is an upper bound ub (computed as in section 4), the strength t and the SUT model S . In line 2, the incremental SAT solver is initialized with the SAT instance $Sat_{CCX}^{N=ub,t,S}$. Additionally, breaking symmetries for the first $lb + 1$ tuples, as described in section 5, are added to the SAT solver. The output is the covering array number and an optimal model.

Algorithm CALOT: Algorithm 2 in [47]

```

1 Input: upper bound  $ub$ , strength  $t$ , SUT model  $S$ 
2  $sat.add(Sat_{CCX}^{N=ub,t,S})$ 
3 Fix  $lb + 1$  value tuples to break symmetries (see Section 5)
4  $b_{model} \leftarrow \emptyset$ 
5 for  $i = N, \dots, lb + 1$  do
6     if  $sat.solve() = UNSAT$  then return  $(i, b_{model})$ 
7      $sat.add(\bigwedge_{\tau \in \mathcal{T}} c_{\tau}^{i-1})$ 
8      $b_{model} \leftarrow sat.model$ 
9     for  $\tau \in \mathcal{T}_a$  do
10        for  $(p, v) \in \tau$  do  $b_{model}[x_{i,p,v}] ? sat.add(\{x_{i,p,v}\}) : sat.add(\{-x_{i,p,v}\})$ 
11 return  $(lb + 1, b_{model})$ 
    
```

The algorithm works in a top-down search manner by iteratively decreasing the ub till it reaches $lb + 1$ (line 5) or the current SAT instance is unsatisfiable (line 6). To decrease the ub by one, the algorithm adds the set of unit clauses $\bigwedge_{\tau \in \mathcal{T}_a} c_{\tau}^{i-1}$ (line 7), which state that every t -tuple is covered by a test case with an index smaller than i .

There is a subtle detail in lines 9 and 10. Whenever the algorithm finds a new upper bound, variables $x_{i,p,v}$ related to the previous upper bound are fixed to the value in the last model found (b_{model} in line 8), so that these variables do not need to be decided in the next iterations. As [47] report, not fixing these variables can have some negative impact on the performance.

Remark 2. The original [47]’s algorithm pseudocode is slightly different. First, it assigns the i -th test at iteration i to the value it had in the previous model found instead of assigning the $i + 1$ -th test. This does not correspond to the description given in the text of the paper and may lead to an incomplete algorithm. Second, the set of constraints (a) (CCX), described in [47], does not set c_{τ}^N to True as we do in this paper, which makes the pseudocode perform a dummy first step that can cause to report a wrong optimum. We think that these are merely errors in the description, and we have fixed them. Since the tool CALOT is not available from the authors for reproducibility, we have tried to do our best to reproduce (or extend) the idea behind their work.

In section 8, we will see that this SAT incremental approach resembles how SAT-based MaxSAT algorithms behave [2, 38]. Actually, in contrast to [47], we show that MaxSAT technology can be effectively applied to solve Covering Arrays.

7 The $CAN(t, S)$ problem as Partial MaxSAT

[3] proposes an encoding into Partial MaxSAT to build covering arrays without constraints of minimum size. The main idea is to use an indicator variable u_i that is True iff test case i is used to build the covering array. The objective function of the optimization problem, which aims to minimize the number of variables u_i set to True, is encoded into Partial MaxSAT by adding the following set of soft clauses:

$$\bigwedge_{i \in [lb+2 \dots N]} (\neg u_i, 1) \quad (SoftU)$$

Notice that we only need to use $N - (lb + 1)$ indicator variables since we know that the covering array will have at least $lb + 1$ tests (see section 4).

To avoid symmetries, it is also enforced that if test case $i + 1$ belongs to the minimum covering array, so does the previous test case i :

$$\bigwedge_{i \in [lb+2 \dots N-1]} (u_{i+1} \rightarrow u_i) \quad (BSU)$$

Then, variables u_i are connected to variables c_τ^i , expressing that if we want test i to be the proof that τ is covered, then test i must be in the optimal solution ²:

$$\bigwedge_{i \in [lb+2 \dots N]} \bigwedge_{\tau \in \mathcal{T}_a} (c_\tau^i \rightarrow u_i) \quad (CU)$$

Proposition 3. Let $PMSat_{CX}^{N,t,S,lb}$ be $SoftU \wedge BSU \wedge CU \wedge Sat_{CX}^{N,t,S}$. If $N \geq CAN(t, S)$, the optimal cost of the Partial MaxSAT instance $PMSat_{CX}^{N,t,S,lb}$ is $CAN(t, S) - (lb + 1)$, otherwise it is ∞ .

In order to build the Partial MaxSAT version of $Sat_{CX}^{N,t,S}$, we just need to change how variables u_i are related to variables c_τ^i . This constraint reflects that if u_i is False (i.e., test i is not in the solution and, therefore, due to constraint BSU , none of the tests $> i$ cannot be in the solution either), then the tuple τ has to be covered at some test below i :

$$\bigwedge_{i \in [lb+2 \dots N]} \bigwedge_{\tau \in \mathcal{T}_a} (\neg c_\tau^{i-1} \rightarrow u_i) \quad (CCU)$$

Proposition 4. Let $PMSat_{CX}^{N,t,S,lb}$ be $SoftU \wedge BSU \wedge CCU \wedge Sat_{CX}^{N,t,S}$. If $N \geq CAN(t, S)$, the optimal cost of the Partial MaxSAT instance $PMSat_{CX}^{N,t,S,lb}$ is $CAN(t, S) - (lb + 1)$, otherwise it is ∞ .

Remark 3. In [3], variables u_i are instead connected to variables $x_{i,p,v}$ in the following way:

$$\bigwedge_{i \in [N]} \bigwedge_{p \in P} (u_i \leftrightarrow \bigvee_{v \in d(p)} x_{i,p,v}) \quad (XU)$$

This is a more compact encoding but it requires equation X to use an AMO constraint instead of an EO constraint.

Finally, we can convert these Partial MaxSAT instances into Weighted Partial MaxSAT modifying $SoftU$ as follows:

$$\bigwedge_{i \in [lb+2 \dots N]} (\neg u_i, w_i) \quad (WSoftU)$$

If we use $w_i = 2^{i-(lb+2)}$ we naturally introduce a lexicographical preference in the soft constraints. This is a key detail to alter the behaviour of SAT-based MaxSAT algorithms when solving Covering Arrays. If the MaxSAT solver applies the stratified approach [5] (see for more details section 8), it suffices to use $w_i = i - (lb + 2) + 1$, i.e., to increase the weights linearly. This is of interest since a high number of tests in $WSoftU$ can result into too large weights for some MaxSAT solvers.

²Notice that τ could be covered by other tests but the respective c_τ^i vars be False.

Proposition 5. Let $WPMSat_{CCX}^{N,t,S,lb}$ be $WSoftU \wedge BSU \wedge CCU \wedge Sat_{CCX}^{N,t,S}$.

If $N \geq CAN(t, S)$ and $w_i = 2^{i-(lb+2)}$ the optimal cost of the Partial MaxSAT instance $PMSat_{CCX}^{N,t,S,lb}$ is $2^{CAN(t,S)-(lb+1)} - 1$, otherwise it is ∞ .

If $N \geq CAN(t, S)$ and $w_i = i - (lb + 2) + 1$ the optimal cost of the Partial MaxSAT instance $PMSat_{CCX}^{N,t,S,lb}$ is $(1 + n) \cdot n/2$ where $n = CAN(t, S) - (lb + 1)$, otherwise it is ∞ .

Example 4. We extend our working example to obtain the Partial MaxSAT and Weighted Partial MaxSAT encodings described in this section. We first describe how we encode *SoftU* (left) and *BSU* (right) constraints:

$$\begin{array}{ll} (\neg u_{10}, 1) & \\ (\neg u_9, 1) & u_{10} \rightarrow u_9 \\ (\neg u_8, 1) & u_9 \rightarrow u_8 \end{array} \quad (\text{Ex. SoftU and BSU})$$

Recall that in our example $ub = 10$ and $lb = 6$. Therefore, we will have $N - (lb + 1) = 10 - (6 + 1) = 3$ u_i indicator variables.

To build the $PMSat_{CCX}^{N=10,t=2,S,lb=6}$ instance we add to $Sat_{CCX}^{N=10,t=2,S}$ the CU constraint:

$$\begin{array}{ll} c_{\tau_1}^{10} \rightarrow u_{10}, \dots, c_{\tau_{33}}^{10} \rightarrow u_{10} & \\ c_{\tau_1}^9 \rightarrow u_9, \dots, c_{\tau_{33}}^9 \rightarrow u_9 & \\ c_{\tau_1}^8 \rightarrow u_8, \dots, c_{\tau_{33}}^8 \rightarrow u_8 & \end{array} \quad (\text{Ex. CU})$$

To build $PMSat_{CCX}^{N=10,t=2,S,lb=6}$ we add to $Sat_{CCX}^{N=10,t=2,S}$ the CCU constraint:

$$\begin{array}{ll} \neg c_{\tau_1}^9 \rightarrow u_{10}, \dots, \neg c_{\tau_{33}}^9 \rightarrow u_{10} & \\ \neg c_{\tau_1}^8 \rightarrow u_9, \dots, \neg c_{\tau_{33}}^8 \rightarrow u_9 & \\ \neg c_{\tau_1}^7 \rightarrow u_8, \dots, \neg c_{\tau_{33}}^7 \rightarrow u_8 & \end{array} \quad (\text{Ex. CCU})$$

The weighted counterparts, $WPMSat_{CCX}^{N=10,t=2,S,lb=6}$ and $WPMSat_{CCX}^{N=10,t=2,S,lb=6}$, need only to replace *SoftU* by *WSoftU* (using $w_i = i - (lb + 2) + 1$), as follows:

$$\begin{array}{ll} (\neg u_{10}, 3) & \\ (\neg u_9, 2) & \\ (\neg u_8, 1) & \end{array} \quad (\text{Ex. WSoftU})$$

In order to build the resulting MCAC from the MaxSAT solver truth assignment, we will discard the $x_{i,p,v}$ vars whose corresponding u_i is assigned to False (i.e. test i does not belong to the solution), and proceed as in Example 1.

8 Solving the $CAN(t, S)$ problem with MaxSAT

In this section, we show that SAT-based MaxSAT approaches can simulate the CALOT algorithm, while the opposite is not true. This is an interesting insight since the MaxSAT approach additionally provides the option of applying a plethora of MaxSAT algorithms.

Let us first introduce a short description of SAT-based MaxSAT algorithms. For further details, please consult [2, 38]. Roughly speaking, SAT-based MaxSAT algorithms proceed by reformulating the MaxSAT optimization problem into a sequence of SAT decision problems. Each SAT instance of the sequence encodes whether there exists an assignment to the MaxSAT instance with a cost less than or equal to a certain k . SAT instances with a k less than the optimal cost are unsatisfiable, while the others are satisfiable. The SAT solver is executed in incremental mode to keep the clauses learnt at each iteration over the sequence of SAT instances. Thus, SAT-based MaxSAT can also be viewed as a particular application of incremental SAT solving.

There are two main types of SAT-based MaxSAT solvers: (i) model-guided and (ii) core-guided. The first ones iteratively refine (decrease) the upper bound and guide the search with satisfying assignments (models) obtained from satisfiable SAT instances. The second ones iteratively refine (increase) the lower bound and guide the search with the unsatisfiable cores obtained from unsatisfiable SAT instances. Both have strengths and weaknesses, and hybrid approaches exist [8, 7].

8.1 The Linear MaxSAT Algorithm

The Linear algorithm [24, 33], described in Algorithm Linear, is a model-guided algorithm for WPMMaxSAT. Let $\phi = \phi_s \cup \phi_h$ (line 1) be the input WPMMaxSAT instance, where ϕ_s (ϕ_h) is the set of soft (hard) clauses in ϕ .

Algorithm Linear: Linear SAT-based algorithm

```

1 Input: Weighted Partial MaxSAT formula  $\phi \equiv \phi_s \cup \phi_h$ , SAT solver sat
2 sat.add( $\phi_h$ )
3 sat.add( $\{c_i \vee b_i \mid (c_i, w_i) \in \phi_s\}$ )
4  $ub \leftarrow \sum_{(c_i, w_i) \in \phi_s} w_i + 1$ 
5  $pb \leftarrow \sum_{(c_i, w_i) \in \phi_s} w_i \cdot b_i \leq ub - 1$ 
6 sat.add(pb.to_cnf)
7 while True do
8   if sat.solve() = UNSAT then return (ub, sat.model)
9    $ub \leftarrow \sum_{(c_i, w_i) \in \phi_s} w_i \cdot \textit{sat.model}[b_i]$ 
10  sat.add(pb.update( $ub - 1$ ))
    
```

At each iteration of the Linear algorithm, the SAT instance solved by the incremental SAT solver is composed of: (i) the hard clauses ϕ_h (line 2), which guarantee that any possible solution is a *feasible* solution; (ii) the reification of each soft clause $(c_i, w_i) \in \phi_s$ into clause $(c_i \vee b_i)$, where b_i is a fresh auxiliary variable which acts as a collector of the truth value of the soft clause (line 3); and (iii) the CNF translation of the PB constraint $\sum_{(c_i, w_i) \in \phi_s} w_i \cdot b_i \leq k$, where $k = ub - 1$ bounds the aggregated cost of the falsified soft clauses, i.e., the value of the objective function.

Initially, ub is set to $(\sum_{(c_i, w_i) \in \phi_s} w_i + 1)$ (line 4), that is semantically equivalent to ∞ . Then, iteratively, if the incremental SAT solver returns satisfiable, ub is updated to $(\sum_{(c_i, w_i) \in \phi_s} w_i \cdot \textit{sat.model}[b_i])$ (line 9)³; otherwise, ub is the optimal cost (line 8). If the input instance is unsatisfiable the algorithm returns $\sum_{(c_i, w_i) \in \phi_s} w_i + 1$ (i.e., ∞).

A technical point to mention is that the PB constraint is translated into SAT thanks to an incremental PB encoding (line 5) so that whenever we tighten the upper bound, instead of retracting the original PB constraint and encode the new one, we just need to add some additional clauses (line 10). Additionally, if all the weights in the soft clauses are equal, instead of using an incremental PB encoding, we can use an incremental cardinality encoding for which more efficient encodings do exist.

Proposition 6. The Linear algorithm with Weighted Partial MaxSAT instance $WPM\textit{Sat}_{CCX}^{N,t,S,lb}$ as input can simulate the CALOT algorithm (excluding lines 9 and 10).

The key point establishing the connection of the Linear algorithm with the CALOT algorithm is to show that, given the same upper bound k to both algorithms, the Linear algorithm can propagate the same set of c_τ^{i-1} variables (line 7 in algorithm CALOT).

Let us recall that the Linear algorithm, with input $\phi \equiv WPM\textit{Sat}_{CCX}^{N,t,S,lb}$, will generate a sequence of SAT instances composed of the original hard clauses ϕ_h , the reification of the soft clauses $\bigwedge_{(c_i, w_i) \in \phi_s} (c_i \vee b_i)$, the translation to CNF of the PB constraint $\sum_{(c_i, w_i) \in \phi_s} w_i \cdot b_i \leq k$, where (c_i, w_i) represents the i -th soft clause in $WPM\textit{Sat}_{CCX}^{N,t,S,lb}$, i.e., $(\neg u_i, 2^{i-(lb+2)})$ when using the exponential increase, and the current upper bound k .

Proposition 7. If $\phi \equiv WPM\textit{Sat}_{CCX}^{N,t,S,lb}$, then

$$CCU \wedge \bigwedge_{(\neg u_i, 2^{i-(lb+2)}) \in \phi_s} (\neg u_i \vee b_i) \wedge \sum_{(\neg u_i, 2^{i-(lb+2)}) \in \phi_s} 2^{i-(lb+2)} \cdot b_i \leq k \vdash_{UP} \bigwedge_{k < i \leq N+1} \bigwedge_{\tau \in \mathcal{T}_a} c_\tau^{i-1}.$$

³*sat.model*[b_i] is 1 if b_i is assigned to True in the model, otherwise it is 0.

First of all, notice that the weight of a higher index test is strictly greater than the aggregated weights of the lower index tests. Given an upper bound k , an *efficient* CNF translation of the PB constraint will allow Unit Propagation (UP) to derive that all b s associated with soft clauses with a weight greater than k must be False. Then, from the set of clauses that reify the soft clauses (of the form $\neg u_i \vee b_i$), UP will also derive that the corresponding u_i vars must be False and, from the set of hard clauses CCU , UP will derive that the corresponding c_τ^{i-1} must be true.

If the input problem is a Partial MaxSAT instance, i.e., $PMSat_{CCX}^{N,t,S,lb}$ where the i -th soft clause is of the form $(\neg u_i, 1)$, the Linear algorithm uses a cardinality constraint instead of a PB constraint to bound the aggregated cost of the falsified soft clauses. In this case, we can only guarantee that $CCU \wedge \bigwedge_{(\neg u_i, 1) \in \phi_s} (\neg u_i \vee b_i) \wedge \sum_{(\neg u_i, 1) \in \phi_s} b_i \leq k \models \bigwedge_{k < i \leq N+1} \bigwedge_{\tau \in \mathcal{T}_a} c_\tau^{i-1}$. Notice that, given an upper bound k , UP cannot derive on $\sum_{(\neg u_i, 1) \in \phi_s} b_i \leq k$ the set of b_i s that must be False, because all correspond to soft clauses of equal weight.

CALOT algorithm cannot simulate the Linear Algorithm: While the CALOT algorithm decreases the upper bound by one at each iteration, the Linear algorithm can decrease it more aggressively. This is the case when it finds a model with a lower cost than $k - 1$ (line 9), which can significantly reduce the number of calls to the SAT solver.

8.2 The WPM1 MaxSAT algorithm

The Fu&Malik algorithm [26] is a core-guided SAT-based MaxSAT algorithm for Partial MaxSAT instances. In contrast to the Linear algorithm, which uses the models to iteratively refine the upper bound, the Fu&Malik algorithm uses the unsatisfiable cores to refine the lower bound. In particular, the initial SAT instance φ_0 explored by the Fu&Malik algorithm is composed of the hard clauses in the input MaxSAT instance ϕ_h plus the SAT clauses c_i extracted from the soft clauses (c_i, w_i) . We refer to these c_i clauses as soft-indicator clauses.

At each iteration, if φ_k is satisfiable, the optimum is k . If φ_k is unsatisfiable, the clauses in the unsatisfiable core retrieved by the SAT solver are analyzed. If none of the clauses is a soft-indicator clause, the Partial MaxSAT formula is declared unsatisfiable and the algorithm stops. Otherwise, the core tells us that we need to relax the soft-indicator clauses, i.e., we need to violate more clauses. To construct the next instance, φ_{k+1} , each soft-indicator clause in the core of φ_k is relaxed with a fresh auxiliary variable b and a hard EO cardinality constraint is added on these new variables, indicating that at least one clause must be violated (this is what the core told us) and at most one clause is violated (this prevents jumping over the optimum).

The WPM1 algorithm [6, 37] is an extension of the Fu&Malik algorithm that solves Weighted Partial MaxSAT instances by applying the split rule for weighted clauses. In particular, we are interested in using the *Stratified* WPM1 algorithm (WPM1) [5], which clusters the input clauses according to their weights⁴. These clusters were originally named as strata in [5]. The algorithm incrementally merges the clusters solving the related subproblem until all clusters have been merged. In its simpler version, all the clauses in a cluster have the same weight (called the representative weight), and clusters are added in decreasing order respect to the representative weight, but other strategies can also be applied [5].

In the WPM1 algorithm, variable ϕ_{wk} represents the formula that is MaxSAT equivalent to the merged clusters (strata) so far, while ϕ_{re} represents the remaining weighted clauses from the original input instance ϕ . Whenever we solve to optimality the current instance ϕ_{wk} , i.e., the SAT solver returned a SAT answer in the last call (line 4) but $\phi_{re} \neq \emptyset$, function *next_stratum* updates variable ϕ_{st} to the new stratum (cluster) to be merged with ϕ_{wk} ⁵ (the working SAT instance (line 5) and variables ϕ_{wk}, ϕ_{re} are updated accordingly (line 6)). Otherwise, the SAT solver returned UNSAT in the previous call, meaning that we are still optimizing the current subproblem ϕ_{wk} and need to call the SAT solver again (line 7).

If the SAT solver returns a SAT answer and all the original clauses in ϕ have been considered, i.e. $\phi_{re} = \emptyset$, then we have optimized the input instance ϕ and return its cost and an optimal model (line 8).

If the SAT solver returns an UNSAT answer, first we analyze the unsatisfiable core returned by the SAT solver (line 10) and return the soft-indicator clauses to be relaxed in variable *to_relax*, if any; otherwise, we have certified that the set of hard clauses is unsatisfiable, i.e., we return cost ∞ and an empty model.

⁴Recall that hard clauses have weight ∞ .

⁵In [5], the first call to *next_stratum* returns the cluster of all hard clauses since their representative weight is ∞

Algorithm WPM1: Stratified WPM1

```

1 Input: Weighted Partial MaxSAT formula  $\phi$ , SAT solver sat
2  $\phi_{wk}, \phi_{re}, status \leftarrow \emptyset, \phi, SAT$ 
3 while True do
4   if  $status = SAT$  then
5      $sat.add(\phi_{st} \leftarrow next\_stratum(\phi_{wk}, \phi_{re}))$ 
6      $\phi_{wk}, \phi_{re} \leftarrow \phi_{wk} \cup \phi_{st}, \phi_{re} \setminus \phi_{st}$ 
7   if  $(status \leftarrow sat.solve()) = SAT$  then
8     if  $\phi_{re} = \emptyset$  then return  $(cost(sat.model, \phi), sat.model)$ 
9   else
10    if  $(to\_relax \leftarrow core\_analysis(\phi_{wk}, sat.core)) = \emptyset$  then return  $(\infty, \emptyset)$ 
11     $relaxed, B, residuals \leftarrow split\_and\_relax(to\_relax, sat.n\_vars)$ 
12     $sat.retract(to\_relax)$ 
13     $sat.add(\phi_{rx} \leftarrow relaxed \cup (CNF(\sum_{b \in B} b = 1), \infty))$ 
14     $\phi_{wk}, \phi_{re} \leftarrow (\phi_{wk} \setminus to\_relax) \cup \phi_{rx}, \phi_{re} \cup residuals$ 

```

Function *split_and_relax* (line 11) first applies the split rule to the soft-indicator clauses in *to_relax* and generates two sets, one where all the clauses are normalized to have the minimum weight, and another with the residuals of each clause respect to the minimum weight in *to_relax*. Second, the set of clauses with the minimum weight are extended, each with an additional fresh variable and stored in the set *relaxed* as in the Fu&Malik algorithm. The new fresh variables are returned in set *B*.

Finally, the original set of clauses *to_relax* is retracted from the SAT solver (line 12), and the new set *relaxed* is added to the working SAT instance plus the cardinality constraint that increases the lower bound as in the Fu&Malik algorithm (line 13)⁶. In line 14, ϕ_{wk} is updated to reflect the changes in the SAT working formula, and the remaining formula ϕ_{re} is extended with the residuals generated from the application of the split rule.

As a final remark, notice that if the statements in grey boxes of the WPM1 algorithm are erased and function *next_stratum* is instructed to report sequentially, first the hard clauses and then the soft clauses, we get the original Fu&Malik algorithm.

In the context of the Covering Array Number problem, the Fu&Malik algorithm on the $PM\text{Sat}_{CCX}^{N,t,S,lb}$ instance will perform a bottom-up search, i.e, the first query will correspond to the question of whether the covering array can be constructed with $k = 0$ tests, then with $k = 1$ tests, etc. This approach does not provide any intermediate upper bounds since the only query answered positively corresponds to the optimum.

However, interestingly, by considering the weighted version of the Fu&Malik algorithm, we can perform a top-down search on the Covering Array problem and provide intermediate upper bounds.

Proposition 8. The Stratified WPM1 algorithm with input $WPM\text{Sat}_{CCX}^{N,t,S,lb}$ can simulate the CALOT algorithm (excluding lines 9 and 10).

Back to the context of covering arrays, each cluster in $WPM\text{Sat}_{CCX}^{N,t,S,lb}$ would be composed of a single soft clause $(\neg u_i, w_i)$, except the cluster containing all the hard clauses. The first subproblem seen by the Stratified WPM1 algorithm encodes the query of whether one can build a covering array using N tests. The next subproblem incorporates the first soft clause $(\neg u_N, w_N)$ and encodes the query of whether one can construct the covering array using $N - 1$ tests. Notice that each $\neg u_i$ will propagate, according to *CCU*, the corresponding c_T^{i-1} vars as in the CALOT algorithm. Notice also that every solution of a subproblem is an upper bound for the covering array.

The discussion of this section has provided insights into how to solve Covering Arrays through MaxSAT, but also into how to fix similar difficulties in other problems where MaxSAT is not yet effective enough.

8.3 Test-based Streamliners for the $CAN(t, S)$ problem

Notice that a solution for a $CAN(t, S)$ problem can be extended to multiple solutions in the previous MaxSAT translations. This happens when $CAN(t, S) < N$, since the assignment to the x vars related to any test i with $i > CAN(t, S)$

⁶Notice that $(CNF(\sum_{b \in b_vars} b = 1), \infty)$ is a set of clauses that have ∞ weight.

(useless from the point of view of the $CAN(t, S)$ problem) still needs to be consistent with the X and $SUTX$ constraints. In general, notice that $SUTX$ can be NP-complete.

Lines 9 and 10 of the CALOT algorithm, as described in section 6, fix that problem but cannot directly be applied within MaxSAT algorithms since the solver is not aware of the $CAN(t, S)$ problem semantics.

However, we can reproduce a similar effect. At the preprocessing step, we can build a *dummy* test case v by computing a solution to S_φ (e.g. with a SAT solver) or select any of the test cases in the solution returned by the ACTS tool when computing the upper bound (see section 4). Then, we can state in the MaxSAT encoding that if a given test i is not part of the optimal solution (i.e., u_i is False), then the corresponding x vars are set to the value in the test case v .

$$\bigwedge_{i \in [lb+2 \dots N]} \left(\neg u_i \rightarrow \bigwedge_{(p,v) \in v} x_{i,p,v} \right) \quad (NUX)$$

The *dummy* test case v exactly plays the role of the so-called streamliner constraints [28], which rule out some of the possible solutions but make the search of the remaining solutions more efficient.

There is yet another way to mitigate that potential bottleneck. We can indeed extend $SUTX$ clauses for test i with literal $\neg u_i$. Therefore, whenever test i is no longer in the optimal solution (i.e. u_i is False), the corresponding SUT constraints are trivially satisfied. However, in the experimental investigation, we confirmed that this option is less efficient than adding NUX clauses.

Example 5. For the SUT in Example 1, let us assume that we use the following dummy test:

L	E	S	M
dy	ur	ra	cb

Then, the NUX encoding is:

$$\begin{aligned} \neg u_{10} &\rightarrow (x_{10,L,dy} \wedge x_{10,E,ur} \wedge x_{10,S,ra} \wedge x_{10,M,cb}) \\ \neg u_9 &\rightarrow (x_{9,L,dy} \wedge x_{9,E,ur} \wedge x_{9,S,ra} \wedge x_{9,M,cb}) \\ \neg u_8 &\rightarrow (x_{8,L,dy} \wedge x_{8,E,ur} \wedge x_{8,S,ra} \wedge x_{8,M,cb}) \end{aligned} \quad (\text{Ex. NUX})$$

9 The $T(N; t, S)$ problem as Weighted Partial MaxSAT

For some applications, we may not be able to use as many test cases as the covering array number (e.g. due to budget restrictions), but we may still be interested in solving the Tuple Number problem, i.e., to determine the maximum number of covered t -tuples we can get with a test suite of fixed size. This problem is also known as the Optimal Shortening Covering Arrays (OSCAR) problem. These *shortened* covering arrays (to which we refer more precisely just as *test suites* since they do not cover all t -tuples) have been used to improve the initialization of metaheuristic approaches for Covering Arrays (without SUT constraints) [19]. These metaheuristics obtain suboptimal Covering Arrays very quickly. Once again, MaxSAT technology can play an important role when SUT constraints are considered. Moreover, the size of the SAT/MaxSAT encodings for this problem are smaller than encodings for computing the Covering Array Number, since fewer tests are taken into consideration.

In the following, we show how we can modify the $Sat_{CX}^{N,t,S}$ and $Sat_{CCX}^{N,t,S}$ formulae to become Partial MaxSAT encodings of the Tuple Number problem.

The basic idea is that we need to soften the hard restriction that enforces all allowed t -tuples to be covered. To this end, we modify the SAT instance $Sat_{CX}^{N,t,S}$ as follows: First, we soften all the clauses from equation C which encode that every t -tuple τ must be covered by at least one test case, therefore allowing to violate (or relax) these constraints. For the sake of clarity, although not required for soundness, we introduce a new set of indicator variables c_τ that reify each ALO constraint in equation C by introducing the following hard constraints:

$$\bigwedge_{\tau \in \mathcal{T}_a} (c_\tau \leftrightarrow \bigvee_{i \in [N]} c_\tau^i) \quad (RC)$$

Then, we add the following set of soft clauses:

$$\bigwedge_{\tau \in \mathcal{T}_a} (c_\tau, 1). \quad (\text{SoftC})$$

Finally, we we replace in $Sat_{CCX}^{N,t,S}$ the set of constraints C (the hard constraint that forced to cover all the tuples) by the previous two sets of constraints.

Proposition 9. Let S be a SUT model and let $TPMSat_{CCX}^{N,t,S}$ be $Sat_{CCX}^{N,t,S} \left\{ \frac{\text{SoftC} \wedge RC}{C} \right\}$. The optimal cost of $TPMSat_{CCX}^{N,t,S}$ is $|\mathcal{T}_a| - T(N; t, S)$.

Remark 4. Even if $N > lb$, we cannot use fixed-tuple symmetry breaking since we do not know whether the t -tuples that we fix will lead to an optimal solution. Therefore, fixed-tuple symmetry is disabled for all the encodings in this section.

Remark 5. When computing the tuple number, we can avoid the step of detecting all forbidden tuples since the encoding remains sound, i.e., we can interchange \mathcal{T}_a by \mathcal{T} . Notice that those c_τ vars related to forbidden tuples will always be set to False. Moreover, notice that a core-guided algorithm may potentially detect easily as many unsatisfiable cores as forbidden tuples which include just the unit soft clause that represents the forbidden tuple.

In case we want to extend $Sat_{CCX}^{N,t,S}$ to compute the tuple number, we just need to notice that the previous defined role of c_τ corresponds exactly to variable c_τ^N in $Sat_{CCX}^{N,t,S}$, so we just need to soften the hard unit clauses c_τ^N (described in CCX) with weight 1.

Proposition 10. Let S be a SUT model and let $TPMSat_{CCX}^{N,t,S}$ be $Sat_{CCX}^{N,t,S} \left\{ \frac{(c_\tau^N, 1)}{(c_\tau^N)} \right\}$. The optimal cost of $TPMSat_{CCX}^{N,t,S}$ is $|\mathcal{T}_a| - T(N; t, S)$.

Example 6. We show how to build $TPMSat_{CCX}^{N=10,t=2,S}$ for the SUT in Example 1.

We must create a new variable c_τ for each value tuple in \mathcal{T}_a and then replace constraint C in $SAT_{CCX}^{N=10,t=2,S}$ (see Example 1) by RC (left). Finally, we have to add the *SoftC* soft clauses (right):

$$\begin{array}{ll} c_{\tau_1} \leftrightarrow (c_{\tau_1}^1 \vee c_{\tau_1}^2 \vee \dots \vee c_{\tau_1}^{10}) & (c_{\tau_1}, 1) \\ \vdots & \vdots \\ c_{\tau_{33}} \leftrightarrow (c_{\tau_{33}}^1 \vee c_{\tau_{33}}^2 \vee \dots \vee c_{\tau_{33}}^{10}) & (c_{\tau_{33}}, 1) \end{array} \quad (\text{Ex. RC and SoftC})$$

For the $TPMSat_{CCX}^{N=10,t=2,S}$, we just have to soften, with weight 1, the set of clauses from CCX (b) in $SAT_{CCX}^{N=10,t=2,S}$ (see Example 1).

In what follows, we present two extensions.

9.1 Combining the $CAN(t, S)$ and $T(N; t, S)$ problems

The Covering Array and Tuple Number problems can lead to think about a more general formulation of the optimization problem where we want to maximize the number of covered t -tuples while minimizing the number of test cases. Notice that it will depend on the value of N respect to the covering array number (not necessarily known a priori) whether we are, in essence, solving the covering array number or the tuple number problem.

To this end, we take the $PMSat_{CCX}^{N,t,S,lb}$ encoding of the Covering Array Number problem for a SUT model S , N tests and strength t . As earlier shown in this section, we first replace the set of hard constraints C by RC and *SoftCWU*.

$$\bigwedge_{\tau \in \mathcal{T}_a} (c_\tau, |u_i| + 1). \quad (\text{SoftCWU})$$

Notice that we prefer violating all soft clauses $(\neg u_i, 1)$ over violating a single soft clause $(c_\tau, |u_i| + 1)$. This way, we guarantee that any solution to our new Weighted Partial MaxSAT instance maximises the number of covered t -tuples while minimises the number of needed test cases.

Proposition 11. If $N \geq CAN(t, S)$, the optimal cost of the Weighted Partial MaxSAT instance $PM\text{Sat}_{CCX}^{N,t,S,lb} \left\{ \frac{\text{Soft}CWU \wedge RC}{C} \right\}$ is $CAN(t, S) - (lb + 1) + (|\mathcal{T}_a| - T(N; t, S)) \cdot (|u_i| + 1)$, otherwise it is $N - (lb + 1) + (|\mathcal{T}_a| - T(N; t, S)) \cdot (|u_i| + 1)$.⁷

The same idea can be applied to $PM\text{Sat}_{CCX}^{N,t,S,lb}$ by softening the unit hard clauses (c_τ^N) in equation (b) from CCX with weight $|u_i| + 1$. Here, it is important to recall the discussion in section 3 on the need of equation (c) in CCX . The other, perhaps more natural, alternative was to replace equation (c) in CCX by $\bigwedge_{\tau \in \mathcal{T}_a} \neg c_\tau^0$. The problem arises when, in an optimal solution, τ is not covered, what also implies that (c_τ^N) is False. Notice that we need to satisfy all clauses related to τ in CCX but, in order to do that, we need to set all c_τ^i vars to False. This may not be compatible with equation CCU (clauses of the form $\neg c_\tau^{i-1} \rightarrow u_i$) when some test i is discarded to be in the solution and variable u_i is set to False, since UP will derive in CCU that c_τ^{i-1} is True. In this case, a contradiction is reached. On the other hand, as discussed in section 3, equation (c) allows to set all c_τ^i vars to True when (c_τ^N) is False and trivially satisfy all clauses in CCX related to τ .

Proposition 12. If $N \geq CAN(t, S)$, the optimal cost of the Weighted Partial MaxSAT instance $PM\text{Sat}_{CCX}^{N,t,S,lb} \left\{ \frac{\text{Soft}CWU}{(c_\tau^N)} \right\}$ is $CAN(t, S) - (lb + 1) + (|\mathcal{T}_a| - T(N; t, S)) \cdot (|u_i| + 1)$, otherwise it is $N - (lb + 1) + (|\mathcal{T}_a| - T(N; t, S)) \cdot (|u_i| + 1)$.⁷

9.2 The $CAN(t, S)$ problem with Relaxed Tuple Ratio Coverage as MaxSAT

We can tackle other realistic settings where we still want to use the minimum number of tests, but there is no need to achieve a 100% ratio of covered t -tuples (mandatory per definition in Covering Arrays). Notice that the last tests that shape the covering array number tend to cover very few not yet covered t -tuples. Therefore, if these tests are expensive enough in our setting, we may consider relaxing the ratio coverage and skip these tests.

The mentioned problem can be encoded by replacing the previously soft constraints on the c_τ vars with a hard cardinality constraint on the minimum number of t -tuples to be covered as follows:

$$\sum_{\tau \in \mathcal{T}_a} c_\tau \geq \lceil |\mathcal{T}_a| \cdot rt \rceil \quad (CCard)$$

where rt is the ratio of allowed t -tuples that we want to cover. Notice that, for efficiency reasons, $CCard$ can be also described as $\sum_{\tau \in \mathcal{T}_a} \neg c_\tau \leq \lceil |\mathcal{T}_a| \cdot (1 - rt) \rceil$.

Remark 6. With this formulation, we cannot use the fixed-tuples symmetry breaking since we do not know whether we will require at least lb tests to cover the specified ratio of allowed t -tuples.

Proposition 13. Let $RTPMSat_{CCX}^{N,t,S,rt}$ be $PM\text{Sat}_{CCX}^{N,t,S,lb=0} \left\{ \frac{CCard}{(c_\tau^N)} \right\}$. The optimal cost of $RTPMSat_{CCX}^{N,t,S,rt}$ is the minimum N' such that $T(N', t, S) \geq \lceil |\mathcal{T}_a| \cdot rt \rceil$.

10 Incomplete MaxSAT Algorithms for the $T(N; t, S)$ problem

As argued earlier, if certifying optimality is not a requirement and we are just interested in obtaining a good suboptimal solution in a reasonable amount of time, we can apply incomplete MaxSAT algorithms on the encodings of the Tuple Number problem described in the previous section. Additionally, in this section, we present a new incomplete algorithm to compute suboptimal solutions for the Tuple Number problem.

10.1 MaxSAT based Incremental Test Suite Construction

A way to reduce the search space of any constraint problem is to add the so-called streamliner constraints [28]. We recall that these constraints rule out some of the possible solutions but make the search of the remaining solutions more efficient. However, in practice, streamliners can rule out all the solutions.

In our context, the streamliners constraints correspond to a set of tests that we think have the potential to be part of optimal solutions. By fixing these tests, we generate a new covering array problem, easier to solve, but whose Covering Array Number can be greater than or equal to that of the original covering array, because we may have missed all the

⁷Notice that if $N \geq CAN(t, S)$, then $|\mathcal{T}_a| - T(N; t, S)$ is 0. However, we keep this expression in case we want to interchange \mathcal{T}_a by \mathcal{T} (see Remark 5).

optimal solutions. We iterate this process until all t -tuples get covered. To select the k candidate test to be fixed at each iteration, we solve the Tuple Number problem restricted to length k .

In the context of the Tuple Number problem, this iterative process of fixing tests should not only finish when all t -tuples have been covered but also when the requested N tests have been fixed.

To that end, here we combine a greedy iterative approach with the SAT-based MaxSAT approaches from section 9 in the IncrementalCA algorithm.

Algorithm IncrementalCA: MaxSAT based Incremental Test Suite Construction

```

1 Input: SUT model S, Tests  $N_i$  per iteration, SAT-based MaxSAT solver msat
2  $\mathcal{T}_r, \Upsilon \leftarrow \mathcal{T}_a, \emptyset$ 
3 while  $\mathcal{T}_r \neq \emptyset$  and  $|\Upsilon| < N$  do
4    $N' \leftarrow \min(N_i, N - |\Upsilon|)$ 
5    $msat.add\left(TPM\text{Sat}_{CCX}^{N',t,S}\right)$ 
6    $msat.solve()$ 
7    $v \leftarrow \text{tests from } msat.model$ 
8    $\Upsilon \leftarrow \Upsilon \cup v$ 
9    $\mathcal{T}_r \leftarrow \mathcal{T}_r \setminus \{\tau \mid v \models \tau\}$ 
10 return  $\Upsilon$ 

```

In this algorithm, we begin with the remaining tuples to cover \mathcal{T}_r , initially assigned to allowed tuples \mathcal{T}_a , as well as an empty test suite Υ (line 2). Then, we first check how many tests should be encoded; the minimum between the tests in iteration N_i and the remaining number of tests left to complete the test suite, $N - |\Upsilon|$ (line 4), storing the result into N' . Next, we solve the Tuple Number problem for these N' tests, encoded as a $TPMSat_{CCX}^{N',t,S}$ formula (lines 5, 6) from section 9. We extract the model from the MaxSAT solver, interpreting it into newly found test cases v (line 7). Then, those new tests are added to test suite Υ (line 8). Finally, the tuples covered by these new test cases are removed from \mathcal{T}_r (line 9). This iteration is repeated until no more tuples are left in \mathcal{T}_r or we have reached the requested N test cases (line 3), in which case we return the constructed test suite Υ (line 10).

11 Experimental Evaluation

In this section, we report on an extensive experimental investigation conducted to assess the approaches proposed in the preceding sections. We start by defining the benchmarks, which include 28 industrial, real-world or real-life instances and 30 crafted instances, and the algorithms involved in the evaluation.

We contacted the authors of [47] and [46] to obtain the benchmarks used in their experiments. In particular, the available benchmarks are: (i) Cohen et al. [20], with 5 real-world and 30 artificially generated (crafted) covering array problems; (ii) Segall et al. [44], with 20 industrial instances; (iii) Yu et al. [48], with two real-life systems reported by ACTS users; and (iv) Yamada et al. [46], with an industrial instance named “Company_B”.

Table 3 provides information about the System Under Test of each instance, where S_P is the number of parameters and their domain (e.g. the meaning of $2^{29}3^1$ in instance 7 is that the instance contains 29 parameters of domain 2 and 1 parameter of domain 3); S_φ is the number of SUT constraints and their sizes (e.g. the meaning of $2^{13}3^2$ in instance 7 is that the instance contains 13 constraints of size 2 and 2 constraints of size 3); and $\# \text{ lits } CNF(S_\varphi)$ is the number of literals of the CNF representation of S_φ (i.e. the sum of the sizes of all clauses).

Table 3 also reports, for $t = 2$, the following data: ub^{ACTS} , which indicates the upper bound returned by the ACTS tool (see section 4); ub^{\approx} , which is the best known upper bound (a star indicates that it is optimal, i.e., $CAN(2, S)$); lb , which reports the lower bound (computed as in section 4); and $|\mathcal{T}_a|$ and $|\mathcal{T}_f|$, which report the number of allowed and forbidden tuples, respectively.

Finally, we also show, for the $PM\text{Sat}_{CCX}^{N,t=2,S,lb}$ encoding of each instance, the following information: $\# \text{ vars}$, which is the number of variables used by this encoding; $\# \text{ clauses}$, which is the number of clauses; $\# \text{ lits}$, which is the number of literals; and $size (MB)$, which is the file size of the WCNF formula in MB.

Notice that in this paper we focus on $t = 2$ strength coverage.

Regarding existing tools for solving Mixed Covering Arrays with Constraints, the main tool we compare with is CALOT [47]. Unfortunately, CALOT is not available from the authors but we did our best to reproduce it (see section

Instance	System Under Test (SUT)			Bounds for $t = 2$					$PM\text{Sat}_{CCX}^{N,t=2,s,lb}$			
	S_P	S_φ	# lits $CNF(S_\varphi)$	ub^{ACTS}	ub^\approx	lb	$ T_a $	$ T_f $	# vars	# clauses	# lits	size (MB)
Cohen et al. [20]												
1	$2^{86}3^41^55^62$	$2^{20}3^34^1$	53	48	37	35	23876	474	1158588	2620675	7463282	60.01
2	$2^{86}3^43^45^16^1$	$2^{19}3^3$	47	32	30*	29	20331	237	657890	1371738	3984183	29.91
3	$2^{27}4^2$	2^93^1	21	19	18*	15	1838	14	36217	79008	222390	1.47
4	$2^{51}3^44^25^1$	$2^{15}3^2$	36	22	20*	19	7530	386	168852	358291	1025536	7.33
5	$2^{155}3^74^35^56^4$	$2^{32}3^64^1$	86	54	45	35	76259	73	4142574	9720622	27451121	236.74
6	$2^{73}4^36^1$	$2^{26}3^4$	64	25	24*	23	11382	1878	289001	597814	1730859	12.72
7	$2^{29}3^1$	$2^{13}3^2$	32	12	9	5	1567	231	19566	49758	132435	0.85
8	$2^{109}3^24^25^36^3$	$2^{32}3^44^1$	80	47	36*	35	33680	1098	1597165	3590459	10247230	84.50
9	$2^{57}3^14^15^16^1$	$2^{30}3^7$	81	22	20*	19	6835	1720	153584	325984	932515	6.63
10	$2^{130}3^64^55^66^4$	$2^{40}3^7$	101	47	41	35	52659	2029	2493173	5608369	16010703	135.34
11	$2^{84}3^44^25^26^4$	$2^{28}3^4$	68	47	39	35	23636	707	1123311	2523897	7200149	57.70
12	$2^{136}3^44^35^16^3$	$2^{23}3^4$	58	43	36*	35	49522	978	2144718	4675267	13461992	108.23
13	$2^{124}3^44^15^26^2$	$2^{22}3^4$	56	40	36*	35	38862	1701	1567084	3319517	9632256	75.77
14	$2^{81}3^54^36^1$	$2^{13}3^2$	32	39	36*	35	20544	618	810618	1697204	4936072	37.18
15	$2^{50}3^44^15^26^1$	$2^{20}3^2$	46	32	30*	29	8388	155	273410	569181	1650514	12.10
16	$2^{81}3^44^26^1$	$2^{30}3^4$	72	25	24*	23	14600	2303	370051	765960	2218422	16.44
17	$2^{128}3^34^25^16^3$	$2^{25}3^4$	62	41	36*	35	43390	66	1792402	3835891	11100545	88.14
18	$2^{127}3^24^45^66^2$	$2^{23}3^44^1$	62	52	41	35	50128	28	2625808	6092947	17250882	146.38
19	$2^{172}3^94^55^66^4$	$2^{38}3^5$	91	51	43	35	98778	114	5064366	11694341	33170488	287.31
20	$2^{138}3^44^55^67$	$2^{42}3^6$	102	60	54	35	64620	3320	3903864	9411047	26386102	227.62
21	$2^{76}3^44^25^16^3$	$2^{40}3^6$	98	39	36*	35	15442	2742	610938	1279170	3717471	27.90
22	$2^{72}3^44^16^1$	$2^{20}3^2$	46	37	36*	35	13405	1181	503127	1028139	3008516	22.48
23	$2^{25}3^16^1$	$2^{13}3^2$	32	14	12*	11	1495	173	21856	47740	132915	0.85
24	$2^{110}3^25^36^4$	$2^{25}3^4$	62	48	41	35	34204	570	1656252	3748659	10679658	88.30
25	$2^{118}3^64^25^26^6$	$2^{23}3^44^1$	59	52	49	35	46968	52	2461280	5710735	16167454	136.81
26	$2^{87}3^14^35^4$	$2^{28}3^4$	68	34	26	24	20921	667	719347	1643461	4647485	36.52
27	$2^{55}3^24^25^16^2$	$2^{17}3^3$	43	37	36*	35	9714	43	365524	746797	2183919	16.18
28	$2^{167}3^{16}4^25^36^6$	$2^{31}3^6$	80	57	50	35	96599	74	5535861	13181074	37087871	322.33
29	$2^{134}3^75^3$	$2^{19}3^3$	47	29	25*	24	45839	32	1338905	2899941	8321499	64.34
30	$2^{73}3^34^3$	$2^{31}3^4$	74	22	16*	15	12453	1308	277976	640938	1792681	13.16
apache	$2^{158}3^84^55^16^1$	$2^{33}4^25^1$	22	33	30*	29	66927	3	2221926	4701044	13619419	109.16
bugzilla	$2^{49}3^14^2$	2^43^1	11	19	16*	15	5818	4	112768	247130	697953	4.82
gcc	$2^{189}3^{10}$	$2^{37}3^3$	83	23	15	8	82770	39	1913568	5063264	13685896	112.78
spins	$2^{13}4^5$	2^{13}	26	26	19*	15	979	13	27050	64498	177169	1.23
spinv	$2^{42}3^24^{11}$	$2^{47}3^2$	100	45	33	15	8741	56	401069	1063265	2888090	22.53
Segall et al. [44]												
Banking1	3^44^1	$5^{11}2$	560	15	13*	11	102	0	1938	5864	19573	0.11
Banking2	$2^{14}4^1$	2^3	6	11	10*	7	473	3	5591	12845	34672	0.21
CommProtocol	$2^{10}7^1$	$2^{10}3^{10}4^{12}5^{24}$ $6^{30}7^{30}8^{12}$	704	19	16*	13	285	35	6047	15914	50363	0.29
Concurrency	2^5	$2^43^15^2$	21	6	5*	3	36	4	278	667	1686	0.01
Healthcare1	$2^63^25^16^1$	2^33^18	60	30	30*	29	361	8	12090	24661	70619	0.44
Healthcare2	$2^53^64^1$	$2^13^65^{18}$	110	16	14	11	466	1	8212	18853	52268	0.31
Healthcare3	$2^{16}3^64^55^16^1$	2^31	62	38	34*	29	3092	59	121950	271023	768538	5.38
Healthcare4	$2^{13}3^{12}4^95^26^17^1$	2^{22}	44	49	46*	41	5707	38	287980	619634	1783211	13.14
Insurance	$2^63^15^16^211^113^117^131^1$	-	0	527	527*	526	4573	0	2509047	5009492	14863678	122.95
NetworkMgmt	$2^24^15^310^211^1$	2^{20}	40	112	110*	109	1228	20	148402	301059	877206	6.28
ProcessorComm1	$2^33^64^6$	2^{13}	26	29	21	15	1058	13	32957	80475	219601	1.54
ProcessorComm2	$2^33^{12}4^85^2$	1^42^{121}	246	32	25*	24	2525	854	85287	193248	541399	3.73
Services	$2^33^52^810^2$	$3^{386}4^2$	1166	106	100*	99	1819	16	204692	460866	1346965	9.78
Storage1	$2^13^14^15^1$	4^95	380	17	17*	14	53	18	1294	4270	13468	0.07
Storage2	3^46^1	-	0	18	18*	17	126	0	2826	5652	15552	0.09
Storage3	$2^93^15^36^18^1$	$2^{38}3^{10}$	106	50	50*	39	1020	120	54810	122009	344328	2.47
Storage4	$2^53^74^15^66^27^110^113^1$	2^{24}	48	136	130*	129	3491	24	495046	1012862	2970799	22.45
Storage5	$2^53^55^36^68^19^110^211^1$	2^{151}	302	218	215	109	5342	246	1206084	3020149	8366680	72.16
SystemMgmt	$2^53^45^1$	$2^{13}3^4$	38	17	15*	14	310	14	5935	12813	35376	0.21
Telecom	$2^53^14^25^16^1$	$2^{11}3^14^9$	61	32	30*	29	440	11	15650	32761	93262	0.59
Yu et al. [48]												
RL-A	$2^53^47^54^65^74^81^{12}3^9$	$1^{12}2^{491}3^{345}$	2029	155	153	143	7066	7156	1142671	2491775	7220414	59.32
RL-B	$2^83^24^35^36^19^1$	$1^{82}1^{127}3^{277}$	27721	767	727	519	17018	5597	13365222	35733711	109278283	1026.60
Yamada et al. [46]												
Company2	$2^63^48^4$	$1^{22}2^{35}3^{89}4^{54}5^{34}$ $6^{20}7^{34}8^{16}9^4$	1247	81	72	55	1149	261	100546	252543	744203	5.35

Table 3: General information of all benchmarks used.

6), showing our experimental investigation that the results are consistent with those of [47]. Our implementation of CALOT and all algorithms presented in this paper will be available for reproducibility, which we think is also a nice contribution for both the combinatorial testing and satisfiability communities.

Since all the algorithms presented in this paper are built on top of a SAT solver, we compared, when possible, all the algorithms with the same underlying SAT solver. That is not the case in [47], which may lead to flawed conclusions. In our experimental investigation we choose Glucose (version 4.1) [10], as most of the state-of-the-art MaxSAT solvers are built on top of it.

We also use the ACTS tool [17] to compute fast and good enough upper bounds of the Covering Array Number problem, although it is not competitive with SAT-based approaches.

The environment of execution consists of a computer cluster with machines equipped with two Intel Xeon Silver 4110 (octa-core processors at 2.1GHz, 11MB cache memory) and 96GB DDR4 main memory. Unless otherwise stated, all the experiments were executed with a timeout of 2h and a memory limit of 18GB. To mitigate the impact of randomness we executed all the algorithms using five different seeds for each instance.

The rest of the experimental section is organized as follows. Regarding the Covering Array Number, in subsection 11.1, we compare the CALOT algorithm with the MaxSAT encodings and SAT-based MaxSAT approaches described in sections 7 and 8. Regarding the Tuple Number problem, in subsection 11.2, we evaluate the complete and incomplete MaxSAT algorithms on the encoding described in section 9. Then, in subsection 11.3, we evaluate the incomplete approach for computing the Tuple Number described in section 10.

11.1 SAT-based MaxSAT approaches for the Covering Array Number problem

In this experiment, we compare the performance of state-of-the-art SAT-based MaxSAT solvers with the CALOT algorithm described in section 6. We hypothesise that since these SAT-based MaxSAT algorithms, once executed on the suitable MaxSAT encodings, can simulate the behaviour of the CALOT algorithm (see Propositions 6 and 8) but the opposite is not true, MaxSAT algorithms may perform similarly or outperform the CALOT algorithm. This hypothesis would contradict the findings in [47], where it was reported that the CALOT algorithm clearly dominates the MaxSAT-based approach in [9]. If our hypothesis is correct, MaxSAT approaches for solving the Covering Array Number problem would be put back on the agenda. We focus in anytime algorithms that must be able to report suboptimal solutions⁸.

Solvers: The CALOT algorithm (described in section 6) and the model-guided Linear SAT-based MaxSAT algorithm Linear (described in section 8) were implemented on top of a custom python framework for SAT solving. This framework includes python bindings for several state-of-the-art SAT solvers and the python binding to the PBLib [35].

We additionally tested several complete and incomplete algorithms from the MaxSAT Evaluation 2020 [13]. From complete MaxSAT solvers we tested MaxHS [12], EvalMaxSAT [11], RC2 [31] and maxino [1]. We only report results for RC2 and one seed⁹, as this was the complete solver that reported better results. MaxHS obtained the best results for 2 of the tested instances, but we decided to exclude it from the comparison since it cannot report upper bounds for most of the instances and it uses another underlying SAT solver different than Glucose41.

Regarding incomplete MaxSAT algorithms we tested Loandra [15], tt-open-wbo-inc [39] and SatLike [34]. We report results for Loandra and tt-open-wbo-inc as SatLike crashed in some of the tested instances.

MaxSAT encodings: Respect to the MaxSAT encodings we report results on $PM\text{Sat}_{CCX}^{N,t,S,lb}$ and the weighted version $WPMSat_{CCX}^{N,t,S,lb}$ using a linear increase for the weights ($w_i = i - (lb + 2) + 1$, see equation $WSoftU$ in section 7). We found that $WPMSat_{CCX}^{N,t,S,lb}$ with the linear and exponential increase ($w_i = 2^{i-(lb+2)}$) lead to the same performance, but the exponential increase represented a problem for some MaxSAT solvers when i was high enough.

We further tested the three different alternatives for equation (a) from CCX , where two reported good results. The first one is the original (a) equation shown in section 3, ($c_\tau^i \rightarrow c_\tau^{i-1} \vee x_{i,p,v}$), which we will refer to as a.0. The second one is the variation ($c_\tau^i \rightarrow c_\tau^{i-1} \vee x_{i,p,v} \wedge (c_\tau^i \leftarrow c_\tau^{i-1})$), which we will refer to as a.1.

Results: Table 4 shows the results of our experimentation. For each row and solver column, we give the average size of the minimum MCAC (out of the 5 executions per instance) and the average runtime. Bold values represent the best results. In case there are ties in size, the best time is marked. Sizes that have a star represent that the optimum has been certified in at least one of the five seeds executed for the current benchmark instance.

Table 5 aggregates the information presented in Table 4 to analyze the dominance relations among approaches, i.e., the number of instances where algorithm A finds a smaller MCAC than algorithm B (size) and the number of instances where A needs less runtime than B (time). If A finds a smaller MCAC than B, then it is also considered that it needs less runtime. In this sense, we will say that an approach outperforms another if it provides a strictly better solution within the given timeout or finds the same best suboptimal solution faster.

We observe how both *tt-open-wbo-inc* and *loandra* outperform the results obtained by *CALOT*, improving the sizes in more than 10 of the 58 available instances and, in the case of *tt-open-wbo-inc*, we also improve runtimes in more

⁸We adapted RC2 MaxSAT solver to report suboptimal solutions when applying the stratified strategy (see section 8)

⁹Unfortunately RC2 MaxSAT solver does not allow to specify a seed.

Instance	ACTS		CALOT		CALOT		CALOT		RC2-B		RC2-B		linear		loandra		loandra		tt-open-wbo-inc		tt-open-wbo-inc		tt-open-wbo-inc		tt-open-wbo-inc		tt-open-wbo-inc						
	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time			
Cohen et al. [20]																																	
1	4800	0.83	3800	1470.88	-1.00	1.00	3700	7146.52	3700	5367.02	3700	1718.97	3700	426.06	3700	902.46	3700	1332.26	3700	4724.61	3700	1332.26	3700	4724.61	3700	1332.26	3700	4724.61	3700	1332.26	3700	4724.61	
2	3200	1.08	3000*	3.57	3000*	0.81	3000*	7.19	3000*	2.91	3000*	7.19	3000*	7.68	3000*	0.81	3000*	1.42	3000	4724.61	3000	1.42	3000	4724.61	3000	1.42	3000	4724.61	3000	1.42	3000	4724.61	
3	1900	1.21	1800*	0.12	1800*	0.73	1800*	0.36	1800*	0.13	1800*	0.36	1800*	0.47	1800*	0.06	1800*	0.08	1800	4724.61	1800	0.08	1800	4724.61	1800	0.08	1800	4724.61	1800	0.08	1800	4724.61	
4	2200	1.04	2000*	0.95	2000*	3.19	2000*	1.34	2000*	0.74	2000*	1.34	2000*	2.24	2000	0.30	2000	0.29	2000	4724.61	2000	0.29	2000	4724.61	2000	0.29	2000	4724.61	2000	0.29	2000	4724.61	
5	5400	1.31	4700*	2411.55	-1.00	4800	378.46	4700	1693.29	-1.00	4600	1629.29	4580	1795.81	4600	1629.29	4580	1348.31	4600	1803.99	4600	1348.31	4600	1803.99	4600	1348.31	4600	1803.99	4600	1348.31	4600	1803.99	
6	2500	1.11	2400*	1.46	2400*	5.47	2400*	2.98	2400*	1.48	2400*	2.98	2400*	3.51	2400	0.43	2400	0.48	2400	4724.61	2400	0.48	2400	4724.61	2400	0.48	2400	4724.61	2400	0.48	2400	4724.61	
7	1200	1.13	900	0.08	900	0.61	900	0.98	900	0.13	900	0.98	900	30.57	900	0.06	900	0.06	900	4724.61	900	0.06	900	4724.61	900	0.06	900	4724.61	900	0.06	900	4724.61	
8	4700	1.01	3800	4727.41	-1.00	1.00	3700	3407.87	3800	547.77	3680*	2849.90	3700	3731.63	3700	1178.58	3660	2948.62	3700	2333.83	3700	2333.83	3700	2333.83	3700	2333.83	3700	2333.83	3700	2333.83	3700	2333.83	
9	2200	1.21	2000*	0.37	2000*	2.66	2000*	2.66	2000*	0.49	2000*	1.22	2000*	2.30	2000	0.18	2000	0.27	2000	4724.61	2000	0.27	2000	4724.61	2000	0.27	2000	4724.61	2000	0.27	2000	4724.61	
10	4700	1.28	4400	6860.63	-1.00	4400	4759.95	4400	1343.51	41.80	1235.56	41.80	1235.56	1891.73	41.80	2720.18	41.00	970.10	41.00	5823.09	41.00	970.10	41.00	5823.09	41.00	970.10	41.00	5823.09	41.00	970.10	41.00	5823.09	
11	4700	1.26	4100	4167.69	-1.00	4200	6172.69	4200	1916.85	39.00	6908.22	40.00	6908.22	4323.47	40.00	741.96	40.00	1086.24	39.00	1628.02	39.00	1086.24	39.00	1628.02	39.00	1086.24	39.00	1628.02	39.00	1086.24	39.00	1628.02	
12	4300	1.29	3600*	9.80	3600*	24.01	3600*	49.94	3600*	20.66	3600*	36.16	3600*	36.16	3600*	8.18	3600	10.46	3600	4724.61	3600	10.46	3600	4724.61	3600	10.46	3600	4724.61	3600	10.46	3600	4724.61	
13	4500	1.15	3600*	14.13	3600*	6.78	3600*	29.91	3600*	6.73	3600*	14.91	3600*	14.91	3600*	3.31	3600	3.31	3600	4724.61	3600	3.31	3600	4724.61	3600	3.31	3600	4724.61	3600	3.31	3600	4724.61	
14	3900	0.83	3600*	3.76	3600*	15.56	3600*	4.88	3600*	4.80	3600*	8.84	3600*	8.84	3600*	3.60	3600	2.48	3600	4724.61	3600	2.48	3600	4724.61	3600	2.48	3600	4724.61	3600	2.48	3600	4724.61	
15	3200	1.05	3000*	1.02	3000*	4.73	3000*	4.88	3000*	1.30	3000*	2.91	3000*	3.70	3000	0.47	3000	0.66	3000	4724.61	3000	0.66	3000	4724.61	3000	0.66	3000	4724.61	3000	0.66	3000	4724.61	
16	2500	0.84	2400*	1.36	2400*	6.60	2400*	6.20	2400*	1.65	2400*	4.30	2400*	4.46	2400	0.56	2400	0.60	2400	4724.61	2400	0.60	2400	4724.61	2400	0.60	2400	4724.61	2400	0.60	2400	4724.61	
17	4100	1.17	3600*	13.60	3600*	30.57	3600*	35.75	3600*	10.98	3600*	22.72	3600*	31.18	3600	0.58	3600	7.48	3600	4724.61	3600	7.48	3600	4724.61	3600	7.48	3600	4724.61	3600	7.48	3600	4724.61	
18	5100	1.43	4600	128.65	-1.00	4200	201.42	-1.00	4200	-1.00	4600	178.28	4600	128.89	44.00	41.00	637.62	41.00	626.83	41.00	2943.99	41.00	626.83	41.00	2943.99	41.00	626.83	41.00	2943.99	41.00	626.83	41.00	2943.99
19	6000	1.53	5700	129.90	-1.00	5700	224.31	-1.00	5700	1092.85	54.80	1422.09	54.60	1736.92	55.00	32.47	55.00	500.42	55.00	437.94	55.00	500.42	55.00	437.94	55.00	500.42	55.00	437.94	55.00	500.42	55.00	437.94	
20	3900	1.08	3600*	3.86	3600*	7.74	3600*	9.95	3600*	3.40	3600*	7.08	3600*	8.88	3600	1.06	3600	1.38	3600	4724.61	3600	1.38	3600	4724.61	3600	1.38	3600	4724.61	3600	1.38	3600	4724.61	
21	3700	0.75	3600*	2.70	3600*	8.57	3600*	8.57	3600*	2.17	3600*	8.57	3600*	8.57	3600*	0.65	3600	0.88	3600	4724.61	3600	0.88	3600	4724.61	3600	0.88	3600	4724.61	3600	0.88	3600	4724.61	
22	1400	1.40	4400	185.06	-1.00	4400	185.06	-1.00	4400	12.00*	0.42	12.00*	0.42	12.00*	0.30	12.00	0.03	12.00	0.05	12.00	4724.61	12.00	0.05	12.00	4724.61	12.00	0.05	12.00	4724.61	12.00	0.05	12.00	4724.61
23	4800	1.28	5200	100.56	-1.00	5200	117.86	-1.00	5200	13.58	42.00	1693.19	49.80	1133.70	51.00	136.11	50.00	2101.71	50.00	884.17	50.00	2101.71	50.00	884.17	50.00	2101.71	50.00	884.17	50.00	2101.71	50.00	884.17	
24	5200	1.37	2700	49.99	2700	31.54	2700	6.70	2700	551.57	2700	103.46	2700	57.66	26.00	6740.38	26.00	790.74	2700	2104.04	2700	790.74	2700	2104.04	2700	790.74	2700	2104.04	2700	790.74	2700	2104.04	
25	3700	1.32	3600*	1.68	3600*	5.35	3600*	6.70	3600*	1.33	3600*	2.96	3600*	3.56	3600	0.39	3600	0.61	3600	4724.61	3600	0.61	3600	4724.61	3600	0.61	3600	4724.61	3600	0.61	3600	4724.61	
26	5700	1.54	5300	78.94	-1.00	5300	380.79	5300	120.36	51.20	333.93	51.00	333.93	1236.07	51.00	3112.51	51.00	607.78	52.00	61.99	49.00	607.78	52.00	61.99	49.00	607.78	52.00	61.99	49.00	607.78	52.00	61.99	
27	2900	1.15	2500*	5.63	2500*	8.77	2500*	20.73	2500*	7.81	2500*	12.69	2500*	17.48	25.00	30.65	25.00	4.02	25.00	3.55	25.00	4.02	25.00	3.55	25.00	4.02	25.00	3.55	25.00	4.02	25.00	3.55	
28	2200	0.81	1600*	1.58	1600*	4.35	1600*	3.98	1600*	1.56	1600*	3.57	1600*	3.94	1600	0.68	1600	0.83	1600	4724.61	1600	0.83	1600	4724.61	1600	0.83	1600	4724.61	1600	0.83	1600	4724.61	
29	3500	1.25	3000*	8.96	3000*	13.68	3000*	38.18	3000*	11.28	3000*	21.65	3000*	32.14	3000	0.57	3000	0.70	3000	4724.61	3000	0.70	3000	4724.61	3000	0.70	3000	4724.61	3000	0.70	3000	4724.61	
apache	1900	1.03	1600*	0.61	1600*	1.72	1600*	1.79	1600*	0.36	1600*	0.89	1600*	1.19	1600	0.14	1600	0.19	1600	4724.61	1600	0.19	1600	4724.61	1600	0.19	1600	4724.61	1600	0.19	1600	4724.61	
bugzilla	2600	0.93	1500	13.09	1500	19.06	1500	47.85	1500	22.14	1500	65.99	1500	68.18	1500	9.44	1500	12.76	1500	11.82	1500	12.76	1500	11.82	1500	12.76	1500	11.82	1500	12.76	1500	11.82	
gcc	2500	1.22	1900*	0.13	1900*	0.56	1900*	0.70	1900*	0.16	1900*	0.28	1900*	0.28	1900	0.13	1900	0.21	1900	4724.61	1900	0.21	1900	4724.61	1900	0.21	1900	4724.61	1900	0.21	1900	4724.61	
spins	2600	0.84	3300	188.97	-1.00	3300	21.39	3300	55.75	33.00	61.37	32.00	61.37	97.82	32.00	26.18	32.00	82.41	32.00	53.36	32.0												

	CALOT CCX a.0		CALOT CCX a.1		RC2-B CCX a.0		RC2-B CCX a.0 wpm		linear CCX a.0		loandra CCX a.0		loandra CCX a.1		tt-open-wbo-inc CCX a.0		tt-open-wbo-inc CCX a.1		tt-open-wbo-inc CCX a.0 wpm		tt-open-wbo-inc CCX a.1 wpm	
	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
ACTS	0 52	2 56	0 52	2 56	21 32	23 35	2 51	4 54	0 52	2 56	3 52	4 54	3 52	4 54	1 52	2 56	0 53	1 57	1 52	2 56	0 53	1 57
CALOT CCX a.0	--	--	5 3	52 6	21 0	57 1	5 4	53 5	3 2	47 11	3 12	44 14	3 12	44 14	1 13	13 45	0 13	12 46	2 12	15 43	0 12	19 39
CALOT CCX a.1	--	--	--	--	21 0	57 1	4 3	50 8	5 5	27 31	3 12	42 16	3 11	42 16	1 12	9 49	0 13	6 52	1 10	7 51	0 11	11 47
RC2-B CCX a.0	--	--	--	--	--	--	0 19	31 25	0 21	1 57	2 20	3 54	2 20	6 51	0 20	1 56	0 21	1 57	0 20	1 56	0 21	1 57
RC2-B CCX a.0 wpm	--	--	--	--	--	--	--	--	3 5	7 51	2 11	8 49	2 10	10 47	0 11	1 56	1 13	4 54	0 9	2 55	1 11	6 52
linear CCX a.0	--	--	--	--	--	--	--	--	--	--	3 11	41 17	3 11	41 17	1 12	7 51	1 13	4 54	2 11	3 55	1 12	7 51
loandra CCX a.0	--	--	--	--	--	--	--	--	--	--	--	--	2 7	38 17	4 5	9 48	4 7	9 49	5 3	12 45	3 5	10 48
loandra CCX a.1	--	--	--	--	--	--	--	--	--	--	--	--	--	--	5 3	13 44	6 5	9 49	7 3	13 44	6 5	13 45
tt-open-wbo-inc CCX a.0	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	2 3	39 19	4 2	38 19	2 4	42 16	
tt-open-wbo-inc CCX a.1	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	6 3	21 37	2 3	39 19	
tt-open-wbo-inc CCX a.0 wpm	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	1 4	43 15

Table 5: Dominance relations for CALOT and SAT-based MaxSAT approaches for the Covering Array Number problem. Bold values highlight winning algorithm per size or runtime.

than 40 instances. This confirms our hypothesis that MaxSAT approaches can simulate and even improve the results obtained by the *CALOT* algorithm.

Regarding the different variations of the *CCX* encoding, we notice that for *tt-open-wbo-inc* and *loandra*, variation a.1 slightly improves results obtained by the original variation a.0. In particular, we observe that *tt-open-wbo-inc* with this specific encoding obtains the best size in instance *RL-B (727)*, while algorithm *CALOT* reports a size of 760. However, this behaviour of the encoding a.1 is not observed in algorithm *CALOT*, as in this case the best variation of equation (a) seems to be a.0. These results suggest that in case we use a new MaxSAT solver we should not discard at front any encoding variation.

For *RC2* and *linear* approaches we can observe clear differences among them when applying the $PM\text{Sat}_{CCX}^{N,t,S,lb}$ encoding, as *linear* obtains better sizes and times in 21 and 57 instances respectively, showing that for the Covering Array Number problem is more effective to perform a search that incrementally refines the upper bound as the *linear* approach does (see section 8). However, we observe a substantial improvement when using the $WPM\text{Sat}_{CCX}^{N,t,S,lb}$ with the *RC2* MaxSAT solver, improving the sizes obtained by its unweighted counterpart in 19 of the 58 instances, which produces similar results than *CALOT* and $PM\text{Sat}_{CCX}^{N,t,S,lb}$ *linear* approaches. This is expected since the weighted version forces *RC2* to perform a top-down search as discussed in section 8.

We also tested the $WPM\text{Sat}_{CCX}^{N,t,S,lb}$ encoding over the *tt-open-wbo-inc*, which is not core-guided MaxSAT solver. We observe that results are similar or slightly worse than with the $PM\text{Sat}_{CCX}^{N,t,S,lb}$. We believe the $WPM\text{Sat}_{CCX}^{N,t,S,lb}$ encoding could be more useful for core-guided MaxSAT solvers as it modifies their refinement strategy (i.e. improve the upper bound instead of the lower bound). We also observed that refining the lower bound for the Covering Array Number problem is more challenging than refining the upper bound, as there are some instances where encoding $PM\text{Sat}_{CCX}^{N,t,S,lb}$ with *RC2* (which would refine the lower bound) is not able to report any results, usually on instances where the CAN is not found.

11.2 Weighted Partial MaxSAT approaches for the Tuple Number problem

Encouraged by the good results of the proposed MaxSAT approaches for the Covering Array Number problem, we now evaluate the MaxSAT approach described in section 9 on SAT-based MaxSAT approaches for solving the Tuple Number problem. Notice that the *CALOT* algorithm only works for solving the Covering Array Number problem. In this sense, this is a pioneering work on applying SAT technology to solve the Tuple Number problem.

Solvers: We choose the *tt-open-wbo-inc* MaxSAT solver to perform these experiments, as this has been the approach that achieved better results in section 11.1.

MaxSAT encodings: We recall there are also some variations of the $TPM\text{Sat}_{CCX}^{N,t,S,lb}$ encoding, due to the way constraint *CCX* is formulated, i.e. the relation among c_{τ}^i vars and $x_{i,p,v}$ vars (see remark 1 in section 3). According some preliminary experimentation we observed that variation $(c_{\tau}^i \leftrightarrow c_{\tau}^{i-1} \vee x_{i,p,v})$, to which we refer as a.2, reported also good results, while variation a.1 did not and was excluded.

We additionally noticed that, when computing the tuple number, the cost of the solution returned by the MaxSAT solver when using the original encoding of equation (a) in *CCX*, $(c_{\tau}^i \rightarrow c_{\tau}^{i-1} \vee x_{i,p,v})$, can indeed overestimate the real cost of the solution induced by the value of the $x_{i,p,v}$ vars, i.e., the assignments that represent the actual tests used in the solution. This can happen since it is possible to set to False a c_{τ}^i even if the right-hand side of the implication is True. Enforcing the other side of the implication corrects this issue. For these reasons we will use the $(c_{\tau}^i \leftrightarrow c_{\tau}^{i-1} \vee x_{i,p,v})$ variation of *CCX*.

Results: We would like to study the evolution of the number of covered tuples as a function of the number of tests, as we hypothesise that adding a new test close to the Covering Array Number (that guarantees all tuples can be covered) will allow to add very few additional tuples. In that sense, if these tests are expensive enough, they will not pay off in terms of the available budget and the additional percentage of coverage we can achieve.

In Figure 1, we show the number of tests required to reach a certain percentage of the tuples to cover for the *tt-open-wbo-inc* approach. Notice that *tt-open-wbo-inc* is an incomplete MaxSAT solver and we are therefore reporting a lower bound on the possible percentage by a particular number of tests. For lack of space, we only show the most representative instances of all the benchmark families.

We observe, for all the tested instances, that most of the tuples are covered using a relatively small number of tests and the remaining tuples require a relatively large additional number of tests. In our experiments, with only 52% of tests required for the Covering Array Number or for the best suboptimal solution from Table 4 in section 11.1, we are able to reach a 95% coverage, whereas the remaining 5% of tuples need the remaining 48% of tests.

We also notice that the Tuple Number problem is more challenging than the Covering Array Number problem. According to some experimentation that we performed using complete MaxSAT solvers, none of the tested approaches has been able to certify any optimum for $N > 1$, even for the instances that were easy to solve for the Covering Array Number problem.

Another interesting observation is the erratic behavior on the *RL-B* instance [48] (Figure 1, bottom right). *RL-B* is the biggest instance in the available benchmarks, having 27 parameters with domains up to 37, and with a suboptimal solution for the Covering Array Number (for $t = 2$) of 727 tests. After 100 tests, the results for the Tuple Number problem become quite unstable in contrast to the behaviour on the rest of instances. This phenomenon might point out that the approach analyzed in this section has some limitations when instances are large enough. For a fixed set of parameters, instances become bigger when we increase the strength t or the number of tests as in this case.

To conclude this section, we have confirmed that MaxSAT could be a good approach to solve the Tuple Number problem with constraints. We have also observed that with a relatively small number of tests we can cover most of the tuples, and that this approach can be useful for medium-sized instances that do not need a large number of tests to reach a reasonable coverage percentage.

In the next section, we explore the Incremental Test Suite Construction for the Tuple Number problem described in section 10.1. It allows us to tackle more efficiently those Tuple Number problems involving a relatively large number of tests.

11.3 MaxSAT based Incremental Test Suite Construction for $T(N; t, S)$

In section 11.2, we have analyzed an approach that can be used to maximise the number of tuples covered by a number of tests inferior to $CAN(t, S)$. However, we have seen that this might not be the most efficient solution if we require to compute the Tuple Number problem for a large enough number of tests.

Solving approaches: Here we propose three incomplete alternatives for solving the Tuple Number problem, with the aim of improving the results obtained in section 11.2. Our hypothesis is that the application of incomplete approaches can be more suitable when solving bigger instances.

The first approach is the greedy algorithm presented in [46], referred as *maxh - its*. This algorithm incrementally adds a test at a time. The test is constructed through a heuristic [22] that tries to increase the number of covered tuples so far, by selecting at each step the parameter tuple with most value tuples yet to be covered.

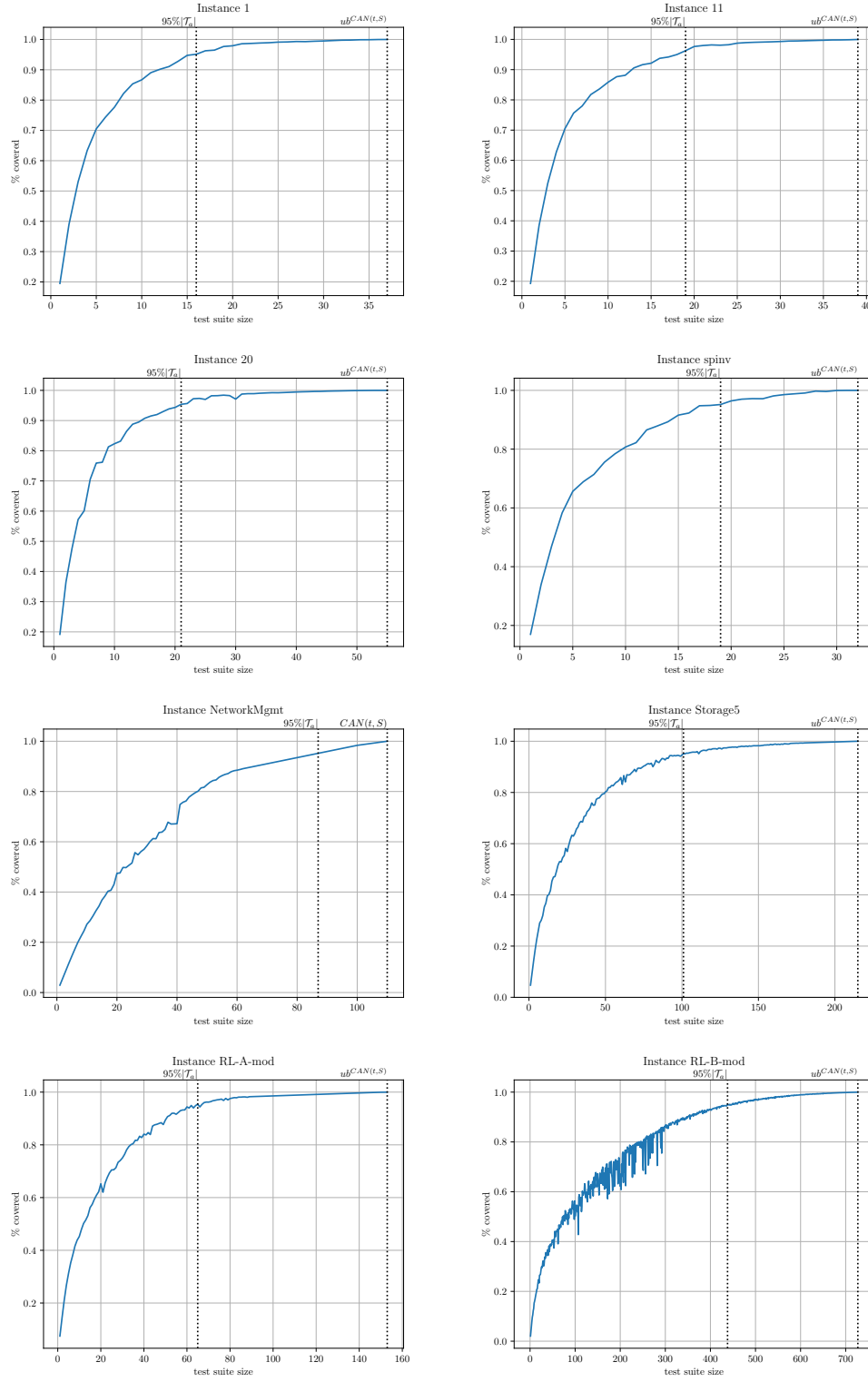
The second approach is the Incremental Test Suite Construction from section 10.1 (referred here as *maxsat - its*), which also adds a test at a time¹⁰, but this test is built by solving the Tuple Number problem through an incomplete MaxSAT solver instead of using a heuristic as in the previous approach.

In the third approach, instead of a MaxSAT query, as in the second approach, we apply a SAT query to return a test that covers at least one more tuple (referred as *sat - its*) than the incremental test suite built so far.

We also evaluate the approach described in section 9.2. The idea is to relax the Covering Array Number problem by allowing to cover only a 95% of the allowed tuples (τ_a). We refer to this approach as *minits - 95% $|\tau_a$* . As for the Covering Array Number problem, we use the upper bound returned by the ACTS tool (see section 4) for the initial number of tests.

¹⁰The algorithm allows to add more than one test at a time, but this experiment is out of reach in this paper.

Figure 1: Number of tests required to reach a certain coverage percentage for the *tt-open-wbo-inc* approach.



Results: We present the relative performance of the previous four approaches respect to the best incomplete MaxSAT approach (*tt-open-wbo-inc*) for solving the Tuple Number problem from section 11.2, referred as $\simeq T(N; t, S)$ (we use the symbol \simeq to indicate that the values reported for $\simeq T(N; t, S)$ correspond to suboptimal solutions). All

the approaches shown in this section also use the incomplete SAT-based MaxSAT solver *tt-open-wbo-inc*, except *sat - its* which uses the Glucose41 SAT solver. For the encoding of equation (a) of *CCX* we use variation a.2 ($c_\tau^i \leftrightarrow c_\tau^{i-1} \vee x_{i,p,v}$) as in section 11.2.

To perform a fair comparison we tried to execute all the algorithms within the same runtime conditions. We use the runtime *maxsat - its* needs to cover all the allowed tuples as a reference. In more detail, we set a timeout of 100s to each iteration of the *maxsat - its* approach¹¹. Therefore, the total runtime in seconds consumed by *maxsat - its* is the number of test it reaches multiplied by 100. For *maxh - its* and *sat - its*, the timeout is the total runtime consumed by *maxsat - its*. For *mint*s - 95% $|\tau_a|$, we use as timeout the runtime consumed by $\simeq T(N; t, S)$ to reach 95% of coverage. Finally, for $\simeq T(N; t, S)$, we use a timeout of $N \cdot 100$ seconds for each N . Notice that in this last case we are ensuring that for a given N , both $\simeq T(N; t, S)$ and *maxsat - its* approaches will have the same execution time limits.

All approaches have been executed with 3 seeds and the mean is reported. The experimental results are presented in Figures 2 and 3. As in section 11.2, we only plot the most representative instances.

Figure 2 shows the increment (or decrement) of the number of tests required by *maxsat - its*, *maxh - its* and *mint*s - 95% $|\tau_a|$ to cover the same number of tuples as $\simeq T(N; t, S)$. On the other hand, Figure 3 shows the increment (or decrement) of tests required to reach the same coverage ratio as $\simeq T(N; t, S)$. For *sat - its* approach we found that in most cases it is able to cover only one tuple per test, so we decided to exclude these results in the figures as they were clearly outperformed by the rest of the presented approaches.

In both figures, we plot a vertical line to show the points where $\simeq T(N; t, S)$ reaches 95% and 100% of tuples covered.

In general, *maxsat - its* clearly outperforms *maxh - its*. This can be expected since the nature of the incremental approach is to do the best at each possible iteration, and *maxsat - its* tackles exactly this goal by solving the Tuple Number problem, while *maxh - its* do not.

We also observe that *maxsat - its* outperforms the tuple coverage that $\simeq T(N; t, S)$ can achieve on the first tests. Particularly, *maxsat - its* is able to improve the number of tests required to cover 95% of the allowed tuples in 7 of the 8 instances we show in Figures 2 and 3. On the other hand, above 95%, $\simeq T(N; t, S)$ seems to be the best approach in terms of using fewer tests for the same coverage. This makes sense since the incomplete nature of *maxsat - its* make it less efficient when approaching the complete coverage, what may not be need it for several applications.

In figure 2 we observe an erratic behaviour of instance *RL-B*, which is the largest instance that we had available. These results are in line with the ones in figure 1 of section 11.2, and might show the possible issues that $\simeq T(N; t, S)$ can suffer when dealing with large instances. In particular, figure 4 shows the number of literals of the MaxSAT instance solved by $\simeq T(N; t, S)$ and *maxsat - its* as the size of the test suite increases for the *RL-B* benchmark. We observe that $\simeq T(N; t, S)$ has to deal with an increasing size of the Partial MaxSAT instance proportional to the number of tests in the test suite. In contrast, for *maxsat - its*, the size of the instance decreases since only encodes one test and the number of tuples to cover decreases along with the size of the test suite built so far. This is an interesting insight since *RL-B* instance comes from an industrial application and it may reflect what we can face in harder real-world scenarios. Therefore, *maxsat - its* may seem more well suited for these harder real-world domains and may extend the reach of Combinatorial Testing for more complex SUTs.

Finally, we also observe that the *mint*s - 95% $|\tau_a|$ approach might not be the best option to obtain a good suboptimal test suite that covers 95% of the total tuples. However, for some instances, it obtains better results than any other tested method (*NetworkMgmt* and *Storage5*). We also have to note that this is the only tested method that can certify the optimality of the obtained test suite when combined with a complete MaxSAT solver.

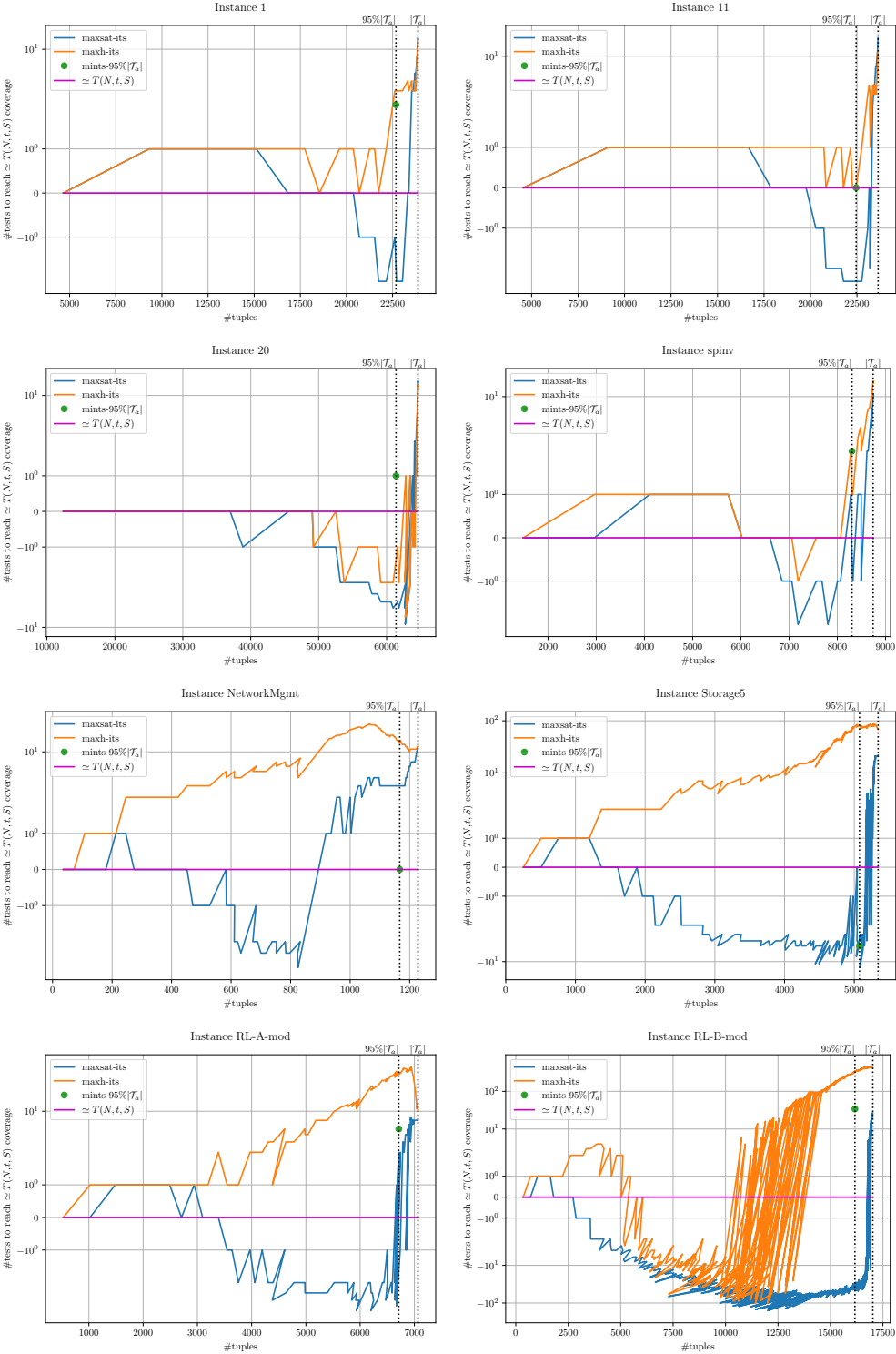
12 Conclusions

We have shown that MaxSAT technology is well-suited for solving the Covering Array Number problem for Mixed Covering Arrays with Constraints through SAT technology. In particular, we discussed efficient encodings and how MaxSAT algorithms perform on them.

We also presented MaxSAT encodings for the Tuple Number problem. To our best knowledge, this is the first time that this problem is studied with SUT Constraints. Additionally, we presented a new incomplete algorithm which can be applied efficiently to solve those instances where the Tuple Number problem encoding into MaxSAT is too large. In particular, we proved we can build good enough solutions by incrementally adding a new test synthesized

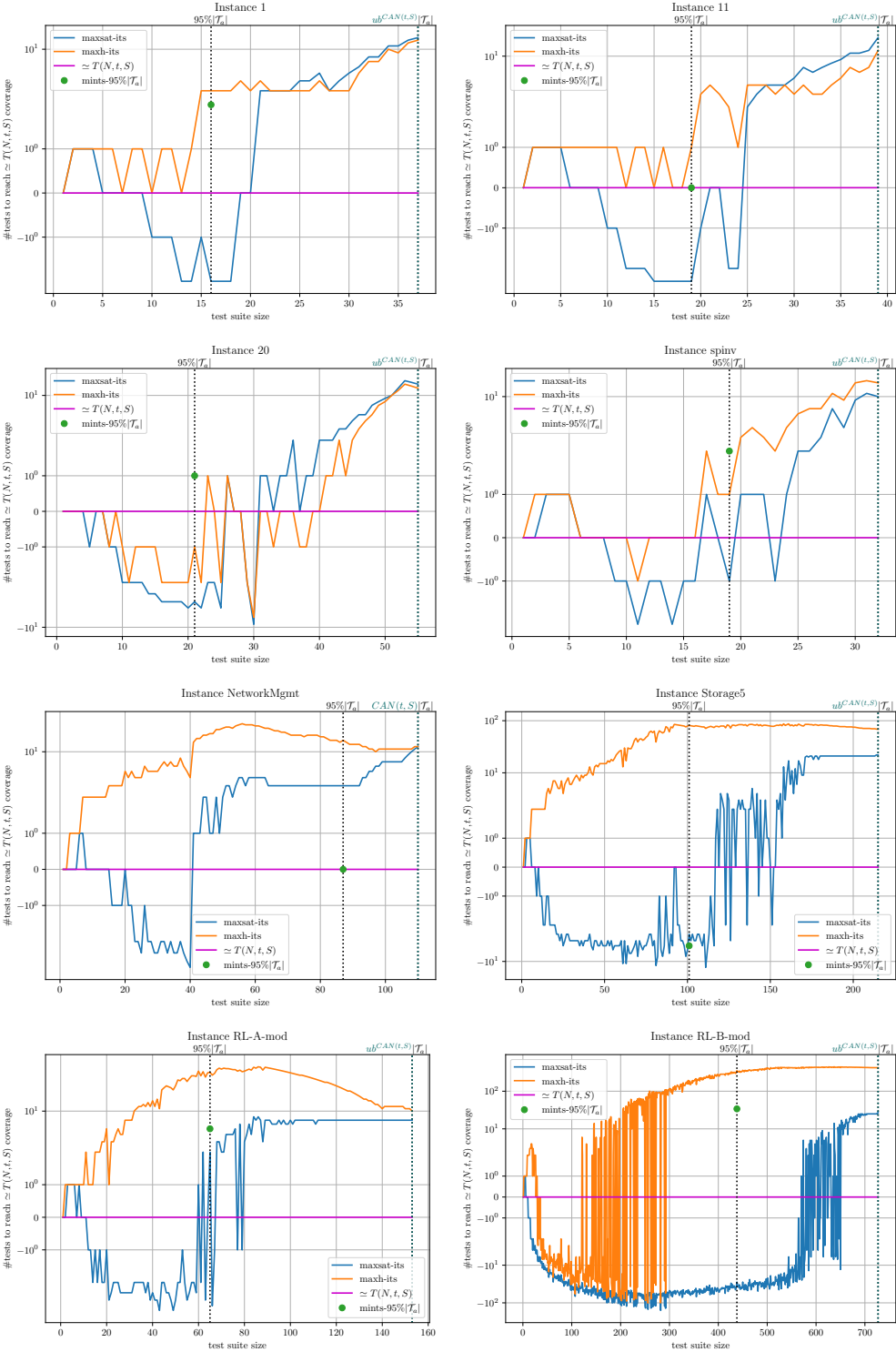
¹¹We assume that *maxsat - its* is able to cover at least one more tuple in 100 seconds

Figure 2: Comparison of the required number of tests for different methods with regards to the number of test used by $\approx T(N, t, S)$ (as base) to cover each number of tuples.



through a MaxSAT query that aims to maximize the coverage of additional allowed tuples, respect to the test suite under construction.

Figure 3: Comparison of the required number of tests for different methods to cover as much tuples at each test from $\approx T(N, t, S)$ (as base).



Another interesting result that we obtained is that if we do not aim to cover all t -tuples but a *statistically significant* fraction, we can save a great amount of tests. We experimentally showed that, to cover a 95% percentage, we just need,

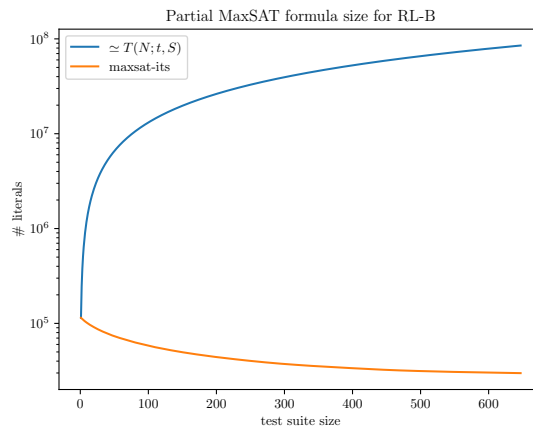


Figure 4: Partial MaxSAT formula size for RL-B in literals as a function of test suite size.

on average, a 52% percentage of the best suboptimal solution reported so far. This is of high practical importance for applications where test cases are expensive according to the budget.

From the point of view of Combinatorial Testing, it is reasonable to say that the practical and theoretical interest application of our findings and approaches will grow proportionally to the hardness or complexity of the SUT constraints. This will certainly extend the reach of Combinatorial Testing to more challenging SUTs.

From the point of view of Constraint programming, the lessons learnt on how to design efficient encodings for MaxSAT solvers can be exported to solve similar problems. These problems are roughly characterized by having an objective function whose size is proportional to the best known upper bound.

SAT and MaxSAT communities will also benefit from new challenging benchmarks to test the new advances in the field. Moreover, any future advance in MaxSAT technology can be applied to solve more efficiently the Covering Array Number and Tuple Number problems with no additional cost.

Acknowledgements

We would like to thank specially Akihisa Yamada for the access to several benchmarks for our experiments and solving some questions about his previous work on Combinatorial Testing with Constraints.

This work was partially supported by the MINECO-FEDER project TASSAT3 (TIN2016-76573-C2-2-P), the MICINN's project PROOFS (PID2019-109137GB-C21) and ISINC (PID2019-111544GB-C21), and the MICINN FPU fellowship (FPU18/02929).

References

- [1] M. Alviano, C. Dodaro, and F. Ricca. A maxsat algorithm using cardinality constraints of bounded size. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.
- [2] C. Ansótegui, M. L. Bonet, and J. Levy. Sat-based maxsat algorithms. *Artificial Intelligence*, 196:77–105, 2013.
- [3] C. Ansótegui, I. Izquierdo, F. Manyà, and J. 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, Vic, Catalonia, Spain, October 23-25, 2013*, pages 51–59, 2013.
- [4] C. Ansótegui and F. Manyà. Mapping problems with finite-domain variables into problems with boolean variables. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, pages 1–15, 2004.
- [5] C. Ansótegui, M. L. Bonet, J. Gabàs, and J. Levy. Improving SAT-Based Weighted MaxSAT Solvers. In M. Milano, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 86–101, Berlin, Heidelberg, 2012. Springer.

- [6] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, Lecture Notes in Computer Science, pages 427–440, Berlin, Heidelberg, 2009. Springer.
- [7] C. Ansótegui and J. Gabàs. WPM3: An (in)complete algorithm for weighted partial MaxSAT. *Artificial Intelligence*, 250:37–57, Sept. 2017.
- [8] C. Ansótegui, J. Gabàs, and J. Levy. Exploiting subproblem optimization in SAT-based MaxSAT algorithms. *Journal of Heuristics*, 22(1):1–53, Feb. 2016.
- [9] C. Ansótegui, I. Izquierdo, F. Manyà, and J. T. Jiménez. A max-sat-based approach to constructing optimal covering arrays. *Frontiers in Artificial Intelligence and Applications*, 256:51–59, 2013.
- [10] G. Audemard, J.-M. Lagniez, and L. Simon. Improving glucose for incremental sat solving with assumptions: Application to mus extraction. In *International conference on theory and applications of satisfiability testing*, pages 309–317. Springer, 2013.
- [11] F. Avellaneda. A short description of the solver evalmaxsat. *MaxSAT Evaluation 2020*, page 8.
- [12] F. Bacchus. Maxhs in the 2020 maxsat evaluation. *MaxSAT Evaluation 2020*, page 19.
- [13] F. Bacchus, J. Berg, M. Jarvisalo, and R. Martins. Maxsat evaluation 2020: Solver and benchmark descriptions. 2020.
- [14] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue. Generating combinatorial test cases by efficient sat encodings suitable for cdcl sat solvers. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 112–126, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [15] J. Berg, E. Demirovic, and P. Stuckey. Loandra in the 2020 maxsat evaluation. *MaxSAT Evaluation 2020*, page 10.
- [16] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
- [17] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of ACTS: A case study. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 591–600, 2012.
- [18] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 146–155, 2005.
- [19] O. Carrizales-Turrubiates, N. Rangel-Valdez, and J. Torres-Jiménez. Optimal shortening of covering arrays. In I. Z. Batyrshin and G. Sidorov, editors, *Advances in Artificial Intelligence - 10th Mexican International Conference on Artificial Intelligence, MICAI 2011, Puebla, Mexico, November 26 - December 4, 2011, Proceedings, Part I*, volume 7094 of *Lecture Notes in Computer Science*, pages 198–209. Springer, 2011.
- [20] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.
- [21] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche*, 59(1,2):125–172, 2004.
- [22] J. Czerwonka. Pairwise testing in real world. In *Proc. of the Twenty-fourth Annual Pacific Northwest Software Quality Conference, 10-11 October 2006, Portland, Oregon*, pages 419–430, 2006.
- [23] F. Duan, Y. Lei, L. Yu, R. N. Kacker, and D. R. Kuhn. Optimizing ipog’s vertical growth with constraints based on hypergraph coloring. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 181–188, 2017.
- [24] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, Jan. 2006. Publisher: IOS Press.
- [25] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, pages 462–476, 2002.
- [26] Z. Fu and S. Malik. On solving the partial max-sat problem. In A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 252–265, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [27] I. P. Gent and P. Nightingale. A new encoding of alldifferent into sat. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.

- [28] C. Gomes and M. Sellmann. Streamlined constraint reasoning. In M. Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 274–289, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [29] B. Hnich, S. Prestwich, and E. Selensky. Constraint-based approaches to the covering test problem. In B. V. Faltings, A. Petcu, F. Fages, and F. Rossi, editors, *Recent Advances in Constraints*, pages 172–186, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [30] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint Models for the Covering Test Problem. *Constraints*, 11(2):199–219, July 2006.
- [31] A. Ignatiev. Rc2-2018@ maxsat evaluation 2020. *MaxSAT Evaluation 2020*, page 13.
- [32] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.
- [33] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, Jan. 2010. Publisher: IOS Press.
- [34] Z. Lei and S. Cai. Solving (weighted) partial maxsat by dynamic local search for sat. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1346–1352. International Joint Conferences on Artificial Intelligence Organization, 7 2018.
- [35] Logic and Optimization Group. Pypbilib. <https://pypi.org/project/pypbilib/>, 2019. University of Lleida.
- [36] E. Maltais and L. Moura. Finding the best CAFE is np-hard. In *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, pages 356–371, 2010.
- [37] V. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for Weighted Boolean Optimization. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, Lecture Notes in Computer Science, pages 495–508, Berlin, Heidelberg, 2009. Springer.
- [38] A. Morgado, F. Heras, M. H. Liffiton, J. Planes, and J. Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013.
- [39] A. Nadel. Tt-open-wbo-inc-20: an anytime maxsat solver entering mse’20. *MaxSAT Evaluation 2020*, page 32.
- [40] T. Nanba, T. Tsuchiya, and T. Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E95.A(9):1501–1505, 2012.
- [41] T. Nanba, T. Tsuchiya, and T. Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Transactions*, 95-A(9):1501–1505, 2012.
- [42] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):1–29, 2011.
- [43] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.
- [44] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, page 254–264, New York, NY, USA, 2011. Association for Computing Machinery.
- [45] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.
- [46] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E.-H. Choi. Greedy combinatorial test case generation using unsatisfiable cores. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, page 614–624, New York, NY, USA, 2016. Association for Computing Machinery.
- [47] A. Yamada, T. Kitamura, C. Artho, E. Choi, Y. Oiwa, and A. Biere. Optimization of combinatorial testing by incremental SAT solving. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.
- [48] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn. Constraint handling in combinatorial test generation using forbidden tuples. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–9, 2015.