Engineering an Efficient Probabilistic Exact Model Counter

Mate Soos[1][0000−0002−7355−881X] and Kuldeep S. Meel[2][0000−0001−9423−5270]

[1] Department of Computer Science, University of Toronto, Toronto, Canada
soos.mate@gmail.com
[2] School of Computer Science, Georgia Institute of Technology, GA, USA
meel@gatech.edu

Abstract. Given a formula F, the problem of model counting, also known as #SAT, is to compute the number of satisfying assignments of F. While model counting has emerged as a crucial primitive in diverse domains from quantitative information flow analysis to neural network verification, scalability remains a fundamental challenge despite advances in both exact and approximate counting techniques.

We present Ganak2, a novel framework that achieves substantial performance improvements through three key technical innovations: (1) a dual independent set framework maintaining distinct SAT-eligibility and decision sets, (2) chronological backtracking specifically adapted to model counting, and (3) refined residual formula processing incorporating SAT-specific techniques while maintaining seamless state transitions.

Our empirical evaluation on 1600 previous model counting competition instances demonstrates that Ganak2 successfully computes counts for 1121 instances within the one hour time limit, compared to 1032 instances by the prior state of the art approach, representing an 8.7% improvement. This progress is especially remarkable considering the extensive development and refinement of model counting tools over the years, driven by yearly competitive evaluation in the field.

1 Introduction

Given a Boolean formula F, the problem of model counting, also known as #SAT, is to compute $|\mathsf{Sol}(F)|$, i.e., the number of satisfying assignments of F. Model counting is #P-complete [37], a complexity class that characterizes counting problems associated with NP decision problems. Toda's theorem established that a single call to a #P oracle suffices to solve any problem in the polynomial hierarchy [36].

Despite its computational intractability, the practical significance of model counting has driven sustained research into developing effective algorithmic techniques, particularly in response to applications across diverse domains, including quantitative information flow analysis [8], network reliability [6], neural network verification [1], and probabilistic inference. For example, given a neural network $\mathcal{N}$ and an input domain $\mathcal{X}$, verifying robustness properties involves encoding the

network's behavior as a Boolean formula F such that each satisfying assignment of F corresponds to an input-output pair (x, y) where $x \in \mathcal{X}$ and $y = \mathcal{N}(x)$ violates the desired property. The verification question then reduces to determining if $|\mathsf{Sol}(F)| = 0$, while quantitative guarantees about the network's behavior can be obtained by computing $|\mathsf{Sol}(F)|/|\mathcal{X}|$, representing the fraction of inputs leading to property violations. The practical impact of model counting in these domains has led to sustained interest in developing scalable counters, as evidenced by yearly model counting competitions [9].

The development of model counting techniques has followed two primary trajectories. The first trajectory focuses on exact counting techniques, which integrate core technical advances from SAT solving: component caching to exploit problem decomposition, conflict-driven clause learning (CDCL) to prune the search space, and decision heuristics informed by structural properties of the formula. The second trajectory pursues approximate model counting through universal hashing techniques, exemplified by ApproxMC, which provides theoretical guarantees of (ε, δ) while achieving improved scalability. Recent efforts have focused on the development of specialized preprocessing techniques and advanced component caching schemes to handle industrial-scale instances. Although significant progress has been made in both exact and approximate counting techniques, the fundamental challenge of scalability remains a crucial bottleneck for practical deployment.

This work focuses on advancing exact model counting through careful algorithmic improvements to Ganak, a well-performing probabilistic exact model counter. We call the resulting counter Ganak2, which achieves significant performance improvements through the following technical contributions:

1. **Enhanced Residual Formula Processing:** Development of an optimized SAT solver architecture for residual formula processing that incorporates VSIDS scoring, restarts, and polarity caching.
2. **Dual Independent Set Framework:** Development of a novel algorithmic framework that maintains distinct SAT-eligibility (S) and decision (D) sets, where the S-set determines SAT solver transitions while the D-set guides branching decisions. This separation enables more efficient search space exploration, especially when combined with enhanced residual formula processing.
3. **Chronological Backtracking:** Adaptation of chronological backtracking to model counting, with the aim to mitigate challenges associated with learnt clause retention, and a specific emphasis on addressing the complications that arise in the context of weighted model counting.

Our extensive empirical evaluation demonstrates significant performance improvements across different configurations of Ganak2. The optimal configuration of Ganak2 solves 1121 instances within the timeout period of 3600 seconds, compared to the baseline configuration that solves only 784 instances. This represents a substantial improvement of over 40% in the number of solved instances. In comparison to prior state of the art, Ganak shows significant runtime

performance improvement: prior state of the art counter finishes only on 1032 instances within the same one hour time limit, a loss of 89 instances. Furthermore, the time-to-solution curves show that Ganak2 maintains better performance throughout the solving process, with particularly strong gains in the 500-1000 second range. These results demonstrate that careful algorithmic engineering can substantially improve the practical efficiency of exact model counting while maintaining theoretical guarantees.

The rest of the paper is organized as follows: Section 2 provides essential preliminaries and background on model counting and top-down counters. Section 3 surveys related work in model counting. Section 4 presents our key technical contributions: chronological backtracking integration, SAT solver optimizations for residual formula processing, and our dual independent set framework. Section 5 provides a comprehensive experimental evaluation of Ganak2, analyzing the impact of each algorithmic improvement on counter performance. Finally, Section 6 provides a summary of the key contributions and findings of this work.

2 Preliminaries

Let $X = \{x_1, x_2, \dots x_n\}$ be a set of Boolean variables. A literal is either a variable (x) or its negation ($\neg x$). A clause is a disjunction of literals, and a Boolean formula F is in conjunctive normal form (CNF) if it is a conjunction of clauses. An assignment $\sigma : X \mapsto \{0, 1\}$ is called a *satisfying assignment* or *solution* of F if it makes F evaluate to true. We denote the set of all solutions of F by $\mathsf{Sol}(F)$.

In weighted model counting, each literal l is assigned a weight $W(l) \in [0, 1]$. The weight of an assignment σ, denoted $w(\sigma)$, is the product of weights of all literals that are satisfied by σ: $W(\sigma) = \prod_{l:\sigma \text{ satisfies } l} w(l)$. Given a formula F and weight function W, the weighted model count of F, denoted $W(F)$, is the sum of weights of all satisfying assignments: $W(F) = \sum_{\sigma \in \mathsf{Sol}(F)} w(\sigma)$

A probabilistic exact counter takes in a formula F, a weight function W, and δ, and returns c such that $\Pr[c = W(F)] \geq 1 - \delta$. The unweighted model counting problem is a special case where all literal weights are 1, in which case $W(F) = |\mathsf{Sol}(F)|$. Often, we are interested in counting over a subset of variables. Given a subset of variables $P \subseteq X$, we denote by $\mathsf{Sol}(F)_{\downarrow P}$ the projection of $\mathsf{Sol}(F)$ onto variables in P. Formally, two assignments σ_1 and σ_2 belong to the same equivalence class if they agree on their assignments to P. For weighted model counting, the weight of a projected solution is the sum of weights of all solutions in its equivalence class.

Components and Residual Formulas Modern model counters employ component decomposition and caching [30] to achieve scalability. A component is a subformula that can be solved independently of the rest of the formula. Formally, given a formula F and an assignment σ to a subset of variables, a component is a maximal set of clauses $C \subseteq F|_\sigma$ such that for any other component C', we have $\mathsf{Vars}(C) \cap \mathsf{Vars}(C') = \emptyset$, where $F|_\sigma$ denotes the formula obtained by substituting

the assignment σ in F. This $F|_\sigma$ is called a *residual formula* when it is being created by the model counter while running: residual formulas are being created at a high pace in model counters, as after each decision and propagation, the remaining formula $F|_\sigma$ is a residual formula that needs to be counted. A residual formula is made up of one or more components, each component being a connected, independent (sub)formula of a residual formula.

Probabilistic Component Caching Component caching is crucial for the performance of model counters. Each component is uniquely identified by its signature, which typically consists of (1) the list of variables appearing in the component (vars), and (2) the list of clause IDs in the component (cls). The efficiency of component caching significantly impacts the overall performance of model counters, as it allows reuse of previously computed results when identical components are encountered during the counting process. Ganak [32] introduced probabilistic component caching, where the vars + cls signature is hashed to reduce the memory footprint. This introduces the possibility of hash collisions, which may lead to incorrect results with arbitrarily small (but nonzero) probability, controlled by the user-specified parameter δ.

Tree Decompositions Treewidth is a graph parameter that measures how closely a graph resembles a tree [4], capturing the complexity of problems [29] by decomposing the graph into a tree-like structure of bounded-size subsets.

Independent Set Instances to be counted always contain a so-called projection set P, which in the case of so-called unprojected instances is the set of all variables. The independent set of an instance is a set of variables I such that if P was replaced with I, the instance would have the same exact model count. Hence, the most trivial independent set is $I = P$. Minimization of the independent set is a key optimization in model counting, as demonstrated by the work of Lagniez et al. [21]. While I is often a subset of P (as is the case in [21,33]), this does not have to be the case. It can be a subset, superset, or entirely different than P. As long as the model count remains the same, the independent set is valid. In our work, we consider independent sets that are either subsets or supersets of P.

3 Related Work

The development of efficient model counting techniques has witnessed sustained research effort over the past two decades, with several distinct algorithmic paradigms emerging. The earliest approaches to practical model counting extended the DPLL framework through systematic enumeration of partial solutions [3]. A significant breakthrough came with Bayardo and Pehoushek's introduction of component caching [18], which exploits the observation that for a formula φ decomposable into components $C_1, C_2, \ldots, C_n$ with disjoint variable sets, $|\mathsf{Sol}(\varphi)| = \prod_i |\mathsf{Sol}(C_i)|$. The key insight was that identical components often recur in different parts of the search, making caching an effective optimization strategy.

Integration of component caching with Conflict Driven Clause Learning (CDCL) marked another pivotal development, first realized in the Cachet model counter [30]. This approach was further refined by Thurley through sharpSAT [35], which introduced improved component encoding schemes and enhanced decision heuristics. More recently, Ganak [32] advanced this line of research by incorporating probabilistic caching strategies and leveraging independent set information to guide search heuristics. Recent years have seen growing interest in exploiting low-width tree decompositions for model counting. Notable implementations include gpusat [11], NestHDB [15], and DPMC-LG [5], each taking distinct approaches to leveraging tree structure. While gpusat and the tensor implementation of DPMC-LG employ pure dynamic programming approaches with time complexity exponential in treewidth, NestHDB adopts a hybrid strategy, incorporating sharpSAT-style[‡] search for high-treewidth subproblems. SharpSAT-TD [19] represents a significant advancement in this direction by integrating tree decomposition information directly into variable selection heuristics while maintaining the core CDCL architecture.

Recent work has focused on integrating algorithmic techniques from satisfiability solving to improve performance. Enhanced preprocessing techniques, such as those pioneered by B+E [21], SharpSAT-TD's preprocessor [20], and Arjun [33] have been developed that preserve model count while reducing formula size. In a similar vein, blocked clause elimination [17] has been adopted by Lagniez et al. [22] to the counter d4 with great success. Despite these advances, substantial challenges remain in scaling to challenging instances and handling diverse formula structures.

4 Technical Overview

This section presents our key technical contributions in enhancing the scalability of exact model counting. Section 4.1 describes our refined SAT solver integration, which incorporates restarts, VSIDS scoring, and polarity caching for residual formula processing. Section 4.2 introduces our dual independent set framework, which maintains distinct sets for SAT-eligibility and decision variables to optimize search space exploration. Finally, Section 4.3 describes our adaptation of chronological backtracking for model counting, with particular attention to the challenges arising in weighted counting scenarios.

4.1 Enhanced Residual Formula Processing

A key optimization in modern model counting is the integration of SAT solving for residual formula processing in gpmc by Suzuki et al. [34]. This approach leverages the observation that once all variables in any given independent set are assigned, the residual formula can be processed using a SAT solver rather than

[‡]Note that the respective authors chose to capitalize sharpSAT and SharpSAT-TD differently.

continuing with the more expensive model counting procedure. The intuition behind this approach is that if the SAT solver finds a satisfying assignment, the count of the residual formula is 1; otherwise, it is 0.

The original implementation in gpmc ensures consistency with the model counter through several design choices. It shares core data structures including watchlists, propagation queues, and assignment stacks between the SAT solving and model counting phases. Our implementation builds upon this foundation through several enhancements to the SAT solving phase. We adopt VSIDS scoring[§] [24] in place of VSADS [31] for more effective variable decisions, make use of polarity caching [28] and introduce Luby-based restarts [16] that were previously absent. We also seamlessly integrate it with chronological backtracking [26] as later explained, to allow smooth transition between SAT solving and model counting phases.

These improvements integrate established SAT solving techniques within the model counter's SAT solving engine. Our implementation maintains the same core data structures and algorithms as the model counter, with two key differences: component analysis is disabled, and the component cache is ignored during SAT solving phases, as they hold no relevance to the SAT solver, and would only slow down solving. Conflict analysis, clause database management, and propagation proceed as normal. The system handles two critical scenarios elegantly: when a learned clause necessitates backtracking to a level prior to the SAT solver's initialization, the system smoothly reverts to model counting. Conversely, upon finding a satisfying assignment, the solver computes the appropriate count, 1 for unweighted, or the product of weights for weighted counting, and returns control to the model counting procedure. In case of weighted model counting and a satisfying assignment, the count maybe different than 1, because we allow projected (and hence often weighted) variables to be part of the SAT solver's assignment, as explained below.

4.2 Dual Independent Set Framework

Model counting techniques have traditionally sought to minimize the given projection set, which is an independent set, driven by two key objectives: (1) enabling Bounded Variable Elimination (BVE) [7] for non-independent set variables, and (2) facilitating early transition to SAT solving through a small independent set. However, this unified approach imposes unnecessary constraints on model counting performance. We propose a novel framework that explicitly maintains two distinct independent set sets optimized for different purposes.

Definition 1 (Dual Independent Set). *Given a Boolean formula F defined over variables V and a projection set $\mathcal{P}$, a dual independent set consists of:*

[§]VSADS is a variant of VSIDS that uses additional literal frequency information called DLCS, to make better decisions. However, DLCS is expensive to compute, and hence no high-performance SAT solver uses it. We follow suit, using VSIDS in SAT solving mode, improving performance.

S-set *A SAT-eligibility set $S \subseteq \mathcal{P}$ that determines when SAT solver mode transition is permissible.*

D-set *A decision set $D \subseteq V$ that guides branching variable selection, where $D \supseteq \mathcal{P}$.*

This formulation generalizes traditional approaches where $D = S \subseteq P$, instead aiming for $S \subseteq P \subseteq D$.

The key insight underlying our framework is that the D and S-sets serve fundamentally different purposes in the model counting process. Consider a variable y that is functionally determined by a set of variables X, where $y \in P, X \subset P, y \notin X$. While y can be excluded from S since its value is fully determined once all variables in P are assigned (as established by Padoa's Theorem [27]), including y in D may still enable more efficient search space exploration as it allows more flexible branching. Note that this works even if y is weighted, as its value (and hence contribution to the weight) is always fixed given the values of X[¶].

The aim of maintaining a larger decision set (D-set) than SAT-eligibility set (S-set) is to reduce the treewidth of the residual formula after the decision has been made and propagated. Treewidth has been established to be a key factor in model counting performance [19]: lower treewidth is known to lead to significantly better performance. A larger D-set provides more flexibility in variable ordering, which can reduce the residual formulas' treewidth and minimize the formula's interconnectedness. This means the counter can choose variables that create more independent subproblems, effectively breaking the original complex formula into smaller, more manageable components. Hence, fewer components can cover the same search space. This behavior is demonstrated in our experimental results in Section 5, where the number of components encountered *decreases* as we enable (and extend) dual independent set—and count significantly more instances.

[¶]In our framework if a variable x_w is weighted, it can be removed from D but cannot be removed from S and hence cannot be eliminated. The elimination of x_w would require the function $x_w = f(X)$ to be computed, a technique from functional synthesis [13], which is beyond the scope of this work.

Example 1. Consider the following formula F over variables $X = \{x_1, x_2, x_3, y, z\}$ and projection set $\mathcal{P} = \{x_1, x_2, x_3\}$

$$F = (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (y \leftrightarrow (x_1 \wedge x_2)) \wedge (x_3 \leftrightarrow (x_1 \oplus x_2)) \wedge (z \leftrightarrow (x_1 \vee y))$$

Observe that x_3 is functionally determined by $\{x_1, x_2\}$, y is determined by $\{x_1, x_2\}$, and z is determined by $\{x_1,\ y\}$. With these dependencies in mind, here is one possible S and D-set: $S = \{x_1, x_2\} \subset \mathcal{P}$, and $D = \{x_1, x_2, x_3, y, z\} \supset \mathcal{P}$ This dual set configuration reduces the S-set by one variable while allowing a set one larger than $\mathcal{P}$ for decisions. This allows the SAT solver to be invoked earlier, while maintaining flexible branching choices during the model counting phase.

Given an initial projection set $\mathcal{P}$, we are interested in computing approximations of $D_{\max}$ and $S_{\min}$ such that $D_{\max} = \text{maximize}_{D \supseteq \mathcal{P}}\{|D|\}$ and $S_{\min} = \text{minimize}_{S \subseteq \mathcal{P}}\{|S|\}$. We aim for approximations only, since computing these can be computationally expensive, as they both rely on definability, which is known to be hard [21,33,12].

***S*-Set Minimization** We now turn our attention to the computation of minimal S-sets, which happens to coincide with the well-studied problem of independent set minimization. We demonstrate the non-confluence property using a formula F defined over the variable set $X = \{x_1, x_2, x_3\}$ with projection set $\mathcal{P} = X$:

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

Starting with $S = X$, let us examine variable removal sequences. Variable x_3 can be removed from S since it is functionally determined by assignments to x_1 and x_2. Similarly, x_2 can be removed since it is functionally determined by assignments to x_1 and x_3. However, attempting to remove both x_2 and x_3 simultaneously yields an invalid S-set $\{x_1\}$.

Thus, the minimal S-set obtained depends critically on the order of variable removal operations, as previously noted by Lagniez et al. [21]. We employ Arjun [33], a state of the art independent set minimization tool to efficiently compute our SAT-eligibility sets.

***D*-Set Maximization** A trivial D-set that is often (much) larger than the S-set is the projection set given by the instance, $\mathcal{P}$. In fact, for unprojected instances, this encompasses all variables. For projected instances however, it maybe possible to enlarge $\mathcal{P}$, as we explain below.

Algorithm 1 formalizes our approach to D-set maximization. The first phase performs syntactic expansion through gate-based analysis (identifying gates as per [33]), systematically identifying variables that could be part of the decision set (D-set) due to structural properties of the formula. This phase exploits the

Algorithm 1 Computing maximal decision set $D_{\max}$. SemanticExpansion is step-limited, aborting if it takes too many computing steps

Require: Formula F, projection set $\mathcal{P}$
Ensure: Maximal decision set $D_{\max}$ where $\mathcal{P} \subseteq D_{\max}$
$D \leftarrow \mathcal{P}$ ▷ Initialize with projection variables
$G \leftarrow$ ExtractGates(F) ▷ Extract OR, ITE, XOR gates
procedure SyntacticExpansion(V)
    changed $\leftarrow V$
    while changed is not empty **do**
        $v \leftarrow$ changed.*pop*()
        for gate $g \in G$ where v is input **do**
            if Inputs(g) $\subseteq D \wedge$ Output(g) $\notin D$ **then**
                $D \leftarrow D \cup \{$Output(g)$\}$
                changed.*append*(Output(g))
    return D
end procedure
procedure SemanticExpansion
    for $v \notin D$ **do**
        if ValidateDecisionVar($D \cup \{v\}, F$) **then**
            $D \leftarrow D \cup \{v\}$
            $D \leftarrow$ SyntacticExpansion($\{v\}$) ▷ Quick check with syntactic analysis
    return D
end procedure
$D \leftarrow$ SyntacticExpansion(P) ▷ Phase 1: Syntactic analysis
$D \leftarrow$ SemanticExpansion ▷ Phase 2: Semantic analysis
return D

observation that if all inputs to a logical gate (OR[‖], ITE, XOR) are in the D-set, its output variable can be added without requiring expensive semantic analysis. The algorithm iteratively applies this rule until reaching a fixed point, ensuring complete coverage of syntactically derivable additions.

The second phase deals with variables whose inclusion cannot be determined through purely syntactic means. Here, we employ semantic analysis through the ValidateDecisionVar routine, which verifies whether adding a variable to the D-set preserves model counting correctness. This is exactly the algorithm as presented in [21] except when it is definable, we include the variable, rather than exclude it, thus making the algorithm confluent. While computationally more expensive than syntactic analysis, this phase is necessary to identify more valid decision variables that may elude syntactic detection.

The separation into syntactic and semantic phases offers significant practical benefits. By exhaustively applying lightweight syntactic analysis before moving to costly semantic checks, the algorithm often achieves substantial decision set expansion while minimizing computational overhead.

[‖]Note that AND gates are OR gates, with all inputs and the output negated. See De Morgan's laws [23]. Since we deal with literals, OR gates are sufficient to extract.

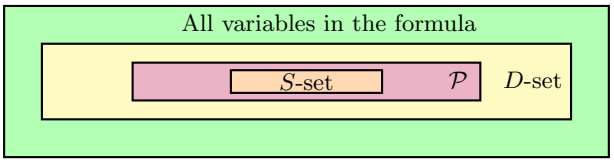

Fig. 1: Illustration of the relationship between the projection set $\mathcal{P}$, the decision set D-set, and the SAT-eligibility set S-set. Previous model counters have $D = S$. In our framework we first set $D = \mathcal{P}$, and then extend D to include more variables, if possible, using Algorithm 1.

4.3 Chronological Backtracking

Chronological backtracking [26] (i.e. ChronoBT) is a technique in SAT solving invented to mitigate the issue that SAT solvers would backtrack to decision level 0 whenever a unit clause is learnt. The requirement to backtrack to level 0 in case of a learned unit clause is due to the strict invariants imposed by classical, non-chronological CDCL architecture. In case of very large industrial instances, such as those that the inventors of ChronoBT were working on [25], this lead to a lot of wasted work: solver would go back to level 0, re-decide and re-propagate much of the current trail, soon find another unit clause, go back to level 0, etc.

Within the context of model counting, the idea of chronological backtracking serves a different purpose. Whenever a clause is learnt that would necessitate backtracking to a level lower than the one below, all current top-down model counters discard the learned clause and backtrack only one level, considering the branch to have zero solutions. While this avoids counting a branch that contains no solutions, it also means that the same fact is either re-learnt again, or never learnt. Hence, the counter may find itself attempting to repeatedly count parts of the space that contain no solutions—solutions that would already have been banned by the discarded learned clause. As shown in Table 2, our experiments demonstrate that the number of conflicts is significantly reduced when using ChronoBT: to count *more* formulas, we need to conflict on average *2.5x less*.

The algorithm that decides whether chronological backtracking is performed in [26] is shown in Algorithm 2. The key insight of this algorithm is that it only performs non-chronological backtracking when it is beneficial to do so. This is the case when the analyzer suggests backtracking more than a certain threshold of levels, which can lead to much wasted work. In in our framework, ChronoBT is used to avoid re-learning the same clause or re-counting already counted components, and is always on.

The adaptation of chronological backtracking to model counting introduces additional complexities related to component caching and solution counting that are not present in SAT solving. In weighted model counting, when a solver performs non-chronological backtracking, it must not only maintain the logical consistency of the search, but also ensure the correctness of weight computations. In particular, each partial assignment contributes to the final weight multiplica-

Algorithm 2 Deciding when to perform chronological backtracking

Input: backtrack level b, current decision level d, threshold t
Output: New decision level b
$h \leftarrow$ highest decision level in learnt clause except d
if only one literal from highest level in conflict clause **then**
    return $h - 1$ ▷ Chronological backtrack
if $d - b > t$ **then return** $h - 1$ ▷ Chronological backtrack
else return b ▷ Non-Chronological backtrack

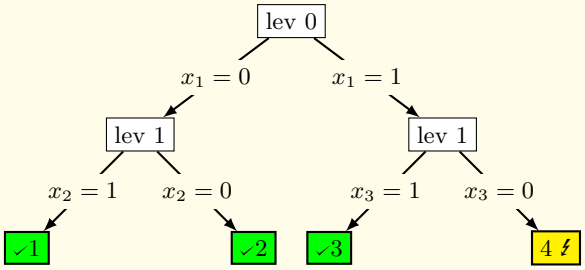

Fig. 2: In this example, we illustrate what happens when the system explores the left side of a graph, nodes 1 and 2. Then backtracks to level 0 and explores the right side, nodes 3 and 4. At node 4, the system learns the unit clause x_3. This unit clause's level is 0, but due to ChronoBT, we only backtrack to level 1. However, the system has already multiplied in the weight of x_3 into nodes 1, 2, and 3. These nodes' weights, which are all on the left side of an already explored branch (branches "lev 0" and "lev 1"), need to be compensated

tively. When chronological backtracking is employed, the solver must carefully track which weights need to be preserved and which should be discarded.

In our implementation, we took the approach that all components start with a weight of one, and only when a literal is unset, and the literal is part of the currently counted component, the literal's weight is multiplied into the current component's weight. In order to know if a variable is part of the current component, during decision analysis, we set $incomp[lev][var] = mark$ for all variables considered for decision (i.e. part of the D-set of the component), where $mark$ is a 64-bit number starting at 0 and incremented at each decision. This $mark$ is saved for each level at $marks[lev] = mark$. Hence, while unsetting the literal (i.e. while backtracking), it is a cheap check of $incomp[lev][var] == marks[lev]$ to decide if at decision level lev variable var was in the component. If it was, its weight is multiplied in. This approach is cheap and works well, counting all components' weights correctly as long as there is no chronological backtracking.

Weight Management During Counting. Chronological backtracking introduces significant complexities to the weight management system in model counting.

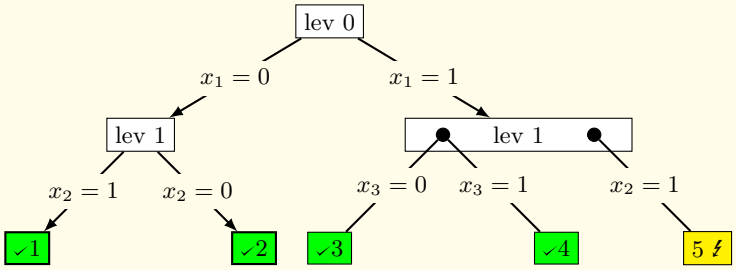

Fig. 3: In this example, x_4, which is part of components 1,2,3, and 4, but not part of component 5 is learned to be false at level 0. This sounds impossible, as x_4 is clearly not part of component 5 (since it is part of 3 and 4), so it should never be part of a learned clause while examining component 5. However, components are decided *purely* based on irredundant clauses. It is possible that learned clauses connect components. These can lead to contradictions over variables *not* part of the component currently examine, and hence a learned clause that implies a literal that's *not* in the component we are examining.

Consider the scenario illustrated in Figure 2. The counter first explores left, and then the right branches. The complications arise during right branch exploration, where a weighted literal previously set&unset during left branch exploration may be unset again at a higher decision level, if the variable is assigned at a lower decision level due to ChronoBT. When backtracking, the literal will be unset again, multiplying its weight twice into the left components. To maintain correct weight calculations, the system must compensate by dividing he left branches' current counts by the literal's weight, in order to avoid double-counting.

A second, more subtle complexity emerges from the interaction between learned clauses and component decomposition, illustrated in Figure 3. Component separation in state of the art model counters examine only the original formula clauses, not learned clauses. Consider variables x and y where y is not in the current component under examination. A learned clause of the form $\neg x \vee y$ may enable propagation of y, despite y not being part of the component. This creates an interesting scenario: had $\neg x \vee y$ been an original clause, x and y would necessarily belong in the same component. As demonstrated in Figure 3, this interaction allows learned clauses within the current component to force assignments to variables outside the component's scope. This can lead to learned clauses asserting literals outside the scope of the current component, attached at a higher level than the current component. When backtracking, this literal will be unset, and its weight multiplied in. However, since it is not part of the currently examined component, it may have *already* been multiplied in, when examining previous components. To account for this case, we need to check sibling components if the asserted, lower level literal is in them, and compensate accordingly, as in Algorithm 3.

Algorithm 3 Weight fixing for chronological backtracking. We use the remaining components to figure out if the variable has been counted by the already processed components. This is because sharpSAT, on which Ganak2 is based, only keeps detailed track of remaining components.

Executed: every time a literal is set
Inputs: literal to set: lit decision level to set: lit_lev current decision level: lev
Invariants: $lit\_lev \leq lev$
for $i \leftarrow lit\_lev$ **to** lev **do**
  $inside \leftarrow incomp[i][var] == marks[i]$
  if $\neg inside$ **then break** ▷ Cannot be in greater decision levels
  if $i > lit\_lev$ AND on right side of branch at decision level i **then**
    divide left side of branch at level i by $weight(lit)$
  $already\_counted \leftarrow true$
  $d \leftarrow$ decision node at level i
  for all remaining components $comp$ of d **do**
    for all variable v in $comp$ **do**
      if $v == lit.var$ **then** {$already\_counted \leftarrow false$; **break**}
  if $already\_counted$ **then** divide active branch at level i by $weight(lit)$

5 Experimental Evaluation

We implemented Ganak2 in C++[**], building upon the codebase of the original Ganak probabilistic exact model counter. Our implementation integrates Arjun [33] for preprocessing with Ganak's core architecture for exact model counting, with the improvements described in Section 4: enhanced SAT solving for residual formula processing, dual independent set framework with SAT-eligibility set (S-set) minimization and decision set (D-set) maximization, and chronological backtracking adapted to model counting.

The experimental evaluation was conducted on a cluster consisting of AMD EPYC 7713 CPUs, with each particular benchmark running on a single core with a memory limit of 9 GB and a time limit of one hour. For all other counters, we used a memory limit of 45 GB[††]. Both Arjun and Ganak2 uses the GNU MP infinite precision arithmetic library[‡‡] for weighted computations. While this is known to be often slower and more memory-hungry than high-precision floating-point arithmetic, it is exact. All other tools used high-precision floating-point arithmetic for weighted model counting, as default. To establish the correctness of Ganak2, it was extensively fuzzed via SharpVelvet[§§], a tool that generates random

[**]We plan to release Ganak2 as open-source software in case of paper acceptance

[††]Counters are optimized for the large amount of memory available during the Model Counting Competition (32GB). While Ganak2 can deal with smaller memory footprint, partially due to its probabilistic component caching scheme, we did not want to unfairly penalize other counters

[‡‡]`https://gmplib.org/`

[§§]`https://github.com/meelgroup/SharpVelvet`

(un)projected and (un)weighted benchmarks in the Model Counting Competition format, and compares the count across different model counters.

Our benchmark suite comprised of the Model Counting Competition [10] benchmark suite from 2023 and 2024, for all standard standard tracks: un/weighted and un/projected in all combinations. This makes up 2x4x200=1600 publicly accessible benchmarks[¶¶] ranging from small to large benchmarks with diverse structural characteristics, sourced from a broad range of application domains.

To rigorously evaluate Ganak2's performance, we conduct comprehensive comparisons against three leading state of the art model counters: SharpSAT-TD, gpmc, and d4. The selection of these tools is motivated by their distinct technical approaches and established performance profiles. SharpSAT-TD represents the current performance frontier in preprocessing [20] and introduced treewidth decomposition techniques to exact model counting [19]. The d4 counter has long been at the forefront of d-DNNF compilation-based model counting, and recently introduced a novel approach based on dynamic blocked clause elimination [22]. The inclusion of gpmc is motivated by its original idea of using a SAT solver to compute a solution to the residual formula [34], and its strong performance in previous competitions. All tools were obtained from the 2024 Model Counting Competition Zenodo repository[***]. and were run with their configurations as per their respective competition scripts. All tools have preprocessing systems similar to Arjun, integrated. Note that SharpSAT-TD only supports unprojected benchmarks, hence it has been left out of comparisons where projected benchmarks are considered. It is worth remarking that Ganak2, Ganak, and SharpSAT-TD rely on probabilistic component caching, and therefore are probabilistic exact counters. The value of δ is determined by the bit-width of the hash functions, which is set to 64, leading to a δ of 0.001.

The primary objectives of our experimental evaluation were twofold: (1) To systematically evaluate Ganak2's performance relative to other state of the art model counters, and (2) to quantitatively assess the runtime performance improvements achieved by Ganak2 compared to the baseline Ganak.

5.1 Comparison with Prior State of the Art

We conducted an extensive empirical evaluation to compare Ganak2 against state of the art model counters: Ganak, gpmc, d4, and SharpSAT-TD.

Figure 4 presents the cumulative distribution function (CDF) plots showing the number of instances solved within different time limits. For unprojected benchmarks (left plot), Ganak2 exhibits consistently superior performance, solving 550 instances compared to 523 instances by SharpSAT-TD, the next best performer. For projected benchmarks (right plot), where SharpSAT-TD is not applicable, the improvement is even more substantial, with Ganak2 solving 571

[¶¶] MCComp 2023 benchmarks at: `https://zenodo.org/records/10012864`
MCComp 2024 benchmarks at: `https://zenodo.org/records/14249068`

[***] `https://zenodo.org/records/14249109`

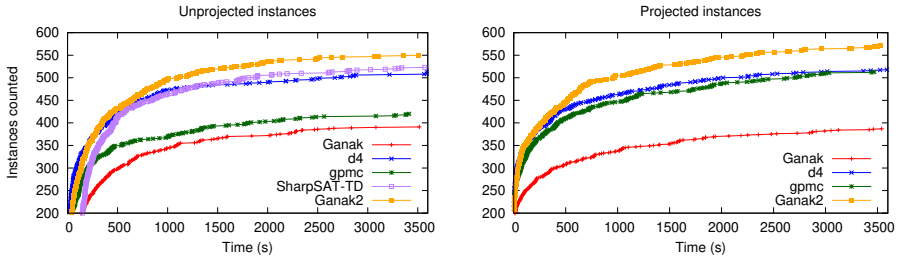

Fig. 4: Cumulative distribution function of Ganak2 against prior state of the art counters

instances compared to 518 by d4, representing a 10% improvement over the closest competitor.

Table 1 provides a detailed quantitative analysis of the performance metrics. We begin with results on all the instances. We see that Ganak2 is able to return counts for 1121 instances while d4 finishes only on 1032 instances, an increase of 89 instances. As model counting tools have matured over the years, such an increase in the number of instances counted would be considered significant. Furthermore, observe that the PAR2[†††] score for d4 is 1478 while Ganak2 achieves a PAR2 score of 1322. For unprojected benchmarks, Ganak2 achieves a PAR2 score of 1362, representing an 11% improvement over d4's score of 1496. The results are similarly impressive for projected benchmarks, where Ganak2 achieves a PAR2 score of 1283, significantly better than d4's 1461 and gpmc's 1519. The memory consumption pattern is consistent across both benchmark sets, with Ganak2 maintaining an approximately 25-50% lower memory footprint compared to other counters.

5.2 Analysis of Algorithmic Improvements

To understand the relative contribution of different algorithmic innovations in Ganak2, we conducted a comprehensive evaluation across multiple configurations. Table 2 presents a comprehensive analysis of Ganak2's performance across different configurations, where each row represents an incremental enhancement to the system's capabilities. The columns show the most relevant metrics for model counting performance: the number of benchmarks counted, the PAR2 score, average memory usage (Mem), the average number of components processed (Comps), and the average number of conflicts encountered (Confls).

Let's turn our attention to the dual independent set framework and its improvement, the decision set extension algorithm. The 1600 benchmarks had 2297 median number of variables, of which a median of 208 were projection variables.

[†††]The PAR2 score is a performance metric, the Penalized Average Runtime for each benchmark: if the benchmark is counted, the score is the runtime, and if a timeout occurs, a value of double the time limit (7200 in our case) is used.

(a) Unprojected benchmarks

Tool	Counted	PAR2	Avg Mem(MB)
Ganak	391	2030	1176
gpmc	420	1881	1506
d4	514	1496	1314
SharpSAT-TD	523	1530	1129
Ganak2	550	1362	869

(b) Projected benchmarks

Tool	Counted	PAR2	Avg Mem(MB)
Ganak	387	2027	1383
gpmc	513	1519	1588
d4	518	1461	1867
Ganak2	571	1283	925

Combined Results

Tool	Counted	PAR2	Avg Mem(MB)
Ganak	784	2022	1279
gpmc	933	1700	1151
d4	1032	1478	1591
Ganak2	1121	1322	898

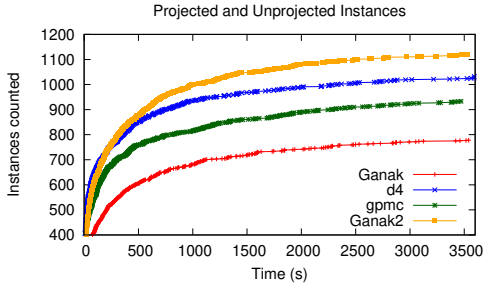

Table 1: Performance analysis of Ganak2 vis-a-vis other state of the art counters across different benchmark categories

After simplification algorithms such as backbone detection [2], equivalent literal substitution [14], and bounded variable elimination [7], the median number of variables was reduced to 410 by Arjun [33]. Further, Arjun computed a median S-set 89 for these CNFs. By allowing the now reduced projection set to be the D-set, this number could be increased to a median of 182. Once our maximal decision set algorithm (Algorithm 1) is enabled, this climbs to 208. This extension took an average of 0.26s for the SyntacticExpansion and 44.7s for the SemanticExpansion functions. They extended the projection set, whenever they could, by a median of 9 and 87 variables, respectively.

In Table 3, we present the performance of Ganak2 on a single benchmark, `149.cnf` from the Model Counting Competition 2023 Track 3 (projected model counting), with different configurations. The benchmark originally had 6629 variables and 19329 clauses, with a projection set size of 68. Arjun reduced this to 638 variables and 4110 clauses, meanwhile reducing the projection set size to 41. It then computed an SAT-eligibility (S-set) of size 39. Using our maximal decision set algorithm, we extended the decision set (D-set) size from 41 to 47. As we turn on our enhancements, the instance is solved significantly faster, with less memory usage, fewer components, and fewer conflicts.

The results demonstrate several key insights. First, using Chronological Backtracking and a SAT solver significantly improves performance while decreasing memory usage, decreasing the number of conflicts, and visited components.

Table 2: Ablation study of Ganak2 with improvements added incrementally, on all benchmarks. First row is baseline Ganak, last row is equivalent to Ganak2

Configuration	Counted	PAR2	Avg Mem(MB)	Avg Confls(M)	Avg Comps(M)
Ganak baseline	778	2029	1279	1.04	64.05
+ ChronoBT + SAT solver	906	1766	1059	0.42	56.94
+ SAT solver Enhancements	924	1738	1048	0.36	55.56
+ $D = \mathcal{P}$	1071	1418	900	0.28	45.72
+ D-set Max (=Ganak2)	1121	1322	898	0.28	45.41

Table 3: Benchmark `149.cnf` from the Model Counting Competition 2023 Track 3 (projected model counting) with different configurations of Ganak2

Configuration	Time (s)	Mem (MB)	Confls (K)	Decs (M)	Comps (K)	S-set	D-set
Ganak baseline	timeout	962	N/A	N/A	N/A	39	–
+ ChronoBT + SAT solver	1049.25	554	2063	112.21	5464	39	–
+ SAT solver Enhancements	821.05	550	1319	20.65	5315	39	–
+ $D = \mathcal{P}$	815.54	547	1214	20.53	5307	39	41
+ D-set Max (=Ganak2)	738.32	549	1230	20.90	5292	39	47

Furthermore, the dual independent set framework significantly enhances performance, leading to measurable reductions across all key metrics. Finally, extending the D-set yields additional performance improvements, though these gains are comparatively modest relative to the other enhancements. Comparing Ganak2 vis-a-vis the baseline, particularly interesting is the reduction in the number of components processed, dropping from an average of 64.05M in the baseline configuration to 45.41M for Ganak2. This substantial decrease suggests that enhancements in Ganak2 effectively guide the counter toward more efficient benchmark decomposition strategies.

6 Conclusion

In this paper, we presented Ganak2, a novel framework for model counting that achieves significant performance improvements thanks to three key algorithmic contributions. First, our dual independent set framework demonstrates that maintaining distinct SAT-eligibility and decision sets enables more efficient space exploration while preserving counting correctness. Second, our enhanced chronological backtracking mechanism, specifically adapted for model counting, addresses the computational challenges in both projected and unprojected scenarios through careful management of component caching and learned clauses. Finally, our refined SAT solver integration achieves seamless state transitions while incorporating advanced features from modern SAT solvers.

Our comprehensive empirical evaluation demonstrates that these technical innovations translate into substantial practical improvements. The ability to solve approximately 1100 instances within the timeout period, compared to 800 instances for baseline configurations, represents a significant advancement in the state of the art for exact model counting.

References

1. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proc. of CCS. pp. 1249–1264 (2019)
2. Biere, A., Froleyks, N., Wang, W.: Cadiback: Extracting backbones with cadical. In: Mahajan, M., Slivovsky, F. (eds.) SAT. LIPIcs, vol. 271, pp. 3:1–3:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.SAT.2023.3, https://doi.org/10.4230/LIPIcs.SAT.2023.3
3. Birnbaum, E., Lozinskii, E.L.: The good old Davis-Putnam procedure helps counting models. J. Artif. Intell. Res. **10**, 457–477 (1999). https://doi.org/10.1613/JAIR.601
4. Bodlaender, H.L.: Discovering treewidth. In: Vojtás, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 1–16. Springer (2005)
5. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: DPMC: Weighted model counting by dynamic programming on project-join trees. In: Simonis, H. (ed.) CP. LNCS, vol. 12333, pp. 211–230. Springer (2020)
6. Duenas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Proc. of AAAI (2017)
7. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Proc. of SAT. vol. 3569, pp. 61–75 (2005)
8. Eiers, W., Saha, S., Brennan, T., Bultan, T.: Subformula caching for model counting and quantitative program analysis. In: Proc. of ASE. pp. 453–464 (2019)
9. Fichte, J.K., Hecher, M., Hamiti, F.: The model counting competition 2020. Journal of Experimental Algorithmics (JEA) **26**, 1–26 (2021)
10. Fichte, J.K., Hecher, M., Hamiti, F.: The model counting competition 2020. ACM J. Exp. Algorithmics **26** (oct 2021). https://doi.org/10.1145/3459080
11. Fichte, J.K., Hecher, M., Zisser, M.: An improved GPU-based SAT model counter. In: Schiex, T., de Givry, S. (eds.) CP. pp. 491–509. Springer International Publishing, Cham (2019)
12. Fleury, M., Biere, A.: Mining definitions in kissat with kittens. Formal Methods Syst. Des. **60**(3), 381–404 (2022). https://doi.org/10.1007/S10703-023-00421-2, https://doi.org/10.1007/s10703-023-00421-2
13. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based boolean functional synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 402–421. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_22
14. Gelder, A.V.: Toward leaner binary-clause reasoning in a satisfiability solver. Ann. Math. Artif. Intell. **43**(1), 239–253 (2005). https://doi.org/10.1007/S10472-005-0433-5, https://doi.org/10.1007/s10472-005-0433-5
15. Hecher, M., Thier, P., Woltran, S.: Taming high treewidth with abstraction, nested dynamic programming, and database technology. In: Pulina, L., Seidl, M. (eds.) SAT. LNCS, vol. 12178, pp. 343–360. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_25

16. Huang, J.: The effect of restarts on the efficiency of clause learning. In: Veloso, M.M. (ed.) IJCAI 2007. pp. 2318–2323 (2007)
17. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Proc. of TACAS. LNCS, vol. 6015, pp. 129–144. Springer (2010)
18. Jr., R.J.B., Pehoushek, J.D.: Counting models using connected components. In: Kautz, H.A., Porter, B.W. (eds.) AAAI/IAAI. pp. 157–162. AAAI Press / The MIT Press (2000)
19. Korhonen, T., Järvisalo, M.: Integrating tree decompositions into decision heuristics of propositional model counters. In: CP (2021)
20. Korhonen, T., Järvisalo, M.: SharpSAT-TD in model counting competitions 2021-2023 (2023), https://arxiv.org/abs/2308.15819
21. Lagniez, J., Lonca, E., Marquis, P.: Improving model counting by leveraging definability. In: Proc. of IJCAI. pp. 751–757 (2016)
22. Lagniez, J., Marquis, P., Biere, A.: Dynamic blocked clause elimination for projected model counting. In: Chakraborty, S., Jiang, J.R. (eds.) SAT. LIPIcs, vol. 305, pp. 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/LIPICS.SAT.2024.21
23. Morgan, A.D.: Formal Logic. Taylor and Walton, London (1847)
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC 2001. pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
25. Nadel, A.: Introducing Intel(R) SAT solver. In: Proc. of SAT. pp. 8:1–8:23 (2022)
26. Nadel, A., Ryvchin, V.: Chronological backtracking. In: Proc. of SAT. pp. 111–121 (2018)
27. Padoa, A.: Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une theorie déductive quelconque. In: Bibliothèque du Congrès international de philosophie. vol. 3, pp. 309–365 (1901)
28. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer (2007). https://doi.org/10.1007/978-3-540-72788-0_28
29. Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects of tree-width. Journal of Algorithms **7**(3), 309–322 (1986)
30. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: Proc. of SAT (2004)
31. Sang, T., Beame, P., Kautz, H.A.: Heuristics for fast exact model counting. In: Bacchus, F., Walsh, T. (eds.) SAT. LNCS, vol. 3569, pp. 226–240. Springer (2005). https://doi.org/10.1007/11499107_17
32. Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: A scalable probabilistic exact model counter. In: Proc. of IJCAI. pp. 1169–1176 (2019)
33. Soos, M., Meel, K.S.: Arjun: an efficient independent support computation technique and its applications to counting and sampling. In: ICCAD. pp. 1–9 (2022)
34. Suzuki, R., Hashimoto, K., Sakai, M.: Improvement of projected model-counting solver with component decomposition using SAT solving in components. Tech. Rep. SIG-FPAI-103-B506, JSAI Technical Report (Mar 2017)
35. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) SAT. LNCS, vol. 4121, pp. 424–429. Springer (2006). https://doi.org/10.1007/11814948_38
36. Toda, S.: PP is as hard as the polynomial-time hierarchy. SIAM J. Comput. **20**(5), 865–877 (1991). https://doi.org/10.1137/0220053

37. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM Journal on Computing **8**(3), 410–421 (1979)