Dynamic Blocked Clause Elimination for Projected Model Counting

Jean-Marie Lagniez
Univ. Artois, CNRS, CRIL, France

Pierre Marquis
Univ. Artois, CNRS, CRIL, IUF, France

Armin Biere
University Freiburg, Germany

Abstract

In this paper, we explore the application of blocked clause elimination for projected model counting. This is the problem of determining the number of models $\|\exists X.\Sigma\|$ of a propositional formula Σ after eliminating a given set X of variables existentially. Although blocked clause elimination is a well-known technique for SAT solving, its direct application to model counting is challenging as in general it changes the number of models. However, we demonstrate, by focusing on projected variables during the blocked clause search, that blocked clause elimination can be leveraged while preserving the correct model count. To take advantage of blocked clause elimination in an efficient way during model counting, a novel data structure and associated algorithms are introduced. Our proposed approach is implemented in the model counter `d4`. Our experiments demonstrate the computational benefits of our new method of blocked clause elimination for projected model counting.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Projected model counting, blocked clause elimination, propositional logic

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.21

Supplementary Material *Software*: `https://zenodo.org/records/11854002`

Funding This work has benefited from the support of the AI Chair EXPEKCTATION (ANR-19-CHIA-0005-01) of the French National Research Agency.

1 Introduction

Propositional model counting consists determines the number of models of a propositional formula Σ, typically represented in conjunctive normal form (`CNF`). Many applications however require a projected variant focusing on a specific set X of variables of interest: given a propositional formula Σ and a set X of propositional variables to be forgotten, the projected model counting problem consists in computing the number of interpretations over the variables occurring in Σ but not in X, which coincide on X with a model of Σ. In other words, the goal is to count the number of models of the quantified Boolean formula $\exists X.\Sigma$ over its variables (i.e., those present in Σ but absent in X).

The projected model counting problem is significant for various application domains of artificial intelligence (AI). For instance, in planning scenarios, it helps to evaluate the robustness of a plan by determining the number of initial states from which the plan execution leads to a goal state [4]. Additionally, its applicability extends beyond AI to formal verification problems [19] and database operations [1]. As a generalization of the standard model counting problem `#SAT`, for the special case $X = \emptyset$, the projected model counting problem is at least as complex as `#SAT` (`#P-hard`). However, the possibility to eliminate some variables actually also might simplify the problem, such as when all variables in Σ belong to X, reducing the problem to simply determining the satisfiability of Σ. However, in practice projected model

27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024).
Editors: Supratik Chakraborty and Jie-Hong Roland Jiang; Article No. 21; pp. 21:1–21:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

counting often turns out to be more challenging than the standard model counting problem. This can be explained by the additional constraints imposed on the branching heuristic, i.e., in which order variables can be used as decisions, making the problem inherently harder. This is also reflected for instance in upper known bounds in the literature [9] on the time complexity of model counting even formulas with fixed treewidth k, i.e., $\mathcal{O}(2^k)$ for standard and $\mathcal{O}(2^{2^k})$ projected model counting.

One way to speed up model counting is to employ preprocessing which simplifies the formula before tackling the model counting task. Preprocessing methods have shown to be effective across various automated reasoning tasks, notably in SAT solving and QBF solving [6]. Among these preprocessing techniques, blocked-clause elimination (BCE) [15] significantly improves solver performance by emulating several other, more complex preprocessing techniques [16]. Blocked clauses, initially introduced by Kullmann [20] as a generalization of extended resolution clauses, are pivotal in propositional preprocessing techniques. In essence, a clause α is deemed blocked within a CNF formula Σ if it includes a literal ℓ for which all conceivable resolvents of α over ℓ yield tautologies. Removal of blocked clauses can significantly enhance the performance of SAT solvers [15]. Furthermore, generalized forms of BCE have demonstrated remarkable performance improvements in solving problems beyond NP, such as QBF [12], DQBF [31] and even first-order theorem proving [18].

However, while several preprocessing techniques used for SAT solving can be adapted to improve model counting [25, 22], others, such as the blocked clause elimination technique, are unsuitable due to their inability to preserve the number of models. In this paper, we address this challenge by delineating conditions under which the use of BCE is correct in projected model counting. Specifically, we demonstrate that focusing on projected variables during the blocked clause search is correct, i.e., gives the same projected count. The rationale behind this lies in the fact that when concentrating on sub-formulas containing only projected variables, the requirement boils down to ensure satisfiability. Consequently, clauses blocked on projected variables can safely be removed.

When used for model counting, simplification techniques are typically applied up-front during preprocessing and even though modern SAT solvers make heavily use of interleaving formula simplification with CDCL search, also called *inprocesssing* [17], this form of simplification is currently performed only at the root-level (decision level zero). In this paper we go beyond root-level simplification and propose to dynamically apply the blocked clause elimination technique dynamically during search at every decision level in the form of *dynamic blocked clause elimination*. In this sense our approach is similar to look-ahead solving [14], which use simplification techniques during search, i.e., probing techniques, at every decision level.

To accomplish this, we introduce novel data structures and associated algorithms tailored for dynamic inprocessing. Our method efficiently identifies clauses eligible for elimination by employing a mechanism akin to watched literals. Importantly, this methodology is not tied to a specific model counter; it seamlessly integrates into any state-of-the-art model counter.

To assess the efficiency of our approach, we conducted experiments using the model counter d4 [24], modified to integrate our newly developed data structures and algorithms for projected model counting. We evaluated the performance of this new version of d4 across various benchmarks from recent model counting competitions (available at https://mccompetition.org/). Our experimental results underscore the computational advantages of employing blocked clause elimination for projected model counting. For certain benchmarks, the adoption of BCE dynamic inprocessing led to a substantial reduction in computation time, with time savings of up to one order of magnitude compared to the baseline version of d4. To ensure that the improvements are indeed attributable to the use of BCE inprocessing, we also examined a version of d4 that implements BCE dynamically during preprocessing only.

Interestingly, our findings indicate that employing BCE in preprocessing had no discernible impact on the effectiveness of the model counter d4, underscoring the possibility to take advantage of BCE eagerly during the model counting process.

The remainder of the paper is structured as follows. The next section provides formal preliminaries. Following this, we delve into theoretical insights and give implementation details on how to perform BCE dynamically during search. Then, we outline the experimental protocol adopted for our empirical evaluations, along with the corresponding results. Finally, we conclude the paper, offering insights into potential avenues for future research. The source code and benchmarks utilized in our experiments are provided as supplementary materials.

2 Preliminaries

Let $\mathcal{L}$ be a propositional language built up from a finite set of propositional variables $\mathcal{P}$ and the standard logical connectives. The symbols $\bot$ and $\top$ represent the Boolean constants for falsehood and truth, respectively. A *literal* ℓ is either a propositional variable (e.g., x) or its negation ($\neg x$). For a literal ℓ defined over variable x, its *complementary literal* $\overline{\ell}$ is defined as $\overline{\ell} = \neg x$ if $\ell = x$, and $\overline{\ell} = x$ if $\ell = \neg x$, with $\mathit{Var}(\ell) = x$ denoting the variable of ℓ. A *term* is a conjunction of literals. A *clause* is a disjunction of literals. Terms and clauses are also interpreted as their sets of literals whenever convenient.

A clause is a *tautology* if it contains $\top$, or both x and $\neg x$ for some variable x. A CNF formula Σ is a conjunction of clauses, also viewed as set of clauses when needed. The set of propositional variables occurring in Σ is denoted $\mathit{Var}(\Sigma)$. If a variable $x \in X$ does not belong to $\mathit{Var}(\Sigma)$, then x is said to be *free* in Σ. Each clause is associated with a unique identifier represented as an integer. A clause α_i of a CNF formula Σ can be accessed using its identifier through square bracket notation, denoted as $\Sigma[i]$. Thus α_i is also noted $\Sigma[i]$. We denote by $S_\ell(\Sigma)$ the set of clauses of Σ that contain literal ℓ. When no ambiguity about Σ is possible, we simply use the shorthand notation S_ℓ instead of explicitly writing $S_\ell(\Sigma)$.

▶ **Example 1.** Consider the CNF formula $\Sigma = \{\alpha_1, \alpha_2, \ldots, \alpha_{11}\}$ with

$1 : x_1 \vee x_2$	$2 : \neg x_2 \vee x_3$	$3 : \neg x_1 \vee \neg x_2 \vee \neg y_1$	$4 : x_1 \vee \neg x_3 \vee y_1$
$5 : x_2 \vee \neg x_3 \vee y_2$	$6 : x_1 \vee \neg x_3 \vee \neg y_2$	$7 : y_3 \vee x_2$	$8 : \neg y_3 \vee \neg x_2 \vee \neg x_3$
$9 : \neg y_3 \vee x_1$	$10 : \neg y_3 \vee \neg y_2 \vee x_3$	$11 : y_3 \vee y_2 \vee x_2$	

$\mathit{Var}(\Sigma) = \{x_1, x_2, x_3, y_1, y_2, y_3\}$, $S_{x_1}(\Sigma) = \{\alpha_1, \alpha_4, \alpha_6, \alpha_9\}$, and $\Sigma[2] = \alpha_2 = \neg x_2 \vee x_3$.

An *interpretation* (or world) over $\mathcal{P}$, denoted by ω, is a mapping from $\mathcal{P}$ to $\{0, 1\}$. Interpretations ω are often represented by sets of literals (one per variable in $\mathcal{P}$), of exactly those literals set to 1 by ω. The collection of all interpretations is denoted by $\mathcal{W}$. An interpretation ω is a *model* of a formula $\Sigma \in \mathcal{L}$ if and only if it satisfies the formula in accordance with the usual truth-functional interpretation. The set of models of the formula Σ is denoted by $\mathrm{mod}(\Sigma)$, defined as $\{\omega \in \mathcal{W} \mid \omega \models \Sigma\}$. The symbol $\models$ denotes logical entailment, while $\equiv$ denotes logical equivalence. For any formulas $\Sigma, \Psi \in \mathcal{L}$, we have $\Sigma \models \Psi$ if and only if $\mathrm{mod}(\Sigma) \subseteq \mathrm{mod}(\Psi)$ and $\Sigma \equiv \Psi$ if and only if $\mathrm{mod}(\Sigma) = \mathrm{mod}(\Psi)$. The notation $\|\Sigma\|$ indicates the number of models of Σ over $\mathit{Var}(\Sigma)$.

▶ **Example 2** (Example 1 cont'd). $\|\Sigma\| = 9$ and the models of Σ are:

$\{\neg x_1, x_2, x_3, y_1, \neg y_2, \neg y_3\}$	$\{x_1, \neg x_2, x_3, y_1, y_2, y_3\}$	$\{x_1, \neg x_2, x_3, \neg y_1, y_2, y_3\}$
$\{x_1, \neg x_2, \neg x_3, \neg y_1, y_2, y_3\}$	$\{x_1, \neg x_2, \neg x_3, y_1, \neg y_2, y_3\}$	$\{x_1, \neg x_2, \neg x_3, y_1, y_2, y_3\}$
$\{x_1, \neg x_2, \neg x_3, \neg y_1, \neg y_2, y_3\}$	$\{x_1, x_2, x_3, \neg y_1, y_2, \neg y_3\}$	$\{x_1, x_2, x_3, \neg y_1, \neg y_2, \neg y_3\}$

For a formula $\Sigma \in \mathcal{L}$ and a subset $X \subseteq \mathcal{P}$, $\exists X.\Sigma$ represents, up to logical equivalence, the most general consequence of Σ that is independent of the variables in X (see for instance [28] for details). We note $Var(\exists X.\Sigma) = Var(\Sigma) \setminus X$.

▶ **Example 3** (Example 1 cont'd). Let $X = \{y_1, y_2, y_3\}$. We have $\|\exists X.\Sigma\| = 4$ and the models of $\exists X.\Sigma$ over $Var(\exists X.\Sigma)$ are $\{\{\neg x_1, x_2, x_3\}, \{x_1, \neg x_2, x_3\}, \{x_1, \neg x_2, \neg x_3\}, \{x_1, x_2, x_3\}\}$.

The *conditioning* of a CNF formula Σ by a consistent term γ results in the formula denoted by $\Sigma_{|\gamma}$, where $\Sigma_{|\gamma}$ is obtained from Σ by removing each clause from containing a literal of γ and simplifying the remaining clauses, by removing from them complementary literals to those in γ. If, during the simplification, a clause becomes empty, then $\Sigma_{|\gamma}$ is unsatisfiable.

The conditioning of Σ on ℓ is equivalent to the formula $\exists\, Var(\ell).(\Sigma \wedge \ell)$. When ℓ is a unit clause of Σ, $\Sigma_{|\ell}$ is satisfiable if and only if Σ is satisfiable. *Boolean Constraint Propagation* (BCP) [29] is the algorithm that, given a CNF formula Σ, returns a CNF formula closed under unit propagation, i.e., that does not contain any unit clauses. The resulting formula is obtained by repeating the unit propagation of a unit clause of Σ in the formula Σ while such a unit clause exists. The identifiers assigned to clauses in Σ remain unaltered by BCP. Consequently, $\texttt{BCP}(\Sigma)[i]$ will retrieve the clause α_i resulting from the application of BCP on Σ, which could be $\bot$, $\top$, or a subset of α_i.

▶ **Example 4** (Example 1 cont'd). The formula $\texttt{BCP}(\Sigma_{|\neg x_1}) = (x_2) \wedge (x_3) \wedge (y_1) \wedge (\neg y_2) \wedge (\neg y_3)$ is the result of conditioning Σ with the literal $\neg x_1$ and applying BCP to $\Sigma_{|\neg x_1}$. $\texttt{BCP}(\Sigma_{|\neg x_1})[1] = \top$ and $\texttt{BCP}(\Sigma_{|\neg x_1})[4] = y_1$.

The resolution rule asserts that given two clauses $\alpha_1 = \{\ell, a_1, \dots, a_n\}$ and $\alpha_2 = \{\overline{\ell}, b_1, \dots, b_m\}$, the resulting clause $\alpha = \{a_1, \dots, a_n, b_1, \dots, b_m\}$, is the *resolvent* of α_1 and α_2 on the literal ℓ. This operation is denoted as $\alpha = \alpha_1 \oplus \alpha_2$. This concept extends naturally to sets of clauses: for two sets S_ℓ and $S_{\overline{\ell}}$ containing clauses that all involve ℓ and $\overline{\ell}$, respectively, we define $S_\ell \oplus S_{\overline{\ell}} = \{\alpha_1 \oplus \alpha_2 | \alpha_1 \in S_\ell, \alpha_2 \in S_{\overline{\ell}},$ and $\alpha_1 \oplus \alpha_2$ is not a tautology$\}$.

▶ **Example 5** (Example 1 cont'd). Let $S_{\neg x_1} = \{(\neg x_1 \vee \neg x_2 \vee \neg y_1)\}$, $S_{y_3} = \{(y_3 \vee x_2), (y_3 \vee y_2 \vee x_2)\}$ and $S_{\neg y_3} = \{(\neg y_3 \vee \neg x_2 \vee \neg x_3), (\neg y_3 \vee x_1), (\neg y_3 \vee \neg y_2 \vee x_3)\}$. We have $S_{y_3} \oplus S_{\neg y_3} = \{(x_1 \vee x_2), (\neg y_2 \vee x_3 \vee x_2), (y_2 \vee x_2 \vee x_1)\}$ and $\{(x_1 \vee x_2)\} \oplus S_{x_1} = \emptyset$.

The simplification technique known as *Blocked Clause Elimination* (BCE) [15, 12], targets the removal of specific clauses termed *blocked clauses* from CNF formulas [20]. In the context of a CNF formula Σ, a literal ℓ within a clause α is termed a *blocking literal* if it blocks α with respect to Σ. This occurs when, for every clause α' in Σ containing $\overline{\ell}$, the resulting resolvent $\alpha \oplus \alpha'$ on ℓ is a tautology. In essence, for a given CNF and its clauses, a clause is considered blocked if it contains a literal that can effectively block it. Applying BCE to Σ leads to remove every clause containing a blocking literal and by repeating the process iteratively until no blocked literal exists. [15, 12] illustrates that the outcome of BCE remains satisfiable equivalent regardless of the sequence in which blocked clauses are eliminated. More generally, blocked clause elimination converges to a unique fixed point for any CNF formula, establishing the confluence of the method.

▶ **Example 6** (Example 1 cont'd). Above we have shown that the clause $(x_1 \vee x_2)$ is blocked by x_1 and therefore can be eliminated. Following this, both $(\neg x_1 \vee \neg x_2 \vee \neg y_1)$ and $(x_1 \vee \neg x_3 \vee y_1)$ can be removed interchangeably, as they are respectively blocked by y_1 and $\neg y_1$. Subsequently, $(\neg y_3 \vee \neg y_2 \vee x_3)$, blocked by $\neg y_2$, is eliminated, along with the two clauses, $(\neg y_3 \vee x_1)$ and $(x_1 \vee \neg x_3 \vee \neg y_2)$, both blocked by x_1. Next, $(y_3 \vee x_2)$ is removed as it is blocked by y_3. Following this, both $(x_2 \vee \neg x_3 \vee y_2)$ and $(y_3 \vee y_2 \vee x_2)$, blocked by y_2, can be eliminated. Then, $(\neg y_3 \vee \neg x_2 \vee \neg x_3)$ is removed because it is blocked by y_3, and finally, the last clause $(\neg x_2 \vee x_3)$ is removed because it is blocked by both $\neg x_2$ and x_3. Thus, $\texttt{BCE}(\Sigma) = \emptyset$.

As highlighted in [20], the removal of any blocked clause ensures the preservation of unsatisfiability. However, as illustrated by the previous example, utilizing blocked clause elimination (BCE) on a CNF formula Σ does not ensure that the resulting formula $\mathtt{BCE}(\Sigma)$ has the same number of models as Σ. In the next section, we will delve into the specific conditions under which BCE can be effectively used for projected model counting.

3 Blocked Clause Elimination for Projected Model Counting

Our goal is to use blocked clause elimination dynamically during search in projected model counting. The primary challenge is to identify conditions under which such simplification is allowed. Section 3.1 provides novel theoretical insights permitting the removal of blocked clauses and Section 3.2 introduces new algorithms to efficiently identify them.

3.1 Theoretical Insights

As illustrated by Example 6, the BCE rule cannot be applied indiscriminately. When applied to the formula Σ provided in Example 1, the result is a tautological formula, indicating that $\|\mathtt{BCE}(\Sigma)\| = 1$ (since $Var(\mathtt{BCE}(\Sigma)) = \emptyset$ this correspond to $2^6 = 64$ models over $Var(\Sigma)$), which differs from $\|\Sigma\| = 9$. It is essential to note that blocked clause elimination guarantees the preservation of satisfiability but not necessarily equivalence or the number of models. However, the picture changes when addressing the projected model counting problem. As we will demonstrate in Proposition 7, it is feasible to eliminate clauses that are blocked on projected variables. The rationale behind this lies in the fact that when focusing on sub-formulas containing only projected variables, the requirement is only to ensure satisfiability. Consequently, clauses blocked on projected variables can be removed in this case:

▶ **Proposition 7.** *Let $\exists x.\Sigma$ be an existentially quantified* CNF *formula. If a non-tautological clause $\alpha \in \Sigma$ is blocked by a literal $\ell \in \alpha$ with $Var(\ell) = x$, then $\exists x.\Sigma$ is logically equivalent to $\exists x.\Sigma'$, where $\Sigma' = \Sigma \setminus \{\alpha\}$.*

Proof. To establish the logical equivalence $\exists x.\Sigma \equiv \exists x.\Sigma'$, we need to demonstrate both (1) $\exists x.\Sigma \models \exists x.\Sigma'$ and (2) $\exists x.\Sigma' \models \exists x.\Sigma$. For condition (1) since $\Sigma \models \Sigma'$ it follows directly that $\exists x.\Sigma \models \exists x.\Sigma'$. Now, let us demonstrate the second condition. We have to prove for any interpretation ω satisfying $\exists x.\Sigma'$, that ω also satisfies $\exists x.\Sigma$. Consider an interpretation ω satisfying $\exists x.\Sigma'$. This means that ω satisfies $(\Sigma'_{|x} \vee \Sigma'_{|\neg x})$. We need to address two scenarios depending on whether ω satisfies $\Sigma'_{|x}$ or $\Sigma'_{|\neg x}$. If ω satisfies $\Sigma'_{|x}$, then $\Sigma'_{|x} \equiv \Sigma_{|x}$. Since $\Sigma_{|x}$ entails $\Sigma_{|x} \vee \Sigma_{|\neg x}$, we conclude that ω satisfies $\exists x.\Sigma$. Let us consider the second scenario where ω satisfies $\Sigma'_{|\neg x}$ but not $\Sigma'_{|x}$ (the case when $\omega \models \Sigma'_{|x}$ has just been discussed). First, both $\Sigma'_{|x}$ and $\Sigma'_{|\neg x}$ contain clauses from Σ' that do not involve variable x. Therefore, if ω does not satisfy Σ'_{x} but satisfies $\Sigma'_{\neg x}$, there must be a clause $\beta \in \Sigma'$ with $\neg x \in \beta$ and $\omega \not\models \beta_{|x}$. Now, let us demonstrate that ω satisfies $\Sigma_{|\neg x}$. Since $\Sigma_{|\neg x} \equiv (\Sigma' \wedge \alpha)_{|\neg x} \equiv \Sigma'_{|\neg x} \wedge \alpha_{|\neg x}$, we only need to show that ω satisfies $\alpha_{|\neg x}$. As α is blocked on x in Σ, each resolvent between α and a clause of Σ containing $\neg x$ is a tautology. Particularly, $\beta \oplus \alpha$ is a tautology, implying that there exists a literal $\exists y \in \beta$ such that $\neg y \in \alpha$ and $x \neq y$. Since we have established that $\omega \not\models \beta_{|x}$, this implies that ω satisfies $\neg y$, hence ω satisfies $\alpha_{|\neg x}$. This demonstrates that ω satisfies $\Sigma'_{\neg x} \wedge \alpha_{\neg x}$, and consequently, ω satisfies $\Sigma_{|\neg x}$. Using similar reasoning as before, we can show that ω satisfies $\exists x.\Sigma$. Therefore, for any interpretation ω that satisfies $\exists x.\Sigma'$, it follows that ω satisfies $\exists x.\Sigma$, proving $\exists x.\Sigma' \models \exists x.\Sigma$. ◀

Proposition 7 only considers formulas with a single existentially quantified and thus projected variable. This can be extended to sets of variables:

▶ **Corollary 8.** *Let $\exists X.\Sigma$ be an existentially quantified* CNF *formula. If a non-tautological clause $\alpha \in \Sigma$ is blocked by a literal $\ell \in \alpha$ such that $Var(\ell) \in X$, then $\exists X.\Sigma$ is logically equivalent to $\exists X.\Sigma'$, where $\Sigma' = \Sigma \setminus \{\alpha\}$.*

Proof. The proof is straightforward. Proposition 7 establishes $\exists x.\Sigma \equiv \exists x.\Sigma'$. Therefore, we directly deduce that $\exists X \setminus \{x\}.(\exists x.\Sigma) \equiv \exists X \setminus \{x\}.(\exists x.\Sigma')$. ◀

Corollary 8 demonstrates the potential of utilizing blocked clause elimination to enhance projected model counters. Our objective is not only to identify the set of blocked clauses in preprocessing but also to perform this operation during search dynamically. However, naive algorithms for blocked clause elimination are in the worst case at least quadratic in the size of the formula, which is clearly infeasible for dynamic blocked clause elimination. In the following section, we capitalize on the observation that model counters typically follow the trace of DPLL solvers. To efficiently detect blocking literals and remove blocked clauses, a dedicated data structure along with associated algorithms are designed.

3.2 Implementation Details

To improve the efficiency of identifying clauses eligible for removal through the blocked clause elimination rule, we introduce the `BlockedClauseManager` object in this section. This specialized utility integrates efficient structures and algorithms crafted for this purpose, and is not exclusive to the projected model counter `d4`. It can be seamlessly employed in any state-of-the-art projected model counter.

To identify clauses eligible for elimination due to being blocked by a literal, we use a mechanism akin to the concept of watched literals. Given a formula $\exists X.\Sigma$, we aim to capture scenarios where a clause α cannot be eliminated via the blocked clause elimination rule, which occurs when there is no literal $\ell \in \alpha$ such that α is blocked on x, and $Var(\ell) \in X$. Specifically, a clause α is not blocked on a literal $\bar{\ell} \in \alpha$ if there exists another clause α' such that $\bar{\ell} \in \alpha'$ and $\alpha \oplus \alpha'$ is not a tautology. Consequently, the invariant we adopt stipulates that for each literal $\ell \in \alpha$ such that $Var(\ell) \in X$, either ℓ is assigned or there must exist a clause α' where $\bar{\ell} \in \alpha'$, and $\alpha \oplus \alpha' \not\equiv \top$.

▶ **Example 9** (Example 1 cont'd). When evaluating α_3, it is not feasible to associate the literal $\neg y_1$ with a clause from Σ without resulting in a tautology. Therefore, α_3 can be safely removed from Σ. Conversely, when examining α_{11}, it is feasible to associate the literal y_3 with clause α_9 and the literal y_2 with clause α_6, demonstrating that α_{11} cannot be eliminated from the formula using the blocked clause elimination rule.

Since blocked elimination can ignore (implied) learned clauses [17], the set $\{\alpha\} \oplus S_{\bar{\ell}}$, representing possible resolutions on a literal ℓ concerning a clause $\alpha \in \Sigma$, can be computed once at the outset. Consequently, when the watched clause to assess whether α is blocked on ℓ is deactivated, it suffices to consider clauses in $\{\alpha\} \oplus S_{\bar{\ell}}$ rather than re-evaluating each clause of $S_{\bar{\ell}}$ to determine if the resolution rule yields a tautology. The first data structure incorporated into our `BlockedClauseManager` is thus a set of triples $(\ell, \alpha, \{\alpha\} \oplus S_{\bar{\ell}})$, referred to as `protectedTriple`.

The function `initProtectedTriple`, outlined in Algorithm 1, is designed for this purpose. When provided with the existentially quantified CNF formula $\exists X.\Sigma$, it begins by enumerating all variables x in X (lines 2–5). Subsequently, it iterates through each possible triple $(\ell, \alpha, \{\alpha\} \oplus S_{\bar{\ell}})$ such that $\ell \in \{x, \neg x\}$, $\ell \in \alpha$, and $\alpha \in S_\ell$ (lines 3–5), adding them into `protectedTriple` (line 5). Moving forward, we will primarily work with clause identifiers rather than the clauses themselves. Therefore, when referring to a clause α in the following discussions, we are actually addressing its identifier. This applies similarly to sets of clauses; we will focus on the set of identifiers corresponding to the clauses within the set.

Algorithm 1 `initProtectedTriple`.

Input: $\exists X.\Sigma$ an existentially quantified `CNF` formula.

`protectedTriple` $\leftarrow \emptyset$
for $x \in X$ **do**
  for $\ell \in \{x, \neg x\}$ **do**
    for $\alpha \in S_\ell(\Sigma)$ **do**
      `protectedTriple` $\leftarrow$ `protectedTriple` $\cup \{(\ell, \alpha, \{\alpha\} \oplus S_{\bar{\ell}}(\Sigma))\}$

▶ **Example 10** (Example 1 cont'd). Upon invoking the function `initProtectedTriple` on the existentially quantified formula $\exists X.\Sigma$ provided in Example 1, the set `protectedTriple` contains the following triples: $(y_1, 4, \{\})$, $(\neg y_1, 3, \{\})$, $(y_2, 5, \{6\})$, $(y_2, 11, \{6\})$, $(\neg y_2, 10, \{\})$, $(\neg y_2, 6, \{5, 11\})$, $(y_3, 7, \{9, 10\})$, $(y_3, 11, \{9\})$, $(\neg y_3, 8, \{\})$, $(\neg y_3, 9, \{7, 11\})$, $(\neg y_3, 10, \{7\})$.

For each triple $(\ell, \alpha, \mathcal{C})$ in `protectedTriple`, we need to watch a clause from $\mathcal{C}$ to ensure that clause α is not blocked by ℓ. To achieve this, we incorporate into `BlockedClauseManager` a map of watching lists, denoted as `watches`. This structure associates each clause $\alpha \in \Sigma$ with a set of triples `watches[`α`]` that are being watched by α.

Algorithm 2 presents the pseudo-code for the function `initWatchList`. Given an existentially quantified `CNF` formula $\exists X.\Sigma$, this function initializes the `watches` structure and returns the indices of blocked clauses U, which are the clauses for which it is impossible to associate a sentinel. The function begins by initializing the set of blocked clauses as empty (line 2). Then, it initializes the map `watches` by associating an empty set with each clause of Σ (lines 2–3). Next, it iterates over the triples in the `protectedTriple` set to associate a sentinel with each of them (lines 4–6). For each triple $t = (\ell, \alpha, \mathcal{C})$, where $\mathcal{C}$ represents the set of non-tautological clauses, the algorithm checks whether $\mathcal{C}$ is empty. If it is, α is added to the set of blocked clauses (line 5). Otherwise, a clause α' from $\mathcal{C}$ is selected, and the triple t is added to the watching list of α' (line 6).

Algorithm 2 `initWatchList`.

Input: $\exists X.\Sigma$ an existentially quantified `CNF` formula.
Output: B is a set of identifiers of clauses that are blocked.

1 $U \leftarrow \emptyset$
2 Let `watches` an empty map
3 **for** $\alpha \in \Sigma$ *s.t.* $Var(\alpha) \cap X \neq \emptyset$ **do** `watches`$[\alpha] = \{\}$
4 **for** $t = (\ell, \alpha, \mathcal{C}) \in$ `protectedTriple` **do**
5   **if** $\mathcal{C} = \emptyset$ **then** $U \leftarrow U \cup \{\alpha\}$
6   **else** `watches`$[\alpha'] \leftarrow$ `watches`$[\alpha'] \cup \{t\}$ with $\alpha' \in \mathcal{C}$
7 **return** U

▶ **Example 11** (Example 1 cont'd). Upon calling the function `initWatchList` on the existentially quantified formula $\exists X.\Sigma$ provided in Example 1, the following represents a potential initialization of the `watched` structure:

`watches`$[6] = \{(y_2, 5, \{6\}), (y_2, 11, \{6\})\}$ `watches`$[5] = \{(\neg y_2, 6, \{5, 11\})\}$
`watches`$[9] = \{(y_3, 7, \{9, 10\}), (y_3, 11, \{9\})\}$ `watches`$[7] = \{(\neg y_3, 9, \{7, 11\}), (\neg y_3, 10, \{7\})\}$
`watches`$[3] =$ `watches`$[4] =$ `watches`$[8] =$ `watches`$[10] =$ `watches`$[11] = \emptyset$

To finalize the initialization of the `BlockedClauseManager` object, we incorporate two arrays for maintaining records of assigned variables and satisfied clauses. The first array, named `isAssignedVar`, associates each variable in X with a Boolean value set to `true` if the variable is assigned, and `false` otherwise. The second array, named `isActiveClause`, associates each clause of Σ (identified by their identifier) with a Boolean variable set to `true` if the clause is active, and `false` otherwise. The arrays `isAssignedVar` and `isActiveClause` are initialized with `false` and `true`, respectively, for all their elements. We also need a stack S of pairs, each consisting of variables and clauses. This stack is used to track the changes made to `isAssignedVar` and `isActiveClause` during each call of the function `propagate`.

Algorithm 3 outlines all the necessary steps for the initialization process. It begins by initializing the two arrays (lines 1–2). Then, the set of triples is initialized by invoking the function `initProtectedTriple` on $\exists X.\Sigma$. Next, the `watches` structure is initialized by calling the function `initWatchList` on $\exists X.\Sigma$, and the set of blocked clauses is collected in U. We initialize S as an empty stack of pairs, where each pair consists of a set of variables and a set of clauses. Finally, the function `propagate` is called to gather all the initially blocked clauses. This function, described afterwards, takes a set of inactive clauses and a set of freshly assigned variables as input, and returns a set of clauses that are identified as blocked (further details will be provided later).

Algorithm 3 `init`.

Input: $\exists X.\Sigma$ an existentially quantified `CNF` formula.
Output: B is a set of identifiers of clauses that are blocked.

1 Let `isAssigned` be an array s.t. `isAssigned`$[x]$ = `false` for each $x \in X$
2 Let `isActiveClause` be an array s.t. `isActiveClause`$[\alpha]$ = `true` for each $\alpha \in \Sigma$
3 `initProtectedTriple`$(\exists X.\Sigma)$
4 $U \leftarrow$ `initWatchList`$(\exists X.\Sigma)$
5 S is an empty stack of pairs of the form (variables, clauses)
6 **return** `propagate(`U`,` $\emptyset$`)`

Before delving into the specifics of how the function `propagate` operates, it is important to highlight that when conditioning a formula by a literal x, without rendering it unsatisfiable, there is no need to consider clauses shortened by this assignment. Consider a clause $\alpha \in \Sigma$ with $\bar{\ell} \in \alpha$. We aim to demonstrate that $\alpha \setminus \{\bar{\ell}\} \in \Sigma_{|\ell}$ cannot be blocked by any literal $\ell' \in \alpha \setminus \{\ell\}$ in $\Sigma_{|\ell}$. Given that α is not blocked in Σ, for every $\ell' \in \alpha \setminus \{\bar{\ell}\}$, there exists $\alpha' \in \Sigma$ such that $\bar{\ell}' \in \alpha'$ and $\alpha \oplus \alpha' \not\equiv \top$. Firstly, note that $\ell \notin \alpha'$; otherwise, $\alpha \oplus \alpha'$ would be a tautology. We then consider two cases based on whether $\bar{\ell}$ belongs to α'. In the first case when $\bar{\ell} \in \alpha'$, we have $\alpha' \setminus \{\bar{\ell}\} \in \Sigma_{|\ell}$, and since the resolution between $\alpha \setminus \{\bar{\ell}\}$ and $\alpha' \setminus \{\bar{\ell}\}$ is not a tautology, it follows that $\alpha \setminus \{\ell\}$ is not blocked on ℓ'. In the second case where $\bar{\ell} \notin \alpha'$, $\alpha' \in \Sigma_{|\ell}$, and once more, $\alpha \setminus \{\ell\}$ is not blocked on ℓ' because $\alpha' \setminus \{\ell\} \oplus \alpha' \not\equiv \top$.

▶ **Example 12** (Example 9 cont'd)**.** Let us examine the formula $\Sigma_{|\neg x_2}$. It is evident that α_{11} remains an unblocked because, with the literals present in the resulting clauses $y_3 \vee y_2$, we can still reference the same clauses from $\Sigma_{|\neg x_2}$ to maintain the invariant.

There are two scenarios where it becomes pertinent to evaluate whether a clause can be eliminated due to being blocked: when an active clause has been satisfied by a literal, or when an active clause has been blocked by a literal. Thus, once certain clauses become inactive, that are clauses satisfied or blocked, it becomes imperative to update the `watches` structure accordingly. The process for this update closely resembles the mechanism for updating watched literals in modern SAT solvers. Specifically, for each newly inactive clause α, we need to iterate through the list of triples from `watches`$[\alpha]$ associated with α. For each triple $t = (\ell, \alpha, \mathcal{C})$ of `watches`$[\alpha]$, if ℓ is not assigned and α is active, we must search for another sentinel in $\mathcal{C}$ – that is, an active clause. The concept here is to ensure that each triple $t = (\ell, \alpha, \mathcal{C})$ is linked with a clause. Additionally, if t is active, meaning α is active, it should be watched by an active clause. Otherwise, if t is watched by an inactive clause α', we must ensure that when we reactivate α, α' is also made active again. This aspect is crucial as it guarantees the backtrack freeness of our structure.

Algorithm 4 outlines the pseudo-code for the function `propagate`, which fulfills the aforementioned requirements. It takes as input a set of clauses identified by their identifiers, denoted as U, that have become inactive, and a set of freshly assigned variables Y. The function returns the set B of clauses detected as being blocked. The algorithm begins by updating the two arrays `isAssignedVar` and `isActiveClause` to reflect the assignment of variables in Y and the inactivation of clauses in U (lines 1–2). Next, the set of blocked clauses B is initialized to be empty (line 3). Then, for each inactive clause α stored in U, the `watches` map is updated (lines 4–18). To accomplish this, an inactive clause α is selected and removed from U (lines 5–6). Subsequently, the set `tmpWatch` is initialized to be empty, and it is used to store triples that will be watched by α, containing triples with assigned variables or inactive clauses (line 18). Since α is now inactive, active triples associated with α need to be redistributed to other active clauses.

This operation is conducted in the for loop where each triple $t = (\ell, \alpha', \mathcal{C})$ from `watches`$[\alpha]$ is considered (lines 8–17). If ℓ is assigned or if α' is inactive (line 9), α can continue to watch t, and thus t is added to `tmpWatch` (line 10). However, if there exists an active clause α'' in $\mathcal{C}$ (line 11), then triple t is added to the watch list of α'' (line 12). Lastly, if it is impossible to associate t with an active clause (lines 13–17), then clause α' is considered blocked, implying that α can continue to watch α' since they will both become active together upon backtracking (line 14). Moreover, α' is added to B, α' is added to U to handle α' later (line 16), and clause α' is marked as inactive (line 17). Upon completing the update of the `watches` map, the assigned variables and inactivated clauses are pushed onto stack S (line 19). Finally, the set of clauses identified as blocked is returned at line 20.

▶ **Example 13** (Example 1 cont'd). Consider the scenario where the literal x_1 is assigned to `true`. In this case, clauses $1, 4, 6, 9$ become satisfied. Invoking the function `propagate` with this information will result in the function returning $\{11\}$ as the set of detected blocked clauses. The various structures within our `BlockedClauseManager` object will be updated as follows:

`watches`$[7] = \{(\neg y_3, 9, \{7, 11\}), (\neg y_3, 10, \{7\})\}$ `watches`$[5] = \{(\neg y_2, 6, \{5, 11\}), (\neg y_2, 10, \{5\})\}$
`watches`$[9] = \{(y_3, 11, \{9\})\}$ `watches`$[6] = \{((y_2, 11, \{6\})\}$
`watches`$[10] = \{(y_2, 5, \{6, 10\}), (y_3, 7, \{9, 10\})\}$
`watches`$[3]$ = `watches`$[4]$ = `watches`$[8]$ = `watches`$[11] = \emptyset$ $S = (\{x_1\}, \{1, 4, 6, 9, 11\})$
`isActiveClause` $= [0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0]$ `isAssignedVar` $= [1, 0, 0]$

As mentioned earlier, our structure is designed to be backtrack-free. Therefore, the only operation needed during backtracking is to retrieve from the stack S the elements of the two arrays `isAssignedVar` and `isActiveClause` that require reinitialization. Algorithm 5 outlines the steps involved in the backtracking process.

To conclude this section, let us illustrate how `BlockedClauseManager` is used within a `DPLL`-style projected model counter, such as the one employed in the model counter `d4` [24]. It is worth noting that our proposed approach can also be applied to other types of projected

Algorithm 4 `propagate`.

Input: U represents a set of clause identifiers corresponding to the newly inactive clauses, and Y denotes the set of variables that have been newly assigned.
Output: B is a set of identifiers of clauses that are blocked.

```
for x ∈ Y do isAssigned[y] = true
for α ∈ U do isActiveClause[α] = false
B ← ∅
while U ≠ ∅ do
    Let α ∈ U
    U ← U \ {α}
    tmpWatch = ∅
    for t = (x, α′, C) ∈ watches[α] do
        if isAssigned[x] or not isActiveClause[α′] then
            tmpWatch ← tmpWatch ∪ {t}
        else if ∃α″ ∈ C s.t. isActiveClause[α′] then
            watches[α″] ← watches[α″] ∪ {t}
        else
            tmpWatch ← tmpWatch ∪ {t}
            B ← B ∪ {α′}
            U ← U ∪ {α′}
            isActiveClause[α′] ← false
    watches[α] ← tmpWatch
push the couple (Y, B ∪ U) in S
return B
```

model counters, such as those discussed in [30, 26, 8, 7, 10]. Algorithm 6 outlines the `count` function, which is invoked on $\exists X.\Sigma$, an existentially quantified `CNF` formula, returning the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$. Specifically, this function creates a global variable named *bce*, which is a `BlockedClauseManager` object (line 1), initializes it (line 2), removes the detected blocked clauses (line 3), and calls the recursive algorithm `count_main` on the simplified formula (line 4). It is important to note that while this function initializes the object necessary for enforcing `BCE` during model counting, the actual computation of the number of models is performed within the `count_main` function, which is described afterwards.

Algorithm 7 outlines the recursive function `count_main`, which serves as a pseudo-code representation of a `DPLL`-style projected model counter. This function operates on $\exists X.\Sigma$, an existentially quantified `CNF` formula, and computes the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$. The blue portion of the algorithm, which differs from the baseline due to the inclusion of `BCE` management, will be discussed later.

The function begins by invoking `BCP` on the input formula Σ at line 1. For simplicity, we assume that `BCP` returns a triple consisting of the set of unit literals *units*, the set of satisfied clauses S, and the simplified formula Σ without clauses from S and the unit literals from *units*. If the formula returned by `BCP` contains an empty clause, indicating unsatisfiability, the function returns 0 (line 3). At line 5, the algorithm visits a cache to determine whether the current formula Σ has been previously encountered during the search. The cache, which

Algorithm 5 `backtrack`.

$(X, \mathcal{C}) \leftarrow$ `top`(S)
`pop`(S)
for $x \in X$ **do** `isAssignedVar`$[x] =$ `false`
for $\alpha \in \mathcal{C}$ **do** `isActiveClause`$[\alpha] =$ `true`

Algorithm 6 `count`.

Input: $\exists X.\Sigma$ an existentially quantified `CNF` formula.
Output: the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$

---

global *bce* is an `BlockedClauseManager` object
$B \leftarrow$ `init`(bce)
$\Sigma \leftarrow \Sigma \setminus \{\Sigma[i] \text{ s.t. } i \in B\}$
return `count_main`(Σ)

---

starts empty, stores pairs comprising a `CNF` formula and its corresponding projected model count with respect to X. Whenever Σ is found in the cache, instead of recalculating $\|\exists X.\Sigma\|$ from scratch, the algorithm retrieves `cache`(Σ) (line 7) to streamline the computation. If the formula is satisfiable, `connectedComponents` is called (line 8) on Σ to partition it into a set of `CNF` formulae that are pairwise variable-disjoint. This procedure is a standard method employed in model counters. It identifies connected components of the primal graph of Σ and returns a set *comps* of `CNF` formulae, ensuring that each pair of distinct formulae in *comps* does not share any common variable. The variable *cpt*, used to accumulate intermediate model counts, is initialized to 1 (line 9). Then, the function iterates over the connected components Σ' identified in *comps* (lines 10–14) to count the number of models of each component. If the considered component Σ' only contains variables from X, the model count accumulated in *cpt* is multiplied by 1 if Σ' is satisfiable and 0 otherwise. If $Var(\Sigma') \setminus X$ is not empty, a variable v from this set is chosen, and the function `count_main` is recursively called on Σ' where v is assigned true and on Σ' where v is assigned false. The results returned by the two recursive calls are then summed up and multiplied by the variable *cpt* (line 14). Before returning the accumulated model count in *cpt* (line 17), the formula Σ is added to the cache associated with the corresponding projected model count *cpt* (line 15).

To integrate `BCE` into the search process (blue part), we first call the function `propagate` on S and *units* to update the information managed by the *bce* object and compute the set of blocked clauses B (line 3). Then, at line 4, we eliminate the stored set of blocked clauses B. To maintain consistency between the `BCE` manager information and the ongoing recursive call, the backtrack function must be executed before returning the calculated model count (lines 6 and 16).

4 Experimental Evaluation

Our aim was to empirically assess the advantages of employing blocked clause elimination in solving instances of the projected model counting problem. For our experimentation, we used 500 CNF instances from the three recent model counting competitions (the 2021, 2022, and 2023 editions documented at `https://mccompetition.org/`). We excluded instances from the 2020 competition due to incompatibility with our software caused by changes in

Algorithm 7 `count_main`.

Input: $\exists X.\Sigma$ an existentially quantified `CNF` formula.
Output: the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$

```
(units, S, Σ) ← bcp(Σ)
if ⊥ ∈ Σ then return 0
B ← bce.propagate(S, {x|ℓ ∈ units and Var(ℓ) = x})
Σ ← Σ \ {Σ[i] s.t. i ∈ B}
if cache(Σ) ≠ nil then
    bce.backtrack()
    return cache(Σ)
comps ← connectedComponents(Σ)
cpt ← 1
for Σ' ∈ comps do
    if Var(Σ') \ X = ∅ then cpt ← cpt × (SAT(Σ')?1 : 0)
    else
        Let v ∈ Var(Σ') \ X
        cpt ← cpt × (count_main(Σ' ∧ v) + count_main(Σ' ∧ ¬v))
cache(Σ) ← cpt
bce.backtrack()
return cpt
```

the input format. The instances were categorized into three datasets: 200 from the 2021 competition, 200 from the 2022 competition, and 100 from the 2023 competition. Notably, as the full set of 2023 instances was unavailable at the time of writing, we only included the 100 public instances provided by the organizers.

The projected model counter used for the evaluation was `d4` [24]. Our experiments were conducted on Intel Xeon E5-2643 processors running at 3.30 GHz with 32 GiB of RAM, operating on Linux CentOS. Regarding the model counting competition, each instance was subject to a time-out of 3600 seconds and a memory limit of 32 GiB. For each instance, we measured the computation times required by three different versions of `d4` for counting the numbers of projected models. These versions include:

- `d4`: This is the standard version of `d4`, as given at `https://github.com/crillab/d4v2`.
- `d4+BCE`$_p$: This version of `d4` incorporates blocked clause elimination performed once during a preprocessing phase.
- `d4+BCE`$_i$: In this version of `d4`, blocked clause elimination is performed dynamically throughout the search achieved by the model counter.

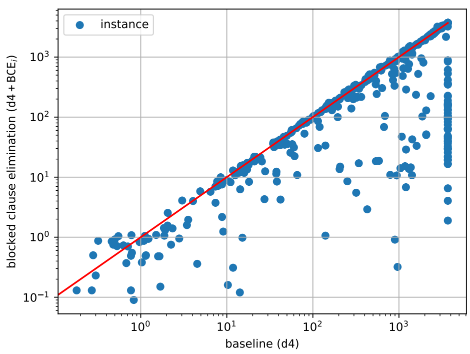
For all the versions under consideration, a preprocessing step of 60 seconds was conducted. This preprocessing involves running `BiPe` [27], followed by the occurrence elimination and vivification preprocessing for 10 iterations as described in [23] (we only replace the gate simplification with `BiPe`).

Table 1 presents the number of instances for which different versions of `d4` terminated within the specified time and memory constraints. The correctness of the extended versions of `d4` was verified by comparing their returned model counts with those of the baseline version. For all instances solved by the baseline version, the extended versions returned the same model counts. The table clearly demonstrates that leveraging dynamical blocked

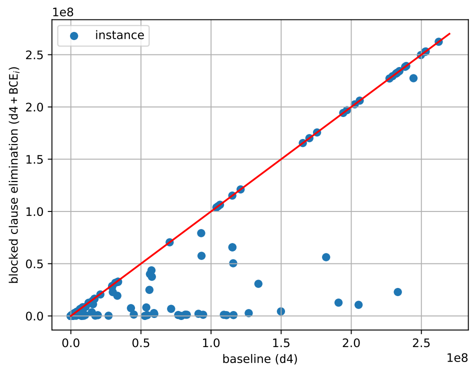
Table 1 The table shows the numbers of instances solved by different versions `d4` within a time limit of 3600 seconds and a memory limit of 32 GiB. The number of memory out (MO) are reported between brackets.

	2021 (200)	2022 (200)	2023 (100)	All (500)
`d4`	139 (56 MO)	149 (24 MO)	73 (9 MO)	361 (89 MO)
`d4+BCE`$_p$	139 (56 MO)	149 (25 MO)	73 (9 MO)	361 (90 MO)
`d4+BCE`$_i$	**172** (23 MO)	**163** (8 MO)	**78** (4 MO)	**413** (35 MO)

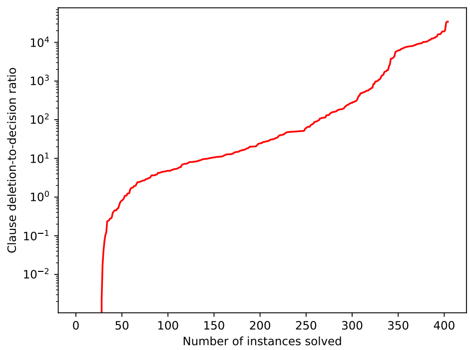
clause elimination significantly improves the performance of the model counter in practice. Furthermore, regardless the benchmark set considered, the version of `d4` equipped with dynamic blocked clause elimination systematically solved more instances than the two other versions. This indicates that the improvement is not limited to specific benchmark sets. Moreover, Table 1 shows that using blocked clause elimination solely during preprocessing phase did not lead to increase the number of instances solved. This demonstrates that for effective results, blocked clause elimination needs to be performed eagerly.



(a) Comparing the run times of `d4` and `d4+BCE`$_i$.



(b) Comparing the number of decisions made by `d4` and `d4+BCE`$_i$ on the instances solved by both.



(c) Plot used to estimate the extent of clause deletion relative to the number of decisions made for instances solved by `d4+BCE`$_i$.

Figure 1 Experimental results.

Figure 1a presents a pairwise comparison between `d4` and `d4+BCE`$_i$ on a scatter plot. Each data point represents an instance, with the x-axis indicating the time (in seconds) required to solve it using the baseline version of `d4`, and the y-axis representing the time for the enhanced version d4+BCE$_i$. The experimental results unequivocally demonstrate that the version of d4 with dynamic blocked clause elimination generally outperforms the baseline version of d4. Furthermore, the figure reveals instances where d4+BCE$_i$ achieves speeds one order of magnitude faster than the baseline version d4.

Focusing on instances solved by both approaches and exhibiting a solving time difference of more than five seconds, the baseline version d4 beat d4+BCE$_i$ for 75 instances, achieving an average speedup of 4%. This speedup is calculated as the ratio between the running times of the methods, with a peak improvement of 14% and the third quartile indicating a speedup of 7%. We investigated the factors contributing to the greater efficiency of the baseline version compared to the one employing dynamic blocked clause elimination, but we were unable to identify a definitive reason. It is hypothesized that the removal of clauses may negatively impact branching, potentially leading to a slightly larger search space explored by the model counter. On the contrary, we discovered 150 instances for which d4+BCE$_i$ outperformed the baseline version of d4. For them, d4+BCE$_i$ exhibits an average speed increase of 40 times compared to the baseline, with a peak improvement of up to 3000 times. The second quartile of the time distribution demonstrates a 50% improvement, while the third quartile shows a remarkable 600% enhancement.

Figure 1b showcases the number of decisions made by d4 and d4+BCE$_i$ on instances solved by both methods. This visualization sheds light on the fact that the performance enhancement cannot be solely attributed to a reduction in memory usage, which might otherwise account for the observed decrease in memory consumption with d4+BCE$_i$. It is widely recognized that the cache structure frequently serves as the primary memory bottleneck. Consequently, removing clauses reduces the size of cache entries, which typically results in decreased memory consumption and associated memory overhead. Nevertheless, as depicted in Figure 1b, the use of blocked clause dynamic elimination also results in a decrease in the number of decisions required by the model counter to complete its task. This underscores that the performance gain is not solely a consequence of an inadequate memory limit setting. Thus, even with a significant increase in the memory limit, employing dynamic blocked clause elimination proves highly advantageous in practice. Specifically, for 71 instances, d4 required fewer decisions than d4+BCE$_i$, with an average difference of 6460 decisions in favor of d4. The second quartile exhibited a difference of 102 decisions, while the third quartile showed a difference of 306 decisions across these instances. Conversely, d4+BCE$_i$ required fewer decisions than d4 for 231 instances, with an average difference of 12,249,678 decisions in favor of d4+BCE$_i$. The second quartile exhibited a difference of 33,005 decisions, while the third quartile showed a difference of 1,602,452 decisions across these instances.

Figure 1c gives the proportion of clauses removed through dynamic blocked clause relative to the number of decisions made for instances solved by d4+BCE$_i$. As observed in the plot, for approximately 300 instances, the average number of blocked clauses removed at each decision is at least 10. For about 100 of these instances, the average number of blocked clauses removed at each decision is at least 100. Additionally, for certain benchmarks, more than 1000 clauses where removed at each decision. While there is some variation in the extent of deletion across different steps, the plot clearly demonstrates that a substantial number of clauses are generally eliminated when employing dynamic blocked clause elimination.

5 Conclusion and Perspectives

In conclusion, this paper has explored the utilization of the blocked clause elimination dynamically during projected model counting. Despite its widespread application in the satisfiability problem, the blocked clause elimination rule posed challenges for model counting due to its inability to maintain the number of models unchanged. However, through focused attention on projected variables during the search for blocked clauses, we have demonstrated the feasibility of leveraging this rule while preserving the correct model count. To achieve this, we introduced a new data structure and corresponding algorithms tailored for leveraging blocked clause elimination dynamically during search. This innovative machinery has been integrated into the projected model counter `d4`, enabling us to conduct comprehensive experiments that showcase the computational benefits of our approach. Our results underscore the efficacy of leveraging the blocked clause elimination rule technique for projected model counting, opening avenues for further exploration and refinement in this domain.

Exploring extensions of blocked clause elimination (BCE) in the context of projected model counting is interesting future work. This particularly includes considering the elimination of resolution asymmetric tautologies (RAT) [17], or even covered [11, 5] or propagation redundant (PR) [13] clauses. These approaches hold the potential to uncover additional redundant clauses, that can be eliminated and thus improve efficiency of projected model counting. In addition, we envision the development of novel branching heuristics designed to prioritize the elimination of clauses that prevent removal of blocked clauses. These improved decision heuristics, could create more instances where clauses become blocked and thus eliminated, again with the goal to improve solver efficiency. Furthermore, we want to explore the applicability of blocked clause elimination to other reasoning tasks, particularly to the weighted `Max#SAT` problem [3, 2] or counting tree models of QBF formulas [21].

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.
2. Gilles Audemard, Jean-Marie Lagniez, and Marie Miceli. A new exact solver for (weighted) max#sat. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPICS.SAT.2022.28`.
3. Gilles Audemard, Jean-Marie Lagniez, Marie Miceli, and Olivier Roussel. Identifying soft cores in propositional formulæ. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Proceedings of the 14th International Conference on Agents and Artificial Intelligence, ICAART 2022, Volume 2, Online Streaming, February 3-5, 2022*, pages 486–495. SCITEPRESS, 2022. `doi:10.5220/0010892700003116`.
4. Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter J. Stuckey. #∃sat: Projected model counting. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015 – 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 2015. `doi:10.1007/978-3-319-24318-4_10`.
5. Lee A. Barnett, David M. Cerna, and Armin Biere. Covered clauses are not propagation redundant. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning – 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2020. `doi:10.1007/978-3-030-51074-9_3`.
6. Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2021. `doi:10.3233/FAIA200992`.

7 Jeffrey M. Dudek, Vu Phan, and Moshe Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1468–1476. AAAI Press, 2020. doi:10.1609/AAAI.V34I02.5505.

8 Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. DPMC: weighted model counting by dynamic programming on project-join trees. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming – 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 211–230. Springer, 2020. doi:10.1007/978-3-030-58475-7_13.

9 Johannes Klaus Fichte, Markus Hecher, Michael Morak, Patrick Thier, and Stefan Woltran. Solving projected model counting by utilizing treewidth and its limits. *Artif. Intell.*, 314:103810, 2023. doi:10.1016/J.ARTINT.2022.103810.

10 Markus Hecher, Patrick Thier, and Stefan Woltran. Taming high treewidth with abstraction, nested dynamic programming, and database technology. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2020. doi:10.1007/978-3-030-51825-7_25.

11 Marijn Heule, Matti Järvisalo, and Armin Biere. Covered clause elimination. In Andrei Voronkov, Geoff Sutcliffe, Matthias Baaz, and Christian G. Fermüller, editors, *Short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR-17-short, Yogyakarta, Indonesia, October 10-15, 2010*, volume 13 of *EPiC Series in Computing*, pages 41–46. EasyChair, 2010. doi:10.29007/CL8S.

12 Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res.*, 53:127–168, 2015. doi:10.1613/JAIR.4694.

13 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020. doi:10.1007/S10817-019-09516-0.

14 Marijn J. H. Heule and Hans van Maaren. Look-ahead based SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 183–212. IOS Press, 2021. doi:10.3233/FAIA200988.

15 Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010. doi:10.1007/978-3-642-12002-2_10.

16 Matti Järvisalo, Armin Biere, and Marijn Heule. Simulating circuit-level simplifications on CNF. *J. Autom. Reason.*, 49(4):583–619, 2012. doi:10.1007/S10817-011-9239-9.

17 Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning – 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012. doi:10.1007/978-3-642-31365-3_28.

18 Benjamin Kiesl, Martin Suda, Martina Seidl, Hans Tompits, and Armin Biere. Blocked clauses in first-order logic. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 31–48. EasyChair, 2017. doi:10.29007/C3WQ.

19 Vladimir Klebanov, Norbert Manthey, and Christian J. Muise. Sat-based analysis and quantification of information flow in programs. In Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems – 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8054 of *Lecture Notes in Computer Science*, pages 177–192. Springer, 2013. doi:10.1007/978-3-642-40196-1_16.

20 Oliver Kullmann. On a generalization of extended resolution. *Discret. Appl. Math.*, 96-97:149–176, 1999. doi:10.1016/S0166-218X(99)00037-2.

21 Jean-Marie Lagniez, Florent Capelli, Andreas Plank, and Martina Seidl. A top-down tree model counter for quantified boolean formulas. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, 3rd-9th August 2024, Jeju, South Korea*. ijcai.org, 2024.

22 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Definability for model counting. *Artif. Intell.*, 281:103229, 2020. doi:10.1016/J.ARTINT.2019.103229.

23 Jean-Marie Lagniez and Pierre Marquis. Preprocessing for propositional model counting. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2688–2694. AAAI Press, 2014. doi:10.1609/AAAI.V28I1.9116.

24 Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. doi:10.24963/IJCAI.2017/93.

25 Jean-Marie Lagniez and Pierre Marquis. On preprocessing techniques and their impact on propositional model counting. *J. Autom. Reason.*, 58(4):413–481, 2017. doi:10.1007/S10817-016-9370-8.

26 Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 – February 1, 2019*, pages 1536–1543. AAAI Press, 2019. doi:10.1609/AAAI.V33I01.33011536.

27 Jean-Marie Lagniez and Pierre Marquis. Boosting definability bipartition computation using SAT witnesses. In Sarah Alice Gaggl, Maria Vanina Martinez, and Magdalena Ortiz, editors, *Logics in Artificial Intelligence – 18th European Conference, JELIA 2023, Dresden, Germany, September 20-22, 2023, Proceedings*, volume 14281 of *Lecture Notes in Computer Science*, pages 697–711. Springer, 2023. doi:10.1007/978-3-031-43619-2_47.

28 Jérôme Lang, Paolo Liberatore, and Pierre Marquis. Propositional independence: Formula-variable independence and forgetting. *J. Artif. Intell. Res.*, 18:391–443, 2003. doi:10.1613/JAIR.1113.

29 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.

30 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1169–1176. ijcai.org, 2019. doi:10.24963/IJCAI.2019/163.

31 Ralf Wimmer, Karina Gitina, Jennifer Nist, Christoph Scholl, and Bernd Becker. Preprocessing for DQBF. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015 – 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 2015. doi:10.1007/978-3-319-24318-4_13.