Artificial Intelligence 345 (2025) 104346

Contents lists available at ScienceDirect

Artificial Intelligence

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

Disjoint projected enumeration for SAT and SMT without blocking clauses

Giuseppe Spallitta [a,*], Roberto Sebastiani [a], Armin Biere [b]

[a] *University of Trento, Via Sommarive 9, Trento, 38123, Italy*
[b] *University of Freiburg, Fahnenbergplatz, Freiburg, 79085, Germany*

ARTICLE INFO

Keywords:
AllSAT
AllSMT
Chronological backtracking
Implicant shrinking

ABSTRACT

All-Solution Satisfiability (AllSAT) and its extension, All-Solution Satisfiability Modulo Theories (AllSMT), have become more relevant in recent years, mainly in formal verification and artificial intelligence applications. The goal of these problems is the enumeration of all satisfying assignments of a formula (for SAT and SMT problems, respectively), making them useful for test generation, model checking, and probabilistic inference. Nevertheless, traditional AllSAT algorithms face significant computational challenges due to the exponential growth of the search space and inefficiencies caused by blocking clauses, which cause memory blowups and degrade unit propagation performance in the long term. This paper presents two novel solvers: TabularAllSAT, a projected AllSAT solver, and TabularAllSMT, a projected AllSMT solver. Both solvers combine Conflict-Driven Clause Learning (CDCL) with chronological backtracking to improve efficiency while ensuring disjoint enumeration. To retrieve compact partial assignments we propose a novel aggressive implicant shrinking algorithm, compatible with chronological backtracking, to minimize the number of partial assignments, reducing overall search complexity. Furthermore, we extend the solver framework to handle projected enumeration and SMT formulas effectively and efficiently, adapting the baseline framework to integrate theory reasoning and the distinction between important and non-important variables. An extensive experimental evaluation demonstrates the superiority of our approach compared to state-of-the-art solvers, particularly in scenarios requiring projection and SMT-based reasoning.

1. Introduction

Given a propositional formula F over a set of Boolean variables V, the All-Solution Satisfiability (AllSAT) problem consists of identifying all possible satisfying assignments for F. AllSAT has seen significant use across various fields, particularly in hardware and software verification. For example, it has been applied in the automatic generation of program test suites [1], as well as in both bounded and unbounded model checking [2]. Additionally, AllSAT has been employed in data mining, specifically in solving the frequent itemset mining problem [3].

AllSAT can be extended to richer logical frameworks in the form of AllSMT (All Satisfiability Modulo Theories) [4,5]. Whereas AllSAT focuses on Boolean variables and propositional formulas, AllSMT expands to formulas F with variables from more complex domains, interpreted over one or multiple specific first-order logic theories $\mathcal{T}$ (e.g., linear integer arithmetic ($\mathcal{LIA}$), linear real

* Corresponding author.
E-mail addresses: giuseppespallitta@yahoo.it, gs81@rice.edu (G. Spallitta).

https://doi.org/10.1016/j.artint.2025.104346
Received 25 November 2024; Received in revised form 8 April 2025; Accepted 21 April 2025

Available online 29 April 2025

arithmetic ($\mathcal{LRA}$) or bit-vectors ($\mathcal{BV}$)). The goal in AllSMT is to enumerate all satisfying assignments η for F under the constraints of the theory $\mathcal{T}$. Recently, AllSMT has gained interest in artificial intelligence and has been used for tasks such as probabilistic inference in hybrid domains via Weighted Model Integration (WMI) [6,7]. Moreover, model counting over first-order theories (#SMT) [8], static analysis for quantifying information flow [9], and the extraction of theory lemmas to generate canonical decision diagrams modulo theories [10] rely on enumeration strategies from AllSMT.

In both AllSAT and AllSMT, the concept of *projection* plays an essential role. It involves restricting the enumeration to a specific subset V_r of the variables in V, thereby simplifying the search of satisfying models by ignoring the truth value of variables outside of V_r. Projection is particularly beneficial for the enumeration of non-CNF formulas. When formulas are transformed into CNF—often via the Tseitin transformation—projection is essential to exclude the newly introduced auxiliary variables. Projection has been applied in numerous domains other than enumeration, including predicate abstraction [4], image computation [11,12], quantifier elimination [13], and model checking [14].

Computational challenges in enumeration Enumerating all solutions of a given formula F is a significantly more computationally intensive task than solving a single SAT instance. When dealing with complex problems, several aspects must be carefully considered to make enumeration viable.

One major challenge is the growth of the search space when dealing with complex instances. For a formula F with n variables, there are 2^n possible total assignments. Explicitly enumerating all of these solutions would require exponential space, which is impractical for large values of n. To mitigate this issue, **partial models** can be utilized to provide more concise representations of the solution set. A partial model is an assignment that leaves some variables unspecified, implying that the truth value of these variables does not influence the satisfiability of the formula in that particular assignment. Consequently, a partial assignment with m specified variables represents 2^{n-m} total assignments, effectively reducing the solution space to explore.

Another critical aspect to consider in enumeration is the handling of repeated models. In some contexts, allowing the repetition of the same model in the enumeration might be acceptable or even desirable, such as in predicate abstraction applications. However, in other scenarios like Weighted Model Integration and #SMT, repeating the same model can lead to incorrect results or inefficiencies. In this paper, we focus on disjoint enumeration, where repetitions of the same assignment are strictly prohibited.

Related work SAT-based enumeration algorithms can be categorized into two main types: **blocking solvers** and **non-blocking solvers**.

Blocking AllSAT solvers [15,2,16] are built on top of Conflict-Driven Clause Learning (CDCL) and non-chronological backtracking (NCB). These solvers work by adding **blocking clauses** to the formula each time a model is found. A blocking clause is designed to exclude the current satisfying assignment, ensuring that the solver does not find and return the same assignment in subsequent searches. This process is repeated until all possible satisfying assignments have been enumerated, effectively scanning the entire search space. Even though blocking solvers are relatively straightforward to implement and can be modified to retrieve partial assignments, they face significant efficiency challenges as the number of models increases. Specifically, an exponential number of blocking clauses might be required to cover the search space entirely. As more blocking clauses are added, unit propagation—the process of deducing variable assignments from the existing clauses—becomes increasingly difficult, leading to degraded performance.

Non-blocking AllSAT solvers [12,17] address the inefficiencies associated with blocking clauses by avoiding their use altogether. Instead, these solvers employ chronological backtracking (CB) [18]. In chronological backtracking, when a conflict arises during the search process, the solver backtracks to the most recently assigned variable, rather than jumping back non-chronologically as in CDCL. This method avoids covering the same model multiple times without the performance degradation caused by an excessive number of blocking clauses. However, non-blocking solvers have their limitations. Publicly available implementations of non-blocking solvers generate total assignments, as obtaining short partial assignments with chronological backtracking remains a complex and largely unexplored challenge. This is due to the strong dependency on decision order, which limits the effectiveness of standard minimization techniques. Additionally, non-blocking solvers can struggle to efficiently escape regions of the search space that contain no solutions, which can lead to inefficiencies in certain scenarios.

[19] introduces a formal calculus for disjunctive model counting that seeks to combine the strengths of both chronological backtracking and CDCL. This approach offers a promising direction, but the original work did not include an implementation or empirical results to demonstrate its effectiveness. Moreover, the calculus does not address how to effectively handle projected enumeration, and extending these methods to problems that include first-order logic theories.

Another SAT-based approach for enumeration that is particularly useful for non-CNF formulas is based on the idea of *entailment* [20]. Typically, when given a partial assignment μ, SAT solvers check whether μ satisfies F by substituting all the assigned variables in F with their corresponding truth values and recursively propagating these values through the formula. If this process results in $\top$, then μ is said to satisfy F, a concept known as "evaluation to true." Entailment, on the other hand, operates differently. A partial assignment μ entails F if every total assignment η that extends μ also satisfies F. In other words, after substituting the variables in F with the values from μ and propagating, the residual formula must be valid. Whereas determining whether a partial assignment entails a formula is computationally more expensive than simply checking if the formula evaluates to true given μ, it has been shown to be effective in generating compact partial models for enumeration [21]. A few formal calculi have been developed to implement enumeration algorithms that leverage dual reasoning during enumeration [22,23].

A second SAT-based approach for enumeration is known as *dual encoding.* This technique relies on a dual representation of the formula that enables efficient detection of partial models while pruning the search space. Unlike traditional approaches, dual encoding avoids explicit satisfiability checks and clause watching mechanisms, reducing overhead in identifying when a partial assignment satisfies the formula. The main achievement is that, instead of shrinking total assignments, as is typically done in other AllSAT solvers, it directly extracts partial solutions. However, experimental results indicate that DUALIZA is not yet competitive with state-of-the-art AllSAT solvers.

An alternative to SAT-based methods involves compiling a formula into a decomposable, deterministic negation normal form (d-DNNF). This structure allows for efficient retrieval of partial assignments that satisfy F. To extract a single satisfying assignment, one can traverse the d-DNNF from the root, selecting exactly one child at each OR node but follow all children from AND nodes. The properties of d-DNNF ensure that each assignment is mutually exclusive from the others. Building on this concept, a recently proposed tool leverages a depth-first traversal of the d-DNNF representation of a Boolean formula to enumerate models, ensuring that memory usage remains bounded by the size of F [24]. It is important to note, however, that this approach is inherently designed for AllSAT and does not consider projection or first-order logic theories. Recently a new calculus to convert CNF formulas to d-DNNF using chronological backtracking has been presented [25].

The literature on AllSMT is very limited, and AllSMT algorithms are highly based on AllSAT techniques and tools. For instance, MATHSAT5 [26] implements an AllSMT functionality based on the procedure by [4], relying on learning blocking clause to search all possible satisfying assignments.

Our contribution Based on the formal calculus in [19], in this paper we present TABULARALLSAT and TABULARALLSMT, respectively a projected AllSAT solver and a projected AllSMT solver that combine CDCL and chronological backtracking to avoid the introduction of blocking clauses. In particular, our main contributions can be summarized as follows:

- (a) We discuss the AllSAT procedure to perform disjoint partial enumeration of propositional formulas by combining the best of current All- SAT state-of-the-art literature: (i) CDCL, to escape search branches where no satisfiable assignments can be found; (ii) chronological backtracking, to ensure no blocking clauses are introduced; (iii) efficient implicant shrinking, to reduce in size partial assignments, by exploiting the 2-literal watching scheme.
- (b) We propose two implicant shrinking algorithms, intending to reduce the number of partial assignments retrieved by the AllSAT procedure while making sure the calculus in [19] is not violated.
- (c) We extend our procedure to deal with projected enumeration, showing how chronological backtracking and CDCL have been adapted to enumerate only a subset of important variables.
- (d) We extend our procedure to deal with the enumeration of SMT formulas, showing how chronological backtracking and CDCL have been adapted to integrate theory reasoning.
- (e) We perform an extensive experimental evaluation, showing the superiority of our proposed techniques against the state-of-the-art algorithms.

Disclaimer A preliminary and much shorter version of this paper was presented at the AAAI24 conference [27], presenting only the baseline algorithm to perform disjoint enumeration without introducing blocking clauses (a) and the first of the two chronological implicant shrinking algorithm discussed in this work (b). We refer to this baseline algorithm in the manuscript as TABULARALL-SAT$_{AAAI24}$.

This paper leverages the algorithm from [27] by proposing a novel implicant shrinking that allows the retrieval of shorter partial assignments without affecting computational times and is not heavily affected by variable ordering as the baseline algorithms (b). Moreover, we extended the algorithm to deal with projected enumeration (c) and SMT enumeration (d). Finally we provided an extensive and detailed experimental evaluation, to compare the implicant shrinking algorithms and the novel TABULARALLSAT algorithm against the state-of-the-art solvers (e).

Structure of the paper The rest of the paper is organized as follows. In §2 we introduce the background, focusing on the notation adopted, the CDCL algorithm, and chronological backtracking. In §3 we briefly summarize the algorithm to perform AllSAT integrating CDCL and chronological backtracking with no need for blocking clauses, representing a summary of [27] and the baseline of extensions discussed in this work. In §4 we discuss the novel implicant shrinking algorithm, to make it more effective. In §6 and §7 we extend the baseline algorithm to address respectively projected SAT enumeration[19] and SMT enumeration, showing the main difference and design choices that were required to make it compliant with the calculus in [19]. At the end of sections §4, §6 and §7 an extensive experimental evaluation is presented, comparing our tool's latest version against the few publicly available state-of-the-art competitors.

2. Background

2.1. Notation

We assume F is a propositional formula defined on the set of Boolean variables $V = \{v_1, ..., v_n\}$, with cardinality $|V|$. A *literal* ℓ is a variable v or its negation $\neg v$. The function $var(\ell)$ maps a literal to the associated variable. When dealing with projected enumeration, the set of variables V is split into two disjoint sets: the set of relevant variables V_r and the set of irrelevant variables V_i. $L(V)$ denotes the set of literals on V. We implicitly remove double negations: if ℓ is $\neg v$, by $\neg\ell$ we mean v rather than $\neg\neg v$. A *clause* is the disjunction of literals $\bigvee_{\ell\in c} \ell$. A *cube* is the conjunction of literals $\bigwedge_{\ell\in c} \ell$.

A function $M : V \mapsto \{\top, \bot\}$ mapping variables in F to their truth value is known as *assignment*. An assignment can be represented by either a set of literals $\{\ell_1, ..., \ell_n\}$ or a cube conjoining all literals in the assignment $\ell_1 \wedge ... \wedge \ell_n$. We distinguish between *total assignments* η or *partial assignments* μ depending on whether all variables are mapped to a truth value or not, respectively.

A *trail* is an ordered sequence of literals $I = \ell_1, ..., \ell_n$ with no duplicate variables. The empty trail is represented by ε. Two trails can be conjoined one after the other $I = KL$, assuming K and L have no variables in common. We use superscripts to mark literals in a trail I: ℓ^d indicates a literal assigned during the decision phase, whereas ℓ^* refers to literals whose truth value is negated due to chronological backtracking after finding a model (we will refer to this action as *flipping*). Trails can be seen as ordered *total* (resp. *partial*) assignments; for the sake of simplicity, we will refer to them as *total* (resp. *partial*) trails.

Definition 1. The *decision level function* $\delta : V \mapsto \mathbb{N} \cup \{\infty\}$ returns the decision level of variable V, where ∞ means unassigned. We extend this concept to literals ($\delta(\ell) = \delta(var(\ell))$) and clauses ($\delta(C) = \{max(\delta(\ell)) | \ell \in C\}$).

Definition 2. The *decision literal function* $\sigma : \mathbb{N} \mapsto L(V) \cup \{\varepsilon\}$ returns the decision literal of a specified level. If we have not decided on a literal at the specified level yet, we return ε.

Definition 3. The *reason function* $\rho(\ell)$ returns the reason that forced literal ℓ to be assigned a truth value:

- DECISION, if the literal is assigned by the decision selection procedure;
- UNIT, if the literal is unit propagated at decision level 0, thus it is an initial literal;
- PROPAGATED(c), if the literal is unit propagated at a decision level higher than 0 due to clause c;

In addition to these standard values discussed in the literature, in this paper, we discuss a new value, BACKTRUE, which is used in the case a literal is unit propagated after a model has been found. More details on it are discussed in §3.2.

2.2. AllSAT, AllSMT and projection

AllSAT is the task of enumerating all the truth assignments propositionally satisfying a propositional formula. The task can be found in the literature in two versions: *disjoint* AllSAT, in which the assignments are required to be pairwise mutually inconsistent, and *non-disjoint* AllSAT, in which they are not. For instance, given the formula $F = A \vee B$, a *non-disjoint* AllSAT solver could generate the partial assignments $\mu_1 = A$ and $\mu_2 = B$. Since the total model $\eta_1 = A \wedge B$ is a superset of both assignments, the enumeration is *not* disjoint. In contrast, a *disjoint* AllSAT solver ensures that each assignment is mutually inconsistent with the others. For the same formula, it could generate $\mu_1 = A$ and $\mu_2 = \neg A \wedge B$ (or alternatively, $\mu_1 = B$ and $\mu_2 = \neg B \wedge A$), preventing overlap between assignments and maintaining disjointness. In this paper, we will focus on disjoint enumeration. A generalization to the SMT($\mathcal{T}$) case is AllSMT($\mathcal{T}$), defined as the task of enumerating all the *$\mathcal{T}$-satisfiable* truth assignments propositionally satisfying a SMT($\mathcal{T}$) formula.

Projection is a process related to SAT and AllSAT solving that involves ignoring irrelevant variables V_i from a Boolean formula while preserving the satisfiability of the formula with respect to the remaining relevant variables V_r. The goal of projection is to reduce the dimensionality of a Boolean expression by "projecting" it onto a subset of its variables, effectively discarding those that are not relevant to the problem at hand. In particular, given a formula F under the set of variables $V_r \cup V_i$ s.t. $V_r \cap V_i = \emptyset$, the enumeration of F projected onto the relevant variables V_r consists of:

$$ProjAllSAT(F(V_r, V_i)) = AllSAT(\exists V_i . F(V_r, V_i)) \tag{1}$$

For example, the enumeration of a non-CNF formula F can be carried out by first converting it into CNF and then enumerating its satisfying assignments by means of *Projected AllSAT*. Specifically, given a non-CNF formula $F(\mathbf{A})$, we can apply either the Tseitin [28] or Plaisted-Greenbaum [29] transformation to obtain $F_{CNF}(\mathbf{A} \cup \mathbf{B})$, where $\mathbf{B}$ represents the Boolean variables introduced by the transformation. Enumeration is then performed over the partial assignments to $\mathbf{A}$ that can be extended to total truth assignments satisfying F_{CNF} over $\mathbf{A} \cup \mathbf{B}$. Here, the original set of variables $\mathbf{A}$ corresponds to V_r, whereas the additional variables $\mathbf{B}$, introduced during the CNF transformation, correspond to V_i. We refer the reader to [30] for an analysis of CNF-ization for enumeration.

2.3. The 2-watched literal scheme

The *2-watched literal scheme* [31] is an indexing technique that efficiently checks if the currently-assigned literals do not cause a conflict. For every clause, two literals are tracked. If at least one of the two literals is set to $\top$, then the clause is satisfied. If one of the two literals is set to $\bot$, then we scan the clause searching for a new literal ℓ' that can be paired with the remaining one, being sure ℓ' is not mapped to $\bot$. If we reach the end of the clause and both watches for that clause are set to false, then we know the current assignment falsifies the formula. Additionally, if after updating the watches only one remaining literal in the clause is unassigned, that literal must be set to $\top$ to satisfy the clause, triggering unit propagation. The 2-watched literal scheme is implemented through watch lists.

Definition 4. The *watch list function* $\omega(\ell)$ returns the set of clauses $\{c_1, ..., c_n\}$ currently watched by literal ℓ.

2.4. CDCL and non-chronological backtracking

Conflict Driven Clause Learning (CDCL) is the most popular SAT-solving technique [32]. It is an extension of the older Davis-Putnam-Logemann-Loveland (DPLL) algorithm [33], improving the latter by dynamically learning new clauses during the search process and using them to drive backtracking.

Every time the current trail falsifies a formula F, the SAT solver generates a conflict clause c starting from the falsified clause, by repeatedly resolving against the clauses which caused unit propagation of falsified literals. This clause is then learned by the solver and added to F. After an analysis based on c, we backtrack to flip the value of one literal, potentially jumping more than one decision level (thus we talk about *non-chronological backtracking*, or NCB). CDCL and non-chronological backtracking allow for escaping regions of the search space where no satisfying assignments are admitted, which benefits both SAT and AllSAT solving. The idea behind conflict clauses has been extended in AllSAT to learn clauses from partial satisfying assignments (known in the literature as *good learning* or *blocking clauses* [34,35]) to ensure no total assignment is covered twice.

2.5. Chronological backtracking

Chronological backtracking (CB) is the core of the original DPLL algorithm. Considered inefficient for SAT solving once NCB was presented in [31], it was recently revamped for both SAT and AllSAT in [18,36]. The intuition is that non-chronological backtracking after conflict analysis can lead to redundant work, due to some assignments that could be repeated later on during the search. Instead, independently of the generated conflict clause c we chronologically backtrack and flip the last decision literal in the trail. Consequently, we explore the search space systematically and efficiently, ensuring no assignment is covered twice during execution. Chronological backtracking combined with CDCL is effective in SAT solving when dealing with satisfiable instances. In AllSAT solving, it guarantees blocking clauses are no more needed to ensure termination.

2.6. Benefits and drawbacks of CDCL and CB for enumeration

Enumeration is generally more challenging compared to SAT solving. In SAT solving, the search terminates as soon as a solution is found, whereas enumeration requires exploring the entire search space to identify all possible solutions. This makes the task of enumeration strictly more difficult than finding a single satisfying assignment.

Considering SAT-based enumeration algorithms, there is no clear supremacy between blocking and non-blocking AllSAT solvers. In particular, we can highlight the following strengths and weaknesses:

- **Systematic Search:** chronological backtracking systematically scans the entire search space, ensuring that all regions are visited without repetition, particularly regions with no solution. CDCL, on the other hand, may not guarantee that some areas are not visited more than once, revisiting areas with no solution multiple times, unless blocking clauses are added.
- **Blocking clauses:** due to the systematic nature of chronological backtracking, there is no need for additional blocking clauses to prevent redundant exploration of the search space. In contrast, CDCL relies on blocking clauses to avoid revisiting previously explored areas. This may require adding an up-to exponential number of blocking clauses, causing memory blowups and a degradation of unit propagation performances.
- **Conflict Analysis:** in areas of the search space with no solution, CDCL can leverage conflict analysis to escape and redirect the search to other regions quickly. Chronological backtracking, on the other hand, may become trapped in such regions until the entire sub-search space is fully explored.
- **Time efficiency:** due to its ability to escape regions of the search space with no solution, CDCL-based approaches can generally enumerate solutions faster than algorithms based on chronological backtracking.
- **Shrinking Techniques and Partial Assignments:** whereas there is extensive discussion in the literature regarding shrinking techniques associated with CDCL for enumeration and the generation of partial assignments, so far these topics have not been addressed in the context of chronological backtracking.

3. Enumerating disjoint partial models without blocking clauses

We summarize the approach allowing for enumerating disjoint partial models with no need for blocking clauses discussed in [27], that integrates: Conflict-Driven Clause-Learning (**CDCL**), to escape search branches where no satisfiable assignments can be found; Chronological Backtracking (**CB**), to ensure no blocking clauses are introduced; and methods for shrinking models (**Implicant Shrinking**), to reduce in size partial assignments.

Several algorithms are proposed in this section, and we use a colored notation to mark significant differences with respect to baseline AllSAT solving and extensions to the original algorithm presented in [27]. In particular:

- For Algorithms 1-3, we color in **red** (For interpretation of the references to color please refer to the web version of this article.) all lines that differ from the baseline CDCL AllSAT solving algorithm.
- For all algorithms, we color in **green** additional conditions and procedures that must be executed to perform projected enumeration.
- For all algorithms, we color in **blue** additional conditions and procedures that must be executed to perform SMT-based enumeration.

Algorithm 1 CHRONO-CDCL(F, V).

```
1: T ← ε
2: dl ← 0
3: while true do
4:     T, c ← UnitPropagation()
5:     if c ≠ ε then
6:         AnalyzeConflict(T, c, dl)
7:     else if |T| = |V| then
8:         c_T ← Check-Theory-Consistency(T)
9:         if c_T ≠ ε then
10:            AnalyzeConflict(T, c_T, dl)
11:            continue
12:        end if
13:        AnalyzeAssignment(T, dl)
14:    else
15:        Decide(T)
16:        dl ← dl + 1
17:    end if
18: end while
```

Algorithm 2 ANALYZECONFLICT(T, c, dl).

```
1: if δ(c) < dl then
2:     T ← Backtrack(δ(c))
3: end if
4: if dl = 0 then
5:     terminate with all models found
6: end if
7: ⟨uip, c′⟩ ← LastUIP-Analysis()
8: T ← Backtrack(dl − 1)
9: T.push(¬uip)
10: limit ← dl − 1
11: ρ(¬uip) ← Propagated(c′)
```

3.1. Disjoint AllSAT by integrating CDCL and CB

The work in [19] exclusively describes the calculus and a formal proof of correctness for a model counting framework on top of CDCL and CB, with neither any algorithm nor any reference in modern state-of-the-art solvers. To this extent, we start by presenting an AllSAT procedure for the search algorithm combining the two techniques, which are reported in this section. In particular, we highlight the major differences to a classical AllSAT solver implemented on top of CDCL and NCB.

Algorithm 1 presents the main search loop of the AllSAT algorithm. (In Alg. 1 the reader is supposed to ignore temporarily the blue lines from 8 to 12, which refer to the SMT version of the algorithm and which will be illustrated in §6.)

The goal is to find a total trail T that satisfies F. At each decision level, it iteratively decides one of the unassigned variables in F and assigns a truth value (Algorithm 1, lines 15-16); it then performs unit propagation (Algorithm 1, line 4) until either a conflict is reached (Algorithm 1, lines 5-6), or no other variable can be unit propagated leading to a satisfying total assignment (Algorithm 1, lines 7-13) or DECIDE has to be called again (Algorithm 1, lines 15-16).

Notice that the main loop is identical to an AllSAT solver based on non-chronological CDCL; the only differences are embedded in the procedure to get the conflict and the partial assignments. (We remark that from now on we color in **red** the lines that differ from the baseline CDCL AllSAT solver.)

Suppose UNITPROPAGATION finds a conflict, returning one clause c in F which is falsified by the current trail T, so that we invoke ANALYZECONFLICT. Algorithm 2 shows the procedure to either generate the conflict clause or stop the search for new assignments if all models have been found.

We first compute the maximum assignment level of all literals in the conflicting clause c and backtrack to that decision level (Algorithm 2, lines 1-2) if strictly smaller than dl. This additional step, not contemplated by AllSAT solvers that use NCB, is necessary to support out-of-order assignments, the core insight in chronological backtracking when integrated into CDCL as described in [18].

Apart from this first step, Algorithm 2 behaves similarly to a standard conflict analysis algorithm. If the solver reaches decision level 0 at this point, it means there are no more variables to flip and the whole search space has been visited, and we can terminate the algorithm (Algorithm 2, lines 4-5). Otherwise, we perform conflict analysis up to the last Unique Implication Point (last UIP, i.e. the decision variable at the current decision level), retrieving the conflict clause c' (Algorithm 2, line 7), as proposed in [19]. Finally, we perform backtracking (notice how we force chronological backtracking independently from the decision level of the conflict clause), push the flipped UIP into the trail, and set c' as its assignment reason for the flipping (Algorithm 2, lines 8-11). (The reader should temporarily skip line 10: the role of variable $limit$ is explained in §4.2).

Suppose instead that every variable is assigned a truth value without generating conflicts (Algorithm 1, line 7); then the current total trail T satisfies F, and we invoke ANALYZEASSIGNMENT, which possibly shrinks the assignment T and updates the decision level dl, stores T, and then continues the search.

Algorithm 3 ANALYZEASSIGNMENT(T, dl).

1: $dl' \leftarrow$ IMPLICANT-SHRINKING(T)
2: **if** $dl' < dl$ **then**
3: $\quad T \leftarrow$ BACKTRACK(dl')
4: **end if**
5: **store model** T
6: **if** $dl' = 0$ **then**
7: $\quad$ **terminate with all models found**
8: **else**
9: $\quad \ell_{flip} \leftarrow \neg(\sigma(dl'))$
10: $\quad T \leftarrow$ BACKTRACK($dl' - 1$)
11: $\quad T.push(\ell_{flip})$
12: $\quad limit \leftarrow dl' - 1$
13: $\quad \rho(\ell_{flip}) =$ BACKTRUE
14: **end if**

ANALYZEASSIGNMENT is illustrated in Algorithm 3. First, IMPLICANT-SHRINKING checks if, for some decision level dl', we can backtrack up to $dl' < dl$ and obtain a partial trail still satisfying the formula (Algorithm 3, lines 1-3). (We discuss the details of chronological implicant shrinking in the next subsection.) We can produce the current assignment from the current trail T (Algorithm 3, line 5). Then we check if all variables in T are assigned at decision level 0. If this is the case, then this means that we found the last assignment to cover F, so that we can end the search (Algorithm 3, lines 6-7). Otherwise, we perform chronological backtracking, flipping the truth value of the currently highest decision variables and searching for a new total trail T satisfying F (Algorithm 3, lines 9-13).

We remark that in [19] it is implicitly assumed that one can determine if a partial trail satisfies the formula right after being generated, whereas modern SAT solvers cannot check this fact efficiently, and detect satisfaction only when trails are total. To cope with this issue, in our approach we first *temporarily* generate a total trail satisfying the formula, then the partial trail is computed *a posteriori* from the total one by implicant shrinking, mimicking the construction of the partial trail in [19]. The mutual exclusivity among different assignments is guaranteed, since the shrinking of the assignments is performed so that the generated partial assignments fall under the conditions of §3 in [19]).

Notice that the calculus discussed in [19] assumes the last UIP is the termination criteria for the conflict analysis. We provide the following counter-example to show that the first UIP does not guarantee mutual exclusivity between returned assignments.

Example 1. Let F be the propositional formula:

$$F = \overbrace{(x_1 \vee \neg x_2)}^{c_1} \wedge \overbrace{(x_1 \vee \neg x_3)}^{c_2} \wedge \overbrace{(\neg x_1 \vee \neg x_2)}^{c_3}$$

For the sake of simplicity, we assume CHRONO-CDCL to return total truth assignments. If the initial variable ordering is x_3, x_2, x_1 (all set to false) then the first two total and the third partial trails generated by Algorithm 1 are:

$$T_1 = \neg x_3^d \neg x_2^d \neg x_1^d; \;\; T_2 = \neg x_3^d \neg x_2^d x_1^*; \;\; T_3 = \neg x_3^d x_2^*$$

Notice how T_3 leads to a falsifying assignment: x_2 forces x_1 due to c_1 and $\neg x_1$ due to c_3 at the same time. A conflict arises and we adopt the first UIP algorithm to stop conflict analysis. We identify x_2 as the first unique implication point (UIP) and construct the conflict clause $\neg x_2$. Since this is a unit clause, we force its negation $\neg x_2$ as an initial unit. We can now set x_3 and x_1 to $\perp$ and obtain a satisfying assignment. The resulting total trail $T = \neg x_3 \neg x_2 \neg x_1$ is covered **twice** during the search process. ⋄

We also emphasize that the incorporation of restarts in the search algorithm (or any method that implicitly exploits restarts, such as rephasing) is not feasible, as reported in [19].

3.2. Implicit solution reasons

Incorporating chronological backtracking into the AllSAT algorithm makes blocking clauses unnecessary. Upon discovering a model, we backtrack chronologically to the most recently assigned decision variable ℓ and flip its truth value, as if there were a reason clause c - containing the negated decision literals of T - that forces the flip. These reason clauses c are typically irrelevant to SAT solving and are not stored in the system. On the other hand, when CDCL is combined with chronological backtracking, these clauses are required for conflict analysis.

Example 2. Let F be the same formula from Example 1. We assume the first trail generated by Algorithm 1 is $T_1 = \neg x_3^d \neg x_2^d \neg x_1^d$. Algorithm 4 can reduce x_1 since $\neg x_2$ suffices to satisfy both c_1 and c_3. (More details about the minimization procedure are discussed in the next section, and they are not relevant for this example). Consequently, we obtain the assignment $\mu_1 = \neg x_3 \wedge \neg x_2$, then flip $\neg x_2$ to x_2. The new trail $I_2 = \neg x_3^d x_2^*$ forces x_1 to be true due to c_1; then c_3 would not be satisfiable anymore and cause the generation of a conflict. The last UIP is x_3, so that the reason clause c' forcing x_2 to be flipped must be handled by the solver to compute the conflict clause. ⋄

To cope with this fact, a straightforward approach would be storing these clauses in memory with no update to the literal watching indexing; this approach would allow for c to be called exclusively by the CDCL procedure without affecting variable propagation. If F admits a large number of models, however, storing these clauses would negatively affect performance, so either we had to frequently call flushing procedures to remove inactive backtrack reason clauses, or we could risk going out of memory to store them.

To overcome the issue, we introduce the notion of *virtual backtrack reason clauses*. When a literal ℓ is flipped after a satisfying assignment is found, its reason clause contains the negation of decision literals assigned at a level lower than $\delta(\ell)$ and ℓ itself. Consequently, we introduce an additional value, BACKTRUE, to the possible answers of the reason function ρ. This value is used to tag literals flipped after a (possibly partial) assignment is found. When the conflict analysis algorithm encounters a literal ℓ having $\rho(\ell) =$ BACKTRUE, the resolvent can be easily reconstructed by collecting all the decision literals with a lower level than ℓ and negating them. This way we do not need to explicitly store these clauses for conflict analysis, allowing us to save time and memory for clause flushing.

It is important to note that an implicant shrinking algorithm cannot remove literals marked with a BACKTRUE flag, as these are essential for ensuring that subsequent assignments remain mutually exclusive from previous ones. Specifically, for each literal with a BACKTRUE reason, there exists an implicit blocking clause C_b that includes ℓ and the negation of all decision literals up to ℓ. While these blocking clauses are not explicitly generated, the implicant shrinking algorithm must ensure that no literal flagged with BACKTRUE is dropped. Failing to preserve these literals would break the implicit blocking clauses, thereby compromising the disjointness of the assignments. With a similar reasoning, the first literal unit-propagated after conflict analysis cannot be removed from a trail, since it guarantees that the search space is sistematically scanned without repetitions. These remarks are fundamental when dealing with assignment shrinking and are further discussed in §4.2.

3.3. Decision variable ordering

As shown in [19], different orders during DECIDE can lead to a different number of partial trails retrieved if chronological backtracking is enabled. After an empirical evaluation, we set DECIDE to select the priority score of a variable depending on the following ordered set of rules.

First, we rely on the Variable State Aware Decaying Sum *(VSADS)* heuristic [37] and set the priority of a variable according to two weighted factors: (i) the count of variable occurrences in the formula, as in the Dynamic Largest Combined Sum (DLCS) heuristics; and (ii) an "activity score", which increases when the variable appears in conflict clauses and decreases otherwise, as in the Variable State Independent Decaying Sum (VSIDS) heuristic. If two variables have the same score, we set a higher priority to variables whose watch list is not empty. If there is still a tie, we rely on the lexicographic order of the names of the variables.

4. Chronological implicant shrinking

Effectively shrinking a total trail T when chronological backtracking is enabled is not trivial.

In principle, we could add a flag for each clause c stating if c is currently satisfied by the partial assignment or not, and check the status of all flags iteratively adding literals to the trail. Despite being easy to integrate into an AllSAT solver and avoiding assigning all variables a truth value, this approach is unfeasible in practice: every time a new literal ℓ is added/removed from the trail, we should check and eventually update the value of the flags of clauses containing it. In the long term, this would negatively affect performance, particularly when the formula has a large number of models.

Also, relying on implicant shrinking algorithms from the literature for NCB-based AllSAT solvers does not work for chronological backtracking. Prime-implicant shrinking algorithms do not guarantee the mutual exclusivity between different assignments, so that they are not useful in the context of disjoint AllSAT. Other assignment-shrinking algorithms, as in [38], work under the assumption that a blocking clause is introduced. For instance, suppose we perform disjoint AllSAT on the formula $F = x_1 \vee x_2$ and the ordered trail is $T_1 = x_1^d x_2^d$. A general assignment shrinking algorithm could retrieve the partial assignment $\mu = x_2$ satisfying F, but obtaining it by using chronological backtracking is not possible (it would require us to remove x_1 from the trail despite being assigned at a lower decision level than x_2) unless blocking clauses are introduced.

In this context, we need an implicant shrinking algorithm such that: (i) it is compatible with chronological backtracking, i.e. we remove variables assigned at level dl or higher as if they have never been assigned; (ii) it tries to cut the highest amount of literals while still ensuring mutual exclusivity.

4.1. Chronological implicant shrinking based on 2-watched literals

Considering all the aforementioned issues, [27] proposed a *chronological implicant shrinking* algorithm that used state-of-the-art SAT solver data structures (thus without requiring dual encoding), which is reported in Algorithm 4.

The idea is to pick literals from the current trail starting from the latest assigned literals (Algorithm 4, lines 3-4) and determine the lowest decision level b to backtrack and shrink the implicant. First, we check if ℓ was not assigned by DECIDE (Algorithm 4, line 5). If this is the case, we set b to be at least as high as the decision level of ℓ ($\delta(\ell)$), ensuring that it will not be dropped by implicant shrinking (Algorithm 4, line 6), since ℓ has a role in performing disjoint AllSAT.

If this is not the case, we compare its decision level $\delta(\ell)$ to b (Algorithm 4, line 7). If $\delta(\ell) > b$, then we actively check if it is necessary for T to satisfy F (Algorithm 4, line 8) and set b accordingly. If CHECK-LITERAL tries to remove ℓ from the trail, we check for each clause c watched by ℓ if the other 2-watched literal ℓ_2 is in T to determine if ℓ is necessary for the satisfiability of F, as if

Algorithm 4 IMPLICANT-SHRINKING(T).

1: $b \leftarrow 0$
2: $T' \leftarrow T$
3: **while** $T' \neq \varepsilon$ **do**
4: $\quad \ell \leftarrow T'.pop()$
5: $\quad$ **if** $\rho(\ell) \neq$ DECISION **then**
6: $\quad\quad b \leftarrow max(b, \delta(\ell))$
7: $\quad$ **else if** $\delta(\ell) > b$ **then**
8: $\quad\quad b \leftarrow$ CHECK-LITERAL(ℓ, b, T')
9: $\quad$ **else if** $\delta(\ell) = 0$ or ($\delta(\ell) = b$ and $\rho(\ell) =$ DECISION) **then**
10: $\quad\quad$ **break**
11: $\quad$ **end if**
12: **end while**
13: **return** b

Algorithm 5 IMPLICANT-SHRINKING-AGGRESSIVE(T).

$T', S, W, N \leftarrow T, \{\}, \{\}, \{\}$
$W, N \leftarrow$ INITIALIZE(W, N)
$S \leftarrow$ GET-IMPORTANT-LITERALS(W, N, S)
$T, dl \leftarrow$ LIFT-LITERALS(S, T, dl)
return dl

the clause c is projected into the binary clause $\ell \vee \ell_2$. If ℓ_2 is not in T, then we force the AllSAT solver to maintain ℓ, setting the backtracking level to at least $\delta(\ell)$; otherwise we move on to the next clause watched by it.

If ℓ is either an initial literal (i.e. assigned at decision level 0) or both $\rho(\ell) =$ DECISION and $\delta(\ell) = b$ hold, all literals in the trail assigned before ℓ would have a decision level lower or equal than b. This means that we can exit the loop early (Algorithm 4, lines 9-10), since scanning further the trail would be unnecessary. Finally, if none of the above conditions holds, we can assume that b is already greater than $\delta(\ell)$, and we can move on to the next literal in the trail.

This variant of implicant shrinking is conservative when it comes to dropping literals from the trail. We do not consider the possibility of another literal ℓ', currently not watching c, being in the current trail T, and having a lower decision level than the two literals watching c.

Example 3. Let F be the formula

$$F = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$$

According to the variable ordering heuristic presented in §3.3, all variables have the same VSADS score and all of them watch at least one clause. Consequently, the variable ordering will be x_1, x_2, x_3, x_4. Assume that every decision variable is set to a positive polarity, obtaining the assignment

$$\eta = x_1^d x_2^d x_3^d x_4^d$$

Whereas x_4 would be removed from the assignment by the shrinking procedure (x_3 ensures that all clauses where x_4 appear are satisfied), x_3 could not be removed, and the procedure would stop with the partial assignment

$$\mu = x_1^d x_2^d x_3^d.$$

Notice that this assignment could be further reduced to $\mu' = x_1^d x_3^d$, but due to the calculus in [19] and the order chosen by the solver, the implicant shrinking procedure is forced to stop the shrinking early on.

4.2. Simulating optimal decision variable ordering

The previous example highlights an important aspect: if the solver knew an optimal variable ordering preventing from assigning as many variables as possible, then it should postpone their assignment to the very end. Whereas this kind of prediction is not feasible, we can try to simulate it. Once a total assignment η is obtained, we can separate the variables in η into two disjoint sets; (i) the variables that are necessary to satisfy all clauses; and (ii) the remaining unnecessary variables. If the search algorithm first assigns all necessary variables before the non-necessary ones, then the literals following the last necessary literal in the trail could be dropped without affecting satisfiability, as they were never critical to the assignment. Essentially, we can remove all non-necessary literals from a trail T, regardless of their position or decision level, assuming their truth value assignments could be deferred to the end of the main search loop. It is important to note that the order of literals in a trail T does not influence whether the assignment satisfies F; any permutation of T will still satisfy F.

Starting from this idea, we present a novel chronological implicant shrinking algorithm, which focuses on performing a more effective shrinking, whose general schema is shown in Algorithm 5.

Algorithm 6 INITIALIZE(W, N).

```
for c ∈ F do
    for ℓ ∈ c do
        if ℓ ∈ T then
            W[ℓ] = W[ℓ] + c
            N[c] = N[c] + 1
        end if
    end for
end for
return W, N
```

Algorithm 7 GET-IMPORTANT-LITERALS(W, N, S).

```
1:  while T′ ≠ ε do
2:      ℓ ← T′.pop()
3:      if δ(ℓ) ≤ limit or ℓ ∉ V_r then
4:          continue
5:      end if
6:      required ← false
7:      for c ∈ W[ℓ] do
8:          if N[c] == 1 then
9:              required = true
10:         end if
11:     end for
12:     if required then
13:         S.push(ℓ)
14:     else
15:         for c ∈ W[ℓ] do
16:             N[c] = N[c] − 1
17:         end for
18:     end if
19: end while
20: return S
```

Algorithm 8 LIFT-LITERALS(S, T, dl).

```
1:  T ← BACKTRACK(limit)
2:  for ℓ ∈ S do
3:      if ℓ ∈ T then
4:          continue
5:      end if
6:      ASSIGN(ℓ)
7:      dl ← dl + 1
8:      T, c ← UNITPROPAGATION()
9:  end for
10: limit ← dl
11: return T, dl
```

We begin by initializing several auxiliary data structures (Algorithm 6): a copy of the satisfying total trail T', an empty ordered list S to store the literals from T that form the shrunk partial assignment, a map $W : \ell \mapsto c_1, \ldots, c_n$ that links each literal in T to the set of clauses containing it, and a map $N : c \mapsto \mathbb{N}$ that tracks how many literals in each clause are present in T.

We then proceed to determine the set of literals that cannot be lifted from the assignment, being necessary to satisfy F (Algorithm 7). (In Alg. 7 and 8 the reader is supposed to ignore the parts "**or** $l \notin V_r$" in green, which refer to the projected version of the algorithm and which will be illustrated in §5.) Starting from the most recently assigned literals (Algorithm 7, line 1), we evaluate each literal ℓ to determine whether there exists a clause that is exclusively watched by ℓ, indicated by a counter $N[c]$ being equal to 1 (Algorithm 7, lines 7-8). If such a clause exists, ℓ cannot be dropped from the assignment and is added to S (Algorithm 7, lines 12-13). Conversely, if no clause necessitates ℓ for the shrunk assignment, ℓ is removed from T, and the map N is updated accordingly for each clause containing ℓ (Algorithm 7, lines 15-16).

The algorithm, as described so far, does not account for chronological backtracking and might remove literals whose truth assignment is fundamental to enforce the disjointness of all assignments (as we remarked at the end of §3.3). Specifically, this issue does arise when minimizing either BACKTRUE literal or literal propagated by conflict analysis. To ensure this does not happen, we introduce an auxiliary variable, $limit$, which stores the lowest level up to which literals cannot be dropped (Algorithm 7, lines 3-4). The $limit$ variable is updated during each conflict analysis (Algorithm 2, line 10) and at the end of each implicant shrinking procedure when a literal is flipped due to a BACKTRUE reason (Alg. 3, line 12).

Once S contains a set of literals satisfying F, we conclude the shrinking procedure by dropping the remaining literals to simulate the best-case decision ordering scenario (Algorithm 8), where these removed literals would have been assigned later on and then dropped. We first backtrack to the decision level stored in $limit$ (Algorithm 8, line 1), to ensure the entire search space is scanned correctly. After that, we add the remaining literals in S that are not yet part of the current trail by assigning them one by one

(Algorithm 8, lines 2-9). Each time a literal is assigned, we perform unit propagation (Algorithm 8, line 8) to ensure that any unassigned literals in S that can have their truth value determined by unit propagation are correctly handled.

Example 4. Consider the formula

$$F = \overbrace{(x_1 \vee x_2)}^{c_1} \wedge \overbrace{(x_3 \vee x_4)}^{c_2},$$

and let

$$\eta_1 = x_1^d x_2^d x_3^d x_4^d$$

be the first total assignment that satisfies F, which we want to minimize. Since no conflict occurred before generating this assignment, limit is set to 0, meaning all literals are candidates for lifting. We initialize all auxiliary data structures, with $N[c_1]$ and $N[c_2]$ both starting at 2.

Starting with the most recent literal, x_4 can be lifted, as no clause is satisfied solely by it. It is not added to S, and $N[c_2]$ is updated to 1. Next, x_3 is processed, but it cannot be lifted: $N[c_2] = 1$, so x_3 is the only literal satisfying c_2, and it is added to S. Similarly, x_2 can be dropped, while x_1 must remain in the partial assignment. The ideal partial assignment generated is

$$\mu_1 = x_1^d x_3^d.$$

To achieve this, we backtrack to level 0, clearing the trail, and explicitly reconstruct the partial trail μ_1 by reassigning the necessary truth values. This trail is stored by the solver, and due to chronological backtracking, x_3 is flipped. At this point, limit is updated to 1, as $\neg x_3$ is assigned at level 1.

After the decision and propagation procedures, the new total trail

$$\eta_2 = x_1^d \neg x_3^* x_4 x_2^d$$

satisfies the formula. Here, limit is set to 1, meaning that to ensure only mutually exclusive partial assignments are generated, all literals assigned at levels up to limit cannot be lifted (in this case, x_1, $\neg x_3$, and x_4). On the other hand, no clause containing x_2 requires it to be satisfied, so it can be dropped. The second partial trail satisfying F is then

$$\mu_2 = x_1^d \neg x_3^* x_4.$$

4.3. About disjointness

The algorithm described in this paper implements the formal calculus introduced in [19], which guarantees disjointness by construction. We analyze the only difference wrt. [19] and show that it does not affect disjointness.

The calculus in [19] produces satisfiable partial trails on the fly, selecting only the necessary literals to detect satisfiability as early as possible. Unfortunately, as previously discussed, the selection process described in [19] is not feasible in practice. To cope with this fact, our algorithm first produces *temporarily* a total trail satisfying the formula, which is used only to drive the actual construction of the partial trail, which mimics the steps in [19]. To this extent, the process in [19] is not modified.

Specifically, the main steps are:

1. **Find a total satisfying trail:** instead of detecting a satisfiable partial trail on the fly, we first construct a total satisfying trail (Algorithm 1, line 7) which is used as the starting point of implicant shrinking (Algorithm 1, line 13).
2. **Detect important literals:** we use the chronological implicant shrinking procedure to determine which literals cannot be dropped to maintain satisfiability of the trail (Algorithm 7).
3. **Backtrack to the starting point:** once a total model is found, we backtrack to the decision level where the last flipped decision literal was assigned (Algorithm 8, line 1).
4. **Reinsert the important literals:** after backtracking, only the important literals identified in Step 1 are reinserted into the trail, leading to the discovery of a disjoint partial model (Algorithm 8, lines 2-8).

Notice that only Step 4 is the actual implementation of the calculus in [19], as it mimics its execution pattern. Steps 1, 2 and 3 are a sort of "look ahead" additional operations that allow the solver to extract the necessary information to produce the partial trail (i.e., the important literals). Since these steps do not modify the core reasoning process, the fundamental properties of the calculus, including disjointness, remain unchanged.

To illustrate this, consider a scenario where a satisfying partial trail has been found (BackTrue rule in [19]). Let ℓ be the last flipped decision literal due to chronological backtracking at decision level x, and let *limit* be set to x accordingly. After flipping ℓ, further literal trail are performed, and one or more conflicts may arise, potentially modifying the flipped literal and the value of *limit* due to conflict analysis (BackFalse rule in [19]). Since our approach does not modify how conflicts are handled—CDCL conflict analysis and resolution remain unchanged—the theoretical invariants established in [19] remain intact.

If, after these steps, a total satisfying trail is found, our algorithm applies the implicant shrinking procedure to extract the important literals. Then it backtracks to ℓ at decision level *limit*. Consequently, all literals assigned beyond *limit* are effectively disregarded, and the solver returns to the state at which the previous BACKTRUE (if no conflict occurred after its execution) or the last BACKFALSE rule (if at least one conflict arose) was triggered.

Once back at *limit*, the solver proceeds by assigning only the important literals using either solver decisions (DECIDE rule in [19]) or unit propagation (UNIT rule in [19]). When the last of these literals is assigned, the BACKTRUE rule is triggered again, yielding a new satisfiable partial trail. From this perspective, the solver behaves exactly as if it had directly followed the execution pattern of [19], assuming an optimal ordering for detecting satisfiability on the fly.

This alignment demonstrates that our approach, despite introducing additional steps, ultimately conforms to the original formal calculus. Since our algorithm ensures that backtracking consistently returns to *limit* before assigning new literals, the search space exploration remains equivalent to that in [19]. Each enumerated trail differs from previous ones by at least one decision literal, preserving all theoretical invariants. Consequently, correctness, termination, and disjointness remain fully guaranteed by the formal framework in [19].

4.4. Chronological implicant shrinking and minimality

The new implicant shrinking algorithm does have the drawback of needing to build W and N for each satisfying total assignment, which can impact performance. However, reducing the number and length of partial assignments is often the goal in many applications. Depending on the number of assignments covering a formula, the significant reduction in partial assignments and the consequent pruning of a large portion of the search space outweigh the minor inefficiency introduced by the implicant shrinking process.

We remark how the novel chronological implicant shrinking does not guarantee that the shrunk partial assignment is minimal. Minimality of a partial assignment means that given a partial assignment μ obtained by the search procedure, there is no other literal that can be removed by μ so that μ still satisfies F and it is pairwise mutually exclusive against all the assignments retrieved before it.

Example 5. Consider the formula:

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$$

We assume the search algorithm favors negative polarity the first time a variable is chosen. The first trail generated by the algorithm is $\mu_1 = \neg x_1^d \neg x_2^d \neg x_3$, the first two variables being chosen by DECIDE and the last one being unit propagated because of the second clause. The first clause, however, is now falsified, and thus a conflict arises, forcing $\neg x_2$ to be flipped. To preserve mutual exclusivity between assignments, $limit$ should be updated up to 1, avoiding dropping anything before x_2. Now the current trail $\mu_2 = \neg x_1^d x_2^*$ is forced to add x_3 to satisfy the third clause. The total trail $\eta_1 = \neg x_1^d x_2^* x_3$ satisfies the formula. The implicant shrinking algorithm is not able to drop any literal; notice, however, that $\neg x_1$ could be dropped by η_1 without altering the satisfiability and the mutual exclusivity of the assignment against the currently empty set of assignments. For this reason, the algorithm is not guaranteed to find a minimal assignment.

4.5. Chronological implicant shrinking and non-disjoint enumeration

One could wonder if the implicant shrinking procedure could be modified for non-disjoint enumeration. In particular, we could argue that if we allow the algorithm to eliminate literals before the *limit* level, then we could get shorter assignments sharing some of the total assignments under it with other partial models. However, the following example shows that under this assumption the AllSAT search is not guaranteed to terminate.

Example 6. Consider the same formula of Example 5, but this time the algorithm favors positive polarity for the first choice, we assume literals whose assignment level is lower than $limit$ can be dropped from the trail.

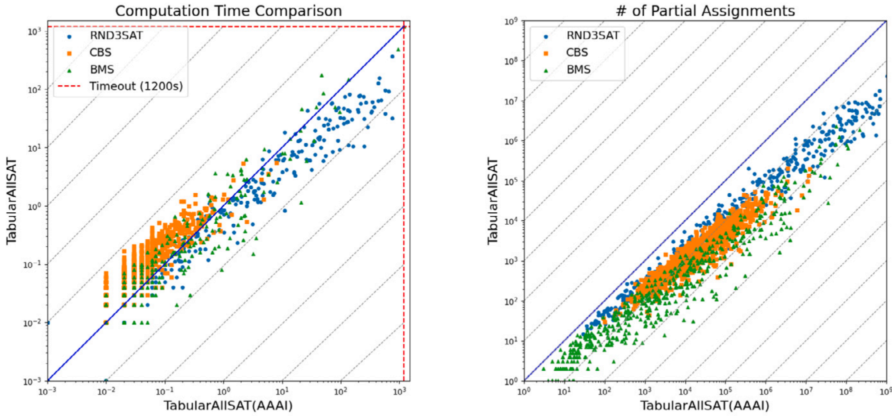
The search algorithm initially generates the first satisfying assignment, $\eta_1 = x_1^d x_2^d x_3^d$. Applying the shrinking process reduces this assignment to $\mu_1 = x_1^d x_2^d$, as x_3 can be removed without affecting the satisfaction of any clause exclusively depending on it.

Following chronological backtracking, we reach the trail $x_1^d \neg x_2^*$, which leads to the second satisfying assignment, $\eta_2 = x_1^d \neg x_2^* x_3$. In this case, x_3 is necessary to satisfy the final clause, while x_2 becomes redundant and can be dropped. The corresponding partial assignment is reduced to $\mu_2 = x_1^d x_3^d$. Notably, due to the reordering of literals, x_3 is now a decision literal, as it is no longer forced by x_1 and $\neg x_2$. Upon flipping x_3, the algorithm encounters the trail $x_1^d \neg x_3^*$, which extends to the satisfying trail $\eta_3 = x_1^d \neg x_3^* x_2$. However, upon reduction, we again obtain μ_1.

At this point, the algorithm cycles between μ_1 and μ_2, without any escape mechanism. The absence of blocking clauses prevents x_1 from being flipped, resulting in a recurring pattern of these partial assignments.

5. AllSAT experimental evaluation

We have implemented the ideas discussed in this paper up until now in our tool TABULARALLSAT, whose source code benchmarks are available on Zenodo [39]. An updated version of the source code is available at https://github.com/giuspek/tabularAllSAT.



(a) CPU Time (in seconds) (b) # of partial models

Fig. 1. Scatter plot comparing CPU time and # of partial models with the two implicant shrinking algorithms. The x and y axes are log-scaled.

Experiments are performed on an Intel Xeon Gold 6238R @ 2.20 GHz 28 Core machine with 128 GB of RAM, running Ubuntu Linux 22.04. Timeout has been set to 1200 seconds. The experiments performed are the following:

- Ablation study to compare the chronological implicant shrinking algorithm in §4.1, against the novel one proposed in §4.2 (§5.1);
- AllSAT experimental evaluation (§6.1.2);

5.1. Comparing implicant shrinking algorithms

We start our experimental evaluation by comparing the two chronological implicant shrinking algorithms discussed respectively in §4.1 (from now on referred as TabularAllSAT$_{\text{AAAI24}}$), and §4.2 (from now on referred as TabularAllSAT). We consider the following benchmark, most of them being used in [27]:

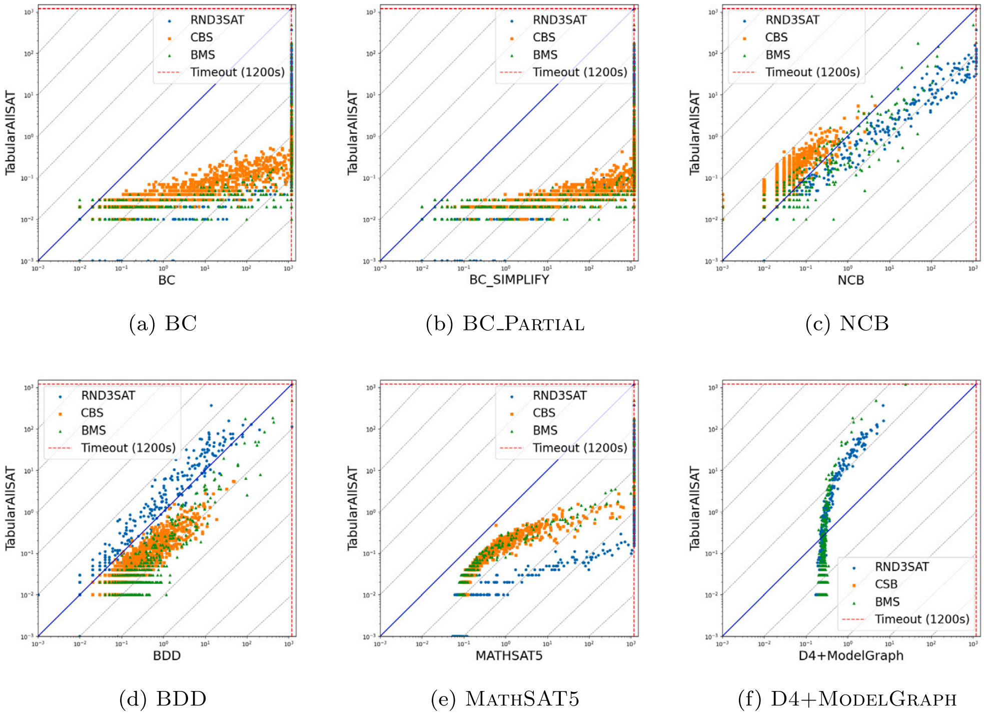
- *Rnd3sat* contains 410 random 3-SAT problems with n variables, $n \in [10, 50]$. In SAT instances, the ratio of clauses to variables needed to achieve maximum hardness is about 4.26, but in AllSAT, it should be set to approximately 1.5 [40]. For this reason, we chose not to use the instances uploaded to SATLIB and we created new random 3-SAT problems accordingly.
- We also tested our algorithms over SATLIB benchmarks, specifically *CBS* and *BMS* [41].

We compared the two implicant shrinking algorithms on two metrics: (i) computational time, and (ii) number of partial assignments retrieved. We checked the correctness of the enumeration by testing if the number of total assignments covered by the set of partial solutions was the same as the model count reported by the #SAT solver Ganak [42], being always correct for both algorithms. Fig. 1 presents a log-scaled scatter plot comparison of two implicant shrinking algorithms, focusing on execution time (left) and the number of partial models generated (right). As expected, the novel implicant shrinking algorithm occasionally incurs a higher overhead due to the additional effort required to shrink assignments, which can result in slightly slower performance compared to the original algorithm with the easiest problems (< 1 s). However, the novel algorithm significantly reduces the number of partial assignments generated, with the impact becoming more pronounced as the number of total assignments for a given instance increases. In these larger instances, the novel implicant shrinking algorithm also demonstrates better performance in terms of execution time. All the following subsections' experiments rely on the novel implicant shrinking algorithm.

5.2. Comparison against state-of-the-art solvers

In these experiments, we considered BC, NCB, and BDD [38], respectively a blocking, a non-blocking, and a BDD-based disjoint AllSAT solver. BC also provides the option to obtain partial assignments (from now on BC_Partial). We also considered MathSAT5 [26], since it provides an interface to compute partial enumeration of propositional problems by exploiting blocking clauses, and the very-recent enumeration approach D4+ModelGraph from [24] that enumerates formulas after transforming them into an equivalent d-DNNF representation. Other AllSAT solvers, such as BASolver [43] and AllSATCC [44], are currently not publicly available, as reported also by other papers [45].

We evaluated the computational performance of TabularAllSAT against several state-of-the-art solvers using the same benchmark set we used in [27]. (See Fig. 3.) The primary objective of this evaluation was to demonstrate that the new chronological implicant shrinking algorithm in TabularAllSAT does not degrade performance in AllSAT problems compared to the previous version. Fig. 2 presents scatter plots comparing TabularAllSAT with other state-of-the-art solvers. The results align with those reported in [27], showing that TabularAllSAT performs competitively or even much better than almost all solvers. The only exception is



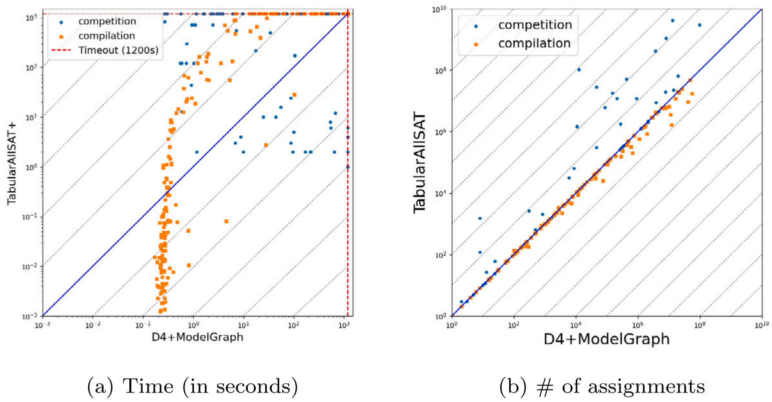
(a) BC (b) BC_PARTIAL (c) NCB

(d) BDD (e) MATHSAT5 (f) D4+MODELGRAPH

Fig. 2. Scatter plots comparing TABULARALLSAT CPU times against other AllSAT solvers. The x and y axes are log-scaled.

	TABULARALLSAT	D4+MODELGRAPH	BDD	NCB	MATHSAT5	BC	BC_PARTIAL
rnd3sat (410)	**410**	**410**	409	396	229	194	210
CSB (1000)	**1000**	**1000**	**1000**	**1000**	997	865	636
BMS (500)	**500**	**500**	498	498	473	368	353
Total (1910)	**1910**	**1910**	1907	1894	1699	1427	1199

Fig. 3. Number of instances solved by each solver within a timeout (1200 seconds) for the AllSAT benchmark.



(a) Time (in seconds) (b) # of assignments

Fig. 4. Scatter plots comparing TABULARALLSAT against D4+MODELGRAPH on AllSAT problems in [24]. The x and y axes are log-scaled.

against the two AllSAT algorithms based on knowledge compilation, respectively BDD and D4+MODELGRAPH. Both approaches perform better than TABULARALLSAT when the problem instances contain few clauses (which is the case of *rnd3sat* problems); in this case, the knowledge compilation procedure is less resource-intensive.

For the sake of completeness, we opted for a more extensive evaluation of TABULARALLSAT and D4+MODELGRAPH by using the benchmarks proposed in [24]. (See Fig. 5.) In this case, with no surprise, TABULARALLSAT is outperformed by the approach based on knowledge compilation, in alignment with results in [24]. We must remark, however, several points. First, the new datasets are based on model counting competition and used for knowledge compilation testing, thus they heavily rely on pre-processing techniques such as partitioning or AND-gate decomposition. In addition to that, and as also stated in [24], TABULARALLSAT has a lower memory

	TabularAllSAT	D4+ModelGraph
compilation(197)	137	**165**
competition(246)	99	**113**
Total(443)	236	**289**

Fig. 5. Number of harder instances solved by TabularAllSAT and D4+ModelGraph within a timeout (1200 seconds) for the AllSAT benchmark.

footprint, never experiencing timeouts during execution, making our tool better suited for situations where memory resources are limited. Finally, it is worth noting that for most of the tested problems, the tools returned the same number of models (see the bisector line in Fig. 4b). In most instances, the model count is equivalent to the number of partial assignments retrieved. This indicates that the structure of these problems prevents implicant shrinking from eliminating even a single atom, thus limiting our algorithm's ability to demonstrate its full potential.

6. From AllSAT to projected AllSAT

To extend Algorithm 1 for projected enumeration, we consider a formula F with two mutually exclusive sets of variables: relevant variables V_r and irrelevant variables V_i. Recall that for a formula $F(V_r, V_i)$, an assignment μ_r projected over V_r satisfies F if μ_r satisfies $\exists V_i.F$. The core search algorithm itself remains unchanged: when we generate a total assignment η during our search loop, we can partition it into η_r and η_i, corresponding to the relevant and irrelevant variables, respectively. Thus, η_r represents the assignment that the algorithm should ultimately produce. As a result, the conflict analysis component of the algorithm does not require modifications. However, it is crucial to note that we are working within the framework of disjoint enumeration. Therefore, the chronological implicant shrinking procedure must be adapted to prevent repetitions while effectively pruning irrelevant variables from the total assignment.

A fundamental adjustment is needed in the variable ordering heuristic, where we prioritize relevant variables over irrelevant ones. This ensures that once the last relevant variable is assigned, any subsequent decision literal from irrelevant variables can be safely ignored. Recalling §3.3, if a non-relevant decision literal ℓ is assigned before the relevant ones, the blocking clause subsumed by chronological backtracking would include ℓ, preventing its removal to guarantee disjointness. By prioritizing relevant variables during the decision phase, we ensure that every partial assignment satisfies F without introducing non-relevant literals as decision literals.

We now discuss how the implicant shrinking algorithm is influenced by projection. All modifications needed for the projected enumeration extensions are highlighted in green. Specifically, when determining which literals to drop from T, we can skip all literals corresponding to variables in V_i, as these irrelevant variables would be dropped regardless. Thus, in Algorithm 7, line 3 we ensure the procedure skips non-relevant literals and lifts them anyway. It is important to note that unit propagation in Algorithm 8, line 8 might force some non-relevant literals back into the trail. However, this does not affect the correctness of the procedure, since no non-relevant variable can be a decision variable this way. Moreover, only literals corresponding to relevant variables are included in the partial model when the model is printed. Algorithm 3, line 5 is updated accordingly, ensuring only variables in V_r are considered.

6.1. Projected AllSAT experimental evaluation

All the additional ideas discussed in §6 to integrate projection in the algorithm have been added in TabularAllSAT. Experiments are performed on an Intel Xeon Gold 6238R @ 2.20 GHz 28 Core machine with 128 GB of RAM, running Ubuntu Linux 22.04. Timeout has been set to 1200 seconds.

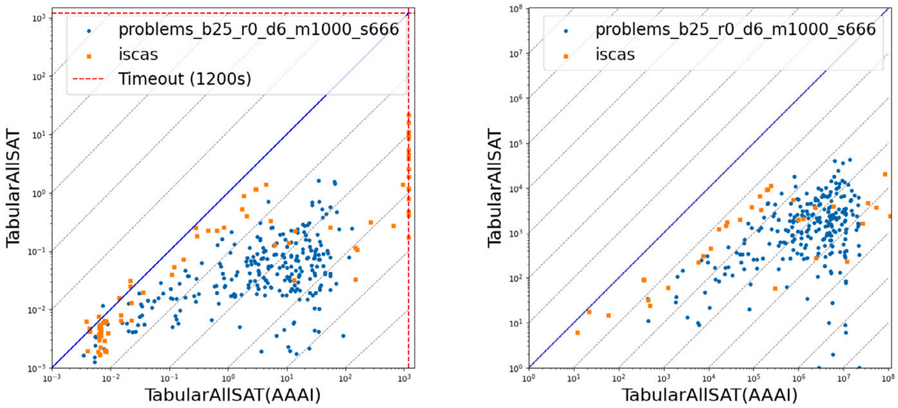
6.1.1. Comparison of implicant shrinking algorithms

We started by comparing the two implicant shrinking algorithms from §4, to ensure that the new method does not introduce negative side effects in projected enumeration, and thus how the more aggressive pruning of total assignments is beneficial for projected enumeration. To evaluate TabularAllSAT on projected AllSAT enumeration, we focused on non-CNF instances that require preprocessing into CNF before conversion to the DIMACS format. This preprocessing step introduces additional CNF-specific variables irrelevant to the final enumeration task. Therefore, we selected benchmarks inspired by [30], specifically: (i) 250 synthetic benchmark instances containing non-CNF formulas with double implications, each with 25 Boolean variables and a formula depth of 6, and (ii) a set of 100 instances from the *iscas* benchmark suite. All of these problems were originally generated as non-CNF formulas. For the CNF transformation, we chose the approach proposed in [30], as it is more suitable for enumeration and the generation of compact partial assignments than the classic Plaisted-Greenbaum transformation.

The results, shown in Fig. 6, indicate that the new implicant shrinking approach positively impacts projected enumeration, yielding improvements in both execution time and the number of partial assignments generated.

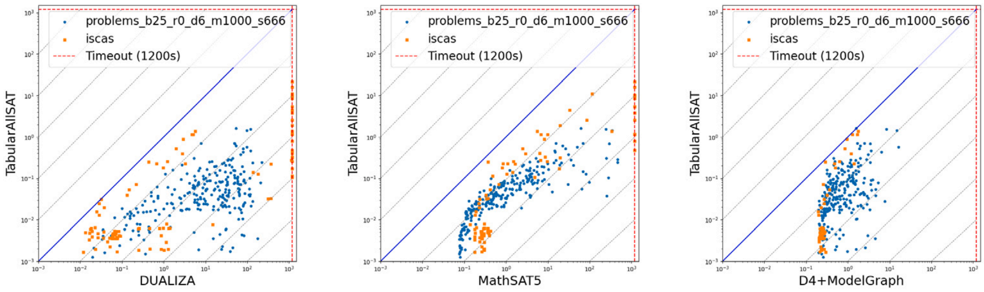
6.1.2. Comparison against state-of-the-art solvers

We compared TabularAllSAT against (i) MathSAT5, (ii) Dualiza, a model counter and AllSAT solver that utilizes dual reasoning [46], and (iii) the D4+ModelGraph tool from [24]. We remark that, despite [24] not directly addressing projected enumeration, it is possible to use D4 to generate projected d-DNNF, on top of which ModelGraph can retrieve partial assignments. The results, depicted in Fig. 7, demonstrate that TabularAllSAT significantly outperforms both Dualiza and MathSAT5, generating partial assignments much faster. (See Fig. 8.)



(a) Time (in seconds) (b) # of partial assignments

Fig. 6. Scatter plot comparing CPU time and number of partial assignments generated of TabularAllSAT against the implicant shrinking algorithm of $\text{TabularAllSAT}_{AAAI24}$ on projected AllSAT problems. The x and y axes are log-scaled.



(a) Dualiza (b) MathSAT5 (c) D4+ModelGraph

Fig. 7. Scatter plot comparing CPU time of TabularAllSAT against Dualiza, MathSAT5, and D4+ModelGraph on projected AllSAT problems. The x and y axes are log-scaled.

	TabularAllSAT	D4+ModelGraph	Dualiza	MathSAT5
iscas (100)	**100**	**100**	76	78
b25_r0_d6 (250)	**250**	**250**	**250**	174
Total (350)	**350**	**350**	326	252

Fig. 8. Table reporting the number of instances solved by each solver within the timeout time (1200 seconds) for projected AllSAT benchmark.

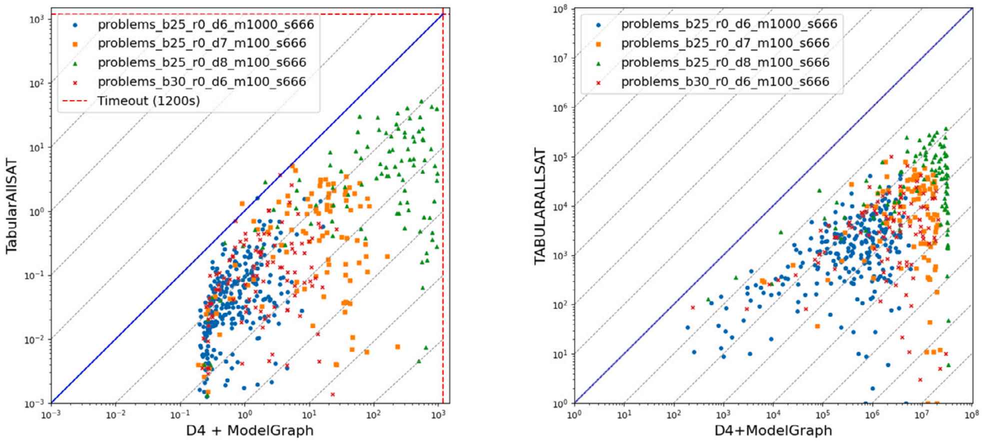
We notice that the performance gap against D4+ModelGraph is not as dramatic as with respect to the other two solvers, so for the sake of completeness, we tested both tools using more complex synthetic benchmarks, where MathSAT5 and Dualiza reached timeout for almost every file. We generated 2 additional benchmarks with 25 Boolean variables and formula depth 7 and 8, respectively. We also provided a set of benchmarks with depth 6 and 30 variables, to check if adding more important variables does impact enumeration. The results, shown in Fig. 9, now clearly show the superiority of TabularAllSAT against the knowledge compilation approach, both considering computation times and number of partial assignments retrieved.

7. From AllSAT to AllSMT

To extend the algorithm to address first-order logic theories, the search algorithm should integrate a theory solver and call it to check if the current assignment that satisfies the Boolean abstraction of a formula F is also theory consistent. These additional checks affect the definition of some of the algorithms of $\text{TabularAllSAT}_{AAAI24}$, and all changes to the original $\text{TabularAllSAT}_{AAAI24}$ algorithms are colored in **blue**.

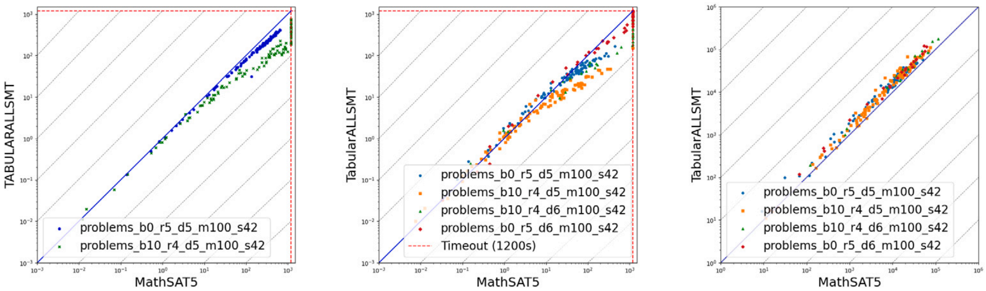
First, in Algorithm 1 once a total trail has been generated, we must verify if there are theory inconsistencies. We perform a T-consistency check (Algorithm 1, line 8) and, if a T-conflict is generated, then we must analyze the conflict and backjump accordingly (Algorithm 1, lines 9-11).

Second, once a literal has been decided and UnitPropagation is executed, there could be some other T-atoms that are implied by the newly added literals in T or there could be a T-conflict. For this reason, UnitPropagation now is a two-step procedure: (i)



(a) Time (b) # of partial assignments

Fig. 9. Scatter plot comparing CPU time and number of partial assignments of TABULARALLSAT against D4+MODELGRAPH on harder projected AllSAT problems. Notice that in this batch of experiments no timeout happens. The x and y axes are log-scaled.



(a) Time - Total enumeration (b) Time - Partial enumeration (c) # of partial assignments

Fig. 10. Scatter plot comparing CPU time for total enumeration (a), CPU time for partial enumeration (b), and the number of partial assignments for partial enumeration (c) generated by TABULARALLSMT against MATHSAT5 on AllSMT problems. The x and y axes are log-scaled.

propositional unit propagation, to satisfy the Boolean abstraction of F, and (ii) $\mathcal{T}$-propagation, which also works as an early pruning algorithm in SMT solving. If either unit propagation call generates a conflict, ANALYZECONFLICT is executed. The $\mathcal{T}$-conflict analysis does not differ from the Boolean conflict analysis algorithm, so additional changes are not required in Algorithm 5. We must remark, however, that theory solvers can add new $\mathcal{T}$-atoms during execution, e.g., in $\mathcal{LIA}$, the branch-and-bound algorithm could generate new inequalities atoms. If this happens, then the trail maximum size increases, and all new $\mathcal{T}$-atoms are flagged as non-relevant variables.

7.1. AllSMT experimental evaluation

We implemented all the ideas discussed in the paper into TABULARALLSMT, whose executable file and all benchmarks are available on Zenodo [47]. An updated version of the source code is available at https://github.com/giuspek/tabularAllSMT.git. TABULARALLSMT integrates MATHSAT5 as the theory solver, which is under a proprietary license, thus TABULARALLSMT code is not publicly available, but the executable file is provided. Experiments are performed on an Intel Xeon Gold 6238R @ 2.20 GHz 28 Core machine with 128 GB of RAM, running Ubuntu Linux 22.04. Timeout has been set to 1200 seconds.

For the final set of experiments, we used benchmarks inspired by [48], generating several synthetic benchmarks with varying numbers of Boolean variables (b), real variables (r), and formula depth (d). We compared TABULARALLSMT against MATHSAT5, which is currently the only publicly available projected AllSMT solver.

We began by evaluating the effect of blocking clauses during total SMT enumeration. This experiment was designed to demonstrate how the introduction of blocking clauses as in MATHSAT5 negatively impacts performance compared to our algorithm, particularly when dealing with first-order logic theories. We performed total enumeration on two smaller benchmark sets, and the results, presented in the scatter plot in Fig. 10a, clearly illustrate that several instances are successfully solved by TABULARALLSMT, whereas MATHSAT5 reaches the timeout limit. Since the theory reasoning is shared among the two tools, it is evident that the lack of blocking clauses is the primary factor contributing to these timeouts.

	TabularAllSMT	MathSAT5
b0_r5_d5 (100)	**100**	**100**
b10_r4_d5 (100)	**100**	99
b0_r5_d6 (100)	**48**	18
b25_r0_d6 (100)	**49**	26
Total (400)	**287**	243

Fig. 11. Table reporting the number of instances solved by each solver within the timeout time (1200 seconds) for AllSMT benchmark.

We continue by comparing results on disjoint partial enumeration, including the same benchmarks used for Fig. 10a plus two other datasets with higher depth. The results, shown in Fig. 10b and 10c, indicate that while MathSAT5 is slightly more effective at shrinking assignments into shorter partial models, TabularAllSMT outperforms it over the long term considering CPU times, especially as the complexity of instances increases. This difference is primarily due to the exponential number of blocking clauses that MathSAT5 adds, which eventually hampers its performance. Additionally, Fig. 11 presents the number of problems solved within the timeout limits, further emphasizing that TabularAllSMT successfully solves a significant number of problems that MathSAT5 cannot handle. It is important to highlight that early pruning, implemented in both MathSAT5 and TabularAllSMT, can significantly reduce the Boolean search space and, as a result, the number of calls to the T-solver. However, while early pruning can be beneficial, it may also lead to unnecessary calls to the T-solver. In the context of enumeration, this overhead can negatively affect performance, as the extra solver calls introduce additional computational cost, explaining why results are less impactful than those shown in §6.

We remark that the experiments in this section focus on linear real arithmetic; TabularAllSMT, however, is compatible with all theories accepted by MathSAT5.

8. Conclusion

In this work, we introduced TabularAllSAT and TabularAllSMT, two new solvers designed for efficient projected enumeration in AllSAT and AllSMT, respectively. By combining CDCL and chronological backtracking, we addressed the inherent inefficiencies of traditional blocking solvers, avoiding the performance degradation caused by excessive blocking clauses. Our novel aggressive implicant shrinking algorithm further reduced the number of partial assignments with respect to its predecessor, ensuring a more compact representation of the solution space. We extended our solver framework to support projected enumeration and SMT formulas, integrating theory reasoning into the search process. Extensive experimental results showed that our solvers outperform existing state-of-the-art techniques, offering a significant advantage in both propositional and SMT-based problems.

CRediT authorship contribution statement

Giuseppe Spallitta: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Roberto Sebastiani:** Writing – original draft, Validation, Supervision, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Armin Biere:** Writing – original draft, Visualization, Validation, Supervision, Software, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

We acknowledge the support of the MUR PNRR project FAIR – Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU. The work was partially supported by the project "AI@TN" funded by the Autonomous Province of Trento. This research was partially supported by TAILOR, a project funded by the EU Horizon 2020 research and innovation program under GA No. 952215.

Data availability

All experimental data of the paper, including the source code of both TabularAllSAT and TabularAllSMT, are publicly available on GitHub and Zenodo. Links are provided in the experimental evaluation.

References

[1] S. Khurshid, D. Marinov, I. Shlyakhter, D. Jackson, A case for efficient solution enumeration, in: Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5–8, 2003, Springer, 2004, pp. 272–286, Selected Revised Papers 6.

[2] H. Jin, H. Han, F. Somenzi, Efficient conflict analysis for finding all satisfying assignments of a Boolean circuit, in: Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, Proceedings 11, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Springer, 2005, pp. 287–300.

[3] I.O. Dlala, S. Jabbour, L. Sais, B.B. Yaghlane, A comparative study of SAT-based itemsets mining, in: Research and Development in Intelligent Systems XXXIII: Incorporating Applications and Innovations in Intelligent Systems XXIV 33, Springer, 2016, pp. 37–52.
[4] S.K. Lahiri, R. Nieuwenhuis, A. Oliveras, SMT techniques for fast predicate abstraction, in: Computer Aided Verification: 18th International Conference, Proceedings 18, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Springer, 2006, pp. 424–437.
[5] Q.-S. Phan, P. Malacaria, All-solution satisfiability modulo theories: applications, algorithms and benchmarks, in: 2015 10th International Conference on Availability, Reliability and Security, IEEE, 2015, pp. 100–109.
[6] G. Spallitta, G. Masina, P. Morettin, A. Passerini, R. Sebastiani, SMT-based weighted model integration with structure awareness, in: Uncertainty in Artificial Intelligence, PMLR, 2022, pp. 1876–1885.
[7] G. Spallitta, G. Masina, P. Morettin, A. Passerini, R. Sebastiani, Enhancing SMT-based weighted model integration by structure awareness, Artif. Intell. 328 (2024) 104067.
[8] D. Chistikov, R. Dimitrova, R. Majumdar, Approximate counting in SMT and value estimation for probabilistic programs, in: Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, Proceedings 21, ETAPS 2015, London, UK, April 11-18, 2015, Springer, 2015, pp. 320–334.
[9] D. Clark, S. Hunt, P. Malacaria, A static analysis for quantifying information flow in a simple imperative language, J. Comput. Secur. 15 (3) (2007) 321–371.
[10] M. Michelutti, G. Masina, G. Spallitta, R. Sebastiani, Canonical Decision Diagrams Modulo Theories, IOS Press - ECAI 2024, 2024.
[11] A. Gupta, Z. Yang, P. Ashar, A. Gupta, SAT-based image computation with application in reachability analysis, in: International Conference on Formal Methods in Computer-Aided Design, Springer, 2000, pp. 391–408.
[12] O. Grumberg, A. Schuster, A. Yadgar, Memory efficient all-solutions SAT solver and its application for reachability analysis, in: Formal Methods in Computer-Aided Design: 5th International Conference, Proceedings 5, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Springer, 2004, pp. 275–289.
[13] J. Brauer, A. King, J. Kriener, Existential quantification as incremental SAT, in: Computer Aided Verification: 23rd International Conference, Proceedings 23, CAV 2011, Snowbird, UT, USA, July 14-20, 2011, Springer, 2011, pp. 191–207.
[14] O. Shtrichman, Pruning techniques for the SAT-based bounded model checking problem, in: Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer, 2001, pp. 58–70.
[15] K.L. McMillan, Applying SAT methods in unbounded symbolic model checking, in: Computer Aided Verification: 14th International Conference, Proceedings 14, CAV 2002 Copenhagen, Denmark, July 27–31, 2002, Springer, 2002, pp. 250–264.
[16] Y. Yu, P. Subramanyan, N. Tsiskaridze, S. Malik, All-SAT using minimal blocking clauses, in: 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, IEEE, 2014, pp. 86–91.
[17] B. Li, M.S. Hsiao, S. Sheng, A novel SAT all-solutions solver for efficient preimage computation, in: Proceedings Design, Automation and Test in Europe Conference and Exhibition, vol. 1, IEEE, 2004, pp. 272–277.
[18] A. Nadel, V. Ryvchin, Chronological backtracking, in: Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, Proceedings 21, FloC 2018, Oxford, UK, July 9–12, 2018, Springer, 2018, pp. 111–121.
[19] S. Möhle, A. Biere, Combining conflict-driven clause learning and chronological backtracking for propositional model counting, in: GCAI, 2019, pp. 113–126.
[20] R. Sebastiani, Are you satisfied by this partial assignment?, arXiv preprint, arXiv:2003.04225, 2020.
[21] D. Fried, A. Nadel, R. Sebastiani, Y. Shalmon, Entailing generalization boosts enumeration, in: 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.
[22] S. Möhle, R. Sebastiani, A. Biere, Four flavors of entailment, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2020, pp. 62–71.
[23] S. Möhle, R. Sebastiani, A. Biere, On enumerating short projected models, Discrete Appl. Math. 361 (2025) 412–439, https://doi.org/10.1016/j.dam.2024.10.021.
[24] J.-M. Lagniez, E. Lonca, Leveraging decision-DNNF compilation for enumerating disjoint partial models, in: 21st International Conference on Principles of Knowledge Representation and Reasoning (KR 2024), 2024.
[25] S. Möhle, An abstract CNF-to-d-DNNF compiler based on chronological CDCL, in: International Symposium on Frontiers of Combining Systems, Springer, 2023, pp. 195–213.
[26] A. Cimatti, A. Griggio, B.J. Schaafsma, R. Sebastiani, The MathSAT5 SMT solver, in: Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, Proceedings 19, ETAPS 2013, Rome, Italy, March 16-24, 2013, Springer, 2013, pp. 93–107.
[27] G. Spallitta, R. Sebastiani, A. Biere, Disjoint partial enumeration without blocking clauses, in: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 38, 2024, pp. 8126–8135.
[28] G.S. Tseitin, On the complexity of derivation in propositional calculus, in: Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970, Symbolic Computation, Springer, 1983, pp. 466–483.
[29] D.A. Plaisted, S. Greenbaum, A structure-preserving clause form translation, J. Symb. Comput. 2 (3) (1986) 293–304, https://doi.org/10.1016/S0747-7171(86)80028-1.
[30] G. Masina, G. Spallitta, R. Sebastiani, On CNF conversion for disjoint SAT enumeration, in: 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.
[31] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proceedings of the 38th Annual Design Automation Conference, 2001, pp. 530–535.
[32] J.P. Marques-Silva, K.A. Sakallah, GRASP: a search algorithm for propositional satisfiability, IEEE Trans. Comput. 48 (5) (1999) 506–521.
[33] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Commun. ACM 5 (7) (1962) 394–397.
[34] R.J. Bayardo Jr., J.D. Pehoushek, Counting models using connected components, in: AAAI/IAAI, 2000, pp. 157–162.
[35] A. Morgado, J. Marques-Silva, Good learning and implicit model enumeration, in: 17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'05), IEEE, 2005, p. 6.
[36] S. Möhle, A. Biere, Backing backtracking, in: Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, Proceedings 22, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Springer, 2019, pp. 250–266.
[37] J. Huang, A. Darwiche, Using DPLL for efficient OBDD construction, in: Theory and Applications of Satisfiability Testing: 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Springer, 2005, pp. 157–172, Revised Selected Papers 7.
[38] T. Toda, T. Soh, Implementing efficient all solutions SAT solvers, ACM J. Exp. Algorithmics 21 (2016) 1–44.
[39] G. Spallitta, A. Biere, R. Sebastiani, Disjoint projected enumeration for SAT and SMT without blocking clauses: TabularAllSAT source code, https://doi.org/10.5281/zenodo.14197776, Nov. 2024.
[40] R.J. Bayardo Jr., R. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: Aaai/iaai, Citeseer, 1997, pp. 203–208.
[41] J. Singer, I.P. Gent, A. Smaill, Backbone fragility and the local search cost peak, J. Artif. Intell. Res. 12 (2000) 235–270.
[42] S. Sharma, S. Roy, M. Soos, K.S. Meel, GANAK: a scalable probabilistic exact model counter, in: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), 2019.
[43] Y. Zhang, G. Pu, J. Sun, Accelerating All-SAT computation with short blocking clauses, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 6–17.
[44] J. Liang, F. Ma, J. Zhou, M. Yin, AllSATCC: boosting AllSAT solving with efficient component analysis, in: IJCAI, 2022, pp. 1866–1872.

[45] D. Fried, A. Nadel, Y. Shalmon, AllSAT for combinational circuits, in: 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[46] S. Möhle, A. Biere, Dualizing projected model counting, in: 2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI), IEEE, 2018, pp. 702–709.

[47] G. Spallitta, A. Biere, R. Sebastiani, Disjoint projected enumeration for SAT and SMT without blocking clauses: TabularAllSMT, https://doi.org/10.5281/zenodo.14198207, Nov. 2024.

[48] G. Masina, G. Spallitta, R. Sebastiani, On CNF conversion for SAT and SMT enumeration, arXiv preprint, arXiv:2303.14971, 2023.