Proceedings of the 23rd International Conference on Principles of Knowledge Representation and Reasoning
Main Track

Efficient Incremental #SAT via Cross-Instance Knowledge Reuse

Uriya Bartal[1] , **Dror Fried**[1] , **Jean-Marie Lagniez**[2]
[1]The Open University of Israel, Raanana, Israel
[2]CRIL, U. Artois & CNRS, F-62300 Lens, France
bauriya2@post.openu.ac.il, dfried@openu.ac.il, lagniez@cril.fr

Abstract

Model counting (#SAT) is a fundamental yet #P-complete problem central to probabilistic reasoning. In this work, we address *incremental model counting*, where sequences of structurally similar formulas must be counted. We propose an approach that amortizes computation via a persistent caching mechanism, retaining component data across solver calls to avoid redundant search. Additionally, we investigate branching heuristics adapted for this setting. We focus on the problems of argumentation and soft core, for which incremental model counting is natural. Experiments demonstrate that our method improves performance compared to current model counters, highlighting the capability of structure-aware reuse in dynamic environments.

1 Introduction

Model counting (also known as #SAT) is a fundamental problem in constraint programming and automated reasoning, with key applications in artificial intelligence (Palacios et al. 2005; Domshlak and Hoffmann 2006; Izza et al. 2023), product configuration (Astesana, Cosserat, and Fargier 2010; Sundermann et al. 2024), and other domains. The goal is to compute or estimate the number of solutions that satisfy a given set of Boolean constraints. Model counting is an inherently challenging task (Valiant 1979), and thus presents significant scalability challenges (Fichte, Hecher, and Hamiti 2021). Nevertheless, the central role of model counting has motivated substantial research into the development of both exact and approximate model counting algorithms capable of handling increasingly large and complex instances. This ongoing progress is exemplified by recent model counting competitions (Fichte and Hecher 2025) that demonstrate the advancement of tools and provide rigorous benchmarks for state-of-the-art solvers. A wide range of (exact) model counters have been introduced and investigated, including search-based approaches such as `cachet` (Sang et al. 2004), `SharpSAT` (Thurley 2006), `SharpSAT-TD` (Korhonen and Järvisalo 2021), and `ganak` (Soos and Meel 2025), as well as compilation-based approaches like `C2D` (Darwiche 2004), `SDD` (Oztok and Darwiche 2015), `dSharp` (Muise et al. 2012), `eadt` (Koriche et al. 2013), and `d4` (Lagniez and Marquis 2017).

Despite this significant progress, current model counters typically treat each problem instance in isolation, necessitating recomputation from scratch even when the constraints are syntactically similar and only minor modifications are made to the constraint set. This approach can be highly inefficient in iterative design or optimization processes where models are refined incrementally. For example, an infrastructure planner optimizing a road network may adjust only a handful of roads at a time, but must still recompute the model count for the entire network after each change.

In this work, we present a model counting framework specifically engineered to exploit sequences of syntactically related problem instances. Specifically, we target a dynamic environment where the propositional theory evolves via arbitrary updates, including the addition or removal of clauses and modifications to the variable set. In this online setting, instances are processed incrementally, without the need for a lookahead, leveraging structural similarity to optimize performance. Crucially, our framework requires weaker assumptions than Incremental SAT (Nadel and Ryvchin 2012), accommodating broader structural modifications. We elaborate on these distinctions and the implications for solver efficiency in Section 2.

The core contribution of our work is a novel method for dynamically sharing computation knowledge across benchmarks. We focus on two forms of sharing: *cache sharing* and *branching heuristic sharing*. Achieving shared caching in this setting is non-trivial, which likely explains why, despite its clear potential, such an approach has not been realized previously. Classical caching techniques for model counting typically assume a static benchmark, where the underlying problem instance remains fixed. This assumption breaks down in our dynamic scenario, where formulas evolve across invocations. In particular, we show that effective cache reuse across different calls requires representing the entire formula in memory. Although this risks high memory consumption, our experiments show that cache reuse can yield orders-of-magnitude performance improvements over isolated strategies. Further, we introduce a *cache management strategy* utilizing quasi-canonical forms for symmetry detection. This significantly improves efficiency by identifying syntactically equivalent subproblems, as we demonstrate on selected benchmarks. Additionally, we explore the reuse of branching heuristics, which affects the choice of which variable to branch on, during the model counting process. We show that sharing the same tree de-

composition, that encapsulates variable ordering constraints, is especially effective on hard cases, in which subsequent formulas are obtained mostly by clause removal.

Our framework is implemented on top of *d4*, a state-of-the-art model counter, featuring advanced cache management techniques suitable for our purposes. We focus our evaluation on the problems of *dynamic argumentation* (Rotstein et al. 2010) and *soft cores computation* (Audemard et al. 2022) where syntactically similar formulas naturally arise and efficient handling is essential. We evaluate our tool on a suite of synthetic and real-world benchmarks, originating from (Järvisalo, Lehtonen, and Niskanen 2023; Fichte, Hecher, and Hamiti 2021; Lagniez and Lonca 2024). Our results show that our tool achieves performance improvements over current leading tools, highlighting its effectiveness on syntactically similar benchmarks.

Section 2 discusses related work, followed by preliminaries in Section 3. Section 4 formalizes the incremental framework. We present persistent caching in Section 5 and heuristic reuse in Section 6. Experiments are detailed in Section 7, followed by conclusions in Section 8. The readers are referred to the full version of the paper for further experimental results and elaboration on the benchmarks that we have used[1].

2 Related Work

We first distinguish our work from Incremental SAT. In the SAT domain, incrementality typically refers to scenarios where the formula remains static but some literals are fixed as assumptions. In practice, allowing certain literals to be set as assumptions often suffices for applications that require the addition or removal of clauses (Nadel 2010). This is achieved by adding an auxiliary literal to each clause, which is set to false to activate the clause and true to deactivate it. Through this mechanism, SAT solvers can retain learnt clauses and internal data structures across different queries, as the base formula itself does not change: only the set of assumed literals does. Consequently, clause-learning and variable activity information can be preserved and reused, thereby improving performance on closely related instances, as is standard in incremental SAT solving.

A similar principle applies to model counting: recent work (Lagniez and Marquis 2019) has shown that maintaining a persistent cache across multiple counting queries can offer significant performance benefits. This allows retaining learned clauses, caches, and variable weights, mirroring incremental SAT strategies to enable efficient information reuse across subproblems. However, this form of incrementality is fundamentally limited. It is only practical when the set of all possible formulas is known in advance and when auxiliary variables can be introduced to activate or deactivate parts of the formula. If future modifications to the formula cannot be predicted, it becomes infeasible to preserve and reuse information between different model counting calls.

A recent model counter for Pseudo-Boolean (PB) formulas that, like our work, targets the incremental setting is `PBCount2` (Yang and Meel 2025). There are, however, fundamental architectural distinctions between the two approaches. Most notably, it relies on compiling constraints into Algebraic Decision Diagrams (ADDs). Its incremental mechanism caches these individual constraint ADDs, which must be retrieved via a linear scan and re-merged to reflect updates in the formula. In contrast, our framework operates directly on the CNF structure using component caching. Rather than storing constraint fragments, we cache subformulas (components) alongside their computed model counts. This allows for immediate retrieval via hash-based lookups during search, bypassing the need for costly diagram reconstruction or merge operations. Furthermore, while `PBCount2` performs a polynomial-time retrieval step once per formula update, our approach employs a constant-time hashing check at every node in the search tree, offering a different trade-off geared towards deep search performance. Consistent with our methodology, `PBCount2` also disables preprocessing in incremental mode to preserve cache validity. We nevertheless include it in our discussion as it represents the primary existing framework for exact incremental counting, although its focus on PB constraints and ADDs differs from our CDCL-based CNF approach. A major difference in the experimental settings between our work and `PBCount2` is that they have a 3-step or 5-step incremental model counting presented in their evaluation section. Namely, each benchmark undergoes 3 or 5 incremental changes. On the contrary, our argumentation setting deals with 1000 steps, while our soft core setting has m steps, where m being the number of clauses in the input formula. Therefore, in our setting, although for the first 5 steps, `PBCount2` may outperform our tool, we handle the remaining many more steps, in which PBCount2 struggles.

Another related work worth mentioning in regards to our work is `Cara` (Illner 2025) which uses similar symmetry techniques in caching but for the purpose of knowledge compilation, rather than model counting.

Despite the sophistication of modern counters, the majority remain strictly static, discarding all learned state between invocations. Aside from `PBCount2`, no existing tool supports the reuse of learned subproblems across sequences of syntactically similar CNF formulas. This gap motivates our contribution: a dedicated incremental CNF counter that enables effective cross-instance cache sharing without the overhead of full knowledge compilation.

3 Background

Let $\mathcal{L}$ be a propositional language constructed from a finite set of propositional variables $\mathcal{P}$ and the standard logical connectives. A *literal* ℓ is either a propositional variable (e.g., x) or its negation ($\neg x$). Given a literal ℓ over a variable x, the *complementary literal* $\overline{\ell}$ is defined as $\overline{\ell} = \neg x$ if $\ell = x$, and $\overline{\ell} = x$ if $\ell = \neg x$. We denote the variable associated with a literal ℓ by $Var(\ell) = x$. A *term* is a conjunction of literals, and a *clause* is a disjunction of literals. Terms and clauses may, when convenient, be regarded as sets of literals. A CNF formula Φ is a conjunction of clauses and may also be treated as a set of clauses when appropriate. $Var(\Phi)$

[1] See full version in https://arxiv.org/abs/2605.00671 .

denotes the set of variables in Φ.

Example 1. *Consider the CNF formula Φ over variables $Var(\Phi) = \{x_1, \ldots, x_5\}$ with clauses: $\{x_1 \vee x_2 \vee x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3, x_4 \vee \neg x_1, x_5 \vee x_1, x_5 \vee x_2 \vee x_3, x_4 \vee \neg x_2 \vee \neg x_3\}$.*

An *interpretation* (or *world*) over $\mathcal{P}$, denoted by ω, is a mapping from $\mathcal{P}$ to $\{0, 1\}$. Interpretations are often represented as sets of literals, one per variable in $\mathcal{P}$, consisting of exactly those literals assigned the value 1 by ω. The set of all possible interpretations is denoted by $\mathcal{W}$. An interpretation ω is said to be a *model* of a formula $\Phi \in \mathcal{L}$ if it satisfies the formula under the standard truth-functional semantics. The *SAT problem* asks whether such a model exists. The set of all models of a formula Φ is denoted by $\text{mod}(\Phi)$ and is defined as $\text{mod}(\Phi) = \{\omega \in \mathcal{W} \mid \omega \models \Phi\}$. The symbol $\models$ denotes logical entailment, and $\equiv$ denotes logical equivalence. For any formulas $\Phi, \Psi \in \mathcal{L}$, we have: $\Phi \models \Psi$ if and only if $\text{mod}(\Phi) \subseteq \text{mod}(\Psi)$ and $\Phi \equiv \Psi$ if and only if $\text{mod}(\Phi) = \text{mod}(\Psi)$. The notation $\|\Phi\|$ denotes the number of models of Φ over the variables in $Var(\Phi)$. For example, the number of models of Φ, as defined in Example 1, is $\|\Phi\| = 10$. The problem of *model counting* determines the number of models of a Boolean formula Φ.

Since model counting is closely related to the SAT problem, many of the most effective strategies are built upon DPLL-based SAT solvers (Gomes, Sabharwal, and Selman 2021). These solvers are extended beyond satisfiability checking to compute the exact number of satisfying assignments. In particular, DPLL-based model counters recursively explore the search space by branching on variables and simplifying the formula at each decision point. Unlike SAT solvers that halt upon finding one solution, model counters must exhaustively enumerate all models, making the task significantly more challenging.

A standard optimization in DPLL-based counting is connected component decomposition. When the formula splits into variable-disjoint subformulas, their counts are computed separately and multiplied, drastically pruning the search space (Sang et al. 2004). Another crucial optimization is component caching: because identical subproblems can arise in different branches of the search tree, model counters store the counts of previously solved components in a cache (Sang et al. 2004; Thurley 2006; Lagniez and Marquis 2021). If the same component is encountered again, the solver can retrieve the cached result instead of recomputing it. Component decomposition and caching underpin state-of-the-art exact counters, enabling scalability on structured instances (Sang et al. 2004; Thurley 2006; Korhonen and Järvisalo 2021; Sharma et al. 2019; Lagniez and Marquis 2017).

In the heart of these optimizations, branching variable selection remains critical for performance. By exploiting structural properties, such as those derived from tree decompositions, heuristics can be guided to significantly reduce the effective search space (Korhonen and Järvisalo 2021).

Nevertheless, all the mechanisms employed by a model counter (such as component decomposition, caching, and branching heuristics) are intrinsically tied to the specific instance being solved. As a result, these structures and strategies cannot typically be reused directly when addressing a new problem instance. This paper challenges this situation. As we demonstrate in the following sections, it is possible, under certain conditions, to fully reuse the cache structure regardless of the subsequent formula.

4 The Incremental Model Counting Framework

In this work, we formalize the problem of incremental model counting as a dynamic process operating on an evolving propositional theory. Unlike the static setting, our framework maintains a current state that is modified by sequences of updates. Crucially, model counting is not necessarily performed after every atomic modification, but rather at specific checkpoints after a batch of operations has been applied. As such, the framework that we describe in the following subsection is a general framework that is intended to be flexible, as to capture the fact that the definition of *incremental* is somehow from the user's perspective. For example, in some benchmarks (e.g. soft core), *incremental* can mean changing a single clause, while in others (e.g. argumentation), *incremental* means much wider changes.

4.1 State and Operations

Let $\mathcal{U}$ be the universe of all propositional variables. We define the state of the system at time step t as a pair $\Phi_t = \langle \mathcal{V}_t, \mathcal{C}_t \rangle$, where $\mathcal{V}_t \subseteq \mathcal{U}$ is the current set of active variables and $\mathcal{C}_t$ is a set of clauses over $\mathcal{V}_t$. The model count of the state Φ_t is defined as the number of satisfying assignments of the CNF formula formed by $\mathcal{C}_t$ over the variables $\mathcal{V}_t$.

The transition between states, denoted $\Phi_{t+1} = \text{update}(\Phi_t, \delta)$, is driven by an update operation δ. We consider the following four atomic rules for the update operation:

- *Clause Addition* (add_clause(c)): Adds a new clause c to the theory: $\mathcal{V}_{t+1} = \mathcal{V}_t, \mathcal{C}_{t+1} = \mathcal{C}_t \cup \{c\}$. *Precondition:* $vars(c) \subseteq \mathcal{V}_t$.
- *Clause Removal* (rem_clause(c)): Removes an existing clause from the theory: $\mathcal{V}_{t+1} = \mathcal{V}_t, \mathcal{C}_{t+1} = \mathcal{C}_t \setminus \{c\}$.
- *Variable Addition* (add_var(v)): Introduces a new variable v into the scope: $\mathcal{V}_{t+1} = \mathcal{V}_t \cup \{v\}, \mathcal{C}_{t+1} = \mathcal{C}_t$. *Note:* Adding a variable without adding constraints essentially doubles the model count.
- *Variable Removal* (rem_var(v)): Removes a variable v from the scope: $\mathcal{V}_{t+1} = \mathcal{V}_t \setminus \{v\}, \quad \mathcal{C}_{t+1} = \mathcal{C}_t$. *Precondition:* The variable v must not appear in any clause in $\mathcal{C}_t$. Formally, $\forall c \in \mathcal{C}_t, v \notin vars(c)$. This ensures that we do not leave "dangling" literals in the clause database.

Problem Definition We define an *update batch*, denoted as Δ, as a finite sequence of atomic operations.

The *Incremental Model Counting* problem is defined as follows: Given an initial state $\Phi_0 = \langle \mathcal{V}_{init}, \mathcal{C}_{init} \rangle$ and a sequence of update batches $\mathbf{\Delta} = \langle \Delta_1, \Delta_2, \ldots, \Delta_k \rangle$, compute the sequence of counts:

$$\langle \|\Phi_1\|, \|\Phi_2\|, \ldots, \|\Phi_k\| \rangle$$

where Φ_i is the state resulting from applying the sequence of operations in Δ_i to Φ_{i-1}. The objective is to minimize the total computational time of the entire sequence of counts. We achieve this by leveraging information learned during the computation of $\Phi_{0\ldots i-1}$ to accelerate the counting of Φ_i.

5 Shared Caching in Model Counting

As detailed in the preliminaries, model counters employ search procedures that recursively decompose the input formula Σ into subformulas. To optimize this process, computed counts are stored in a cache: if a subformula Σ' is encountered again during the search, its model count is retrieved directly from the cache, thereby eliminating the need for redundant re-computation.

Our strategy persists the cache across benchmarks to enable subformula count reuse. While conceptually simple, this presents significant implementation challenges. Section 5.1 analyzes the limitations of standard caching mechanisms in this context, while Section 5.2 introduces a novel optimization to adapt the basic scheme for efficient reuse.

5.1 Cache in Model Counters

During the search, the solver encounters subformulas φ derived from Σ via conditioning. We define a *caching scheme* c as a mapping from each φ to a representation $r_c(\varphi)$, with the cache storing the map $r_c(\varphi) \mapsto \|\varphi\|$. A cache miss occurs if no matching representation is found. A scheme is *correct* if collision implies logical equivalence: $r_c(\varphi_1) = r_c(\varphi_2) \implies \varphi_1 \equiv \varphi_2$.

The performance of a caching scheme is determined by a trade-off between memory footprint and the granularity of equivalence detection. We review three standard approaches and analyze their applicability to our dynamic setting. In the basic scheme used by `cachet`, each subformula φ is represented explicitly as a string of literals, with clauses separated by zeroes. While robust, this representation is memory-intensive. To improve efficiency, `SharpSAT` employs a hybrid scheme. It assigns static indices to the clauses of the input formula Σ. A subformula is represented by a pair: the set of variables present and the sorted list of indices of the *original* clauses that belong to the subformula. Crucially, to save space, `SharpSAT` omits indices of binary clauses from this representation, relying on the static implication graph to handle them implicitly. `d4` adopts a structured explicit approach. It stores the clauses of φ directly but applies aggressive normalization: literals within clauses and the clauses themselves are sorted lexicographically to maximize cache hits. To optimize memory, `d4` does not store clauses that were originally binary in Σ, nor clauses that have not been shortened by the current assignment.

The optimized schemes of `SharpSAT` and `d4` fundamentally rely on the assumption that the input formula Σ is static. They achieve compression by treating binary clauses (or unshortened clauses) as implicit background knowledge that does not need to be part of the cache key.

In our incremental setting, however, where the formula evolves, this assumption leads to unsoundness. Consider two different formulas: $\Sigma_1 = \{x_1 \vee x_2\}$ and $\Sigma_2 = \{x_1 \vee x_2, \neg x_1 \vee \neg x_2\}$. Under the schemes of `SharpSAT` or `d4`, the cache representation of both Σ_1 and Σ_2 would be identical: a variable set $\{x_1, x_2\}$ with an empty clause list (since the clauses are binary and thus omitted). However, Σ_1 has 3 models while Σ_2 has 2. A cache entry created for Σ_1 would be incorrectly retrieved for Σ_2, yielding a wrong count.

Consequently, we cannot use these compressed representations. We must adopt a fully explicit caching scheme similar to `cachet`, which captures the exact semantic content of the subformula regardless of the global context. In the following section, we introduce optimizations to mitigate the memory overhead of this approach.

5.2 The Lazy Symmetry Caching

As discussed in the previous section, we decided to employ an explicit caching scheme to ensure correctness. To maximize efficiency, we adopt `d4`'s normalization: literals and clauses are lexicographically sorted, and duplicates removed. Crucially, our method goes beyond exact variable matching, detecting equivalent subproblems even when defined on disjoint variable sets. In the following, we introduce *symmetry caching* and detail our novel *lazy symmetry caching* mechanism, which exploits our explicit representation to identify these symmetries.

Symmetries are typically exploited via permutations: a permutation σ over $L_{\mathcal{P}}$, the set of all literals over $\mathcal{P}$, is a bijective mapping from $L_{\mathcal{P}} = \mathcal{P} \cup \{\neg x \mid x \in \mathcal{P}\}$ to itself. Any such permutation σ can be naturally extended to a homomorphism on propositional formulas, such that for every Boolean connective c of arity k: $\sigma(c(\alpha_1, \ldots, \alpha_k)) = c(\sigma(\alpha_1), \ldots, \sigma(\alpha_k))$. Throughout this paper, we assume that any permutation σ under consideration satisfies the following *stability condition*: for any pair of literals ℓ_1, ℓ_2, it holds that $\sigma(\ell_1) = \ell_2$ if and only if $\sigma(\overline{\ell_1}) = \overline{\ell_2}$.

Each permutation σ is represented in *simplified cycle notation*: as a product of cycles corresponding to its non-trivial orbits (i.e., those of length at least two). In this notation, for each orbit $\{\ell_1, \ldots, \ell_k\}$, only one of the two cycles $(\ell_1 \ldots \ell_k)$ or $(\overline{\ell_1} \ \ldots \ \overline{\ell_k})$ is shown. For example, if $L_{\mathcal{P}} = \{x_1, \ldots, x_6\}$, the permutation represented by $(x_1\ \neg x_3\ x_4)(x_5\ x_6)$ corresponds to mapping x_1 to $\neg x_3$, $\neg x_1$ to x_3, x_3 to $\neg x_4$, $\neg x_3$ to x_4, x_4 to x_1, $\neg x_4$ to $\neg x_1$, x_5 to x_6, $\neg x_5$ to $\neg x_6$, x_6 to x_5, and $\neg x_6$ to $\neg x_5$, with x_2 and $\neg x_2$ remaining fixed. The identity permutation is represented by the empty product in this notation.

Example 2 (Example 1 cont'd)**.** *Consider the formula Φ from Example 1. Suppose we evaluate the assignment $\{x_3\}$, resulting in the residual formula $\varphi_{x_3} = \{\neg x_1 \vee \neg x_2,\ x_4 \vee \neg x_1,\ x_5 \vee x_1,\ x_4 \vee \neg x_2\}$. Now, consider instead the assignment $\{\neg x_3\}$, for which the residual formula is $\varphi_{\neg x_3} = \{x_1 \vee x_2,\ x_4 \vee \neg x_1,\ x_5 \vee x_1,\ x_5 \vee x_2\}$. Let $\sigma = (x_1\ \neg x_1)(x_2\ \neg x_2)(x_4\ x_5)$ be the permutation that swaps x_1 with $\neg x_1$, x_2 with $\neg x_2$, and x_4 with x_5. Note that applying σ to Φ yields $\sigma(\varphi_{x_3}) = \varphi_{\neg x_3}$.*

As shown in (Bart et al. 2014), in the context of knowledge compilation, symmetry caching is a powerful technique that can significantly reduce the size of the representation. The authors propose a greedy method that searches

the cache for an entry φ_s equivalent to the current formula φ up to variable renaming, filtering candidates by size and literal counts. However, this approach is computationally expensive due to the cost of verifying renamings, and relies on generic hash properties that increase collision rates.

To overcome the limitations of symmetry caching over shared caching, we propose a lazy approach in which each formula inserted into the cache is stored by using a canonical representation. Rather than the standard lexicographic ordering, we employ a representation that incorporates structural information about the formula. We describe our method in steps. First, for each literal, we count the number of clauses in which it appears, and for each pair of literals, we sort them by their frequency of occurrence (increasing order). In the case of a tie, the negative literal is listed first.

Example 3 (Example 2 cont'd)**.** *Consider the two formulas computed in Example 2. For* φ_{x_3} *, the literal pairs (for each variable) are:* $(x_1, \neg x_1)$*,* $(x_2, \neg x_2)$*,* $(\neg x_4, x_4)$*, and* $(\neg x_5, x_5)$*. For* $\varphi_{\neg x_3}$ *, the corresponding pairs are:* $(\neg x_1, x_1)$*,* $(\neg x_2, x_2)$*,* $(\neg x_4, x_4)$*, and* $(\neg x_5, x_5)$*.*

We then sort these pairs (each associated with a variable) lexicographically according to the frequency of occurrence of each literal in the formula, with ties broken by lexicographical order. For example note that the encoding of $(\neg x_4, x_4)$ in φ_{x_3} is $(0, 2)$.

Example 4 (Example 3 cont'd)**.** *For* φ_{x_3}*, after sorting, we obtain the ordering:* $(\neg x_5, x_5)$*,* $(x_2, \neg x_2)$*,* $(\neg x_4, x_4)$*,* $(x_1, \neg x_1)$*. For* $\varphi_{\neg x_3}$ *, the sorted order is:* $(\neg x_4, x_4)$*,* $(\neg x_2, x_2)$*,* $(\neg x_5, x_5)$*,* $(\neg x_1, x_1)$*.*

Finally, for each pair, we select the first literal and use its position in the sorted ordering to construct the associated permutation σ. Specifically, for each variable, we consider the negative literal when associating positions in the permutation. In this way we normalize the formulas to have the same canonical ordering. For example in $\varphi_{\neg x_3}$, $(\neg x_4, x_4)$ is the *first* pair in the order so the permutation $\sigma_{\neg x_3}$ takes $\neg x_4$ to $\neg x_1$, which is the *first* (negated) literal. Similarly the permutation σ_{x_3} takes $\neg x_5$ to $\neg x_1$.

Example 5 (Example 4 cont'd)**.** *Continuing with the formulas from Example 4: For* φ_{x_3}*, we obtain* $\sigma_{x_3} = (\neg x_4\ \neg x_1)(\neg x_2\ \neg x_2)(\neg x_5\ \neg x_3)(\neg x_1\ \neg x_4)$*. For* $\varphi_{\neg x_3}$*, we obtain* $\sigma_{\neg x_3} = (\neg x_5\ \neg x_1)(x_2\ \neg x_2)(\neg x_4\ \neg x_3)(x_1\ \neg x_4)$*.*

Once the permutation associated with a formula is determined, we store the formula obtained by applying the permutation in the cache, rather than the original formula. Since applying a permutation to a formula preserves the number of satisfying assignments, we all in all get that the permuted formula is associated with the model count of the original, non-permuted formula.

Example 6 (Example 5 cont'd)**.** *Consider the two formulas from Example 2 along with their permutations computed in Example 5. We have* $\sigma_{\neg x_3}(\varphi_{x_3}) = \{x_4 \vee x_2,\ x_1 \vee \neg x_4,\ x_3 \vee x_4,\ x_3 \vee x_2\} = \sigma_{x_3}(\varphi_{\neg x_3})$*.*

As this example illustrates, lazy symmetry caching allows us to identify and reuse formulas with identical model counts, even if they are not syntactically equivalent. In this approach, the hash function operates on the permuted (canonicalized) version of the formula. Since we already store the full formula in the cache, this strategy incurs no additional storage cost, aside from the modest computational overhead of determining the appropriate permutation.

It is important to note that this caching strategy is only directly applicable to model counting, where permutations do not affect the count. In the context of weighted model counting, additional care must be taken to ensure that the weights are also preserved under the applied permutation.

6 Shared Heuristics

The decision of which variable to branch on next is a critical factor that significantly affects the practical performance of model counters. Numerous classical SAT branching heuristics have been proposed in the literature, such as MOM (Dubois et al. 1993), JWTS (Jeroslow and Wang 1990), and VSIDS (Moskewicz et al. 2001), along with heuristics specifically designed for model counting, including DLCS and VSADS (Sang, Beame, and Kautz 2005). However, the most effective heuristics often exploit structural properties of the given formula. In particular, the structure of a formula, captured by its primal graph, offers valuable information that can be leveraged to design more efficient decision heuristics. By utilizing tree decompositions (TDs) of the primal graph, a solver can exploit this inherent structure, thereby potentially reducing the effective search space and substantially improving the efficiency of model counting.

A *primal graph* of a CNF formula Φ is an undirected graph whose vertices are the variables of Φ, with edges between variables that appear together in at least one clause. A *tree decomposition* (Robertson and Seymour 1986; Bodlaender 2005) of a graph G is a tree T whose nodes, called *bags*, are subsets of vertices of G, and which collectively cover the vertices and edges of G while guaranteeing a certain connectivity property (see, e.g., (Robertson and Seymour 1986)). The *width* of a decomposition is the size of its largest bag minus one; the *treewidth* of G is the minimum such width.

Tree decomposition is fundamental in model counting, enabling dynamic programming algorithms with time complexity $\mathcal{O}(2^k \cdot n)$, where k is the treewidth. This makes the problem fixed-parameter tractable w.r.t. k, offering efficiency for low-treewidth formulas. However, since practical instances often have large treewidths, state-of-the-art counters use this structural information more flexibly. For instance, Korhonen and Järvisalo (Korhonen and Järvisalo 2021) show that *hybrid scores* (combining tree decomposition data with traditional heuristics) improve performance even when theoretical bounds are not strictly met.

This hybrid approach enables modern model counters to exploit the advantages of structural decomposition even for instances of high treewidth, by strategically prioritizing variables according to their structural relevance. In our setting, we consider the model counting problem over a repository of multiple, closely related Boolean formulas. Since computing a tree decomposition can be computationally expensive, performing this preprocessing step separately for each formula in the repository is generally impractical. Instead, we

propose computing the tree decomposition only once, during the initial iteration, and reusing it for all subsequent formulas in the repository. This approach amortizes the overhead of tree decomposition construction, reducing it from N computations (one per formula) to a single computation per repository, and thus yields significant efficiency gains by eliminating redundant work and enabling more consistent variable ordering across related instances.

The key insight is that when a formula is modified solely by clause removal, its existing tree decomposition remains a valid structural representation, even if it is no longer optimal. Moreover, as tree decompositions encode variable ordering constraints, integrating this structural information into the branching heuristic guides the solver to explore similar subformulas when analyzing related formulas. Empirical results, presented in the next section, confirm that this strategy can yield significant performance improvements.

For similar reasons, we opt for the DLCS heuristic rather than VSADS, the latter being generally more effective when solving a single, standalone benchmark. The DLCS heuristic assigns each variable a weight equal to the number of clauses in which it appears and selects the variable with the highest score for branching. In contrast, VSADS multiplies the DLCS score of a variable v by the number of times v participates in conflicts encountered during the search, thus incorporating conflict-driven information. Our intuition is that DLCS, being less influenced by conflict-driven clause learning, yields more stable and predictable branching decisions across syntactically similar formulas, thereby increasing the likelihood of positive cache hits. Experimental results support this hypothesis, showing that DLCS consistently outperforms VSADS in our setting.

7 Implementation and Evaluation

In this section, we present two evaluation settings. The first examines the benefits of sharing when counting the number of complete extensions in a dynamic argumentation framework (Dung 1993). The second setting addresses the computation of subset-minimal soft cores (Audemard et al. 2022). We describe each setting separately, explain how it fits into our incremental model counting framework, and show by evaluation how the re-use of knowledge across the various instances succeeds over methods in which each instance is a standalone.

7.1 Settings and Research Questions

All the experiments for both settings were carried out on a cluster equipped with dual 32-core Intel® Xeon® Gold 6130 CPUs (2.10GHz) and 32 GiB of memory, running Rocky Linux 8.7 (kernel version 4.18.0-425.13.1.el8_7.x86_64) and using the `gcc` 11.4.0 compiler. All code was implemented in C++. We enforce a timeout limit of 60 minutes per benchmark.

For our experiments, we developed a tool named `d4-dyn`[2], based on the state-of-the-art model counter `d4`[3]. `d4-dyn` implements the incremental framework formalized in Section 4, evolving the formula through step-by-step modifications while maintaining a persistent state, rather than solving a sequence of independent instances. Our tool supports three caching modes: *No-Sharing (*`NoShared`*)*, where each instance is solved independently with a local cache; *Shared-cache (*`Shared`*)*, where the cache persists across updates; and *Shared cache + Symmetry (*`Shared+Sym`*)*, which extends the shared mode by activating `d4-dyn`'s symmetry detection during cache operations. See Section 5 for details. Additionally, `d4-dyn` features an orthogonal `Tree Decomposition Sharing (TD)` mode, as explained in Section 6, which reuses the tree decomposition computed for the initial formula for all subsequent instances. Finally, the tool supports two branching heuristics, `VSADS` and `DLCS`, as detailed in Section 6.

For each setting our objective is to evaluate the following. First (RQ1), how does the deterministic `DLCS` branching heuristic perform compared to the `VSADS` branching heuristic, across all of our tool modes. Second (RQ2), which mode of `d4-dyn` with the best branching heuristic, is the most efficient. Finding that either of the shared modes is the most efficient, demonstrates the advantage of knowledge re-use for our problem settings. Finally (RQ3), we are also interested to learn how this best mode is compared against other state-of-the-art model counters. For that, We benchmark against two external solvers: `ganak` (Soos and Meel 2025), the winner of the 2025 Model Counting Competition,[4] and PBCount2 (Yang, Lai, and Meel 2025), which similarly to us, employs knowledge reuse across instances. These tools were executed using their default (or near-default) settings, following common practice.

7.2 Argumentation

Problem Definition To thoroughly evaluate our approach, we adopt as a test bed the problem of counting extensions within an Argumentation Framework (AF), focusing particularly on the complete semantic of dynamic argumentation, wherein the AF may evolve over time.

Abstract argumentation, introduced by Dung (Dung 1993), provides a foundational paradigm for formalizing argumentative reasoning in artificial intelligence. An *argumentation framework* consists of a set of abstract arguments together with binary attack relations among them. The principal computational task is to identify subsets of arguments, called *extensions*, which are deemed collectively acceptable according to different *semantics*. Formally, a set of arguments S constitutes a *complete extension* if: (i) S is conflict-free (no arguments in S attack each other); (ii) every argument in S is defended by S against attacks from outside; and (iii) S contains all arguments that it defends. Beyond classical acceptance tasks, the *counting* problem in abstract argumentation involves quantifying the number of extensions under a given semantics (Lagniez et al. 2021), offering insight into the diversity and nature of solutions. This is particularly valuable in scenarios such as preference-based decision making and probabilistic reasoning.

[2]https://zenodo.org/records/20008174

[3]https://github.com/crillab/d4v2

[4]https://mccompetition.org/

Dynamic argumentation examines how argument status shifts under modification, a critical concern in evolving multi-agent systems (Bistarelli et al. 2024). To systematically evaluate performance, we employ `crustabri` (crillab 2025) to generate dynamic Argumentation Frameworks (AFs) by iteratively applying random perturbations:

- *Delete argument* (10%): Remove an argument and its attacks.
- *Add argument with attacks* (20%): Add a new argument that attacks others.
- *Delete attacks* (40%): Remove random attack relations.
- *Add attacks* (30%): Add new attack relations between existing arguments.

Each resulting AF is translated into a CNF formula Γ using the standard encoding from (Besnard and Doutre 2004) and implemented in `crustabri`. We adapt our incremental framework terminology from Section 4. However, given the potentially extensive structural differences between consecutive AFs, we implement the transition from Φ_{i-1} to Φ_i as a full reset: the update batch clears the current state and inserts all clauses of Γ. Consequently, each initial AF benchmark yields a sequence $\langle \Phi_1, \ldots, \Phi_{1000} \rangle$ of CNF instances derived from the evolving framework.

For a dataset, we use 264 AFs from the 2023 Abstract Argumentation Competition(Järvisalo, Lehtonen, and Niskanen 2023). The AFs are described in further detail in the full version of our paper. We have used the `Cleaning Expectation` method of `d4` to prevent memory-outs.

Evaluation Details

Answering RQ1 To evaluate the impact of the branching heuristic on the effectiveness of our approach, we performed a comparison between DLCS and VSADS on all benchmarks and modes. `d4-dyn` with `DLCS` reaches a peak of 203 solved benchmarks, outperforming `VSADS`, which tops out at 172. Specifically, `DLCS` solves 187, 188, and 203 benchmarks across the `No Shared`, `Shared`, and `Shared+Sym` configurations, respectively, while `VSADS` solves 166, 172, and 171 in the same settings.

When analyzing the results, we observed that, on average, `DLCS` is approximately 15% faster than `VSADS` in our experimental setup. This outcome is consistent with our intuition: the deterministic nature of `DLCS` increases the likelihood of positive cache hits. Since caching relies on re-encountering similar substructures during the search, a deterministic traversal strategy such as `DLCS` promotes more consistent exploration patterns, thereby enhancing cache reuse and improving overall efficiency of `d4-dyn`. As such, we decided to continue with the DLCS heuristic for our remaining comparisons.

Answering RQ2 We ran `d4-dyn` with all three modes. Since the modifications between formulas are random, we observed that shared Tree Decomposition (TD), does not seem to scale, and indeed after preliminary tests we decided not to include TD in this comparison. This is likely because the tree decomposition computed for Φ_i becomes structurally unsuitable for Φ_{i+1} due to the significant changes in the underlying argumentation framework.

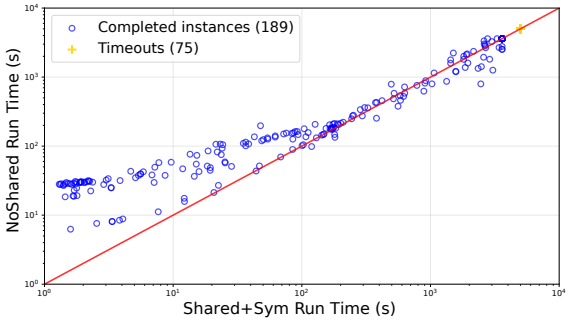

Figure 1: Comparing the runtimes (s) of `d4-dyn` (`NoShared`, `DLCS`) and `d4-dyn` (`Shared+Sym`, `DLCS`)

Moreover, we observed that `Shared` vs. `Shared+Sym` produces almost the same results (see below for more on that). As such, we focus on the comparison between `NoShared` and `Shared+Sym` configurations across all benchmarks, as can be seen in Figure 1. Each point in the plot corresponds to a sequence of formulas derived from a specific AF benchmarks. The y-axis shows the runtime of `d4-dyn` in the `NoShared` mode, while the x-axis shows the runtime in the `Shared+Sym` mode.

As can be seen, in most cases `Shared+Sym` yields a substantial performance gains, with a total improvement rate of 5000% on average (1200% median). We notice that this gain is particular for faster repositories. Specifically, improvement could vary between 20000% on faster benchmarks to almost even a minor slow down of a couple of percents on more complex or slower benchmarks. We suspect that this behavior arises because the benefits of caching diminish as benchmarks increase in complexity. In particular, the memory cost of maintaining the full formula context in the shared caching scheme can be prohibitive compared to the `NoShared` mode, which generally maintains a significantly smaller cache footprint.

For `Shared` vs `Shared+Sym`, the difference was minor and on average is 4% in favor of using symmetry. We suspect that the small gain of using symmetry is that the 1000 instances in each repository are similar, with the same variable names (since all benchmarks originate from the same source). In these cases, using symmetry that intends to exploit similar structures between benchmarks, differ only in the variables names, is mostly redundant.

Cache-hits Analysis To better understand the impact of shared caching, we analyze a specific AF benchmark: $af input\_exp\_cycles\_indvary2\_step1\_batch\_yyy01$. This benchmark comprises a sequence of a 1000 formulas, each with a small model count (2, 3, or 4). While the `NoShared` mode times out after solving 921 benchmarks, both shared caching modes successfully complete the entire sequence. Specifically, `Shared` and `Shared+Sym` achieve runtime improvements of 78% and 121%, respectively, over the `NoShared` baseline.

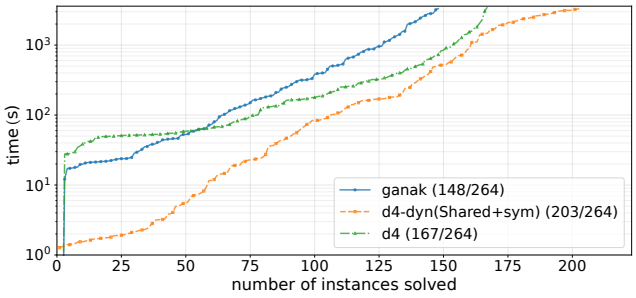

Figure 2: Argumentation - Model counter comparison. Evaluating the runtime of each method

Our observation is that in all three modes of operation, the first iteration takes roughly the same runtime. As the run progresses, however, time differences start to become evident in favor of the shared-cache modes. To understand the reason behind this behavior, we checked the total positive and negative cache hits (= cache hits / misses, resp.) for the entire run. Regarding positive hits, there are 0 positive cache hits during the entire run of the `NoShared` mode on this repository, opposed to 766 on the Shared mode and 764 on `Shared+Sym`. On the other hand, for negative hits, there are 71691 negative hits on the `NoShared` mode on this repository, opposed to only 19758 (`Shared`) and 17588 (`Shared+Sym`). This implies that the runtime of the shared modes should indeed become faster as less branching and recursion is done on the formulas in order to count their models.

Answering RQ3 We ran our tool against other state-of-the-art tools. For that we executed baseline model counters on each repository of argumentation frameworks using a simple `bash` script loop. Our evaluation comprises the following three experimental setups shown in Figure 2. We aimed to run the baseline tools with their default setting: (i) our tool `d4-dyn` (orange) on a Shared+Sym mode with DLCS branching heuristic; (ii) `d4` (Lagniez, Lonca, and Marquis 2016) (green) with *Glucose* as a SAT solver, flow cutter disabled and no preprocessing; (iii) `ganak` (blue) with default settings.

We also tried comparing with `PBCount2` but were limited to its Regular mode, as its Incremental mode is incompatible with our `crustabri`-based batch workflow. After translating CNF files to the required .opb format, PBCount2 (Regular) solved 50 of 264 benchmark families, well below the other tools and therefore is not shown in the plot.

Overall, `d4-dyn` solves 203 benchmarks, compared to 167 for `d4` and 148 with `ganak` . We must qualify these results by noting a structural advantage: `d4-dyn` operates as a persistent process, incurring initialization costs only once. In contrast, competing solvers are re-invoked for each formula, accumulating significant overhead from repeated process instantiation.

7.3 Soft Cores

Problem Definition The notion of a soft core was introduced in (Audemard et al. 2022) as a concept to explain why some system behaviors are exhibited while others are excluded. A *soft core* is a subset of constraints that is both small and highly constrained, meaning it admits only a limited number of models. Identifying a soft core is naturally a bi-criteria optimization problem, where both the size of the subset and the number of models are relevant. Depending on the context, one may prioritize minimizing soft core size or reducing model count. However, computing a minimal soft core is NP^{PP}-hard (Littman, Goldsmith, and Mundhenk 1998), making optimal solutions already infeasible for instances with more than 10 variables.

In our work, we consider a slightly modified variant, which searches for a minimal subset soft core. To achieve this, we employ a simple iterative strategy. For that recall that removal of clauses increases the model-count. Starting from the original formula, we do a single iteration over all clauses, in which we remove one clause at a time and check the new model count. If that model count is still below the threshold, we keep the index of that clause and continue. Otherwise, we return the clause to the formula and continue to the next clause. We return a set of the indices of the removed clauses.

We use `d4-dyn`'s add and remove clause framework to interactively remove clauses according to the soft core algorithm. We use a threshold of $\delta = 20$ percent for all our tests. An example for a typical run in this mode is giving a CNF formula as an input to `d4-dyn` for our baseline soft core application (described in RQ3). The tool would modify that CNF `#Clauses` times and run model counting for each s.t. Φ_1 is the original CNF and Φ_i is the modified CNF in the i'th iteration of the soft core algorithm. Finally, the algorithm outputs the set of clauses comprising the soft core.

We evaluate 3040 formulas across 6 families, including circuit, planning, and competition benchmarks (see the full version of our paper for further details).

A benchmark is considered solved only if all its constituent instances are processed without timeout. We excluded from our comparisons benchmarks that timed out on all the tools. The families are also classified into 5 runtime intervals, according to the runtime of solving Φ_1 with $d4$. This was done to better observe the difference between benchmarks in which already the first CNF instance is easier/harder to solve.

Evaluation Details

Answering RQ1 Consistent with the argumentation results, DLCS outperforms VSADS by comparable margins. Consequently, we employ DLCS as the default branching heuristic for the remainder of our evaluation.

Answering RQ2 Table 1 aggregates the results for `d4-dyn` across its three caching modes, evaluated both with and without shared tree decompositions (six configurations in total), plus the external tools `PBCount2` and `ganak` . To compare against `PBCount2`, we translated the CNFs of our experiments to OPB format that `PBCount2` accepts as input and created an input script of add and remove clause commands to input to `PBCount2`'s incremental mode, in order to emulate our soft core algorithm. Our experiments showed

Family	0–1s	1–10s	10–100s	100–1000s	1000–10000s
BMS (500)	–	S (223/242) G:0/242	S (238/254) G:0/254	SS (3/4) G:0/4	–
CBS (1000)	–	S (344/344) G:0/344	S (646/656) G:0/656	–	–
mcCompetition (113)	SS+TD (16/50) G:18/50	NS+TD (24/34) G:2/34	SS (10/27) G:2/27	SS+TD (1/1) G:0/1	SS+TD (1/1) G:0/1
or (434)	SS (189/193) G:0/193	SS (83/108) G:1/108	SS+TD (11/61) *G(42/61)	SS+TD (2/72) *G(70/72)	–
planning (616)	SS (133/199) G:2/199	SS (41/120) G:0/120	S+TD (35/124) G:1/124	NS+TD (21/83) G:1/83	S+TD (60/90) G:6/90
rnd3sat (377)	SS (234/284) G:0/284	SS (88/89) G:0/89	SS (4/4) G:0/4	–	–

Table 1: Best solver for each Soft Core benchmark family by runtime category
Legend: NS = NoShared, S = Shared, SS = Shared+Sym, +TD = with Tree Decomposition. G = `ganak` .
Tool (X/Y) indicates that Tool outperformed the others tools in X out of Y benchmarks in that cell.

that `PBCount2` timed-out on almost all benchmarks from the 0-1s and 1-10s interval categories, leading us to conclude that its usage of knowledge compilation based model counting may not be suitable for the purpose of our experimental setups. In order to evaluate our approach with `ganak` (in default mode with no extra parameters), we created a soft core tool in C++, based on our own soft core algorithm, that calls `ganak` as a black box for the model counting tasks. Since these system calls are being run in a loop sequentially, they cannot share cache or any other information with each other, as opposed to our tool.

Table 1 should be read as follows. Each line in the table describes a family of benchmarks with the number of benchmarks that completed the computation (i.e. did not time-out) for at least one tool. We checked in how many benchmarks each tool outperformed the others, in terms of runtime. Each cell in the table has in top (black), the `d4-dyn` mode that won the most benchmarks in its category (family + time interval), while the bottom (brown) shows the number of times that `ganak` outperformed the other tools. The cases in which `ganak` won overall more benchmarks than any mode of `d4-dyn` are marked with (*).

We first discuss the comparison of our own modes. Table 1 clearly demonstrates that `Shared+Sym` outperforms `NoShared` on easier benchmarks, solving more instances across families such as *BMS*, *CBS*, *rnd3sat*, *or*, and *planning* (within a 10-second runtime threshold). Similarly to AFs, the lazy symmetry mode added an extra minor boost to the runtime improvement, but the major boost came from using shared cache across benchmarks. On more challenging benchmarks, such as those from model counting competition and the harder intervals for *planning* and *or* families, Tree decomposition sharing becomes increasingly beneficial. For simpler benchmarks, the initial overhead of computing a tree decomposition may exceed the potential search time savings. However, for harder benchmarks this investment pays off; the TD-enabled modes successfully completed several runs where non-TD modes timed out. Among the TD configurations, `Shared` modes achieved improvements up to 50%, though they occasionally favored the `NS+TD` configuration in specific categories (see the full version of our paper. for detailed comparisons). Mirroring our AF results, these results suggest that while cache sharing is highly effective in easier settings, its utility diminishes as complexity increases, at which point shared structural information like tree decompositions becomes the primary driver of efficiency.

Answering RQ3 We next compare `d4-dyn` against `ganak`. As shown in Table 1, `d4-dyn` exhibits superior performance on easier benchmarks. This advantage stems from our integrated framework, which facilitates knowledge sharing and avoids the repeated process initialization overhead inherent in `ganak` 's baseline execution. On the harder benchmarks on which `ganak` is known to be better (Fichte and Hecher 2025) it shines, especially since, as discussed before, there is less utility in sharing on the harder benchmarks. Specifically, on harder benchmarks, `ganak`'s specialized search techniques allow it to excel, particularly as the marginal utility of cache sharing diminishes with increasing problem complexity. Notably, `ganak` exhibits exceptional performance on the *or* benchmarks, where its robust pre-processing is often sufficient to solve instances entirely without further search. This is further illustrated in Figure 3, which compares the runtime of `d4-dyn`'s optimal configuration against that of `ganak` across all benchmarks. These results corroborate Table 1, demonstrating that `d4-dyn` outperforms `ganak` on the majority of benchmark families, with the *or* benchmarks remaining a notable excep-

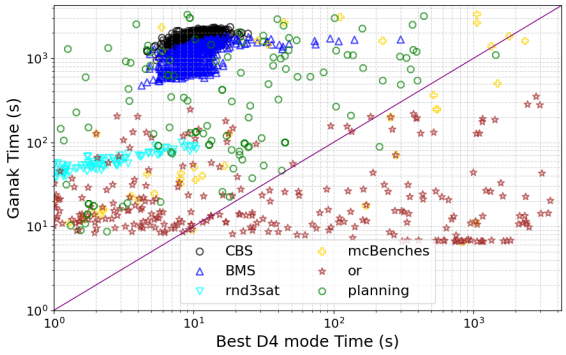

Figure 3: Runtimes of `d4-dyn` best mode (per benchmark) vs `ganak`

tion, where `ganak` 's efficiency is most pronounced.

All in all, our results demonstrate the consistent advantage of knowledge reuse across nearly all benchmarks. While shared caching and symmetry often suffice to improve performance, the reuse of tree decompositions proves particularly effective for the most challenging benchmarks.

8 Conclusion

In this work, we tailored an incremental model counter, `d4-dyn`, that rather than treating each benchmark independently, re-uses both cache information and branching-heuristic knowledge between benchmarks to exploit their similarities. We evaluated our approach on dynamic argumentation benchmarks, where syntactic similarity is crucial, for which we showed how our sharing techniques improve over existing solvers. We also considered the computation of minimal subset soft cores, showing that our framework can leverage cache and heuristic reuse to improve efficiency in this setting, and outperforms existing tools using benchmarks taken from competitions. Our results highlight the benefits of cache reuse within our benchmark sequences. To the best of our knowledge, this is the first work to exploit cache and heuristic reuse across multiple benchmarks in DPLL-based model counting. As this paper introduces a novel setting, it serves as a foundational first step. In both problems that we chose, model counting and the notion of incrementality is natural. Showing that our framework can handle further problems is a promising future work that we intend on pursuing.

Acknowledgements

We thank Mate Soos for his insights during the early stages of this work. We also thank the reviewers for their insightful comments. This work has been partly supported by the CERADOC project of the French National Agency for Research (ANR-25-CE23-3078), and by The Open University of Israel Research Fund 47372.

AI Declaration

Generative AI was used for language polishing and experimental script development. All scientific results, proofs, artifacts, and conclusions were independently verified by the authors, who assume full responsibility for the final content of the paper.

References

Astesana, J.; Cosserat, L.; and Fargier, H. 2010. Constraint-based Vehicle Configuration: A Case Study. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI*.

Audemard, G.; Lagniez, J.; Miceli, M.; and Roussel, O. 2022. Identifying Soft Cores in Propositional Formulæ. In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence, ICAART*.

Bart, A.; Koriche, F.; Lagniez, J.; and Marquis, P. 2014. Symmetry-Driven Decision Diagrams for Knowledge Compilation. In *ECAI - 21st European Conference on Artificial Intelligence - Including Prestigious Applications of Intelligent Systems (PAIS)*.

Besnard, P., and Doutre, S. 2004. Checking the Acceptability of a Set of Arguments. In Delgrande, J. P., and Schaub, T., eds., *10th International Workshop on Non-Monotonic Reasoning (NMR)*.

Bistarelli, S.; Kotthoff, L.; Lagniez, J.-M.; Lonca, E.; Mailly, J.-G.; Rossit, J.; Santini, F.; and Taticchi, C. 2024. The Third and Fourth International Competitions on Computational Models of Argumentation: Design, Results and Analysis. *Argument & Computation*.

Bodlaender, H. L. 2005. Discovering Treewidth. In Vojtás, P.; Bieliková, M.; Charron-Bost, B.; and Sýkora, O., eds., *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science*.

crillab. 2025. crustabri. https://github.com/crillab/crustabri.

Darwiche, A. 2004. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI*.

Domshlak, C., and Hoffmann, J. 2006. Fast Probabilistic Planning through Weighted Model Counting. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS*.

Dubois, O.; André, P.; Boufkhad, Y.; and Carlier, J. 1993. SAT versus UNSAT. In Johnson, D. S., and Trick, M. A., eds., *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science.

Dung, P. M. 1993. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning and Logic Programming. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence.*

Fichte, J. K., and Hecher, M. 2025. The Model Counting Competitions 2021-2023. *CoRR*.

Fichte, J. K.; Hecher, M.; and Hamiti, F. 2021. The Model Counting Competition 2020. *ACM J. Exp. Algorithmics*.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2021. Model Counting. In *Handbook of Satisfiability - Second Edition*.

Illner, P. 2025. New Compilation Languages Based on Restricted Weak Decomposability. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Izza, Y.; Huang, X.; Ignatiev, A.; Narodytska, N.; Cooper, M. C.; and Marques-Silva, J. 2023. On Computing Probabilistic Abductive Explanations. *Int. J. Approx. Reason.*

Järvisalo, M.; Lehtonen, T.; and Niskanen, A. 2023. International Competition on Computational Models of Argumentation.

Jeroslow, R. G., and Wang, J. 1990. Solving Propositional Satisfiability Problems. *Ann. Math. Artif. Intell.*

Korhonen, T., and Järvisalo, M. 2021. Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters (Short Paper). In *27th International Con-*

ference on Principles and Practice of Constraint Programming, CP.

Koriche, F.; Lagniez, J.; Marquis, P.; and Thomas, S. 2013. Knowledge Compilation for Model Counting: Affine Decision Trees. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI*.

Lagniez, J., and Lonca, E. 2024. Leveraging Decision-DNNF Compilation for Enumerating Disjoint Partial Models. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR*.

Lagniez, J., and Marquis, P. 2017. An Improved Decision-DNNF Compiler. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI*.

Lagniez, J., and Marquis, P. 2019. A Recursive Algorithm for Projected Model Counting. In *The Thirty-Third Conference on Artificial Intelligence, AAAI*.

Lagniez, J., and Marquis, P. 2021. About Caching in D4 2.0.

Lagniez, J.; Lonca, E.; Mailly, J.-G.; and Rossit, J. 2021. The Fourth International Competition on Computational Models of Argumentation.

Lagniez, J.; Lonca, E.; and Marquis, P. 2016. Improving Model Counting by Leveraging Definability. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI*.

Littman, M. L.; Goldsmith, J.; and Mundhenk, M. 1998. The Computational Complexity of Probabilistic Planning. *J. Artif. Intell. Res.*

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference*.

Muise, C. J.; McIlraith, S. A.; Beck, J. C.; and Hsu, E. I. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT. In *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, AI*.

Nadel, A., and Ryvchin, V. 2012. Efficient SAT Solving under Assumptions. In *Theory and Applications of Satisfiability Testing, SAT*.

Nadel, A. 2010. Boosting Minimal Unsatisfiable Core Extraction. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD*.

Oztok, U., and Darwiche, A. 2015. A Top-Down Compiler for Sentential Decision Diagrams. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI*.

Palacios, H.; Bonet, B.; Darwiche, A.; and Geffner, H. 2005. Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling, ICAPS*.

Robertson, N., and Seymour, P. D. 1986. Graph Minors. II. Algorithmic Aspects of Tree-Width.

Rotstein, N. D.; Moguillansky, M. O.; García, A. J.; and Simari, G. R. 2010. A Dynamic Argumentation Framework. In *Computational Models of Argument, COMMA*.

Sang, T.; Beame, P.; and Kautz, H. A. 2005. Heuristics for Fast Exact Model Counting. In Bacchus, F., and Walsh, T., eds., *Theory and Applications of Satisfiability Testing, 8th International Conference*.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. In *The Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT*.

Sharma, S.; Roy, S.; Soos, M.; and Meel, K. S. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI*.

Soos, M., and Meel, K. S. 2025. Engineering an Efficient Probabilistic Exact Model Counter. In Piskac, R., and Rakamaric, Z., eds., *Computer Aided Verification - 37th International Conference, CAV 2025*.

Sundermann, C.; Raab, H.; Heß, T.; Thüm, T.; and Schaefer, I. 2024. Reusing d-DNNFs for Efficient Feature-Model Counting. *ACM Trans. Softw. Eng. Methodol.*

Thurley, M. 2006. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *Theory and Applications of Satisfiability Testing, SAT*.

Valiant, L. G. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.*

Yang, S., and Meel, K. S. 2025. Towards Projected and Incremental Pseudo-Boolean Model Counting. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Yang, S.; Lai, Y.; and Meel, K. S. 2025. On Top-Down Pseudo-Boolean Model Counting. In Berg, J., and Nordström, J., eds., *28th International Conference on Theory and Applications of Satisfiability Testing, SAT*.