

Solving Projected Model Counting by Utilizing Treewidth and its Limits

Johannes K. Fichte^a, Markus Hecher^{b,*}, Michael Morak^d, Patrick Thier^c, Stefan Woltran^c

^a*AIICS, IDA, Linköping University,
581 83 Linköping, Sweden*

^b*Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology,
32 Vassar St., Cambridge, MA, United States*

^c*Database and Artificial Intelligence Group, TU Wien,
Favoritenstrasse 9-11, 1040 Vienna, Austria*

^d*Department of Artificial Intelligence and Cybersecurity, University of Klagenfurt,
Universitätsstraße 65-67, 9020 Klagenfurt am Wörthersee, Austria*

Abstract

In this paper, we introduce a novel algorithm to solve *projected model counting* (PMC). PMC asks to count solutions of a Boolean formula with respect to a given set of *projection variables*, where multiple solutions that are identical when restricted to the projection variables count as only one solution. Inspired by the observation that the so-called “treewidth” is one of the most prominent structural parameters, our algorithm utilizes small treewidth of the primal graph of the input instance. More precisely, it runs in time $\mathcal{O}(2^{k+4}n^2)$ where k is the treewidth and n is the input size of the instance. In other words, we obtain that the problem PMC is fixed-parameter tractable when parameterized by treewidth. Further, we take the exponential time hypothesis (ETH) into consideration and establish lower bounds of bounded treewidth algorithms for PMC, yielding asymptotically tight runtime bounds of our algorithm.

While the algorithm above serves as a first theoretical upper bound and although it might be quite appealing for small values of k , unsurprisingly a naive implementation adhering to this runtime bound suffers already from instances of relatively small width. Therefore, we turn our attention to several measures in order to resolve this issue towards exploiting treewidth in practice: We present a technique called nested dynamic programming, where different levels of abstractions of the primal graph are used to (recursively) compute and refine tree decompositions of a given instance. Further, we integrate the concept of hybrid solving, where subproblems hidden by the abstraction are solved by classical

*Corresponding author.

Email addresses: johannes.klaus.fichte@liu.se (Johannes K. Fichte), hecher@mit.edu (Markus Hecher), michael.morak@aau.at (Michael Morak), thier@tuwien.ac.at (Patrick Thier), woltran@dbai.tuwien.ac.at (Stefan Woltran)

search-based solvers, which leads to an interleaving of parameterized and classical solving. Finally, we provide a nested dynamic programming algorithm and an implementation that relies on database technology for PMC and a prominent special case of PMC, namely model counting ($\#SAT$). Experiments indicate that the advancements are promising, allowing us to solve instances of treewidth upper bounds beyond 200.

Keywords: tree decompositions, high treewidth, lower bounds, exponential time hypothesis, graph problems, Boolean logic, counting, projected model counting, nested dynamic programming, hybrid solving, parameterized algorithms, parameterized complexity, computational complexity, database management systems

2010 MSC: 05C05, 05C83, 03B05, 03B70

1. Introduction

A problem that has been used to solve a large variety of real-world questions is the *model counting problem* ($\#SAT$) [1, 2, 3, 4, 5, 6, 7, 8, 9]. It asks to compute the number of solutions of a Boolean formula [10] and is theoretically of high worst-case complexity ($\#P$ -complete [11, 12]). Lately, both $\#SAT$ and its approximate version have received renewed attention in theory and practice [13, 4, 14, 15]. A concept that allows very natural abstractions of data and query results is projection. Projection has wide applications in databases [16] and declarative problem modeling. The problem *projected model counting* (PMC) asks to count solutions of a Boolean formula with respect to a given set of *projection variables*, where multiple solutions that are identical when restricted to the projection variables count as only one solution. If all variables of the formula are projection variables, then PMC is the $\#SAT$ problem and if there are no projection variables then it is simply the SAT problem. Projected variables allow for solving problems where one needs to introduce auxiliary variables, in particular, if these variables are functionally independent of the variables of interest, in the problem encoding, e.g., [17, 18]. Projected model counting is a fundamental problem in artificial intelligence and was also subject to a dedicated track in the first model counting competition [19]. It turns out that there are plenty of use cases and applications for PMC, ranging from a variety of real-world questions in modern society, artificial intelligence [20], reliability estimation [4] and combinatorics [21]. Variants of this problem are relevant to problems in probabilistic and quantitative reasoning, e.g., [2, 3, 9] and Bayesian reasoning [8]. This work also inspired follow-up work, as extensions of projected model counting as well as generalizations for logic programming and quantified Boolean formulas have been presented recently, e.g., [22, 23, 24].

When we consider the computational complexity of PMC it turns out that under standard assumptions the problem is even harder than $\#SAT$, more precisely, complete for the class $\#NP$ [25]. Even though there is a PMC solver [21] and an ASP solver that implements projected enumeration [26], PMC has

received very little attention in parameterized algorithmics so far. Parameterized algorithms [27, 28, 29, 30] tackle computationally hard problems by directly exploiting certain structural properties (parameter) of the input instance to solve the problem faster, preferably in polynomial-time for a fixed parameter value. In this paper, we consider the treewidth of graphs associated with the given input formula as parameter, namely the primal graph [31]. Roughly speaking, small *treewidth* of a graph measures its tree-likeness and sparsity. Treewidth is defined in terms of *tree decompositions (TDs)*, which are arrangements of graphs into trees. When we take advantage of small treewidth, we usually take a TD and evaluate the considered problem in parts, via *dynamic programming (DP)* on the TD. This dynamic programming technique utilizes tree decompositions, where a tree decomposition is traversed in post-order, i.e., from the leaves towards the root, and thereby for each node of the TD tables are computed such that a problem is solved by cracking smaller (partial) problems.

In this work we apply tree decompositions for projected model counting and study precise *runtime dependency on treewidth*. While there are also related works on properties for efficient counting algorithms, e.g., [32, 33, 34], even for treewidth, precise runtime dependency for projected model counting has been left open. We design a novel algorithm that runs in *double exponential time*¹ in the treewidth, but it is quadratic in the number of variables of a given formula. Later, we also establish a conditional lower bound showing that under reasonable assumptions it is quite *unlikely that one can significantly improve* this algorithm.

Naturally, it is expected that our proposed PMC algorithm can be only competitive for instances where the treewidth is very low. Still, despite our new theoretical result, it turns out that in practice there is a way to efficiently implement dynamic programming and tree decompositions for solving PMC. However, most of the existing systems based on dynamic programming guided along a tree decomposition are suffering from maintaining large tables, since the size of these tables (and thus the computational efforts required) are bounded by a function in the treewidth of the instance. Although dedicated competitions [35] for treewidth advanced the state-of-the-art for efficiently computing treewidth and TDs [36, 37], these systems and approaches reach their limits when instances have higher treewidth. Indeed, such approaches based on dynamic programming reach their limits when instances have higher treewidth; a situation which can even occur in structured real-world instances [38]. Nevertheless in the area of Boolean satisfiability, this approach proved to be successful for counting problems, such as, e.g., (weighted) model counting [39, 40, 31].

To further increase the practical applicability of dynamic programming for PMC, novel techniques are required, where we rely on certain simplifications of a graph, which we call *abstraction*². Thereby, we (a) rely on different *levels*

¹Runtimes that are double exponential in the treewidth indicates expressions of the form $2^{2^{\mathcal{O}(k)}} \cdot \text{poly}(n)$, where n indicates the number of variables of a given formula and k refers to the treewidth of its primal graph.

²A formal account on these abstractions will be given in Definition 12.

of *abstraction* of the instance at hand; (b) *treat subproblems* originating in the abstraction by standard solvers whenever widths appear too high; and (c) use highly *sophisticated data management* in order to store and process tables obtained by dynamic programming.

Contributions. In more details, we provide the following contributions.

1. We introduce a novel algorithm to *solve projected model counting* in time $\mathcal{O}(2^{2^{k+4}} n^2)$ where k is the treewidth of the primal graph of the instance and n is the size of the input instance. Similar to recent DP algorithms for problems on the second level of the polynomial hierarchy [41], our algorithm traverses the given tree decomposition multiple times (multi-pass). In the first traversal, we run a dynamic programming algorithm on tree decompositions to solve SAT [31]. In a second traversal, we construct equivalence classes on top of the previous computation to obtain model counts with respect to the projection variables by exploiting combinatorial properties of intersections.
2. Then, we establish that our *runtime bounds are asymptotically tight under the exponential time hypothesis (ETH)* [42] using a recent result by Lampis and Mitsou [43], who established lower bounds for the problem $\exists\forall$ -SAT assuming ETH. Intuitively, ETH states a complexity theoretical lower bound on how fast satisfiability problems can be solved. More precisely, one *cannot* solve 3-SAT in time $2^{s \cdot n} \cdot n^{\mathcal{O}(1)}$ for some $s > 0$ and number n of variables.
3. Finally, we also provide an implementation for PMC that efficiently utilizes treewidth and is highly competitive with state-of-the-art solvers. In more details, we treat above aspects (a), (b), and (c) as follows.
 - (a) To tame the beast of high treewidth, we propose *nested dynamic programming*, where only parts of some abstraction of a graph are decomposed. Then, each TD node also needs to solve a *subproblem* residing in the graph, but may involve vertices outside the abstraction. In turn, for solving such subproblems, the idea of nested DP is to subsequently repeat decomposing and solving more fine-grained graph abstractions in a nested fashion. While candidates for obtaining such abstractions often naturally originate from the problem PMC, nested DP may require computing those during nesting, for which we even present a generic solution.
 - (b) To further improve the capability of handling high treewidth, we show how to apply nested DP in the context of *hybrid solving*, where established, standard solvers (e.g., SAT solvers) and caching are incorporated in nested DP such that the best of two worlds are combined. Thereby, we solve counting problems like PMC, where we apply DP to parts of the problem instance that are *subject to counting*, while depending on the existence of a solution for certain subproblems. Those subproblems that are *subject to searching* for the

existence of a solution reside in the abstraction only and are solved via standard solvers.

- (c) We implemented a system based on a recently published tool [39] for using database management systems (DBMS) to efficiently perform table manipulation operations needed during DP. Our system is called **nestHDB**³ and uses and significantly extends this tool in order to perform hybrid solving, thereby combining nested DP and standard solvers. As a result, we use DBMS for efficiently implementing the handling of tables needed by nested DP. Preliminary experiments indicate that nested DP with hybrid solving can be fruitful, where we are capable of solving instances, whose treewidth upper bounds are beyond 200.

This paper combines research of work that is published at the 21st International Conference on Satisfiability (SAT 2018) [44] and research that was presented at the 23rd International Conference on Satisfiability (SAT 2020) [45]. In addition to these conference versions, we added detailed proofs, further examples, and significantly improved the presentation throughout the document.

2. Preliminaries

We assume familiarity with basic notions from set theory and on sequences. We write a sequence consisting of ℓ elements e_i for $1 \leq i \leq \ell$ in angular brackets, i.e., $\langle e_1, e_2, \dots, e_\ell \rangle$. For a set X , let 2^X be the *power set of X* consisting of all subsets Y with $\emptyset \subseteq Y \subseteq X$. Recall the well-known combinatorial inclusion-exclusion principle [46], which states that for two finite sets A and B it is true that $|A \cup B| = |A| + |B| - |A \cap B|$. Later, we need a generalized version for arbitrary many sets. Given for some integer n a family of finite sets X_1, X_2, \dots, X_n , the number of elements in the union over all sets is $|\bigcup_{j=1}^n X_j| = \sum_{I \subseteq \{1, \dots, n\}, I \neq \emptyset} (-1)^{|I|-1} |\bigcap_{i \in I} X_i|$.

Satisfiability. A literal is a (Boolean) variable x or its negation $\neg x$. A *clause* is a finite set of literals, interpreted as the disjunction of these literals. A (CNF) *formula* is a finite set of clauses, interpreted as the conjunction of its clauses. A 3-CNF has clauses of length at most 3. Let F be a formula. A *sub-formula* S of F is a subset $S \subseteq F$ of F . For a clause $c \in F$, we let $\text{var}(c)$ consist of all variables that occur in c and $\text{var}(F) := \bigcup_{c \in F} \text{var}(c)$. An *assignment* is a mapping $\alpha : V \rightarrow \{0, 1\}$ for a set $V \subseteq \text{var}(F)$ of variables. For $x \in V$, we define $\alpha(\neg x) := 1 - \alpha(x)$. The formula F *under an assignment* α is the formula $F[\alpha]$ obtained from F by removing all clauses c containing a literal set to 1 by α and removing from the remaining clauses all literals set to 0 by α . An assignment α is *satisfying* if $F[\alpha] = \emptyset$, denoted by $\alpha \models F$. Then, F is *satisfiable* if there is such a satisfying assignment α , otherwise we

³nestHDB is open-source and available at github.com/hmarkus/dp_on_dbs/tree/nesthdb.

say F is *unsatisfiable*. Let V be a set of variables. An *interpretation* is a set $J \subseteq V$ and its induced assignment $\alpha_{J,V}$ of J with respect to V is defined as follows $\alpha_{J,V} := \{v \mapsto 1 \mid v \in J \cap V\} \cup \{v \mapsto 0 \mid v \in V \setminus J\}$. We simply write α_J for $\alpha_{J,V}$ if $V = \text{var}(F)$. An interpretation J is a *model* of F if its induced assignment α_J is satisfying, i.e., $\alpha_J \models F$. Given a formula F ; the problem SAT asks whether F is satisfiable and the problem #SAT asks to output the number of models of F , i.e., $|S|$ where S is the set of all models of F .

Projected Model Counting. An instance of the projected model counting problem is a pair (F, P) where F is a (CNF) formula and P is a set of Boolean variables such that $P \subseteq \text{var}(F)$. We call the set P *projection variables* of the instance. The *projected model count* of a formula F with respect to P is the number of total assignments α to variables in P such that the formula $F[\alpha]$ under α is satisfiable. The *projected model counting problem* (PMC) [21] asks to output the projected model count of F , i.e., $|\{M \cap P \mid M \in S\}|$ where S is the set of all models of F .

Example 1. Consider formula $F := \{\overbrace{\neg a \vee b \vee p_1}^{c_1}, \overbrace{a \vee \neg b \vee \neg p_1}^{c_2}, \overbrace{a \vee p_2}^{c_3}, \overbrace{a \vee \neg p_2}^{c_4}\}$ and set $P := \{p_1, p_2\}$ of projection variables. The models of formula F are $\{a, b\}$, $\{a, p_1\}$, $\{a, b, p_1\}$, $\{a, b, p_2\}$, $\{a, p_1, p_2\}$, and $\{a, b, p_1, p_2\}$. However, projected to the set P , we only have models \emptyset , $\{p_1\}$, $\{p_2\}$, and $\{p_1, p_2\}$. Hence, the model count of F is 6 whereas the projected model count of instance (F, P) is 4.

Quantified Boolean Formulas (QBFs). A (prenex) quantified Boolean formula Q is of the form $Q_1 V_1. Q_2 V_2. \dots Q_m V_m. F$ where $Q_i \in \{\forall, \exists\}$, V_i are disjoint sets of Boolean variables, and F is a Boolean formula that contains only the variables in $\bigcup_{i=1}^m V_i$. The truth (evaluation) of quantified Boolean formulas is defined in the standard way, where for Q above if $Q_1 = \exists$, then Q evaluates to true if and only if there exists an assignment $\alpha : V_1 \rightarrow \{0, 1\}$ such that $Q_2 V_2. \dots Q_m V_m. F[\alpha]$ evaluates to true. If $Q_1 = \forall$, then Q evaluates to true if for any assignment $\alpha : V_1 \rightarrow \{0, 1\}$, we have that $Q_2 V_2. \dots Q_m V_m. F[\alpha]$ evaluates to true. Given a quantified Boolean formula Q , the evaluation problem of quantified Boolean formulas QSAT asks whether Q evaluates to true. The problem QSAT is PSPACE-complete and is therefore believed to be computationally harder than SAT [47, 48, 49]. A well known fragment of QSAT is $\forall\exists$ -SAT where the input is restricted to quantified Boolean formulas of the form $\forall V_1. \exists V_2. F$ where F is a Boolean CNF formula. The complexity class consisting of all problems that are polynomial-time reducible to $\forall\exists$ -SAT is denoted by Π_2^P , and its complement is denoted by Σ_2^P . For more detailed information on QBFs we refer to other sources, e.g., [50, 47].

Computational Complexity. We assume familiarity with standard notions in computational complexity [48] and use counting complexity classes as defined by Hemaspaandra and Vollmer [51]. For parameterized complexity, we refer to standard texts [27, 28, 29, 30]. Let Σ and Σ' be some finite alphabets. We call $I \in \Sigma^*$ an *instance* and $\|I\|$ denotes the size of I . Let $L \subseteq \Sigma^* \times \mathbb{N}$ and $L' \subseteq \Sigma'^* \times \mathbb{N}$ be two parameterized problems. An *fpt-reduction* r from L to L'

is a many-to-one reduction from $\Sigma^* \times \mathbb{N}$ to $\Sigma'^* \times \mathbb{N}$ such that for all $I \in \Sigma^*$ we have $(I, k) \in L$ if and only if $r(I, k) = (I', k') \in L'$ such that $k' \leq g(k)$ for a fixed computable function $g : \mathbb{N} \rightarrow \mathbb{N}$, and there is a computable function f and a constant c such that r is computable in time $O(f(k)\|I\|^c)$ [29]. A *witness function* is a function $\mathcal{W} : \Sigma^* \rightarrow 2^{\Sigma'^*}$ that maps an instance $I \in \Sigma^*$ to a finite subset of Σ'^* . We call the set $\mathcal{W}(I)$ the *witnesses*. A *parameterized counting problem* $L : \Sigma^* \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$ is a function that maps a given instance $I \in \Sigma^*$ and an integer $k \in \mathbb{N}$ to the cardinality of its witnesses $|\mathcal{W}(I)|$. We call k the *parameter*. The *exponential time hypothesis* (ETH) states that the (decision) problem SAT on 3-CNF formulas *cannot* be solved in time $2^{s \cdot n} \cdot n^{O(1)}$ for some $s > 0$ where n is the number of variables [42].

Graph Theory. We recall some graph theoretical notations. For further basic terminology on graphs and digraphs, we refer to standard texts [52, 53]. An *undirected graph* or simply a *graph* is a pair $G = (V, E)$ where $V \neq \emptyset$ is a set of *vertices* and $E \subseteq \{\{u, v\} \subseteq V \mid u \neq v\}$ is a set of *edges*. A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$ and an *induced subgraph* if additionally for any $u, v \in V'$ and $\{u, v\} \in E$ also $\{u, v\} \in E'$. Let $G = (V, E)$ be a graph and $A \subseteq V$ be a set of vertices. We define the *subgraph* $G - A$, which is the graph obtained from G by removing vertices A , by $G - A := (V \setminus A, \{e \in E, e \cap A = \emptyset\})$. Graph G is *complete* if for any two vertices $u, v \in V$ there is an edge $uv \in E$. G contains a *clique* on $V' \subseteq V$ if the induced subgraph (V', E') of G is a complete graph. A (*connected*) *component* $C \subseteq V$ of G is a \subseteq -largest set such that for any two vertices $u, v \in C$ there is a path from u to v in G .

Tree Decompositions and Treewidth. For basic terminology on graphs, we refer to standard texts [52, 53]. For a (rooted) tree $T = (N, A)$ with root node $\text{root}(T)$ and a node $t \in N$, we let $\text{children}(t)$ be the sequence of all nodes t' in arbitrarily but fixed order, which have an edge $(t, t') \in A$. Let $G = (V, E)$ be a graph. A *tree decomposition* (TD) of graph G is a pair $\mathcal{T} = (T, \chi)$ where $T = (N, A)$ is a rooted tree and χ a mapping that assigns to each node $t \in N$ a set $\chi(t) \subseteq V$, called a *bag*, such that the following conditions hold: (i) $V = \bigcup_{t \in N} \chi(t)$ and $E \subseteq \bigcup_{t \in N} \{uv \mid u, v \in \chi(t)\}$; (ii) for each $r, s, t \in N$ such that s lies on the path from r to t , we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. Then, $\text{width}(\mathcal{T}) := \max_{t \in N} |\chi(t)| - 1$. The *treewidth* $\text{tw}(G)$ of G is the minimum $\text{width}(\mathcal{T})$ over all tree decompositions \mathcal{T} of G . For arbitrary but fixed $w \geq 1$, it is feasible in linear time to decide if a graph has treewidth at most w and, if so, to compute a tree decomposition of width w [54]. In order to simplify case distinctions in the algorithms, we always use so-called nice tree decompositions, which can be computed in linear time without increasing the width [55] and are defined as follows. For a node $t \in N$, we say that $\text{type}(t)$ is *leaf* if $\text{children}(t) = \langle \rangle$; *join* if $\text{children}(t) = \langle t', t'' \rangle$ where $\chi(t) = \chi(t') = \chi(t'') \neq \emptyset$; *int* (“introduce”) if $\text{children}(t) = \langle t' \rangle$, $\chi(t') \subseteq \chi(t)$ and $|\chi(t)| = |\chi(t')| + 1$; *rem* (“removal”) if $\text{children}(t) = \langle t' \rangle$, $\chi(t') \supseteq \chi(t)$ and $|\chi(t')| = |\chi(t)| + 1$. If for every node $t \in N$, $\text{type}(t) \in \{\text{leaf}, \text{join}, \text{int}, \text{rem}\}$ and bags of leaf nodes and the root are empty, then the TD is called *nice*.

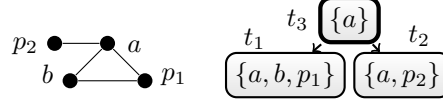


Figure 1: Primal graph G_F of F from Example 2 (left) with a TD \mathcal{T} of graph G_F (right).

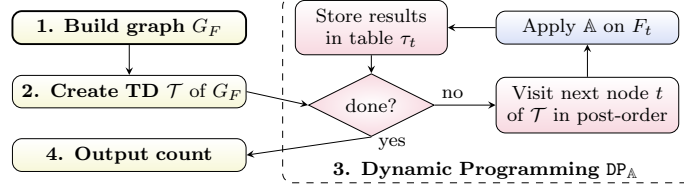


Figure 2: The DP approach, where table algorithm A modifies tables. [56]

3. Dynamic Programming on TDs for SAT

Before we introduce our algorithm, we need some notations for dynamic programming on tree decompositions and recall how to solve the decision problem SAT by exploiting small treewidth. To this end, we present in Section 3.1 basic notation and a simple algorithm for solving SAT and #SAT via utilizing treewidth. The simple algorithm is inspired by related work [31], which is extended by the capability of actually computing some (projected) models in Section 3.2. The algorithm and the definitions of the whole section will then serve as a basis for solving projected model counting in Section 4.

3.1. Dynamic Programming for SAT

Graph Representation of SAT Formulas. In order to use tree decompositions for satisfiability problems, we need a dedicated graph representation of the given formula F . The *primal graph* G_F of F has as vertices the variables of F and two variables are joined by an edge if they occur together in a clause of F . Further, we define some auxiliary notation. For a given node t of a tree decomposition (T, χ) of the primal graph, we let the *bag formula* $F_t := \{c \mid c \in F, \text{var}(c) \subseteq \chi(t)\}$, i.e., clauses entirely covered by $\chi(t)$. The set $F_{\leq t}$ denotes the union over F_s for all descendant nodes s of t . In the following, we sometimes simply write *tree decomposition of formula F* or *treewidth of F* and omit the actual graph representation of F .

Example 2. Consider formula F from Example 1. The primal graph G_F of formula F and a tree decomposition \mathcal{T} of G_F are depicted in Figure 1. Intuitively, \mathcal{T} allows to evaluate formula F in parts. When evaluating $F_{\leq t_3}$, we split into $F_{\leq t_1} = \{c_1, c_2\}$ and $F_{\leq t_2} = \{c_3, c_4\}$, respectively.

Algorithms that solve SAT or #SAT [31] in linear time for input formulas of bounded treewidth proceed by dynamic programming along the tree decomposition (in post-order) where at each node t of the tree information is gathered [57] in a table τ_t . A *table* τ is a set of rows, where a *row* $\vec{u} \in \tau$ is a sequence of fixed

Listing 1: Algorithm $\text{DP}_{\mathbb{A}}((F, P), \mathcal{T}, PP\text{-}Tabs)$ for DP on TD \mathcal{T} .

In: Table algorithm \mathbb{A} , and instance (F, P) of PMC, a TD $\mathcal{T} = (T, \chi)$ of the primal graph G_F of F , and tables $PP\text{-}Tabs$.

Out: Table mapping $\mathbb{A}\text{-}Comp$, which maps each TD node t of T to some computed table τ_t .

```

1  $\mathbb{A}\text{-}Comp \leftarrow \{\}$       /* empty mapping */
2 for iterate  $t$  in post-order( $T$ ) do
3    $Child\text{-}Tabs \leftarrow \langle \mathbb{A}\text{-}Comp[t_1], \dots, \mathbb{A}\text{-}Comp[t_\ell] \rangle$  where  $children(t) = \langle t_1, \dots, t_\ell \rangle$ 
4    $\mathbb{A}\text{-}Comp[t] \leftarrow \mathbb{A}(t, \chi(t), F_t, P \cap \chi(t), Child\text{-}Tabs, PP\text{-}Tabs)$ 
5 return  $\mathbb{A}\text{-}Comp$ 

```

length, which is denoted by angle brackets. Tables are derived by an algorithm, which we therefore call *table algorithm* \mathbb{A} . The actual length, content, and meaning of the rows depend on the algorithm \mathbb{A} that derives tables. Therefore, we often explicitly state $\mathbb{A}\text{-}row$ if rows of this *type* are syntactically used for table algorithm \mathbb{A} and similar $\mathbb{A}\text{-}table$ for tables. For sake of comprehension, we specify the rows before presenting the actual table algorithm for manipulating tables. The rows used by a table algorithm SAT have in common that the first position of these rows manipulated by SAT consists of an interpretation. The remaining positions of the row depend on the considered table algorithm. For each sequence $\vec{u} \in \tau$, we write $I(\vec{u})$ to address the interpretation (first) part of the sequence \vec{u} . Further, for a given positive integer i , we denote by $\vec{u}_{(i)}$ the i -th element of row \vec{u} and define $\tau_{(i)}$ as $\tau_{(i)} := \{\vec{u}_{(i)} \mid \vec{u} \in \tau\}$.

Then, the dynamic programming approach for Boolean satisfiability works as outlined in Figure 2 and performs the following steps:

1. Construct the primal graph G_F of F .
2. Compute a tree decomposition (T, χ) of G_F , obtainable via heuristics.
3. Run DP_{SAT} , as presented in Listing 1, which executes a table algorithm SAT for every node t in post-order of the nodes of T , and returns SAT-Comp mapping every node t to its table. SAT takes as input⁴ bag $\chi(t)$, sub-formula F_t , and tables Child-Tabs previously computed at children of t and outputs a table τ_t .
4. Print a positive result whenever the table for node $\text{root}(T)$ is not empty.

The basic steps of the approach are briefly summarized by Listing 2.

Listing 3 presents table algorithm SAT that uses the primal graph representation. We provide only brief intuition, for details we refer to the original source [31]. The main idea is to store in table τ_t only interpretations restricted to bag $\chi(t)$ that can be extended to a model of sub-formula $F_{\leq t}$. Table algorithm SAT transforms at node t certain row combinations of the tables (Child-Tabs) of child

⁴Actually, SAT takes in addition as input $PP\text{-}Tabs$, which contains a mapping of nodes of the tree decomposition to tables, i.e., tables of the previous pass. Later, we use this for a second traversal to pass results (SAT-Comp) from the first traversal to the table algorithm PROJ for projected model counting in the second traversal.

Listing 2: Algorithm for solving SAT via dynamic programming.

In: A Boolean formula F in CNF.

Out: Satisfiability of F .

```

1  $\mathcal{T} = (T, \chi) \leftarrow \text{Decompose\_via\_Heuristics}(G_F)$     /* Decompose */
2  $\text{SAT-Comp} \leftarrow \text{DP}_{\text{SAT}}((F, P), \mathcal{T}, \emptyset)$     /* DP via table algorithm SAT */
3 return  $\text{PROJ-Comp}[\text{root}(T)] \neq \emptyset$     /* true iff root table is not empty */
```

Listing 3: Table algorithm $\text{SAT}(t, \chi_t, F_t, \cdot, \text{Child-Tabs}, \cdot)$ [31].

In: Node t , bag χ_t , clauses F_t , and sequence $\text{Child-Tabs} = \langle \tau_1, \dots, \tau_\ell \rangle$ of child SAT-tables of t .

Out: SAT-Table τ_t .

```

1 if  $\text{type}(t) = \text{leaf}$  then  $\tau_t \leftarrow \{\langle \emptyset \rangle\}$ 
2 else if  $\text{type}(t) = \text{int}$  and  $a \in \chi_t$  is introduced then
3   |  $\tau_t \leftarrow \{\langle K \rangle \mid \langle J \rangle \in \tau_1, K \in \{J, J \cup \{a\}\}, K \models F_t\}$ 
4 else if  $\text{type}(t) = \text{rem}$  and  $a \notin \chi_t$  is removed then
5   |  $\tau_t \leftarrow \{\langle J \setminus \{a\} \rangle \mid \langle J \rangle \in \tau_1\}$ 
6 else if  $\text{type}(t) = \text{join}$  then
7   |  $\tau_t \leftarrow \{\langle J \rangle \mid \langle J \rangle \in \tau_1 \cap \tau_2\}$ 
8 return  $\tau_t$ 
```

nodes of t into rows of table τ_t . The transformation depends on a case where variable a is added or not added to an interpretation (*int*), removed from an interpretation (*rem*), or where coinciding interpretations are required (*join*). In the end, an interpretation $I(\vec{u})$ from a row \vec{u} of the table τ_n at the root n proves that there is a superset $J \supseteq I(\vec{u})$ that is a model of $F = F_{\leq n}$, and hence that the formula is satisfiable.

Example 3 lists selected tables when running algorithm DP_{SAT} on a nice tree decomposition. Note that illustration along the lines of a nice TD allows us to visualize the basic cases separately. If one was to implement such an algorithm on general TDs, one still obtains the same basic cases, but interleaved.

Example 3. Consider formula F from Example 2. Figure 3 illustrates a nice TD $\mathcal{T}' = (\cdot, \chi)$ of the primal graph of F and tables τ_1, \dots, τ_{12} that are obtained during the execution of $\text{DP}_{\text{SAT}}((F, \cdot), \mathcal{T}', \cdot)$. We assume that each row in a table τ_t is identified by a number, i.e., row i corresponds to $u_{t,i} = \langle J_{t,i} \rangle$.

Table $\tau_1 = \{\langle \emptyset \rangle\}$, due to $\text{type}(t_1) = \text{leaf}$. Since $\text{type}(t_2) = \text{int}$, we construct table τ_2 from τ_1 by taking $J_{1,i}$ and $J_{1,i} \cup \{a\}$ for each $\langle J_{1,i} \rangle \in \tau_1$. Then, t_3 introduces p_1 and t_4 introduces b . $F_{t_1} = F_{t_2} = F_{t_3} = \emptyset$, but since $\chi(t_4) \subseteq \text{var}(c_1)$ we have $F_{t_4} = \{c_1, c_2\}$ for t_4 . In consequence, for each $J_{4,i}$ of table τ_4 , we have $\alpha_{J_{4,i}} \models \{c_1, c_2\}$ since SAT enforces satisfiability of F_t in node t . Since $\text{type}(t_5) = \text{rem}$, we remove variable p_1 from all elements in τ_4 to construct τ_5 . Note that we have already seen all rules where p_1 occurs and hence p_1 can no longer affect interpretations during the remaining traversal. We similarly create $\tau_6 = \{\langle \emptyset \rangle, \langle a \rangle\}$ and $\tau_{10} = \{\langle a \rangle\}$. Since $\text{type}(t_{11}) = \text{join}$, we build table τ_{11} by taking the intersection of τ_6 and τ_{10} . Intuitively, this combines interpretations agreeing on a . By definition (primal graph and TDs), for every $c \in F$, variables $\text{var}(c)$ occur

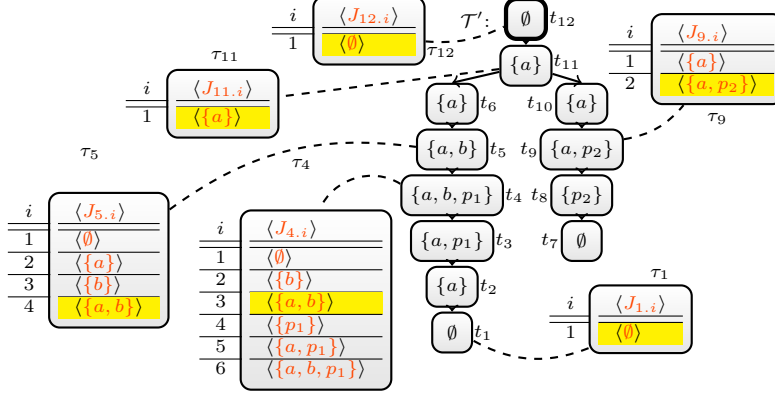


Figure 3: Selected tables obtained by algorithm DP_{SAT} on tree decomposition \mathcal{T}' .

together in at least one common bag. Hence, $F = F_{\leq t_{12}}$ and since $\tau_{12} = \{\langle \emptyset \rangle\}$, we can reconstruct for example model $\{a, b, p_2\} = J_{11.1} \cup J_{5.4} \cup J_{9.2}$ of F using highlighted (yellow) rows in Figure 3. On the other hand, if F was unsatisfiable, τ_{12} would be empty (\emptyset).

Interestingly, the above table algorithm SAT can be easily extended to also count models. Such a table algorithm for solving $\#\text{SAT}$ works similarly to SAT , but additionally also maintains a counter [31]. There, intuitively, rows of tables for leaf nodes set this counter to 1 and introduce nodes basically just copy the counter value of child rows. Then, upon removing a certain variable, one has to add (sum up) counters accordingly, and for join nodes counters need to be multiplied. Finally, the counters of the table for the root node can be summed up to obtain the solution to the $\#\text{SAT}$ problem.

3.2. (Re-)constructing Interpretations and Models

Even further, with the help of the obtained tables during dynamic programming, one can actually construct (projected) models by combining suitable predecessor rows. The idea is to combine those obtained rows that contain parts of models that fit together. To this end, we require the following definition, which we will also use later. At a node t and for a row \vec{u} of the computed table $\text{SAT-Comp}[t]$, it yields the *originating rows* in the tables of the children of t that were involved in computing row \vec{u} by algorithm SAT .

Definition 1 (Origins, cf., [41]). *Let F be a formula, $\mathcal{T} = (T, \chi)$ be a tree decomposition of F , t be a node of T with $\text{children}(t) = \langle t_1, \dots, t_\ell \rangle$, and $\tau_1 \in \text{SAT-Comp}[t_1], \dots, \tau_\ell \in \text{SAT-Comp}[t_\ell]$ be the tables computed by DP_{SAT} .*

For a given SAT-row \vec{u} in $\text{SAT-Comp}[t]$, we define its originating SAT-rows by $\text{Origins}(t, \vec{u}) := \{\vec{s} \mid \vec{s} \in \tau_1 \times \dots \times \tau_\ell, \tau = \text{SAT}(t, \chi(t), F_t, \cdot, \langle \{\vec{s}\} \rangle, \cdot), \vec{u} \in \tau\}$.⁵ We naturally extend this to a SAT-table σ by $\text{Origins}(t, \sigma) := \bigcup_{\vec{u} \in \sigma} \text{Origins}(t, \vec{u})$.

⁵Given a sequence $\vec{s} = \langle s_1, \dots, s_\ell \rangle$, we let $\langle \{\vec{s}\} \rangle := \langle \{s_1\}, \dots, \{s_\ell\} \rangle$, for technical reasons.

Example 4 illustrates Definition 1 for our running example, where we briefly show origins for some rows of selected tables.

Example 4. Consider formula F , tree decomposition $\mathcal{T}' = (T, \chi)$, and tables τ_1, \dots, τ_{12} from Example 3. We focus on $u_{1.1}^- = \langle J_{1.1} \rangle = \langle \emptyset \rangle$ of table τ_1 of the leaf t_1 . The row $u_{1.1}^-$ has no preceding row, since $\text{type}(t_1) = \text{leaf}$. Hence, we have $\text{Origins}(t_1, u_{1.1}^-) = \{\langle \rangle\}$. The origins of row $u_{5.1}^-$ of table τ_5 are given by $\text{Origins}(t_5, u_{5.1}^-)$, which correspond to the preceding rows in table t_4 that lead to row $u_{5.1}^-$ of table τ_5 when running algorithm SAT, i.e., $\text{Origins}(t_5, u_{5.1}^-) = \{\langle u_{4.1}^- \rangle, \langle u_{4.4}^- \rangle\}$. Observe that $\text{Origins}(t_i, \vec{u}) = \emptyset$ for any row $\vec{u} \notin \tau_i$. For node t_{11} of type join and row $u_{11.1}^-$, we obtain $\text{Origins}(t_{11}, u_{11.1}^-) = \{\langle u_{6.2}^-, u_{10.1}^- \rangle\}$ (see Example 3). More general, when using algorithm SAT, at a node t of type join with table τ we have $\text{Origins}(t, \vec{u}) = \{\langle \vec{u}, \vec{u} \rangle\}$ for row $\vec{u} \in \tau$.

Definition 1 refers to the predecessors of rows. In order to reconstruct models, one needs to recursively combine these origins from a node t down to the leafs. This idea of *combining suitable rows* is formalized in the following definition, which introduces the concept of *extensions*. Thereby, rows are *extended* such that one can then reconstruct models from these extensions.

Definition 2 (Extensions). Let F be a formula, $\mathcal{T} = (T, \chi)$ be a tree decomposition, t be a node of T , and \vec{u} be a row of SAT-Comp[t].

An extension below t is a set of pairs where a pair consists of a node t' of $T[t]$ and a row \vec{v} of SAT-Comp[t'] and the cardinality of the set equals the number of nodes in the sub-tree $T[t]$. We define the family of extensions below t recursively as follows. If t is of type leaf, then $\text{Exts}_{\leq t}(\vec{u}) := \{\{\langle t, \vec{u} \rangle\}\}$; otherwise $\text{Exts}_{\leq t}(\vec{u}) := \bigcup_{\vec{v} \in \text{Origins}(t, \vec{u})} \{\{\langle t, \vec{u} \rangle\} \cup X_1 \cup \dots \cup X_\ell \mid X_i \in \text{Exts}_{\leq t_i}(\vec{v}_{(i)})\}$ for the ℓ children t_1, \dots, t_ℓ of t . We lift this notation for a SAT-table σ by $\text{Exts}_{\leq t}(\sigma) := \bigcup_{\vec{u} \in \sigma} \text{Exts}_{\leq t}(\vec{u})$. Further, we let $\text{Exts} := \text{Exts}_{\leq n}(\text{SAT-Comp}[n])$.

Indeed, if we construct extensions below the root n , it allows us to also obtain all models of a formula F . Finally, we define notation that gives us a way to reconstruct interpretations from such (families of) extensions.

Definition 3 (Interpretations of Extensions). Let (F, P) be an instance of PMC, $\mathcal{T} = (T, \chi)$ be a tree decomposition of F , t be a node of T . Further, let E be a family of extensions below t , and P be a set of projection variables. We define the set $I(E)$ of interpretations of E by $I(E) := \{\bigcup_{\langle \cdot, \vec{u} \rangle \in X} I(\vec{u}) \mid X \in E\}$ and the set $I_P(E)$ of projected interpretations by $I_P(E) := \{\bigcup_{\langle \cdot, \vec{u} \rangle \in X} I(\vec{u}) \cap P \mid X \in E\}$.

We briefly illustrate these concepts along the lines of our running example.

Example 5. Consider again formula F and tree decomposition \mathcal{T}' with root n of F from Example 3. Let $X = \{\langle t_{12}, \langle \emptyset \rangle \rangle, \langle t_{11}, \langle \{a\} \rangle \rangle, \langle t_6, \langle \{a\} \rangle \rangle, \langle t_5, \langle \{a, b\} \rangle \rangle, \langle t_4, \langle \{a, b\} \rangle \rangle, \langle t_3, \langle \{a\} \rangle \rangle, \langle t_2, \langle \{a\} \rangle \rangle, \langle t_1, \langle \emptyset \rangle \rangle, \langle t_{10}, \langle \{a\} \rangle \rangle, \langle t_9, \langle \{a, p_2\} \rangle \rangle, \langle t_8, \langle \{p_2\} \rangle \rangle, \langle t_7, \langle \emptyset \rangle \rangle\}$ be an extension below n . Observe that $X \in \text{Exts}$ and that Figure 3 highlights those rows of tables for nodes $t_{12}, t_{11}, t_9, t_5, t_4$ and t_1 that also occur in X (in yellow). Further, $I(\{X\}) = \{a, b, p_2\}$ computes the corresponding model of X , and $I_P(\{X\}) = \{p_2\}$ derives the projected model of X . $I(\text{Exts})$ refers to the set of models of F , whereas $I_P(\text{Exts})$ is the set of projected models of F .

In order to only construct extensions that correspond to (parts of) models of the formula, we simply need to access only those extensions that contain rows that lead to models of the formula. As already observed in the previous example, these rows are precisely the ones contained in Exts . The resulting extensions for a node t are formalized in the following concept of *satisfiable extensions*, whereby we take only those extensions of $\text{Exts}_{\leq t}$ that are also contained in Exts .

Definition 4 (Satisfiable Extension). *Let F be a formula, $\mathcal{T} = (T, \chi)$ be a tree decomposition of F , t be a node of T , and $\sigma \subseteq \text{SAT-Comp}[t]$ be a set of rows. Then, we define the satisfiable extensions below t for σ by $\text{SatExt}_{\leq t}(\sigma) := \bigcup_{\vec{u} \in \sigma} \{X \mid X \in \text{Exts}_{\leq t}(\vec{u}), X \subseteq Y, Y \in \text{Exts}\}$.*

4. Counting Projected Models by Dynamic Programming

While the transition from deciding SAT to solving $\#\text{SAT}$ is quite simple by adding an additional counter, it turns out that the problem PMC requires more effort. We solve this problem PMC by providing an algorithm in Section 4.1 that utilizes treewidth and adheres to multiple passes (rounds) of computation that are guided along a tree decomposition. Then, we give detailed formal arguments on correctness of this algorithm in Section 4.2. Later, in Section 4.3 we discuss complexity results in the form of matching upper and lower bounds, where it turns out that our algorithm cannot be significantly improved.

4.1. Solving PMC by means of Dynamic Programming

Next, we introduce the dynamic programming algorithm PCNT_{SAT} to solve the projected model counting problem (PMC) for Boolean formulas. From a high-level perspective, our algorithm builds upon the table algorithm SAT from the previous section; we assume again a formula F and a tree decomposition $\mathcal{T} = (T, \chi)$ of F , and additionally a set P of projection variables. Thereby, the table for each tree decomposition node t consists of a set σ of assignments restricted to bag variables $\chi(t)$ (as computed by SAT) that agree on their assignment of variables in $P \cap \chi(t)$, and a counter c . Intuitively, this counter c counts those *satisfying assignments* of $F_{\leq t}$ restricted to $P \cap \chi(t)$ that are among satisfiable extensions and extend any assignment in σ . Then, for the (empty) tree decomposition root n , there is only one single counter which is the projected model count of F with respect to P . The *challenge* of our algorithm PCNT_{SAT} is to compute these counts c by only considering local information, i.e., previously computed tables of child nodes of t . To this end, we utilize mathematical combinatorics, namely the principle of inclusion-exclusion principle [46], which we need to apply in an interleaved fashion.

Concretely, our algorithm PCNT_{SAT} traverses the tree decomposition twice following a multi-pass dynamic programming paradigm [41]. Figure 4 illustrates the steps of our algorithm PCNT_{SAT} , which are also presented in the form of Listing 4. Similar to the previous section (cf., Figure 2), we construct a graph representation and heuristically compute a tree decomposition of this graph. Then, we run DP_{SAT} (see Listing 1) in Step 3a as *first pass*. Step 3a can also

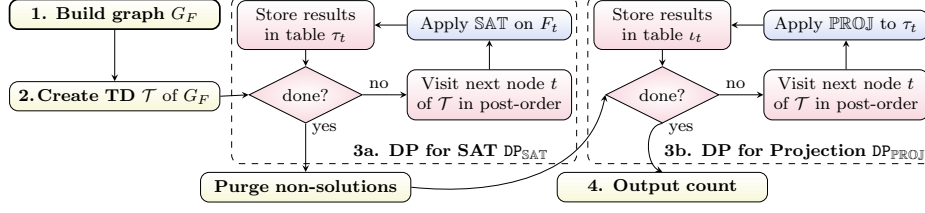


Figure 4: Algorithm PCNT_{SAT} consists of DP_{SAT} and DP_{PROJ} .

Listing 4: Algorithm $\text{PCNT}_{\text{SAT}}(F, P)$ for solving PMC via dynamic programming.

In: An instance (F, P) of PMC.

Out: The projected model count of (F, P) .

```

1  $\mathcal{T} = (T, \chi) \leftarrow \text{Decompose\_via\_Heuristics}(G_F)$  /* Decompose */
2  $\text{SAT-Comp} \leftarrow \text{DP}_{\text{SAT}}((F, P), \mathcal{T}, \emptyset)$  /* DP via table algorithm SAT */
/* Purge non-solutions of SAT-Comp; ensured by using SatExt below. */
3  $\text{PROJ-Comp} \leftarrow \text{DP}_{\text{PROJ}}((F, P), \mathcal{T}, \text{SAT-Comp})$  /* DP via algorithm PROJ */
4 return  $\sum_{\langle \varphi, c \rangle \in \text{PROJ-Comp}[\text{root}(T)]} c$  /* Return projected model count */

```

be seen as a preprocessing step for projected model counting, from which we immediately know whether the formula has a model. However, we keep the SAT-tables that have been computed in Step 3a. These tables form the basis for the next step.

There, we remove all rows from the obtained SAT-tables which cannot be extended to a model of the SAT problem (“*Purge non-solutions*”). In other words, we keep only rows \vec{u} in table $\text{SAT-Comp}[t]$ at node t if its interpretation $I(\vec{u})$ can be extended to a model of F . Thereby, we avoid redundancies and can simplify the description and presentation of our next step, since we then only consider rows that are (parts of) models. Intuitively, the rows involving non-models contributes only non-relevant information, as also observed in related works [37, 58]. Formally, this is achieved by utilizing satisfiable extensions as defined in Definition 4, since these extensions precisely consider the rows that contribute to models.

In Step 3b (DP_{PROJ}), we perform the *second pass*, where we traverse the tree decomposition a second time to count projections of interpretations of rows in SAT-tables. Observe that the tree traversal in DP_{PROJ} is the same as before. Therefore, in the following, we describe the ingredients that lead to table algorithm PROJ. For PROJ, a row at a node t is a pair $\langle \sigma, c \rangle$ where σ is a SAT-table, in particular, a subset of $\text{SAT-Comp}[t]$ computed by DP_{SAT} , and c is a non-negative integer. Below, we characterize σ , which is based on grouping rows in equivalence classes.

Equivalence Classes for SAT-Tables. The following definitions provide central notions for grouping rows of tables according to the given projection of variables, which yields an equivalence relation.

Definition 5. Let (F, P) be an instance of PMC and σ be a SAT-table. We

define the relation $=_P \subseteq \sigma \times \sigma$ to consider equivalent rows with respect to the projection of its interpretations by $=_P := \{(\vec{u}, \vec{v}) \mid \vec{u}, \vec{v} \in \sigma, I(\vec{u}) \cap P = I(\vec{v}) \cap P\}$.

Observation 1. *The relation $=_P$ is an equivalence relation.*

Based on this equivalence relation, we define corresponding equivalence classes.

Definition 6 (Equivalence Classes). *Let τ be a SAT-table and \vec{u} be a row of τ . The relation $=_P$ induces equivalence classes $[\vec{u}]_P$ on the SAT-table τ in the usual way, i.e., $[\vec{u}]_P = \{\vec{v} \mid \vec{v} =_P \vec{u}, \vec{v} \in \tau\}$ [59]. We denote by $\text{EqClasses}_P(\tau)$ the set of equivalence classes of τ , i.e., $\text{EqClasses}_P(\tau) := (\tau / =_P) = \{[\vec{u}]_P \mid \vec{u} \in \tau\}$.*

These classes are briefly demonstrated on our running example.

Example 6. *Consider again formula F and set P of projection variables from Example 1 and tree decomposition $\mathcal{T}' = (T, \chi)$ and SAT-table τ_4 from Figure 3. We have $u_{4.1} =_P u_{4.2}$ and $u_{4.4} =_P u_{4.5}$. We obtain the set $\tau_4 / =_P$ of equivalence classes of τ_4 by $\text{EqClasses}_P(\tau_4) = \{\{u_{4.1}, u_{4.2}, u_{4.3}\}, \{u_{4.4}, u_{4.5}, u_{4.6}\}\}$.*

Indeed, the algorithm PROJ, stores at a node t pairs $\langle \sigma, c \rangle$, where σ is actually a (non-empty) subset of the equivalence classes in $\text{EqClasses}_P(\text{SAT-Comp}[t])$. Next, we discuss how the integer c aids in projected counting for such a subset σ .

Counting for Equivalence Classes. In fact, we store in integer c a count that expresses the number of “intersection” projected models (ipmc) that indicates for σ the number of projected models up to node t that the rows in σ have in common (*intersection of models*). In the end, we aim for the projected model count (pmc), i.e., the *combined* number of projected models (union of models), where σ is involved. However, it turns out that the process of computing these projected model counts will be heavily interleaved with the ipmc counts. In the following, we define both counts for a node t of a tree decomposition by means of the satisfying extensions below t .

Notably, the effort of directly computing these counts when strictly following the definition below would not result in an algorithm that is fixed-parameter tractable. As a result, our approach is then subsequently developed thereafter, without explicitly involving *every* descendant node below t in order to fulfill the desired runtime claims.

Definition 7. *Let (F, P) be an instance of PMC, $\mathcal{T} = (T, \chi)$ be a tree decomposition of F , t be a node of T , and $\sigma \subseteq \text{SAT-Comp}[t]$ be a set of SAT-rows for node t . Then, the intersection projected model count $\text{ipmc}_{\leq t}(\sigma)$ of σ below t is the size of the intersection over projected interpretations of the satisfiable extensions of σ below t , i.e., $\text{ipmc}_{\leq t}(\sigma) := |\bigcap_{\vec{u} \in \sigma} I_P(\text{SatExt}_{\leq t}(\{\vec{u}\}))|$.*

The projected model count $\text{pmc}_{\leq t}(\sigma)$ of σ below t is the size of the union over projected interpretations of the satisfiable extensions of σ below t , formally, $\text{pmc}_{\leq t}(\sigma) := |\bigcup_{\vec{u} \in \sigma} I_P(\text{SatExt}_{\leq t}(\{\vec{u}\}))|$.

Note that this definition relies on satisfiable extensions as given in Definition 4. Intuitively, the counts $\text{ipmc}_{\leq t}$ represent for a set σ of SAT-rows, the cardinality of

those projected models of $F_{\leq t}$ that can be extended to models of F , where *every* row in σ is involved. Consequently, for the root n of a nice tree decomposition of F we have that $\text{ipmc}_{\leq n}(\{\langle \emptyset \rangle\}) = \text{pmc}_{\leq n}(\{\langle \emptyset \rangle\})$ coincides with the *projected model count* of F . This is the case since $F_{\leq n} = F$, the bag of n is empty, and therefore the SAT-table for n contains one row if and only if F is satisfiable.

Observe that when computing these counts for a node t , we cannot directly count models since this would not yield a fixed-parameter tractability algorithm. Instead, in order to count, we may *only utilize counters* for sets σ of rows in tables of t and direct child nodes of t , which is more involved than directly counting models. This is established for pmc next by relying on combinatorial counting principles like inclusion-exclusion [46].

Computing Projected Model Counts (pmc). Since PROJ stores in PROJ-tables an SAT-table together with a counter, in the end we need to describe how these counters are maintained. As the first step, we show how for a node t , these counters (ipmc values) for child tables of t can be used to compute pmc values for t . Intuitively, when we are at a node t in the Algorithm DP_{PROJ} we already computed all tables SAT-Comp by DP_{SAT} according to Step 3a, purged non-solutions, and computed PROJ-Comp[t'] for all nodes t' below t and in particular the PROJ-tables Child-Tabs of the children of t . Then, we compute the projected model count of a subset σ of the SAT-rows in SAT-Comp[t], which we formalize by applying the generalized inclusion-exclusion principle to the stored intersection projected model counts of origins.

The idea behind the following definition is that for every origin of σ , we lift the ipmc counts that are stored in the corresponding child tables. However, if we sum up these counts, those models that two origins have in common are over-counted, i.e., they need to be subtracted. But then, those models that three origins have in common are under-counted, i.e., they need to be (re-)added again. In turn, the inclusion-exclusion principle ensures that we obtain the correct pmc value for σ .

Definition 8. Let (F, P) be an instance of PMC, $\mathcal{T} = (T, \chi)$ be a tree decomposition of F , and t be a node of T with ℓ children. Further, let $\text{Child-Tabs} = \langle \text{PROJ-Comp}[t_1], \dots, \text{PROJ-Comp}[t_\ell] \rangle$ be the sequence of PROJ-tables computed by $\text{DP}_{\text{PROJ}}((F, P), \mathcal{T}, \text{SAT-Comp})$, where $\text{children}(t) = \langle t_1, \dots, t_\ell \rangle$ and $\sigma \subseteq \text{SAT-Comp}[t]$ is a table. We define the (inductive) projected model count of σ :

$$\begin{aligned} \text{pmc}(t, \sigma, \text{Child-Tabs}) &:= \sum_{\emptyset \subsetneq O \subseteq \text{Origins}(t, \sigma)} (-1)^{(|O|-1)} \cdot \text{s-ipmc}(\text{Child-Tabs}, O), \text{ where} \\ \text{s-ipmc}(\text{Child-Tabs}, O) &:= \prod_{\substack{i \in \{1, \dots, \ell\}, \\ \langle O_{(i)}, c \rangle \in \text{PROJ-Comp}[t_i]}} c \quad \text{is the stored ipmc from child tables.} \end{aligned}$$

Vaguely speaking, pmc determines the origins of the set σ of rows, goes over all subsets of these origins and looks up the stored counts (s-ipmc) in the PROJ-tables of the children of t . There, we may simply have several child

nodes, i.e., nodes of type *join*, and hence in this case we need to multiply the corresponding children's (independent) ipmc values.

Example 7 provides an idea on how to compute the projected model count of tables of our running example using pmc.

Example 7. *The function defined in Definition 8 allows us to compute the projected count for a given SAT-table. Therefore, consider again formula F and tree decomposition \mathcal{T}' from Example 2 and Figure 3. Say we want to compute the projected count $\text{pmc}(t_5, \{u_{5.4}\}, \text{Child-Tabs})$ where $\text{Child-Tabs} := \{\langle \{u_{4.3}\}, 1 \rangle, \langle \{u_{4.6}\}, 1 \rangle\}$ for row $u_{5.4}$ of table τ_5 . Note that t_5 has $\ell = 1$ child nodes $\langle t_4 \rangle$ and therefore the product of Definition 8 consists of only one factor. Observe that $\text{Origins}(t_5, u_{5.4}) = \{\langle u_{4.3} \rangle, \langle u_{4.6} \rangle\}$. Since the rows $u_{4.3}$ and $u_{4.6}$ do not occur in the same SAT-table of Child-Tabs , only the value of s-ipmc for the two singleton origin sets $\{\langle u_{4.3} \rangle\}$ and $\{\langle u_{4.6} \rangle\}$ is non-zero; for the remaining set of origins we have zero. Hence, we obtain $\text{pmc}(t_5, \{u_{5.4}\}, \text{Child-Tabs}) = 2$.*

Computing Intersection Projected Model Counts (ipmc). Before we present algorithm PROJ (Listing 5), we give the definition allowing us at a certain node t to obtain the ipmc value for a given SAT-table σ by computing the pmc (using stored ipmc values from PROJ-tables for children of t), and subtracting and adding ipmc values for subsets $\emptyset \subsetneq \rho \subsetneq \sigma$ accordingly.

The intuition is that in order to obtain the number of those common projected models, where *every single row* in σ participates, we take all involved projected models of σ and subtract every single row's projected model count (ipmc values). There, we subtracted those models that two rows have in common more than once. Again, these models need to be re-added. Then, the models that three rows have in common are subtracted and so forth. In turn, we end up with the intersection projected model count, i.e., those projected models, where every row of σ is involved.

Definition 9. *Let $\mathcal{T} = (T, \chi)$ be a tree decomposition, t be a node of T , σ be a SAT-table, and Child-Tabs be a sequence of tables. Then, we define the (recursive) ipmc of σ as follows:*

$$\text{ipmc}(t, \sigma, \text{Child-Tabs}) := \begin{cases} 1, & \text{if } \text{type}(t) = \text{leaf}, \\ \text{pmc}(t, \sigma, \text{Child-Tabs}) + \\ \quad \left| \sum_{\emptyset \subsetneq \rho \subsetneq \sigma} (-1)^{|\rho|} \cdot \text{ipmc}(t, \rho, \text{Child-Tabs}) \right|, & \text{otherwise.} \end{cases}$$

In other words, if a node is of type *leaf* the ipmc is one, since by definition of a tree decomposition the bags of nodes of type *leaf* contain only one projected interpretation (the empty set). Otherwise, using Definition 8, we are able to compute the ipmc for a given SAT-table σ , which is by construction the same as $\text{ipmc}_{\leq t}(\sigma)$ (cf., proof of Theorem 1 later). In more detail, we want to compute for a SAT-table σ its ipmc that represents “all-overlapping” counts of σ with respect to set P of projection variables, that is, $\text{ipmc}_{\leq t}(\sigma)$. Therefore, for ipmc, we rearrange the inclusion-exclusion principle. To this end, we take pmc, which

Listing 5: Table algorithm $\text{PROJ}(t, \cdot, \cdot, P, \text{Child-Tabs}, \text{SAT-Comp})$.

In: Node t , set P of projection variables, Child-Tabs, and SAT-Comp.

Out: Table ι_t consisting of pairs $\langle \sigma, c \rangle$, where $\sigma \subseteq \text{SAT-Comp}[t]$ and $c \in \mathbb{N}$.

1 $\iota_t \leftarrow \{ \langle \sigma, \text{ipmc}(t, \sigma, \text{Child-Tabs}) \rangle \mid C \in \text{EqClasses}_P(\text{SAT-Comp}[t]), \emptyset \subsetneq \sigma \subseteq C \}$
2 **return** ι_t

computes the “non-overlapping” count of σ with respect to P , by once more exploiting the inclusion-exclusion principle on origins of σ (as already discussed) such that we count every projected model only once. Then we have to alternately subtract and add ipmc values for strict subsets ρ of σ , accordingly.

We provide an example on how this definition is carried out below.

The Table Algorithm PROJ. Finally, Listing 5 presents table algorithm PROJ, which stores for given node t a PROJ-table consisting of every non-empty subset of equivalence classes for the given table SAT-Comp[t] together with its ipmc (as presented above).

Example 8. Recall instance (F, P) of PMC, tree decomposition \mathcal{T}' , and tables τ_1, \dots, τ_{12} from Example 1, 3, and Figure 3. Figure 5 depicts selected tables of $\iota_1, \dots, \iota_{12}$ obtained after running DP_{PROJ} for counting projected interpretations. We assume numbered rows, i.e., row i in table ι_t corresponds to $v_{t,i} = \langle \sigma_{t,i}, c_{t,i} \rangle$. Note that for some nodes t , there are rows among different SAT-tables that occur in $\text{Exts}_{\leq t}$, but not in $\text{SatExt}_{\leq t}$. These rows are removed during purging. In fact, rows $u_{4,1}, u_{4,2}$, and $u_{4,4}$ do not occur in table ι_4 . Observe that purging is a crucial trick here that avoids to correct stored counters c by backtracking whenever a certain row of a table has no succeeding row in the parent table.

Next, we discuss selected rows obtained by $\text{DP}_{\text{PROJ}}((F, P), \mathcal{T}', \text{SAT-Comp})$. Tables $\iota_1, \dots, \iota_{12}$ that are computed at the respective nodes of the tree decomposition are shown in Figure 5. Since $\text{type}(t_1) = \text{leaf}$, we have $\iota_1 = \{ \langle \emptyset \rangle, 1 \}$. Intuitively, up to node t_1 the SAT-row $\langle \emptyset \rangle$ belongs to 1 equivalence class. Node t_2 introduces variable a , which results in table $\iota_2 := \{ \langle \{a\} \rangle, 1 \}$. Note that the SAT-row $\langle \emptyset \rangle$ is subject to purging. Node t_3 introduces p_1 and node t_4 introduces b . Node t_5 removes projection variable p_1 . The row $v_{5,2}$ of PROJ-table ι_5 has already been discussed in Example 7 and row $v_{5,1}$ works similar. For row $v_{5,3}$ we compute the count $\text{ipmc}(t_5, \{u_{5,2}, u_{5,4}\}, \langle \iota_4 \rangle)$ by means of pmc. Therefore, take for ρ the sets $\{u_{5,2}\}$, $\{u_{5,4}\}$, and $\{u_{5,2}, u_{5,4}\}$. For the singleton sets, we simply have $\text{pmc}(t_5, \{u_{5,2}\}, \langle \iota_4 \rangle) = \text{ipmc}(t_5, \{u_{5,2}\}, \langle \iota_4 \rangle) = c_{5,1} = 1$ and $\text{pmc}(t_5, \{u_{5,4}\}, \langle \iota_4 \rangle) = \text{ipmc}(t_5, \{u_{5,4}\}, \langle \iota_4 \rangle) = c_{5,2} = 2$. To compute $\text{pmc}(t_5, \{u_{5,2}, u_{5,4}\}, \langle \iota_4 \rangle)$ following Definition 8, take for O the sets $\{u_{4,5}\}$, $\{u_{4,3}\}$, and $\{u_{4,6}\}$ into account, since all other non-empty subsets of origins of $u_{5,2}$ and $u_{5,4}$ in ι_4 do not occur in ι_4 . Then, we take the sum over the values $\text{s-ipmc}(\langle \iota_4 \rangle, \{ \langle u_{4,5} \rangle \}) = 1$, $\text{s-ipmc}(\langle \iota_4 \rangle, \{ \langle u_{4,3} \rangle \}) = 1$, and $\text{s-ipmc}(\langle \iota_4 \rangle, \{ \langle u_{4,6} \rangle \}) = 1$; and subtract $\text{s-ipmc}(\langle \iota_4 \rangle, \{ \langle u_{4,5} \rangle, \langle u_{4,6} \rangle \}) = 1$. Hence, $\text{pmc}(t_5, \{u_{5,2}, u_{5,4}\}, \langle \iota_4 \rangle) = 2$. In order to compute $\text{ipmc}(t_5, \{u_{5,2}, u_{5,4}\}, \langle \iota_4 \rangle) = |\text{pmc}(t_5, \{u_{5,2}, u_{5,4}\}, \langle \iota_4 \rangle) - \text{ipmc}(t_5, \{u_{5,2}\}, \langle \iota_4 \rangle) - \text{ipmc}(t_5, \{u_{5,4}\}, \langle \iota_4 \rangle)| = |2 - 1 - 2| = |-1| = 1$.

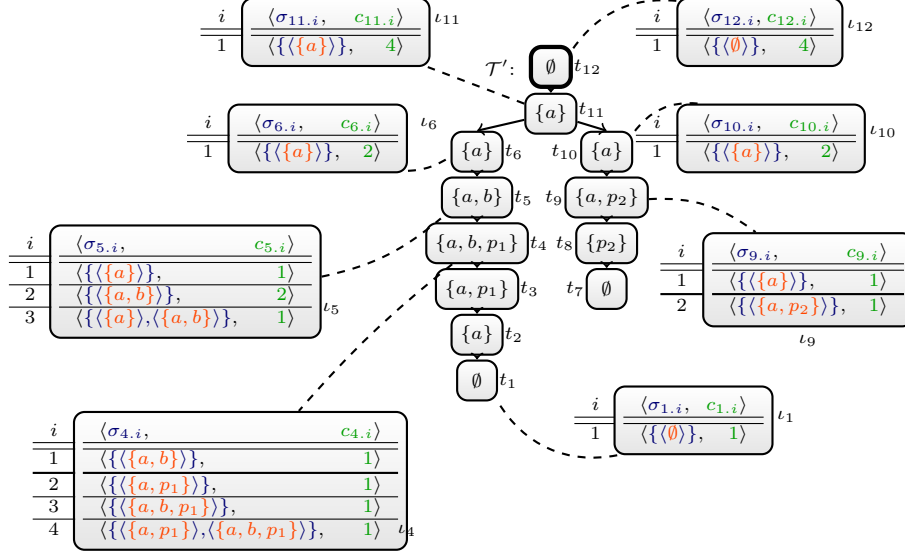


Figure 5: Selected tables obtained by DP_{PROJ} on TD \mathcal{T}' using DP_{SAT} (cf., Figure 3).

Hence, $c_{5,3} = 1$ represents the number of projected models, both rows $u_{5,2}$ and $u_{5,4}$ have in common. We then use it for table t_6 .

For node t_{11} of type join one simply in addition multiplies stored s-ipmc values for SAT-rows in the two children of t_{11} accordingly (see Definition 8). In the end, the projected model count of F corresponds to $\sum_{\langle \sigma, c \rangle \in \ell_{12}} c = c_{12,1} = 4$.

4.2. Correctness of the Algorithm

In the following, we state definitions required for the correctness proofs of our algorithm PROJ . In the end, we only store rows that are restricted to the bag content to maintain runtime bounds. In related work [31], it was shown that this suffices for table algorithm SAT , i.e., SAT is both sound and complete. Similar to related work [56, 31], we proceed in two steps. First, we define properties of so-called PROJ -solutions up to t , and then restrict these to PROJ -row solutions at t .

Assumptions. For the following statements, we assume that we have given an arbitrary instance (F, P) of PMC and a tree decomposition $\mathcal{T} = (T, \chi)$ of formula F , where $T = (N, A)$, node $n = \text{root}(T)$ is the root and \mathcal{T} is of width k . Moreover, for every $t \in N$ of tree decomposition \mathcal{T} , we let $\text{SAT-Comp}[t]$ be the tables that have been computed by running algorithm DP_{SAT} for the dedicated input. Analogously, let $\text{PROJ-Comp}[t]$ be the tables computed by running DP_{PROJ} .

Definition 10. Let $\emptyset \subsetneq \sigma \subseteq \text{SAT-Comp}[t]$ be a table with $\sigma \subseteq C$ for some $C \in \text{EqClasses}_P(\text{SAT-Comp}[t])$. We define a **PROJ**-solution up to t to be the sequence $\langle \hat{\sigma} \rangle = \langle \text{SatExt}_{\leq t}(\sigma) \rangle$.

Next, we recall that we can reconstruct all models from the tables.

Proposition 1. $I(\text{SatExt}_{\leq n}(\text{SAT-Comp}[n])) = I(\text{Exts}) = \{J \in 2^{\text{var}(F)} \mid \alpha_J \models F\}$.

Proof (Sketch). In fact, we can use the construction by Samer and Szeider [31] of the tables. Then, the extensions simply collect the corresponding, preceding rows. By taking the interpretation parts $I(\dots)$ of these collected rows we obtain the set of all models of the formula. A similar construction is used by Pichler, Rümmele, and Woltran [60, Fig. 1], which they use in a general algorithm to enumerate solutions by means of tables obtained during dynamic programming. \square

Before we present equivalence results between $\text{ipmc}_{\leq t}(\dots)$ and the recursive version $\text{ipmc}(t, \dots)$ (Definition 9) used during the computation of DP_{PROJ} , recall that $\text{ipmc}_{\leq t}$ and $\text{pmc}_{\leq t}$ (Definition 7) are key to compute the projected model count. The following corollary states that computing $\text{ipmc}_{\leq n}$ at the root n actually suffices to compute the projected model count $\text{pmc}_{\leq n}$ of the formula.

Corollary 1. $\text{ipmc}_{\leq n}(\text{SAT-Comp}[n]) = \text{pmc}_{\leq n}(\text{SAT-Comp}[n]) = |I_P(\text{SatExt}_{\leq n}(\text{SAT-Comp}[n]))| = |I_P(\text{Exts})| = |\{J \cap P \mid J \in 2^{\text{var}(F)}, \alpha_J \models F\}|$

Proof. The corollary immediately follows from Proposition 1 and the observation that $|\text{SAT-Comp}[n]| \leq 1$ by properties of algorithm **SAT** and since $\chi(n) = \emptyset$. \square

The following lemma establishes that the **PROJ**-solutions up to root n of a given tree decomposition solve the PMC problem.

Lemma 1. The value $\sum_{\langle \hat{\sigma} \rangle}$ is a **PROJ**-solution up to n $|I_P(\hat{\sigma})|$ corresponds to the projected model count c of F with respect to the set P of projection variables.

Proof. (“ \implies ”): Assume that $c = \sum_{\langle \hat{\sigma} \rangle}$ is a **PROJ**-solution up to n $|I_P(\hat{\sigma})|$. Observe that there can be at most one projected solution up to n , since $\chi(n) = \emptyset$. If $c = 0$, then **SAT-Comp** $[n]$ contains no rows. Hence, F has no models, cf., Proposition 1, and obviously also no models projected to P . Consequently, c is the projected model count of F . If $c > 0$ we have by Corollary 1 that c is equivalent to the projected model count of F with respect to P .

(“ \impliedby ”): We proceed similar in the if direction. Assume that c is the projected model count of F and P . If $c = 0$, we have by Proposition 1 that $\text{SatExt}_{\leq n}(\text{SAT-Comp}[n]) = \emptyset$ and therefore **SAT-Comp** $[n] = \emptyset$. As a result for $c = 0$, there does not exist any **PROJ**-solution up to n . Otherwise, i.e., if $c > 0$, the result follows immediately by Corollary 1. \square

In the following, we provide for a given node t and a given **PROJ**-solution up to t , the definition of a **PROJ**-row solution at t .

Definition 11. Let $t, t' \in N$ be nodes of a given tree decomposition \mathcal{T} , and $\hat{\sigma}$ be a PROJ-solution up to t . Then, we define the local table for t' as $\text{local}(t', \hat{\sigma}) := \{\langle \vec{u} \rangle \mid \langle t', \vec{u} \rangle \in \hat{\sigma}\}$, and if $t = t'$, the PROJ-row solution at t by $\langle \text{local}(t, \hat{\sigma}), |I_P(\hat{\sigma})| \rangle$.

Observation 2. Let $\langle \hat{\sigma} \rangle$ be a PROJ-solution up to a node $t \in N$. There is exactly one corresponding PROJ-row solution $\langle \text{local}(t, \hat{\sigma}), |I_P(\hat{\sigma})| \rangle$ at t .

Vice versa, let $\langle \sigma, c \rangle$ be a PROJ-row solution at t for some integer c . Then, there is exactly one corresponding PROJ-solution $\langle \text{SatExt}_{\leq t}(\sigma) \rangle$ up to t .

We need to ensure that storing PROJ-row solutions at a node suffices to solve the PMC problem, which is necessary to obtain runtime bounds (cf., Corollary 3).

Lemma 2. Let $t \in N$ be a node of the tree decomposition \mathcal{T} . There is a PROJ-row solution at root n if and only if the projected model count of F with respect to the set P of projection variables is larger than 0.

Proof. (“ \implies ”): Let $\langle \sigma, c \rangle$ be a PROJ-row solution at root n where σ is a SAT-table and c is a positive integer. Then, by Definition 11, there also exists a corresponding PROJ-solution $\langle \hat{\sigma} \rangle$ up to n such that $\sigma = \text{local}(n, \hat{\sigma})$ and $c = |I_P(\hat{\sigma})|$. Moreover, since $\chi(n) = \emptyset$, we have $|\text{SAT-Comp}[n]| = 1$. Then, by Definition 10, $\hat{\sigma} = \text{SAT-Comp}[n]$. By Corollary 1, we have $c = |I_P(\text{SAT-Comp}[n])|$. Finally, the claim follows.

(“ \impliedby ”): Assume that the projected model count of F with respect to P is larger than zero. Then, by Lemma 1, there is at least one PROJ-solution $\hat{\sigma}$ up to the root n . As a result, by Definition 11, there is also a PROJ-row solution at t , which is precisely $\langle \text{local}(n, \hat{\sigma}), |I_P(\hat{\sigma})| \rangle$. \square

Observation 3. Let X_1, \dots, X_n be finite sets. The number $|\bigcap_{i \in X} X_i|$ is given by $|\bigcap_{i \in X} X_i| = |\bigcup_{j=1}^n X_j| + \sum_{\emptyset \subsetneq I \subsetneq X} (-1)^{|I|} |\bigcap_{i \in I} X_i|$.

$$\begin{aligned} |\bigcap_{i \in X} X_i| = & \left| \bigcup_{j=1}^n X_j \right| - \sum_{\emptyset \subsetneq I \subsetneq X, |I|=1} |\bigcap_{i \in I} X_i| + \sum_{\emptyset \subsetneq I \subsetneq X, |I|=2} |\bigcap_{i \in I} X_i| - \dots \\ & + \sum_{\emptyset \subsetneq I \subsetneq X, |I|=n-1} (-1)^{|I|} |\bigcap_{i \in I} X_i|. \end{aligned}$$

Lemma 3. Let $t \in N$ be a node of the tree decomposition \mathcal{T} with $\text{children}(t) = \langle t_1, \dots, t_\ell \rangle$ and let $\langle \sigma, \cdot \rangle$ be a PROJ-row solution at t . Then,

1. $\text{ipmc}(t, \sigma, \langle \text{PROJ-Comp}[t_1], \dots, \text{PROJ-Comp}[t_\ell] \rangle) = \text{ipmc}_{\leq t}(\sigma)$
2. If $\text{type}(t) \neq \text{leaf}$: $\text{pmc}(t, \sigma, \langle \text{PROJ-Comp}[t_1], \dots, \text{PROJ-Comp}[t_\ell] \rangle) = \text{pmc}_{\leq t}(\sigma)$.

Proof. We prove the statement by simultaneous induction.

(“Induction Hypothesis”): Lemma 3 holds for the nodes in $\text{children}(t)$ and also for node t , but on strict subsets $\rho \subsetneq \sigma$.

(“Base Cases”): Towards showing the base case of the first claim, let $\text{type}(t) = \text{leaf}$. By definition, $\text{ipmc}(t, \emptyset, \langle \rangle) = \text{ipmc}_{\leq t}(\emptyset) = 1$. Next, we establish the base case for the second claim. Since $\text{type}(t) \neq \text{leaf}$, let t be a node that has a node $t' \in N$ with $\text{type}(t') = \text{leaf}$ as child node. Observe that by definition of \mathcal{T} , t has exactly one child. Then, we have $\text{pmc}(t, \sigma, \langle \text{PROJ-Comp}[t'] \rangle) = \sum_{\emptyset \subsetneq O \subseteq \text{Origins}(t, \sigma)} (-1)^{(|O|-1)} \cdot \text{s-ipmc}(\langle \text{SAT-Comp}[t'] \rangle, O) = |\bigcup_{\vec{u} \in \sigma} I_P(\text{SatExt}_{\leq t'}(\{\vec{u}\}))| = \text{pmc}_{\leq t}(\sigma) = 1$ for PROJ-row solution $\langle \sigma, \cdot \rangle$ at t .

(“Induction Step”): We distinguish two cases.

Case (i): Assume that $\ell = 1$. Let $\langle \sigma, c \rangle$ be a PROJ-row solution at t for some integer c , and $t' = t_1$.

First, we show the second claim on pmc values. By Definition 8, we have $\text{pmc}(t, \sigma, \langle \text{PROJ-Comp}[t'] \rangle) = \sum_{\emptyset \subsetneq O \subseteq \text{Origins}(t, \sigma)} (-1)^{(|O|-1)} \cdot \text{s-ipmc}(\text{PROJ-Comp}[t'], O)$, which by definition of s-ipmc results in $\sum_{\emptyset \subsetneq O \subseteq \text{Origins}(t, \sigma)} (-1)^{(|O|-1)} \cdot \text{ipmc}(t', O, \text{PROJ-Comp}[t'])$. By the induction hypothesis, this evaluates to $\sum_{\emptyset \subsetneq O \subseteq \text{Origins}(t, \sigma)} (-1)^{(|O|-1)} \cdot \text{ipmc}_{\leq t'}(O)$. Then, by the construction based on the inclusion-exclusion principle (cf., Observation 3), this expression further simplifies to $\text{pmc}_{\leq t'}(\text{Origins}(t, \sigma))$. By Definition 7, $\text{pmc}_{\leq t'}(\text{Origins}(t, \sigma)) = |\bigcup_{\vec{u} \in \text{Origins}(t, \sigma)} I_P(\text{SatExt}_{\leq t'}(\{\vec{u}\}))|$. However, since by construction of PROJ, $|\text{EqClasses}_P(\sigma)| = 1$, i.e., σ is contained in one equivalence class, we have $|\bigcup_{\vec{u} \in \text{Origins}(t, \sigma)} I_P(\text{SatExt}_{\leq t'}(\{\vec{u}\}))| = |\bigcup_{\vec{u} \in \sigma} I_P(\text{SatExt}_{\leq t}(\{\vec{u}\}))|$. This corresponds to $\text{pmc}_{\leq t}(\sigma)$ and, consequently, $\text{pmc}_{\leq t'}(\text{Origins}(t, \sigma)) = \text{pmc}_{\leq t}(\sigma)$. This concludes the proof for the second claim on pmc values.

The induction step for ipmc works similar. By Definition 9, we have $\text{ipmc}(t, \sigma, \text{PROJ-Comp}[t']) = |\text{pmc}(t, \sigma, \text{PROJ-Comp}[t'])| + \sum_{\emptyset \subsetneq \rho \subsetneq \sigma} (-1)^{|\rho|} \cdot \text{ipmc}(t, \rho, \text{PROJ-Comp}[t'])$. By the proof on the second claim above, $|\text{pmc}(t, \sigma, \text{PROJ-Comp}[t'])| = \text{pmc}_{\leq t}(\sigma)$. Then, by the induction hypothesis on ρ , we have $\text{ipmc}(t, \sigma, \text{PROJ-Comp}[t']) = \text{pmc}_{\leq t}(\sigma) + \sum_{\emptyset \subsetneq \rho \subsetneq \sigma} (-1)^{|\rho|} \cdot \text{ipmc}_{\leq t}(\rho)$. Further, we follow by Definition 7 that $\text{ipmc}(t, \sigma, \text{PROJ-Comp}[t'])$ corresponds to the expression $|\bigcup_{\vec{u} \in \sigma} I_P(\text{SatExt}_{\leq t}(\{\vec{u}\}))| + \sum_{\emptyset \subsetneq \rho \subsetneq \sigma} (-1)^{|\rho|} \cdot |\bigcap_{\vec{u} \in \rho} I_P(\text{SatExt}_{\leq t}(\{\vec{u}\}))|$. Finally, by Observation 3, this yields $|\bigcap_{\vec{u} \in \sigma} I_P(\text{SatExt}_{\leq t}(\{\vec{u}\}))|$, which simplifies to $\text{ipmc}_{\leq t}(\sigma)$. This concludes the proof for the first claim on ipmc values.

Case (ii): Assume that $\ell = 2$.

First, we show the induction step on the second claim over pmc. By Definition 8, we have $\text{pmc}(t, \sigma, \langle \text{PROJ-Comp}[t_1], \text{PROJ-Comp}[t_2] \rangle) = \sum_{\emptyset \subsetneq O \subseteq \text{Origins}(t, \sigma)} (-1)^{(|O|-1)} \cdot \text{s-ipmc}(\langle \text{PROJ-Comp}[t_1], \text{PROJ-Comp}[t_2] \rangle, O)$. This then results in $\sum_{\emptyset \subsetneq O \subseteq \text{Origins}(t, \sigma)} (-1)^{(|O|-1)} \cdot \text{ipmc}(t_1, O_{(1)}, \text{PROJ-Comp}[t_1]) \cdot \text{ipmc}(t_2, O_{(2)}, \text{PROJ-Comp}[t_2])$. By the induction hypothesis, this then evaluates to $\sum_{\emptyset \subsetneq O \subseteq \text{Origins}(t, \sigma)} (-1)^{(|O|-1)} \cdot \text{ipmc}_{\leq t_1}(O_{(1)}) \cdot \text{ipmc}_{\leq t_2}(O_{(2)})$. By expansion via Definition 7 and applying Observation 3, i.e., the inclusion-exclusion principle, this corresponds to $|\bigcup_{(\vec{u}_1, \vec{u}_2) \in \text{Origins}(t, \sigma)} I_P(\text{SatExt}_{\leq t_1}(\{\vec{u}_1\})) \cdot I_P(\text{SatExt}_{\leq t_2}(\{\vec{u}_2\}))|$. Since we have that $|\text{EqClasses}_P(\sigma)| = 1$, i.e., σ is contained in one equivalence class and by Definition 4 of SatExt, this expression simplifies to $|\bigcup_{\vec{u} \in \sigma} I_P(\text{SatExt}_{\leq t}(\{\vec{u}\}))|$. This corresponds to $\text{pmc}_{\leq t}(\sigma)$, which concludes the proof for pmc of Case (ii).

The induction step for ipmc also works analogously to the proof for ipmc

of Case (i), since it does not need to directly consider origins in multiple child nodes. This concludes the proof. \square

Lemma 4 (Soundness). *Let $t \in N$ be a node of the tree decomposition \mathcal{T} with $\text{children}(t) = \langle t_1, \dots, t_\ell \rangle$. Then, each row $\langle \tau, c \rangle$ at node t obtained by PROJ is a PROJ-row solution for t .*

Proof. Observe that Listing 5 computes a row for each subset σ with $\emptyset \subsetneq \sigma \subseteq C$ for some $C \in \text{EqClasses}_P(\text{SAT-Comp}[t])$. The resulting row $\langle \sigma, c \rangle$ obtained by ipmc is indeed a PROJ-row solution for t according to Lemma 3. \square

Lemma 5 (Completeness). *Let $t \in N$ be a node of tree decomposition \mathcal{T} where $\text{children}(t) = \langle t_1, \dots, t_\ell \rangle$ and $\text{type}(t) \neq \text{leaf}$. Given a PROJ-row solution $\langle \sigma, c \rangle$ at t . Then, there is $\langle C_1, \dots, C_\ell \rangle$ where each C_i is a set of PROJ-row solutions at t_i with $\sigma = \text{PROJ}(t, \cdot, \cdot, P, \langle C_1, \dots, C_\ell \rangle, \text{SAT-Comp})$.*

Proof. Since $\langle \sigma, c \rangle$ is a PROJ-row solution for t , there is by Definition 11 a corresponding PROJ-solution $\langle \hat{\sigma} \rangle$ up to t such that $\text{local}(t, \hat{\sigma}) = \sigma$. Then we define $\hat{\sigma}' := \{(t'', \hat{\rho}) \mid (t'', \hat{\rho}) \in \sigma, t'' \neq t\}$ and proceed again by case distinction.

Case (i): Assume that $\ell = 1$ and $t' = t_1$. For each subset $\emptyset \subsetneq \rho \subseteq \text{local}(t', \hat{\sigma}')$, we define $\langle \rho, |I_P(\text{SatExt}_{\leq t}(\rho))| \rangle$ in accordance with Definition 11. By Observation 2, we have that $\langle \rho, |I_P(\text{SatExt}_{\leq t}(\rho))| \rangle$ is a SAT-row solution at t' . Since we defined PROJ-row solutions for t' for all respective PROJ-solutions up to t , we encountered every PROJ-row solution for t' required for deriving $\langle \sigma, c \rangle$ via PROJ (cf., Definitions 8 and 9).

Case (ii): Assume that $\ell = 2$, i.e., t is a join node. Similarly to above, we define PROJ-row solutions at t_1 and t_2 . Analogously, we define for each subset $\emptyset \subsetneq \rho \subseteq \text{local}(t_1, \hat{\sigma}')$, a PROJ-row solution $\langle \rho, |I_P(\text{SatExt}_{\leq t_1}(\rho))| \rangle$ at t_1 . Additionally, for each subset $\emptyset \subsetneq \rho \subseteq \text{local}(t_2, \hat{\sigma}')$, we construct a PROJ-row solution $\langle \rho, |I_P(\text{SatExt}_{\leq t_2}(\rho))| \rangle$ at t_2 in accordance with Definition 11. By Observation 2, we have that these constructed rows are indeed a SAT-row solution at t_1 and a SAT-row solution at t_2 , respectively. Since also for this case we defined PROJ-row solutions for t_1 and t_2 for all respective PROJ-solutions up to t , we encountered every PROJ-row solution for t_1 and t_2 required for deriving $\langle \sigma, c \rangle$ via PROJ. This concludes the proof. \square

Theorem 1. *The algorithm DP_{PROJ} is correct. More precisely, $\text{DP}_{\text{PROJ}}((F, P), \mathcal{T}, \text{SAT-Comp})$ returns tables PROJ-Comp such that $p = \sum_{\langle \sigma, c \rangle \in \text{SAT-Comp}[n]} c$ is the projected model count of F with respect to the set P of projection variables.*

Proof. By Lemma 4 we have soundness for every node $t \in N$ and hence only valid rows as output of table algorithm PROJ when traversing the tree decomposition in post-order up to the root n . By Lemma 2 we know that the projected model count p of F is larger than zero if and only if there exists a certain PROJ-row solution for n . This PROJ-row solution at node n is of the form $\langle \{\emptyset, \dots\}, p \rangle$. If there is no PROJ-row solution at node n , then $\text{SAT-Comp}[n] = \emptyset$ since the table algorithm SAT is correct (cf., Proposition 1). Consequently, we have $p = 0$. Therefore, $p = \sum_{\langle \sigma, c \rangle \in \text{SAT-Comp}[n]} c$ is the pmc of F w.r.t. P in both cases.

Next, we establish completeness by induction starting from root n . Let therefore, $\langle \hat{\sigma} \rangle$ be the PROJ-solution up to n , where for each row in $\vec{u} \in \hat{\sigma}$, $I(\vec{u})$ corresponds to a model of F . By Definition 11, we know that for n we can construct a PROJ-row solution at n of the form $\langle \{\langle \emptyset, \dots \rangle\}, p \rangle$ for $\hat{\sigma}$. We already established the induction step in Lemma 5. Hence, we obtain some row for every node t . Finally, we stop at the leaves. \square

Corollary 2. *The algorithm PCNT_{SAT} is correct, i.e., PCNT_{SAT} solves PMC.*

Proof. The result follows, since PCNT_{SAT} consists of pass DP_{SAT}, a purging step and DP_{PROJ}. For correctness of DP_{SAT} we refer to other sources [56, 31]. By Proposition 1, “purging” neither destroys soundness nor completeness of DP_{SAT}. \square

4.3. Runtime Analysis (Upper and Lower Bounds)

In this section, we first present asymptotic upper bounds on the runtime of our Algorithm DP_{PROJ}. For the analysis, we assume $\gamma(i)$ to be the costs for multiplying two i -bit integers, which can be achieved in time $i \cdot \log(i) \cdot \log(\log(i))$ [61, 62]. Recently, an even faster algorithm was published [62].

Then, we present a lower bound that establishes that there cannot be an algorithm that solves PMC in time that is only single exponential in the treewidth and polynomial in the size of the formula unless the exponential time hypothesis (ETH) fails. This result establishes that there *cannot* be an algorithm exploiting treewidth that is *asymptotically better* than our presented algorithm, although one can likely improve on the analysis and give a better algorithm. One could for example cache pmc values, which, however, overcomplicates worst-case analysis.

Theorem 2. *Given a PMC instance (F, P) and a tree decomposition $\mathcal{T} = (T, \chi)$ of F of width k with g nodes. Algorithm DP_{PROJ} runs in time $\mathcal{O}(2^{2^{k+4}} \cdot \gamma(\|F\|) \cdot g)$.*

Proof. Let $d = k + 1$ be maximum bag size of \mathcal{T} . For each node t of T , we consider table $\tau = \text{SAT-Comp}[t]$ which has been computed by DP_{SAT} [31]. The table τ has at most 2^d rows. In the worst case we store in $\iota = \text{PROJ-Comp}[t]$ each subset $\sigma \subseteq \tau$ together with exactly one counter. Hence, we have 2^{2^d} many rows in ι . In order to compute ipmc for σ , we consider every subset $\rho \subseteq \sigma$ and compute pmc. Since $|\sigma| \leq 2^d$, we have at most 2^{2^d} many subsets ρ of σ . For computing pmc, there could be each subset of the origins of ρ for each child table, which are less than $2^{2^{d+1}} \cdot 2^{2^{d+1}}$ (join and remove case). In total, we obtain a runtime bound of $\mathcal{O}(2^{2^d} \cdot 2^{2^d} \cdot 2^{2^{d+1}} \cdot 2^{2^{d+1}} \cdot \gamma(\|F\|)) \subseteq \mathcal{O}(2^{2^{d+3}} \cdot \gamma(\|F\|))$ since we also need multiplication of counters. Then, we apply this to every node t of the tree decomposition, which results in running time $\mathcal{O}(2^{2^{d+3}} \cdot \gamma(\|F\|) \cdot g)$. \square

Corollary 3. *Given an instance (F, P) of PMC where F has treewidth k . Algorithm PCNT_{SAT} runs in time $\mathcal{O}(2^{2^{k+4}} \cdot \gamma(\|F\|) \cdot \|F\|)$.*

Proof. We compute in time $2^{\mathcal{O}(k^3)} \cdot |V|$ a tree decomposition \mathcal{T}' of width at most k [54] of primal graph G_F . Then, we run a decision version of the algorithm DP_{SAT} by Samer and Szeider [31] in time $\mathcal{O}(2^k \cdot \gamma(\|F\|) \cdot \|F\|)$. Then, we again traverse the decomposition, thereby keeping rows that have a satisfying extension (“purging”), in time $\mathcal{O}(2^k \cdot \|F\|)$. Finally, we run DP_{PROJ} and obtain the claim by Theorem 2 and since \mathcal{T}' has linearly many nodes [54]. \square

The next results also establish the lower bounds for our worst-cases.

Theorem 3. *Unless ETH fails, PMC cannot be solved in time $2^{2^{o(k)}} \cdot \|F\|^{o(k)}$ for a given instance (F, P) where k is the treewidth of the primal graph of F .*

Proof. Assume for a proof by contradiction that there is such an algorithm. We show that this contradicts a recent result [63, Theorem 13], which states that one cannot decide the validity of a quantified Boolean formula $Q = \forall V_1. \exists V_2. E$ in time $2^{2^{o(k)}} \cdot \|E\|^{o(k)}$ under ETH. A version of this result for formulas in disjunctive normal form appeared earlier [43]. Given an instance (Q, k) of $\forall\exists$ -SAT when parameterized by the treewidth k of E , we provide a reduction to an instance $((F, P, n), k)$ of decision version PMC-exactly- n of PMC such that $F = E$, $P = V_1$, and the number n of solutions is exactly $2^{|V_1|}$. The reduction is in fact an fpt-reduction, since the treewidth of F is exactly k . It is easy to see that the reduction gives a yes instance $((F, P, n), k)$ of PMC-exactly- n if and only if $(\forall V_1. \exists V_2. E, k)$ is a yes instance of $\forall\exists$ -SAT. Assume towards a contradiction that $((F, P, n), k)$ is a yes-instance of PMC-exactly- n , but $\forall V_1. \exists V_2. E$ evaluates to false. Then, there is an assignment $\alpha : V_1 \rightarrow \{0, 1\}$ such that $E[\alpha]$ evaluates to false, which contradicts that the projected model count of F with respect to P is $2^{|V_1|}$. In the other direction, assume that $\forall V_1. \exists V_2. E$ evaluates to true, but the projected model count of F and P is $< n$. This, however, contradicts that $\forall V_1. \exists V_2. E$ evaluates to true, which concludes the proof. \square

Corollary 4. *Given any instance (F, P) of PMC where F has treewidth k . Then, under ETH, PMC requires runtime $2^{2^{\Theta(k)}} \cdot \text{poly}(\|F\|)$.*

5. Towards Efficiently Utilizing Treewidth for PMC

Although the tables obtained via table algorithms might be exponential in size, the size is bounded by the width of the given TD of the primal graph G_F of a formula F . Still, practical results of such algorithms show competitive behaviour [64, 40] up to a certain width. As a result, instances with high (tree)width seem out of reach. Even further, as we have shown above, lifting the table algorithm SAT in order to solve problem PMC results in an algorithm that is double exponential in the treewidth.

To mitigate these issues and to enable practical implementations, we present a novel approach to deal with high treewidth, by nesting of DP on graph simplifications (abstractions) of G_F . These abstractions are discussed in Section 5.1 and the basis for nested DP is presented in Section 5.2. As we will see, nested

dynamic programming not only works for #SAT, but also for PMC with adaptations. Finally, Section 5.3 concerns about hybrid dynamic programming, which is a further extension of nested DP. More concretely, hybrid DP tries to combine the best of the two worlds (i) *dynamic programming* and (ii) *applying standard, search-based solvers*, where DP provides the basic structure guidance and delegates hard subproblems that occur during solving to these standard solvers.

5.1. Abstractions are key

In the following, we discuss certain graph simplifications (called abstractions) of the primal graph in the context of the Boolean satisfiability problem, namely for the problem #SAT. Afterwards we generalize the usage of these abstraction to nested dynamic programming for PMC.

To this end, let F be a Boolean formula. Now, assume the situation that a set U of variables of F , called *nesting variables*, appears *uniquely* in the bag of exactly one TD node t of a tree decomposition of G_F . Then, observe that one could do dynamic programming on the tree decomposition as explained in Section 3.1, but no truth value for any variable in U requires to be stored. Instead, clauses of F over variables U could be evaluated within node t , since variables U appear uniquely in the node t . Indeed, for dynamic programming on the non-nesting variables, only the result of this evaluation is essential, as variables U appear uniquely within $\chi(t)$.

Before we can apply nested DP, we require a formal account of abstractions with room for choosing nesting variables between the empty set and the set of all the variables. Let F be a Boolean formula and recall the primal graph $G_F = (\text{var}(F), E)$ of F . Inspired by related work [65, 66, 67, 68], we define the *nested primal graph* G_F^A for a given formula F and a given set $A \subseteq \text{var}(F)$ of variables, referred to by *abstraction variables*. To this end, we say a path P in primal graph G_F is a *nesting path* (between u and v) using A , if $P = u, v_1, \dots, v_\ell, v$ ($\ell \geq 0$), and every vertex v_i is a *nesting variable*, i.e., $v_i \notin A$ for $1 \leq i \leq \ell$. Note that any path in G_F is nesting using A if $A = \emptyset$. Then, the vertices of nested primal graph G_F^A correspond to A and there is an edge between two distinct vertices $u, v \in A$ if there is a nesting path between u and v .

Definition 12. Let F be a Boolean formula and $A \subseteq \text{var}(F)$ be a set of variables. Then, the nested primal graph G_F^A is defined by $G_F^A := (A, \{\{u, v\} \mid u, v \in A, u \neq v, \text{ there is a nesting path in } G_F \text{ between } u \text{ and } v\})$.

Observe that the nested primal graph only consists of abstraction variables and, intuitively, “hides” nesting variables of nesting paths of primal graph G_F . Even further, the connected components of $G_F - A$ are hidden in the nested primal graph G_F^A by means of cliques among A .

Example 9. Recall formula $F := \{\overbrace{\{\neg a, b, p_1\}}^{c_1}, \overbrace{\{a, \neg b, \neg p_1\}}^{c_2}, \overbrace{\{a, p_2\}}^{c_3}, \overbrace{\{a, \neg p_2\}}^{c_4}\}$ and primal graph G_F of Example 1, which is visualized in Figure 6 (left). Given abstraction variables $A = \{a, b\}$, nesting paths of G_F are, e.g., $P_1 = a$, $P_2 = a, p_2$, $P_3 = p_2, a$, $P_4 = a, b$, $P_5 = a, p_1, b$. However, neither path $P_6 = b, a, p_2$, nor path $P_7 =$

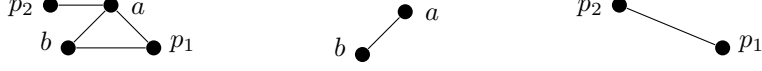


Figure 6: Primal graph G_F of F from Example 1 (left), nested primal graph $G_F^{\{a,b\}}$ (middle), as well as nested primal graph $G_F^{\{p_1,p_2\}}$ (right).

p_2, a, b, p_1 is nesting using A . Nested primal graph G_F^A is shown in Figure 6 (middle) and it contains an edge $\{a, b\}$ over the vertices in A due to, e.g., paths P_4, P_5 . Assume a different set $A' = \{p_1, p_2\}$. Observe that $G_F^{A'}$ as depicted in Figure 6 (right) consists of the vertices A' and there is an edge between p_1 and p_2 due to, e.g., nesting path $P' = p_1, a, p_2$ using A' .

The nested primal graph provides abstractions of needed flexibility for nested DP. Indeed, if we set abstraction variables to $A = \text{var}(F)$, we end up with full dynamic programming and zero nesting, whereas setting $A = \emptyset$ results in full nesting, i.e., nesting of all variables. Intuitively, the nested primal graph ensures that clauses subject to nesting (containing nesting variables) can be safely evaluated in exactly one node of a tree decomposition of the nested primal graph.

To formalize this, we assume a tree decomposition $\mathcal{T} = (T, \chi)$ of G_F^A and say a set $U \subseteq \text{var}(F)$ of variables is *compatible* with a node t of T , and vice versa, if

- (I) U is a connected component of the graph $G_F - A$, which is obtained from primal graph G_F by removing A and
- (II) all neighbor vertices of U that are in A are contained in $\chi(t)$, i.e., $\{a \mid a \in A, u \in U, \text{ there is a nesting path from } a \text{ to } u \text{ using } A\} \subseteq \chi(t)$.

If such a set $U \subseteq \text{var}(F)$ of variables is compatible with a node of T , we say that U is a *compatible set*. By construction of the nested primal graph, any nesting variable is in at least one compatible set. However, a compatible set could be compatible with several nodes of T . Hence, to enable nested evaluation in general, we need to ensure that each nesting variable is evaluated only in one unique node t .

As a result, we formalize for every compatible set U , a *unique* node t of T that is compatible with U , denoted by $\text{comp}_{F,A,\mathcal{T}}(U) := t$. We denote the union of all compatible sets U with $\text{comp}_{F,A,\mathcal{T}}(U) = t$, by *nested bag variables* $\chi_t^A := \bigcup_{U: \text{comp}_{F,A,\mathcal{T}}(U)=t} U$. Then, the *nested bag formula* F_t^A for a node t of T equals $F_t^A := \{c \mid c \in F, \text{var}(c) \subseteq \chi(t) \cup \chi_t^A\} \setminus F_t$, where the bag formula F_t is defined as in the beginning of Section 3. Observe that the definition of nested bag formulas ensures that any connected component U of $G_F - A$ “appears” among nested bag variables of some unique node of T . Consequently, each variable $a \in \text{var}(F) \setminus A$ appears *only* in one nested bag formula F_t^A of a node t of T that is unique for a .

Example 10. Recall formula F , the tree decomposition $\mathcal{T} = (T, \chi)$ of G_F , as depicted in Figure 7 (left), and abstraction variables $A = \{a, b\}$ of Example 9. Consider TD $\mathcal{T}' := (T, \chi')$, where $\chi'(t) := \chi(t) \cap \{a, b\}$ for each node t of T , which is given in Figure 7 (right). Observe that \mathcal{T}' is \mathcal{T} , but restricted to A and that

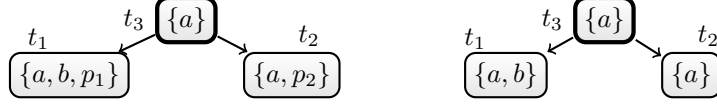


Figure 7: TD \mathcal{T} (left) of the primal graph G_F of Figure 6, and a TD \mathcal{T}' (right) of nested primal graph $G_F^{\{a,b\}}$.

Listing 6: Algorithm $\text{NestDP}_{\mathbb{N}}(\text{depth}, (F, P), A, \mathcal{T})$ for computing solutions of (F, P) via nested DP on TD \mathcal{T} .

In: Nested table algorithm \mathbb{N} , nesting depth ≥ 0 , instance (F, P) of PMC, abstraction variables $A \subseteq \text{var}(F)$, and a TD $\mathcal{T} = (T, \chi)$ of the nested primal graph G_F^A .

Out: Table mapping $\mathbb{N}\text{-Comp}$, which maps each TD node t of T to some computed table τ_t .

```

1  $\mathbb{N}\text{-Comp} \leftarrow \{\}$  /* empty mapping */
2 for iterate  $t$  in post-order( $T$ ) do
3    $\text{Child-Tabs} \leftarrow \langle \mathbb{N}\text{-Comp}[t_1], \dots, \mathbb{N}\text{-Comp}[t_\ell] \rangle$  where  $\text{children}(t) = \langle t_1, \dots, t_\ell \rangle$ 
4    $\mathbb{N}\text{-Comp}[t] \leftarrow \mathbb{N}(\text{depth}, t, \chi(t), F_t, F_t^A, P \cap \text{var}(F_t^A), \text{Child-Tabs})$ 
5 return  $\mathbb{N}\text{-Comp}$ 

```

\mathcal{T}' is a TD of G_F^A of width 1. There are two compatible sets, namely $\{p_1\}$ and $\{p_2\}$. Observe that only for compatible set $U = \{p_2\}$ we have two nodes compatible with U , namely t_2 and t_3 . We assume that $\text{comp}_{F,A,\mathcal{T}'}(U) = t_2$, i.e., we decide that t_2 shall be the unique node for U . Consequently, nested bag formulas are $F_{t_1}^A = \{c_1, c_2\}$, $F_{t_2}^A = \{c_3, c_4\}$, and $F_{t_3}^A = \emptyset$.

5.2. Nested Dynamic Programming on Abstractions

Now, we have established required notation in order to discuss *nested dynamic programming (nested DP)*. Listing 6 presents algorithm NestDP for solving a given problem by means of nested dynamic programming. Observe that Listing 6 is almost identical to algorithm DP as presented in Listing 1. The reason for this is that nested dynamic programming can be seen as a refinement of dynamic programming, cf. algorithm DP of Listing 1. Indeed, the difference of NestDP compared to DP is that NestDP uses labeled tree decompositions of the nested primal graph and that it gets as additional parameter a set A of abstraction variables. Further, instead of a table algorithm \mathbb{A} , algorithm NestDP relies on a *nested table algorithm* \mathbb{N} during dynamic programming, which is similar to a table algorithm that gets as additional parameter an integer depth ≥ 0 that will be used later and a nested bag instance that needs to be evaluated. For simplicity and generality, also the formula is passed as a parameter, which is, however, used only for passing problem-specific information of the instance. Indeed, most nested table algorithm do not require this parameter, which should not be used for direct problem solving instead of utilizing the bag instance. Consequently, nested dynamic programming still follows the basic concept of dynamic programming as presented in Figure 2.

Listing 7: Nested table algorithm $\text{NSAT}(\cdot, t, \chi_t, F_t, F_t^A, \cdot, \text{Child-Tabs})$ for solving $\#SAT$.

In: Node t , bag χ_t , bag formula F_t , nested bag formula F_t^A , and sequence $\text{Child-Tabs} = \langle \tau_1, \dots, \tau_\ell \rangle$ of child tables of t .

Out: Table τ_t .

```

1 if type( $t$ ) = leaf then  $\tau_t \leftarrow \{\langle \emptyset, 1 \rangle\}$ 
2 else if type( $t$ ) = int and  $a \in \chi(t)$  is introduced then
3    $\tau_t \leftarrow \{\langle \langle J, c' \cdot c \rangle \mid \langle I, c \rangle \in \tau_1, J \in I \cup \{a \mapsto v\}, v \in \{0, 1\}, J \models F_t, c' > 0, c' = \#SAT(F_t^A[J]) \rangle\}$ 
4 else if type( $t$ ) = rem and  $a \notin \chi(t)$  is removed then
5   /*  $C(I)$  is the set that contains the rows in  $\tau_1$  for assignments  $J$ 
   that are equal to  $I$  after removing  $a$  */
6    $C(I) \leftarrow \{\langle J, c \rangle \mid \langle J, c \rangle \in \tau_1, J \setminus \{a \mapsto 0, a \mapsto 1\} = I \setminus \{a \mapsto 0, a \mapsto 1\}\}$ 
7    $\tau_t \leftarrow \{\langle \langle I \setminus \{a \mapsto 0, a \mapsto 1\} \rangle, \sum_{\langle J, c \rangle \in C(I)} c \rangle \mid \langle I, \cdot \rangle \in \tau_1\}$ 
7 else if type( $t$ ) = join then
8    $\tau_t \leftarrow \{\langle \langle I, c_1 \cdot c_2 \rangle \mid \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2 \rangle\}$ 
9 return  $\tau_t$ 

```

Similar to above, for the ease of presentation our presented nested table algorithms use nice tree decompositions only. However, this is not a hard restriction. Indeed, it is easy to see that for arbitrary TDs the clear case distinctions of nice decompositions are still valid, but are in general just overlapping. Further, without loss of generality we also assume that each compatible set U gets assigned a unique node $t = \text{comp}_{F,A,\mathcal{T}}(U)$ that is an *introduce node*, i.e., $\text{type}(t) = \text{int}$.

Nested Dynamic Programming for $\#SAT$. In order to design a nested table algorithm for $\#SAT$, assume a Boolean formula F as well as a given labeled tree decomposition $\mathcal{T} = (T, \chi)$ of G_F^A using any set A of abstraction variables. Recall from the discussions above, that each variable $a \in \text{var}(F) \setminus A$ appears *only* in one nested bag formula F_t^A of a node t of T that is unique for a . These unique variable appearances allow us to actually nest the evaluation of nested bag formula F_t^A . This evaluation is performed by a nested table algorithm NSAT in the context of nested dynamic programming. Listing 7 shows this simple nested table algorithm NSAT for solving problem $\#SAT$ by means of algorithm $\text{NestDP}_{\text{NSAT}}$. For comparison, recall table algorithm SAT for solving problem $\#SAT$ by means of dynamic programming, as given by Listing 3. Observe that in contrast to Listing 3, we store here assignments (and not interpretations), which simplifies the presentation of nesting. However, the main difference of NSAT compared to SAT is that the nested table algorithm NSAT maintains a counter c and that it gets called on a nested primal graph, i.e., the algorithm gets additional parameters like the nested bag formula. Then, the nested table algorithm evaluates this nested bag formula in Line 3 via any procedure $\#SAT$ for solving problem $\#SAT(F_t^A[J])$ on the nested bag formula F_t^A simplified by the current assignment J to variables in the bag $\chi(t)$. Note that this subproblem $\#SAT(F_t^A[J])$ itself can be solved by again using nested dynamic programming with the help of algorithm $\text{NestDP}_{\text{NSAT}}$.

In the following, we briefly show the evaluation of nested dynamic program-

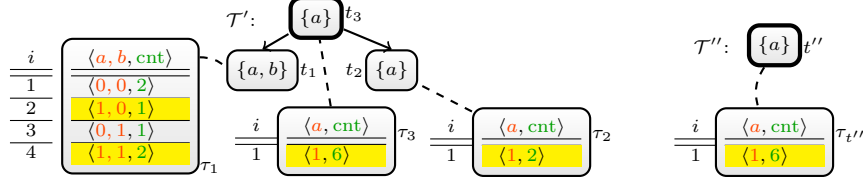


Figure 8: Selected tables obtained by nested DP on TD \mathcal{T}' of $G_F^{a,b}$ (left) and on TD \mathcal{T}'' of G_F^{a} (right) for F of Example 10 via $\text{NestDP}_{\text{NSAT}}$.

ming for $\#SAT$ on an example.

Example 11. Recall formula F , set A of abstraction variables, and TD \mathcal{T}' of nested primal graph G_F^A given in Example 10. As already mentioned, Formula F has six satisfying assignments, namely $\{a \mapsto 1, b \mapsto 0, p_1 \mapsto 1, p_2 \mapsto 0\}$, $\{a \mapsto 1, b \mapsto 0, p_1 \mapsto 1, p_2 \mapsto 1\}$, $\{a \mapsto 1, b \mapsto 1, p_1 \mapsto 0, p_2 \mapsto 0\}$, $\{a \mapsto 1, b \mapsto 1, p_1 \mapsto 0, p_2 \mapsto 1\}$, $\{a \mapsto 1, b \mapsto 1, p_1 \mapsto 1, p_2 \mapsto 0\}$, and $\{a \mapsto 1, b \mapsto 1, p_1 \mapsto 1, p_2 \mapsto 1\}$.

Figure 8 (left) shows TD \mathcal{T}' of G_F^A and tables obtained by $\text{NestDP}_{\text{NSAT}}(0, (F, \cdot), A, \mathcal{T}')$ for model counting ($\#SAT$) on F . We briefly discuss executing NSAT on \mathcal{T}' , resulting in tables τ_1 , τ_2 , and τ_3 as shown in Figure 8 (left). Intuitively, table τ_1 is the result of introducing variables a and b . Recall from Example 10 that $F_{t_1}^A = \{c_1, c_2\}$ with $c_1 = \{\neg a, b, p_1\}$ and $c_2 = \{a, \neg b, \neg p_1\}$. Then, in Line 3 of algorithm NSAT, for each assignment I to $\{a, b\}$ of each row r of τ_1 , we compute $\#SAT(F_{t_1}^A[I])$. Consequently, for assignment $I_1 = \{a \mapsto 0, b \mapsto 0\}$, we have that there are two satisfying assignments of $F_{t_1}^A[I_1]$, namely $\{p_1 \mapsto 0\}$ and $\{p_1 \mapsto 1\}$. Indeed, this count of 2 is obtained for the first row of table τ_1 by Line 3. Analogously, one can derive the remaining tables of τ_1 and one obtains table τ_2 similarly, by using formula $F_{t_2}^A$. Then, table τ_3 is the result of removing b in node t_1 and combining agreeing assignments of rows accordingly. Consequently, we obtain that there are six satisfying assignments of F , which are all required to set a to 1 due to formula $F_{t_2}^A$ that is evaluated in node t_2 .

Figure 8 (right) shows TD \mathcal{T}'' of G_F^{a} and tables obtained by $\text{NestDP}_{\text{NSAT}}(0, (F, \cdot), \{a\}, \mathcal{T}'')$ using TD \mathcal{T}'' . Since $F_{t''}^{a} = F$ and $F[\{a \mapsto 0\}]$ is unsatisfiable, table $\tau_{t''}$ does not contain an entry corresponding to assignment $\{a \mapsto 0\}$, cf. Condition “ $c' > 0$ ” in Line 3 of Listing 7. Thus, there are six satisfying assignments of $F_{t''}^{a}[\{a \mapsto 1\}]$ obtained by computing $\#SAT(F_{t''}^{a}[\{a \mapsto 1\}])$.

While the overall concept of nested dynamic programming as given by algorithm NestDP of Listing 6 is quite general, sometimes in practice it is sufficient to further restrict the set of choices for abstraction vertices A when constructing the nested primal graph.

Nested Table Algorithm for PMC. To this end, we show the approach of nested dynamic programming for the problem PMC.

Example 12. Recall formula F as well as set $A = \{a, b\}$ of abstraction variables from Example 10. Then, we have that (F, A) is an instance of the projected model

Listing 8: Nested table algorithm $\text{NPMC}(\cdot, t, \chi_t, F_t, F_t^A, P, \text{Child-Tabs})$ for solving PMC.

In: Node t , bag χ_t , bag formula F_t , nested bag formula F_t^A , projection variables $P \subseteq \text{var}(F_t^A)$, and sequence $\text{Child-Tabs} = \langle \tau_1, \dots, \tau_\ell \rangle$ of child tables of t .
Out: Table τ_t .

```

1 if type( $t$ ) = leaf then  $\tau_t \leftarrow \{\langle \emptyset, 1 \rangle\}$ 
2 else if type( $t$ ) = int and  $a \in \chi(t)$  is introduced then
3    $\tau_t \leftarrow \{\langle J, c' \cdot c \rangle \mid \langle I, c \rangle \in \tau_1, = I \cup \{a \mapsto v\}, v \in \{0, 1\}, J \models F_t, c' > 0, c' =$ 
                                      $\text{PMC}(F_t^A[J], P)\}$ 
4 else if type( $t$ ) = rem and  $a \notin \chi(t)$  is removed then
5    $C(I) \leftarrow \{\langle J, c \rangle \mid \langle J, c \rangle \in \tau_1, J \setminus \{a \mapsto 0, a \mapsto 1\} = I \setminus \{a \mapsto 0, a \mapsto 1\}\}$ 
6    $\tau_t \leftarrow \{\langle I \setminus \{a \mapsto 0, a \mapsto 1\}, \sum_{\langle J, c \rangle \in C(I)} c \rangle \mid \langle I, \cdot \rangle \in \tau_1\}$ 
7 else if type( $t$ ) = join then
8    $\tau_t \leftarrow \{\langle I, c_1 \cdot c_2 \rangle \mid \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$ 
9 return  $\tau_t$ 
```

counting problem PMC. Restricted to projection set A , the Boolean formula F has two satisfying assignments, namely $\{a \mapsto 1, b \mapsto 0\}$ and $\{a \mapsto 1, b \mapsto 1\}$. Consequently, the solution to PMC on (F, A) , i.e., $\text{PMC}(F, A)$, is 2.

Indeed, for solving projected model counting we mainly focus on the case, where for a given instance (F, P) with Boolean formula F of problem PMC, the abstraction variables A that are used for constructing the *nested primal graph* G_F^A are among the projection variables, i.e., $A \subseteq P$. The approach of nested DP can then be applied for solving projected model counting such that the nested table algorithm naturally extends algorithm NSAT of Listing 7.

The nested table algorithm NPMC for solving projected model counting via nested dynamic programming is presented in Listing 8. Observe that nested table algorithm NPMC does not significantly differ from algorithm NSAT due to $A \subseteq P$. Indeed, the main difference is only in Line 3 of Listing 7, where instead of a procedure for model counting, a procedure PMC for solving a projected model counting question is called.

5.3. Hybrid Dynamic Programming based on nested DP

Now, we have definitions at hand to further refine and discuss nested dynamic programming in the context of *hybrid dynamic programming (hybrid DP)*, which combines using both standard solvers and parameterized solvers exploiting treewidth in the form of nested dynamic programming. We illustrate these ideas for the problem PMC next. Afterwards we discuss how to implement the resulting algorithms in order to efficiently solve PMC and #SAT by means of database management systems.

Listing 9 depicts our algorithm $\text{HybDP}_{\text{HPMC}}$ for solving projected model counting, i.e., problem PMC. This algorithm $\text{HybDP}_{\text{HPMC}}$ takes an instance (F, P) of PMC consisting of Boolean formula F and projection variables P . The algorithm maintains a global, but simple cache mapping a formula to an integer, and consists of the following four subsequent blocks of code, which are separated

Listing 9: Algorithm $\text{HybDP}_{\text{HPMC}}(\text{depth}, F, P)$ for hybrid DP of PMC based on nested DP.

In: Nesting depth ≥ 0 and an instance (F, P) of PMC.

Out: Number $\text{PMC}(F, P)$ of assignments.

```

1  $(F', P') = \text{Preprocessing}(F, P)$ 
2  $A \leftarrow P'$ 
3 if  $F' \in \text{dom}(\text{cache})$  /*Cache Hit*/ then return  $\text{cache}(F') \cdot 2^{|P \setminus P'|}$ 
4 if  $P' = \emptyset$  then return  $\text{SAT}(F') \cdot 2^{|P|}$ 
5  $\mathcal{T} = (T, \chi) \leftarrow \text{Decompose\_via\_Heuristics}(G_{F'}^A)$  /* Decompose */
6 if  $\text{width}(\mathcal{T}) \geq \text{threshold}_{\text{hybrid}}$  or  $\text{depth} \geq \text{threshold}_{\text{depth}}$  /* Standard Solver */
   then
7   if  $\text{var}(F') = P'$  then  $\text{cache} \leftarrow \text{cache} \cup \{(F', \# \text{SAT}(F'))\}$ 
8   else  $\text{cache} \leftarrow \text{cache} \cup \{(F', \text{PMC}(F', P'))\}$ 
9   return  $\text{cache}(F') \cdot 2^{|P \setminus P'|}$ 
10 if  $\text{width}(\mathcal{T}) \geq \text{threshold}_{\text{abstr}}$  /* Abstract via Heuristics & Decompose */ then
11    $A \leftarrow \text{Choose\_Subset\_via\_Heuristics}(A, F')$ 
12    $\mathcal{T} = (T, \chi) \leftarrow \text{Decompose\_via\_Heuristics}(G_{F'}^A)$ 
13  $\text{N-Comp} \leftarrow \text{NestDP}_{\text{HPMC}}(\text{depth}, (F', P'), A, \mathcal{T})$  /* Nested Dynamic Programming */
14  $\text{cache} \leftarrow \text{cache} \cup \{(F', c) \mid \langle \emptyset, c \rangle \in \text{N-Comp}[\text{root}(T)]\}$ 
15 return  $\text{cache}(F') \cdot 2^{|P \setminus P'|}$ 

```

by empty lines: (1) Preprocessing & Cache Consolidation, (2) Standard Solving, (3) Abstraction & Decomposition, and (4) Nested Dynamic Programming, which causes an indirect recursion through nested table algorithm HPMC , as discussed later.

Block (1) spans Lines 1-3 and performs simple preprocessing techniques [69] like *unit propagation*, thereby obtaining a simplified instance (F', P') , where simplified formula F' of F and projection variables $P' \subseteq P$ are obtained. Any preprocessing simplifications are fine, as long as the solution of the resulting PMC instance (F', P') is the same as solving PMC on (F, P) . Then, in Line 2, we set the set A of abstraction variables to P' , and consolidate cache with the updated formula F' . Note that the operations in Line 1 are required to return a simplified instance that preserves satisfying assignments of the original formula when restricted to P . If F' is not cached, in Block (2), we do standard solving if the width is out-of-reach for nested DP, which spans over Lines 4-9. More precisely, if the updated formula F' does not contain projection variables, in Line 4 we employ a SAT solver returning integer 1 or 0. If F' contains projection variables and either the width obtained by heuristically decomposing $G_{F'}$ is above $\text{threshold}_{\text{hybrid}}$, or the nesting depth exceeds $\text{threshold}_{\text{depth}}$, we use a standard $\# \text{SAT}$ or PMC solver depending on P' .

Block (3) spans Lines 10-12 and is reached if no cache entry was found in Block (1) and standard solving was skipped in Block (2). If the width of the computed decomposition is above $\text{threshold}_{\text{abstr}}$, we need to use an abstraction in form of the nested primal graph. This is achieved by choosing suitable subsets $E \subseteq A$ of abstraction variables and decomposing F_t^E heuristically.

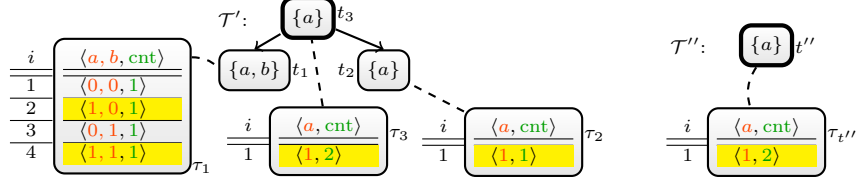


Figure 9: Selected tables obtained by nested DP using **NestDP_{HPMC}** on TD \mathcal{T}' of $G_F^{\{a,b\}}$ (left) and on TD \mathcal{T}'' of $G_F^{\{a\}}$ (right) for instance $(F, \{a, b\})$ of Example 12.

Listing 10: Nested table algorithm **HPMC**(depth, t , χ_t , F_t , F_t^A , P , Child-Tabs) for solving PMC.

In: Nesting depth ≥ 0 , node t , bag χ_t , bag formula F_t , nested bag formula F_t^A , projection variables $P \subseteq \text{var}(F_t^A)$, and sequence Child-Tabs = $\langle \tau_1, \dots, \tau_\ell \rangle$ of child tables of t .

Out: Table τ_t .

```

1 if type( $t$ ) = leaf then  $\tau_t \leftarrow \{\langle \emptyset, 1 \rangle\}$ 
2 else if type( $t$ ) = int and  $a \in \chi(t)$  is introduced then
3    $\tau_t \leftarrow \{\langle J, c' \cdot c \rangle \mid \langle I, c \rangle \in \tau_1, J = I \cup \{a \mapsto v\}, v \in \{0, 1\}, J \models F_t, c' > 0, c' =$ 
       $\text{HybDP}_{\text{HPMC}}(\text{depth} + 1, F_t^A[J], P)\}$ 
4 else if type( $t$ ) = rem and  $a \notin \chi(t)$  is removed then
5    $C(I) \leftarrow \{\langle J, c \rangle \mid \langle J, c \rangle \in \tau_1, J \setminus \{a \mapsto 0, a \mapsto 1\} = I \setminus \{a \mapsto 0, a \mapsto 1\}\}$ 
6    $\tau_t \leftarrow \{\langle I \setminus \{a \mapsto 0, a \mapsto 1\}, \sum_{\langle J, c \rangle \in C(I)} c \rangle \mid \langle I, \cdot \rangle \in \tau_1\}$ 
7 else if type( $t$ ) = join then
8    $\tau_t \leftarrow \{\langle I, c_1 \cdot c_2 \rangle \mid \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$ 
9 return  $\tau_t$ 

```

Finally, Block (4) concerns nested DP, cf. Lines 13-15. This block relies on nested table algorithm **HPMC**, which is given in Listing 10 that is almost identical to nested table algorithm **NPMC** as already discussed above and given in Listing 8. The only difference of **HPMC** compared to **NPMC** is that in Line 3 the nested table algorithm **HPMC** uses the parameter depth and recursively executes algorithm **HybDP_{HPMC}** on the increased nesting depth of depth+1, and the same formula as the one used in the generic PMC oracle call in Line 3 of Listing 8.

As a result, our approach deals with high treewidth by recursively finding and decomposing abstractions of the graph. If the treewidth is too high for some parts, tree decompositions of abstractions are used to guide standard solvers. Towards defining an actual implementation for practical solving, one still needs to find values for the threshold constants $\text{threshold}_{\text{hybrid}}$, $\text{threshold}_{\text{depth}}$, and $\text{threshold}_{\text{abstr}}$. The actual values of these constants will be made more precisely in the next section when discussing our implementation and experiments.

Example 13. Recall instance (F, A) of Example 12, and set A of abstraction variables as well as TD \mathcal{T}' of nested primal graph G_F^A as given in Example 10. Further, recall that restricted to projection set A , formula F has two satisfying assignments. Figure 9 (left) shows TD \mathcal{T}' of G_F^A and tables obtained

by $\text{NestDP}_{\text{HPMC}}(0, (F, A), A, \mathcal{T}')$ for solving projected model counting on (F, A) .

Note that nested table algorithm HPMC of Listing 10 works similar to the nested table algorithm NPMC of Listing 8, but it calls $\text{HybDP}_{\text{HPMC}}$ recursively. We briefly discuss executing HPMC in the context of Line 13 of algorithm $\text{HybDP}_{\text{HPMC}}$ on node t_1 , resulting in table τ_1 as shown in Figure 9 (left). Recall that $F_{t_1}^A = \{\{\neg a, b, p_1\}, \{a, \neg b, \neg p_1\}\}$. Then, in Line 3 of algorithm HPMC , for each assignment J to $\{a, b\}$ of each row of τ_1 , we compute $\text{HybDP}_{\text{HPMC}}(\text{depth}+1, F_{t_1}^A[J], \emptyset)$. Each of these recursive calls, however, is already solved by unit propagation (preprocessing), e.g., $F_{t_1}^A[\{a \mapsto 1, b \mapsto 0\}]$ of Row 2 simplifies to $\{\{p_1\}\}$.

Figure 9 (right) shows $\text{TD } \mathcal{T}''$ of G_F^E with $E := \{a\}$, and tables obtained by algorithm $\text{NestDP}_{\text{HPMC}}(0, (F, A), E, \mathcal{T}'')$. Still, $F_{t''}^E[J]$ for a given assignment J to $\{a\}$ of any row $r \in \tau_{t''}$ can be simplified. Concretely, $F_{t''}^E[\{a \mapsto 0\}]$ evaluates to \emptyset and $F_{t''}^E[\{a \mapsto 1\}]$ evaluates to clause $\{b, c\}$. Thus, restricted to $\{b\} = A \setminus \{a\}$, there are 2 satisfying assignments $\{b \mapsto 0\}$, $\{b \mapsto 1\}$ of $F_{t''}^E[\{a \mapsto 1\}]$.

6. Hybrid Dynamic Programming in Practice

Below, in Section 6.1 we present an implementation of hybrid dynamic programming in order to solve the problems $\#\text{SAT}$ as well as PMC . This is then followed by an experimental evaluation and discussion of the results in Section 6.2, where we also briefly elaborate on existing techniques of state-of-the-art solvers.

6.1. Implementing Hybrid Dynamic Programming

We implemented a solver nestHDB ⁶ based on hybrid dynamic programming in Python3 and using table manipulation techniques by means of *structured query language (SQL)* and the *database management system (DBMS)* PostgreSQL. Our solver builds upon the recently published prototype dpdb [39], which applied a DBMS for the efficient implementation of plain dynamic programming algorithms. This dpdb prototype provides a basic framework for implementing plain dynamic programming algorithms, which can be specified in the form of a plain table algorithm, e.g., the one of Listing 3. However, this system does not have support for neither hybrid nor nested dynamic programming. In order to compare plain dpdb and our solver nestHDB in a fair way, for both systems we used the most-recent version 12 of PostgreSQL and we let it operate on a *tmpfs-ramdisk* instead of disk space (HDD/SDD), i.e., within the main memory (RAM) of a machine. In both dpdb as well as our solver nestHDB , the DBMS serves the purpose of extremely efficient in-memory table manipulations and query optimization required by nested DP, and therefore nestHDB benefits from database technology. Those benefits are already available in the form of different and efficient join manipulations that are selected based on several heuristics that are invoked during SQL query optimizing. Note that especially efficient

⁶ nestHDB is open-source and available at github.com/hmarkus/dp_on_dbs/tree/nesthdb. Instances and detailed results are available online at: tinyurl.com/nesthdb.

join operations have been already designed, implemented, combined, and tuned for decades [70, 71, 72]. Therefore it seems more than natural to rely on this technological advancement that database theory readily provides. We are certain that one can easily replace PostgreSQL by any other state-of-the-art relational database that uses standard SQL in order to express queries. In the following, we briefly discuss implementation specifics that are crucial for a performant system that is competitive with state-of-the-art solvers.

Nested DP & Choice of Standard Solvers. We implemented dedicated nested DP algorithms for solving #SAT and PMC, where we do (nested) DP up to $\text{threshold}_{\text{depth}} = 2$. Note that incrementing nesting depth results in getting again exponentially many (in the largest bag size) rows for each row of tables of the previous depth, i.e., a low nesting limit is highly expected. Currently, we do not see a way to efficiently solve instances of higher nesting depth, which might change in case of further advances allowing to decrease table sizes obtained during dynamic programming. Further, we set $\text{threshold}_{\text{hybrid}} = 1000$ and therefore we do not “fall back” to standard solvers based on the width (cf., Line 6 of Listing 9), but based on the nesting depth.

Also, the evaluation of the nested bag formula is “shifted” to the database if it uses *at most* 40 abstraction variables, since PostgreSQL efficiently handles these small-sized Boolean formulas. Thereby, further nesting is saved by executing optimized SQL statements within the TD nodes. A value of 40 seems to be a nice balance between the overhead caused by standard solvers for small formulas and exponential growth counteracting the advantages of the DBMS. For hybrid solving, we use #SAT solver SharpSAT [73] and for PMC we employ the recently published PMC solver projMC [20], solver SharpSAT and SAT solver picosat [74]. Observe that our solver immediately benefits from better standard solvers and further improvements of the solvers above.

Choosing Non-Nesting Variables & Compatible Nodes. TDs are computed by means of heuristics via decomposition library htd [36]. We implement a heuristic for finding practically sufficient abstractions, i.e., abstraction variables for the nested primal graph, in reasonable using an external solver. Therefore, we encode our heuristic into two logic programs (ASP) for solver clingo [75], which includes techniques for fast solving reachability via nesting paths. The encodings, which in total comprise 11 lines, are publicly available in the online repository of nestHDB. Technically, our focus is on avoiding extremely large abstractions at the cost of larger nested bag formulas. Still, nesting allows to obtain refined abstractions again at higher depths. Thereby, we achieve a good trade off between runtime and quality.

By the first encoding (“guess_min_degree.lp”), we compute a *reasonably-sized subset of vertices* of smallest degree, more precisely, such that the number of neighboring vertices not in the set is minimized. We take a subset of size at most 95, which turned out to be practically useful. We run the ASP solver clingo for *up to 10 seconds*. The solver might not return an optimum within 10 seconds, but always returns a subset of vertices that can be used subsequently.

By the second encoding (“guess_increase.lp”), we guess among the thereby

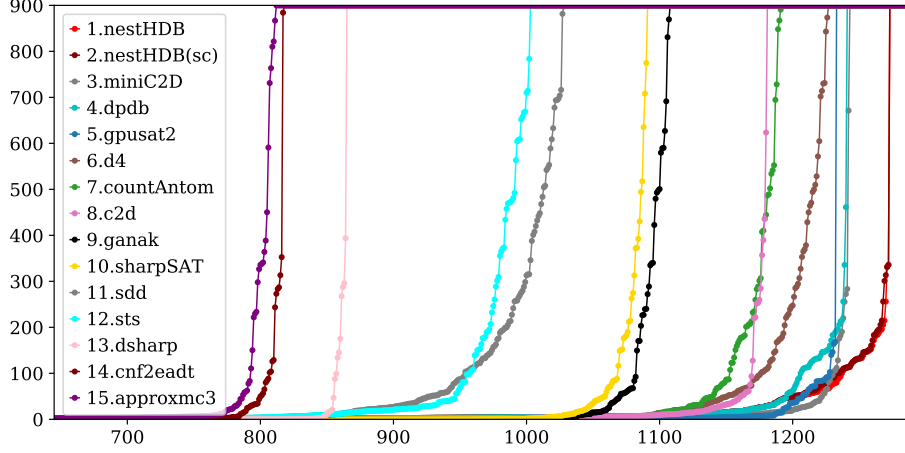


Figure 10: Cactus plot of instances for #SAT, where instances (x-axis) are ordered for each solver individually by runtime[seconds] (y-axis). $\text{threshold}_{\text{abstr}} = 38$.

obtained subset of vertices of preferably smallest degree, a preferably maximal set A of at most 64 abstraction variables such the resulting graph G_F^A is reasonably sparse, which is achieved by minimizing the number of edges of G_F^A . To this end, we also use built-in (cost) optimization, where we take the best results obtained by *clingo* after running *at most 35 seconds*. For more details on ASP, we refer to introductory texts [26, 75].

We expect that this approach, which driven mostly by practical considerations, can be improved. Furthermore, it can also be extending by problem-specific as well as domain-specific information, which might help in choosing promising abstraction variables A .

As rows of tables during (nested) DP can be independently computed and parallelized [40], hybrid solver *nestHDB* potentially calls standard solvers for solving subproblems in parallel using a thread pool. Thereby, the uniquely compatible node for relevant compatible sets U , as denoted in this paper by means of $\text{comp}(\cdot)$, is decided during runtime among compatible nodes on a first-come-first-serve basis.

6.2. Experimental Evaluation

In order to evaluate the concept of hybrid dynamic programming, we conducted a series of experiments considering a variety of solvers and benchmarks, both for model counting (#SAT) as well as projected model counting (PMC). During the evaluation we thereby compared the performance of algorithm *HybDP_{HPMC}* of Listing 9. We benchmarked this algorithm both for the projected model counting problem, but also for the special case of model counting, where all variables are projection variables.

Benchmarked Solvers & Instances. We benchmarked *nestHDB* and 16 other publicly available #SAT solvers on 1,494 instances recently considered [39]. Most

of the existing solvers of other approaches are single-core solvers, which adhere to three different techniques, namely *knowledge-compilation based*, *caching based*, and *approximate*. Among the knowledge-compilation based solvers, which aim to obtain compact representations of the formulas that are concise and easier to solve, are `miniC2D` [76], `d4` [14], `c2d` [77], `sdd` [78], `dsharp` [79], `cnf2eadt` [80], and `bdd_minisat` [81]. These solvers use different variants and flavors of knowledge-compilation, thereby finding decent trade-offs between the time needed to obtain those representations and their succinctness. We also considered the caching-based solvers `cachet` [82], `sharpSAT` [73], and `ganak` [83], which employ existing SAT-based solvers by sophisticated caching techniques. Finally, among the approximate counters, we focused on `sts` [84], `sharpCDCL` [85], and `approxmc3` [86], which employ sampling-based techniques to approximately obtain the model counts. Our comparison also included the multi-core solvers `dpdb` [39], `gpusat2` [40], which is also based on dynamic programming and uses massively parallel graphics processing units (GPUs), as well as `countAntom` [87], which relies on sophisticated techniques for work-balancing. For a more ample description of the used techniques, we refer to the model counting competition report [19]. Note that we excluded distributed solvers such as `dCountAntom` [88] and `DMC` [89] from our experimental setup. Both solvers require a cluster with access to the OpenMPI framework [90] and fast physical interconnections. Unfortunately, we do not have access to OpenMPI on our cluster. Nonetheless, our focus are shared-memory systems and since `dpdb` might well be used in a distributed setting, it leaves an experimental comparison between distributed solvers that also include `dpdb` as subsolver to future work. While `nestHDB` itself is a multi-core solver, we additionally included in our comparison `nestHDB(sc)`, which is `nestHDB`, but restricted to a single core only. The instances [39] we took are *already preprocessed* by `pmc` [69] using recommended options `-vivification -eliminateLit -litImplied -iterate=10 -equiv -orGate -affine`, which guarantee that the model counts are preserved. However, `nestHDB` still uses `pmc` with these options in Line 1 of Listing 9, which is used in the light of nested bag formulas that appear due to nesting.

Further, we considered the problem PMC, where we compare solvers `projMC` [20], `clingo` [75], `ganak` [83], `nestHDB`⁶, and `nestHDB(sc)` on 610 publicly available instances⁷ from `projMC` (consisting of 15 *planning*, 60 *circuit*, and 100 *random* instances) and Fremont, with 170 *symbolic-markov* applications, and 265 *misc* instances. For simplifying nested bag formulas under assignments encountered due to nesting in Line 1 of Listing 9, `nestHDB` uses `pmc` as before, but *without options* `-equiv -orGate -affine` to ensure preservation of models (equivalence).

Benchmark Setup. Solvers ran on a cluster of 12 nodes. Each node of the cluster is equipped with two Intel Xeon E5-2650 CPUs consisting of 12 physical cores each at 2.2 GHz clock speed, 256 GB RAM. For `dpdb` and `nestHDB`, we used

⁷Sources: tinyurl.com/projmc; tinyurl.com/pmc-fremont-01-2020.

bench- mark set	solver	tw upper bound				Σ	time [h]
		max	0-30	31-50	>50		
planning	nestHDB	30	7	0	0	7	2.88
	nestHDB(sc)	30	7	0	0	7	3.31
	projMC	26	6	0	0	6	3.01
	ganak	19	5	0	0	5	3.36
	clingo	4	1	0	0	1	4.00
circ	nestHDB	99	34	10	16	60	2.10
	nestHDB(sc)	99	34	4	14	52	4.60
	projMC	91	28	10	11	49	6.23
	ganak	99	34	10	16	60	1.21
	clingo	99	31	10	16	57	4.44
random	nestHDB	79	30	20	17	67	10.91
	nestHDB(sc)	79	30	20	15	65	11.29
	projMC	84	30	20	15	65	11.09
	ganak	19	19	0	0	19	23.18
	clingo	24	25	0	0	25	21.38
markov	nestHDB	23	62	0	0	62	31.98
	nestHDB(sc)	23	61	0	0	61	32.54
	projMC	8	54	0	0	54	33.65
	ganak	59	64	0	4	68	30.32
	clingo	3	38	0	0	38	37.54
misc	nestHDB	47	38	17	0	55	46.12
	nestHDB(sc)	47	38	13	0	51	48.20
	projMC	47	38	13	0	51	45.90
	ganak	44	38	15	0	53	45.72
	clingo	63	38	15	1	54	44.79
Σ	nestHDB	99	171	47	33	251	93.99
	nestHDB(sc)	99	170	37	29	236	99.95
	projMC	91	156	43	26	225	99.88
	ganak	99	160	25	20	205	103.78
	clingo	99	133	25	17	175	112.15

Figure 11: Number of solved PMC instances, grouped by upper bound intervals of treewidth. time[h] is cumulated wall clock time, timeouts count as 900s. threshold_{abstr}=8.

PostgreSQL 12 on a tmpfs-ramdisk (/tmp) that could grow up to at most 1 GB per run. Results were gathered on Ubuntu 16.04.1 LTS machines with disabled hyperthreading on kernel 4.4.0-139. We mainly compare total wall clock time and number of timeouts. For parallel solvers (dpdb, countAntom, nestHDB) we allow 12 physical cores. Timeout is 900 seconds and RAM is limited to 16 GB per instance and solver. Results for gpusat2 are taken from [39], where a machine equipped with a consumer GPU is used: Intel Core i3-3245 CPU operating at 3.4 GHz, 16 GB RAM, and one Sapphire Pulse ITX Radeon RX 570 GPU running at 1.24 GHz with 32 compute units, 2048 shader units, and 4GB VRAM using driver amdgpu-pro-18.30-641594 and OpenCL 1.2. The system operated on Ubuntu 18.04.1 LTS with kernel 4.15.0-34.

Benchmark Results. The results for #SAT showing the best 14 solvers are summarized in the cactus plot of Figure 10. Overall it shows nestHDB among the

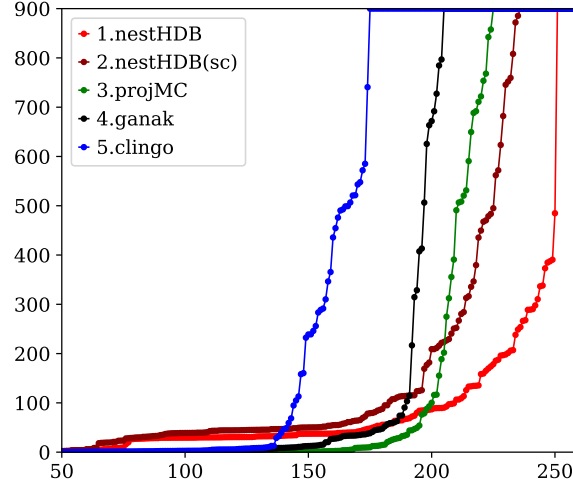


Figure 12: Cactus plot showing the number of solved PMC instances, where the x-axis shows for each solver (configuration) individually, the number of instances ordered by increasing runtime. time[h] is cumulated wall clock time, timeouts count as 900s. $\text{threshold}_{\text{abstr}}=8$.

best solvers, solving 1,273 instances. The reason for this is, compared to `dpdb`, that `nestHDB` can solve instances using TDs of primal graphs of widths larger than 44, up to width 266. This limit is even slightly larger than the width of 264 that SharpSAT on its own can handle. We also tried using `minic2d` instead of SharpSAT as standard solver for solvers `nestHDB` and `nestHDB(sc)`, but we could only solve one instance more. Notably, `nestHDB(sc)` has about the same performance as `nestHDB`, indicating that parallelism does not help much on the instances. Further, we observed that the employed simple cache as used in Listing 9, provides only a marginal improvement.

Figure 11 depicts a table of results on PMC, where we observe that `nestHDB` does a good job on instances with low widths below $\text{threshold}_{\text{abstr}} = 8$ (containing ideas of `dpdb`), but also on widths well above 8 (using nested DP). Notably, `nestHDB` is also competitive on widths well above 50. Indeed, `nestHDB` and `nestHDB(sc)` perform well on all benchmark sets, whereas on some sets the solvers `projMC`, `clingo` and `ganak` are faster. Overall, parallelism provides a significant improvement here, but still `nestHDB(sc)` shows competitive performance, which is also visualized in the cactus plot of Figure 12. Figure 13 shows scatter plots comparing `nestHDB` to `projMC` (left) and to `ganak` (right). Overall, both plots show that `nestHDB` solves more instances, since in both cases the y-axis shows more black dots at 900 seconds than the x-axis. Further, the bottom left of both plots shows that there are plenty easy instances that can be solved by `projMC` and `ganak` in well below 50 seconds, where `nestHDB` needs up to 200 seconds. Similarly, the cactus plot given in Figure 12 shows that `nestHDB` can have some overhead compared to the three standard solvers, which is not surprising. This indicates that there is still room for improvement if, e.g., easy instances are easily

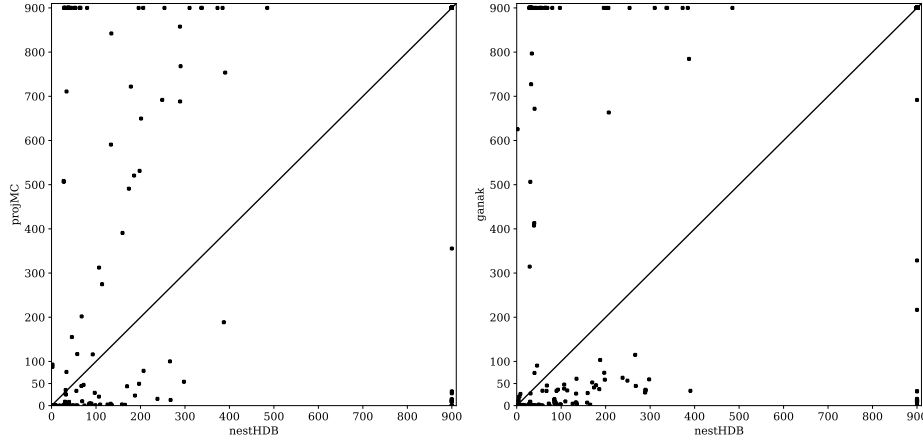


Figure 13: Scatter plot of instances for PMC, where the x-axis shows runtime in seconds of `nestHDB` compared to the y-axis showing runtime of `projMC` (left) and of `ganak` (right). $\text{threshold}_{\text{abstr}} = 8$.

detected, and if standard solvers are used for those instances. Alternatively, one could also just run a standard solver for at most 50 seconds and if not solved within 50 seconds, the heavier machinery of nested dynamic programming is invoked. Apart from these instances, Figure 13 shows that `nestHDB` solves harder instances faster, where standard solvers struggle.

7. Discussion and Conclusion

We introduced a dynamic programming algorithm to solve projected model counting (PMC) by exploiting the structural parameter treewidth. Our algorithm is asymptotically optimal under the exponential time hypothesis (ETH). Its runtime is double exponential in the treewidth of the primal graph of the instance and polynomial in the size of the input instance. We believe that our results can also be extended to another graph representation, namely the incidence graph. Our approach is very general and might be applicable to a wide range of other hard combinatorial problems, such as projection for ASP [56] and QBF [91].

Then, in order to still efficiently deal with projected model counting in practice, we presented nested dynamic programming (nested DP) using different levels of abstractions, which are subsequently refined and solved recursively. This approach is complemented with hybrid solving, where (search-intense) subproblems are solved by standard solvers. We provided nested DP algorithms for problems related to Boolean satisfiability, but the idea can be easily applied for other formalisms. We implemented some of these algorithms and our benchmark results are promising.

In the light of related works on properties for efficient counting algorithms, e.g., [32, 33, 34], we are curious to revisit some of those and potentially study precise runtime dependencies. We expect interesting insights when focusing

on the search for properties of local instance parts that in combination with treewidth allow algorithms that are significantly better than double-exponential in the treewidth. As we demonstrated in the experimental results, we can solve the problem PMC that theoretically requires double-exponential worst-case effort in the treewidth, on instances of decent treewidth upper bounds (up to 99). On plain model counting ($\#SAT$), which is only single-exponential in the treewidth, our solver even deals with instances of larger treewidth upper bounds (up to 260). This also opens the question of whether similar empirical observations can be drawn in other areas and formalisms like constraint solving or database query languages. Further, we plan on deeper studies of problem-specific abstractions, in particular for quantified Boolean formulas. We want to further tune our solver parameters (e.g., thresholds, timeouts, sizes), deepen interleaving with PMC solvers like **projMC**, and to use incremental solving for obtaining abstractions and evaluating nested bag formulas, where intermediate solver references are kept during dynamic programming and formulas are iteratively added and (re-)solved.

Acknowledgements

This work has been supported by the Austrian Science Fund (FWF), Grants J4656, P32830 and Y698, as well as the Vienna Science and Technology Fund, Grant WWTF ICT19-065. Hecher is also affiliated with the University of Potsdam, Germany. The main research was conducted while Fichte and Morak were affiliated with TU Vienna, Austria. Part of the research was carried out while Hecher and Fichte were visiting the Simons Institute for the Theory of Computing at UC Berkeley. We thank the anonymous reviewers for their detailed comments.

References

- [1] B. Abramson, J. Brown, W. Edwards, A. Murphy, R. L. Winkler, Hailfinder: A Bayesian system for forecasting severe weather, *International Journal of Forecasting* 12 (1) (1996) 57–71. doi:10.1016/0169-2070(95)00664-8.
- [2] A. Choi, G. Van den Broeck, A. Darwiche, Tractable learning for structured probability spaces: A case study in learning preference distributions, in: Q. Yang (Ed.), *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, The AAAI Press, 2015.
- [3] C. Domshlak, J. Hoffmann, Probabilistic planning via heuristic forward search and weighted model counting, *J. Artif. Intell. Res.* 30 (2007) 565–620. doi:10.1613/jair.2289.
- [4] L. Dueñas-Osorio, K. S. Meel, R. Paredes, M. Y. Vardi, Counting-based reliability estimation for power-transmission grids, in: S. P. Singh, S. Markovitch (Eds.), *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*, The AAAI Press, San Francisco, CA, USA, 2017, pp. 4488–4494.
- [5] C. D. Manning, P. Raghavan, H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press, Cambridge, 2008.
- [6] O. Pourret, P. Naim, M. Bruce, *Bayesian Networks - A Practical Guide to Applications*, John Wiley & Sons, 2008.
- [7] M. Sahami, S. Dumais, D. Heckerman, E. Horvitz, A Bayesian approach to filtering junk e-mail, in: T. Joachims (Ed.), *Proceedings of the AAAI-98 Workshop on Learning for Text Categorization*, Vol. 62, 1998, pp. 98–105.
- [8] T. Sang, P. Beame, H. Kautz, Performing Bayesian inference by weighted model counting, in: M. M. Veloso, S. Kambhampati (Eds.), *Proceedings of the 29th National Conference on Artificial Intelligence (AAAI'05)*, The AAAI Press, 2005.

- [9] Y. Xue, A. Choi, A. Darwiche, Basing decisions on sentences in decision diagrams, in: J. Hoffmann, B. Selman (Eds.), *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12)*, The AAAI Press, Toronto, ON, Canada, 2012.
- [10] C. P. Gomes, A. Sabharwal, B. Selman, Chapter 20: Model counting, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, Amsterdam, Netherlands, 2009, pp. 633–654. doi:10.3233/978-1-58603-929-5-633.
- [11] L. Valiant, The complexity of enumeration and reliability problems, *SIAM J. Comput.* 8 (3) (1979) 410–421.
- [12] D. Roth, On the hardness of approximate reasoning, *Artificial Intelligence* 82 (1–2) (1996). doi:10.1016/0004-3702(94)00092-1.
- [13] S. Chakraborty, K. S. Meel, M. Y. Vardi, Improving approximate counting for probabilistic inference: From linear to logarithmic SAT solver calls, in: S. Kambhampati (Ed.), *Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, The AAAI Press, New York City, NY, USA, 2016, pp. 3569–3576.
- [14] J.-M. Lagniez, P. Marquis, An improved decision-DNNF compiler, in: C. Sierra (Ed.), *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI'17)*, The AAAI Press, 2017.
- [15] S. H. Sæther, J. A. Telle, M. Vatshelle, Solving #SAT and MAXSAT by dynamic programming, *J. Artif. Intell. Res.* 54 (2015) 59–82.
- [16] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases: The Logical Level*, 1st Edition, Addison-Wesley, Boston, MA, USA, 1995. doi:10.020.153/7710.
- [17] M. Gebser, T. Schaub, S. Thiele, P. Veber, Detecting inconsistencies in large biological networks with answer set programming, *Theory Pract. Log. Program.* 11 (2-3) (2011) 323–360.
- [18] M. L. Ginsberg, A. J. Parkes, A. Roy, Supermodels and robustness, in: C. Rich, J. Mostow (Eds.), *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI'98)*, The AAAI Press, Madison, Wisconsin, USA, 1998, pp. 334–339.
- [19] J. K. Fichte, M. Hecher, F. Hamiti, The model counting competition 2020, *ACM Journal of Experimental Algorithmics* 26 (2021). doi:10.1145/3459080.

- [20] J. Lagniez, P. Marquis, A Recursive Algorithm for Projected Model Counting, in: 33rd Conference on Artificial Intelligence, The AAAI Press, 2019, pp. 1536–1543.
- [21] R. A. Aziz, G. Chu, C. Muise, P. Stuckey, $\#(\exists)$ SAT: Projected Model Counting, in: M. Heule, S. Weaver (Eds.), Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT’15), Springer Verlag, Austin, TX, USA, 2015, pp. 121–137. doi:10.1007/978-3-319-24318-4_10.
- [22] F. Capelli, S. Mengel, Tractable QBF by knowledge compilation, in: 36th International Symposium on Theoretical Aspects of Computer Science (STACS’19), Vol. 126 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 18:1–18:16.
- [23] J. K. Fichte, M. Hecher, Treewidth and counting projected answer sets, in: M. Balduccini, Y. Lierler, S. Woltran (Eds.), 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019), Vol. 11481 of Lecture Notes in Computer Science, Springer, 2019, pp. 105–119. doi:10.1007/978-3-030-20528-7_9.
- [24] J. M. Dudek, V. H. N. Phan, M. Y. Vardi, ProCount: Weighted Projected Model Counting with Graded Project-Join Trees, in: C. Li, F. Manyà (Eds.), Theory and Applications of Satisfiability Testing (SAT 2021), Vol. 12831 of Lecture Notes in Computer Science, Springer, 2021, pp. 152–170. doi:10.1007/978-3-030-80223-3_11.
- [25] A. Durand, M. Hermann, P. G. Kolaitis, Subtractive reductions and complete problems for counting complexity classes, Theoretical Computer Science 340 (3) (2005) 496–513. doi:10.1016/j.tcs.2005.03.012.
- [26] M. Gebser, B. Kaufmann, T. Schaub, Solution enumeration for projected boolean search problems, in: W.-J. van Hoeve, J. N. Hooker (Eds.), Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR’09), Vol. 5547 of Lecture Notes in Computer Science, Springer Verlag, Berlin, 2009, pp. 71–86. doi:10.1007/978-3-642-01929-6_7.
- [27] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, M. P. Dániel Marx, M. Pilipczuk, S. Saurabh, Parameterized Algorithms, Springer Verlag, 2015. doi:10.1007/978-3-319-21275-3.
- [28] R. G. Downey, M. R. Fellows, Fundamentals of Parameterized Complexity, Texts in Computer Science, Springer Verlag, London, UK, 2013. doi:10.1007/978-1-4471-5559-1.
- [29] J. Flum, M. Grohe, Parameterized Complexity Theory, Vol. XIV of Theoretical Computer Science, Springer Verlag, Berlin, 2006. doi:10.1007/3-540-29953-X.

- [30] R. Niedermeier, Invitation to Fixed-Parameter Algorithms, Vol. 31 of Oxford Lecture Series in Mathematics and its Applications, Oxford University Press, New York, NY, USA, 2006.
- [31] M. Samer, S. Szeider, Algorithms for propositional model counting, *J. Discrete Algorithms* 8 (1) (2010) 50–64. doi:10.1016/j.jda.2009.06.002.
- [32] A. Durand, S. Mengel, Structural tractability of counting of solutions to conjunctive queries, in: W. Tan, G. Guerrini, B. Catania, A. Gounaris (Eds.), Joint 2013 EDBT/ICDT Conferences (ICDT’13), ACM, 2013, pp. 81–92. doi:10.1145/2448496.2448508.
- [33] H. Chen, S. Mengel, A Trichotomy in the Complexity of Counting Answers to Conjunctive Queries, in: M. Arenas, M. Ugarte (Eds.), 18th International Conference on Database Theory (ICDT’15), Vol. 31 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 110–126. doi:10.4230/LIPIcs.ICDT.2015.110.
- [34] G. Greco, F. Scarcello, The Power of Local Consistency in Conjunctive Queries and Constraint Satisfaction Problems, *SIAM J. Comput.* 46 (3) (2017) 1111–1145. doi:10.1137/16M1090272.
- [35] H. Dell, C. Komusiewicz, N. Talmon, M. Weller, The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration, in: IPEC’17, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl Publishing, 2017, pp. 30:1–30:13.
- [36] M. Abseher, N. Musliu, S. Woltran, htd – a free, open-source framework for (customized) tree decompositions and beyond, in: CPAIOR’17, Vol. 10335 of Lecture Notes in Computer Science, Springer Verlag, 2017, pp. 376–386.
- [37] H. Tamaki, Positive-instance driven dynamic programming for treewidth, *J. Comb. Optim.* 37 (4) (2019) 1283–1311. doi:10.1007/s10878-018-0353-z.
- [38] S. Maniu, P. Senellart, S. Jog, An experimental study of the treewidth of real-world graph data (extended version), *CoRR* abs/1901.06862 (2019). arXiv:1901.06862.
- [39] J. K. Fichte, M. Hecher, P. Thier, S. Woltran, Exploiting database management systems and treewidth for counting, in: PADL’20, Vol. 12007 of Lecture Notes in Computer Science, Springer Verlag, 2020, pp. 151–167.
- [40] J. K. Fichte, M. Hecher, M. Zisser, An improved gpu-based SAT model counter, in: CP’19, Vol. 11802 of LNCS, Springer, 2019, pp. 491–509.
- [41] J. K. Fichte, M. Hecher, M. Morak, S. Woltran, DynASP2.5: Dynamic programming on tree decompositions in action, in: D. Lokshtanov, N. Nishimura

- (Eds.), Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC'17), Dagstuhl Publishing, 2017. doi:10.4230/LIPIcs.IPEC.2017.17.
- [42] R. Impagliazzo, R. Paturi, F. Zane, Which problems have strongly exponential complexity?, J. of Computer and System Sciences 63 (4) (2001) 512–530. doi:10.1006/jcss.2001.1774.
 - [43] M. Lampis, V. Mitsou, Treewidth with a quantifier alternation revisited, in: D. Lokshantov, N. Nishimura (Eds.), Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC'17), Dagstuhl Publishing, 2017. doi:10.4230/LIPIcs.IPEC.2017.17.
 - [44] J. K. Fichte, M. Hecher, M. Morak, S. Woltran, Exploiting treewidth for projected model counting and its limits, in: 21st International Conference on Theory and Applications of Satisfiability Testing (SAT), Vol. 10929 of Lecture Notes in Computer Science, Springer Verlag, 2018, pp. 165–184. doi:10.1007/978-3-319-94144-8_11.
 - [45] M. Hecher, P. Thier, S. Woltran, Taming High Treewidth with Abstraction, Nested Dynamic Programming, and Database Technology, in: 23rd International Conference on Theory and Applications of Satisfiability Testing SAT, Vol. 12178 of Lecture Notes in Computer Science, Springer Verlag, 2020, pp. 343–360.
 - [46] R. L. Graham, M. Grötschel, L. Lovász, Handbook of Combinatorics, Vol. I, Elsevier Science Publishers, North-Holland, 1995.
 - [47] H. Kleine Büning, T. Lettman, Propositional logic: deduction and algorithms, Cambridge University Press, Cambridge, New York, NY, USA, 1999.
 - [48] C. H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.
 - [49] L. J. Stockmeyer, A. R. Meyer, Word problems requiring exponential time, in: A. V. Aho, A. Borodin, R. L. Constable, R. W. Floyd, M. A. Harrison, R. M. Karp, H. R. Strong (Eds.), Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC'73), Assoc. Comput. Mach., New York, Austin, TX, USA, 1973, pp. 1–9. doi:10.1145/800125.804029.
 - [50] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, Vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, Amsterdam, Netherlands, 2009.
 - [51] L. A. Hemaspaandra, H. Vollmer, The satanic notations: Counting classes beyond #P and other definitional adventures, SIGACT News 26 (1) (1995) 2–13. doi:10.1145/203610.203611.
 - [52] R. Diestel, Graph Theory, 4th Edition, Vol. 173 of Graduate Texts in Mathematics, Springer Verlag, 2012.

- [53] J. A. Bondy, U. S. R. Murty, Graph theory, Vol. 244 of Graduate Texts in Mathematics, Springer Verlag, New York, USA, 2008.
- [54] H. L. Bodlaender, A linear-time algorithm for finding tree-decompositions of small treewidth, *SIAM J. Comput.* 25 (6) (1996) 1305–1317.
- [55] H. L. Bodlaender, A. M. C. A. Koster, Combinatorial optimization on graphs of bounded treewidth, *The Computer Journal* 51 (3) (2008) 255–269. doi:10.1093/comjnl/bxm037.
- [56] J. K. Fichte, M. Hecher, M. Morak, S. Woltran, Answer set solving with bounded treewidth revisited, in: M. Balduccini, T. Janhunen (Eds.), Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’17), Vol. 10377 of Lecture Notes in Computer Science, Springer Verlag, Espoo, Finland, 2017, pp. 132–145. doi:10.1007/978-3-319-61660-5_13.
- [57] H. L. Bodlaender, T. Kloks, Efficient and constructive algorithms for the pathwidth and treewidth of graphs, *J. Algorithms* 21 (2) (1996) 358–402.
- [58] M. Bannach, S. Berndt, Recent Advances in Positive-Instance Driven Graph Searching, *Algorithms* 15 (2) (2022) 42. doi:10.3390/a15020042.
- [59] R. L. Wilder, Introduction to the Foundations of Mathematics, 2nd Edition, John Wiley & Sons, 1965.
- [60] R. Pichler, S. Rümmele, S. Woltran, Counting and enumeration problems with bounded treewidth, in: E. M. Clarke, A. Voronkov (Eds.), Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR’10), Vol. 6355 of Lecture Notes in Computer Science, Springer Verlag, 2010, pp. 387–404. doi:10.1007/978-3-642-17511-4_22.
- [61] D. E. Knuth, How fast can we multiply?, in: The Art of Computer Programming, 3rd Edition, Vol. 2 of Seminumerical Algorithms, Addison-Wesley, 1998, Ch. 4.3.3, pp. 294–318.
- [62] D. Harvey, J. van der Hoeven, G. Lecerf, Even faster integer multiplication, *J. Complexity* 36 (2016) 1–30. doi:https://doi.org/10.1016/j.jco.2016.03.001.
- [63] J. K. Fichte, M. Hecher, A. Pfandler, Lower Bounds for QBFs of Bounded Treewidth, in: 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS’20), Assoc. Comput. Mach., New York, 2020, pp. 410–424.
- [64] M. Bannach, S. Berndt, Practical access to dynamic programming on tree decompositions, *Algorithms* 12 (8) (2019) 172.
- [65] H. Dell, M. Roth, P. Wellnitz, Counting answers to existential questions, in: ICALP’19, Vol. 132 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 113:1–113:15.

- [66] E. Eiben, R. Ganian, T. Hamm, O. Kwon, Measuring what matters: A hybrid approach to dynamic programming with treewidth, in: MFCS'19, Vol. 138 of LIPIcs, Dagstuhl Publishing, 2019, pp. 42:1–42:15.
- [67] R. Ganian, M. S. Ramanujan, S. Szeider, Combining treewidth and backdoors for CSP, in: STACS'17, 2017, pp. 36:1–36:17. doi:10.4230/LIPIcs.STACS.2017.36.
- [68] M. Hecher, M. Morak, S. Woltran, Structural Decompositions of Epistemic Logic Programs, in: AAAI'20, AAAI Press, 2020, pp. 2830–2837. URL <https://aaai.org/ojs/index.php/AAAI/article/view/5672>
- [69] J. Lagniez, P. Marquis, Preprocessing for Propositional Model Counting, in: 28th AAAI Conference on Artificial Intelligence (AAAI), The AAAI Press, 2014, pp. 2688–2694.
- [70] J. D. Ullman, Principles of Database and Knowledge-Base Systems, Volume II, Computer Science Press, New York, NY, USA, 1989.
- [71] H. Garcia-Molina, J. D. Ullman, J. Widom, Database systems: the complete book, 2nd Edition, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2009.
- [72] R. Elmasri, S. B. Navathe, Fundamentals of Database Systems, 7th Edition, Pearson, 2016.
- [73] M. Thurley, sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP, in: 9th International Conference on Theory and Applications of Satisfiability Testing (SAT), Springer Verlag, 2006, pp. 424–429.
- [74] A. Biere, PicoSAT Essentials, J. on Satisfiability, Boolean Modeling and Computation 4 (2-4) (2008) 75–97.
- [75] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (1) (2019) 27–82. doi:10.1017/S1471068418000054.
- [76] U. Oztok, A. Darwiche, A Top-Down Compiler for Sentential Decision Diagrams, in: 24th International Joint Conference on Artificial Intelligence (IJCAI), The AAAI Press, 2015, pp. 3141–3148.
- [77] A. Darwiche, New Advances in Compiling CNF to Decomposable Negation Normal Form, in: 16th European Conference on Artificial Intelligence (ECAI), IOS Press, 2004, pp. 318–322.
- [78] A. Darwiche, SDD: A New Canonical Representation of Propositional Knowledge Bases, in: 22nd International Joint Conference on Artificial Intelligence (IJCAI), AAAI Press/IJCAI, 2011, pp. 819–826.

- [79] S. A. Muise, Christian J. and McIlraith, J. C. Beck, E. I. Hsu, Dsharp: Fast d-DNNF Compilation with sharpSAT, in: 25th Canadian Conference on Artificial Intelligence (AI), Vol. 7310 of Lecture Notes in Computer Science, Springer Verlag, 2012, pp. 356–361.
- [80] F. Koriche, J.-M. Lagniez, P. Marquis, S. Thomas, Knowledge Compilation for Model Counting: Affine Decision Trees, in: 23rd International Joint Conference on Artificial Intelligence (IJCAI), IJCAI/AAAI, 2013.
- [81] T. Toda, T. Soh, Implementing Efficient All Solutions SAT Solvers, ACM Journal of Experimental Algorithmics 21 (2015) 1.12, special Issue SEA 2014.
- [82] T. Sang, F. Bacchus, P. Beame, H. Kautz, T. Pitassi, Combining Component Caching and Clause Learning for Effective Model Counting, in: 7th International Conference on Theory and Applications of Satisfiability Testing (SAT), 2004.
- [83] S. Sharma, S. Roy, M. Soos, K. S. Meel, GANAK: A Scalable Probabilistic Exact Model Counter, in: 28th International Joint Conference on Artificial Intelligence (IJCAI), ijcai.org, 2019, pp. 1169–1176.
- [84] S. Ermon, C. P. Gomes, B. Selman, Uniform Solution Sampling Using a Constraint Solver As an Oracle, in: 28th Conference on Uncertainty in Artificial Intelligence (UAI), AUAI Press, 2012, pp. 255–264.
- [85] V. Klebanov, N. Manthey, C. J. Muise, SAT-Based Analysis and Quantification of Information Flow in Programs, in: 10th International Conference on Quantitative Evaluation of Systems (QEST), Vol. 8054 of Lecture Notes in Computer Science, Springer Verlag, 2013, pp. 177–192.
- [86] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, M. Y. Vardi, Distribution-Aware Sampling and Weighted Model Counting for SAT, in: 28th AAAI Conference on Artificial Intelligence (AAAI), The AAAI Press, 2014, pp. 1722–1730.
- [87] J. Burchard, T. Schubert, B. Becker, Laissez-Faire Caching for Parallel #SAT Solving, in: 18th International Conference on Theory and Applications of Satisfiability Testing (SAT), Vol. 9340 of Lecture Notes in Computer Science, Springer Verlag, 2015, pp. 46–61.
- [88] J. Burchard, T. Schubert, B. Becker, Distributed Parallel #SAT Solving, in: 18th IEEE International Conference on Cluster Computing (CLUSTER), IEEE Computer Society, 2016, pp. 326–335.
- [89] J.-M. Lagniez, P. Marquis, N. Szczepanski, DMC: A Distributed Model Counter, in: 27th International Joint Conference on Artificial Intelligence (IJCAI), The AAAI Press, 2018, pp. 1331–1338.

- [90] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation, in: 11th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science, 2004, pp. 97–104.
- [91] G. Charwat, S. Woltran, Dynamic programming-based QBF solving, in: F. Lonsing, M. Seidl (Eds.), Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF'16), Vol. 1719, CEUR Workshop Proceedings (CEUR-WS.org), 2016, pp. 27–40, co-located with 19th International Conference on Theory and Applications of Satisfiability Testing (SAT'16).

Appendix A. Omitted Proofs

Observation 1. *The relation $=_P$ is an equivalence relation.*

Proof. One can easily see that $=_P (A, B) := (A \cap P) = (B \cap P)$ is

- reflexive: $A \cap P = A \cap P$ for any two sets A, P ,
- symmetric: $A \cap P = B \cap P$ if and only if $B \cap P = A \cap P$ for given sets A, B, P , and
- transitive: if $A \cap P = B \cap P$ and $B \cap P = C \cap P$, then $A \cap P = C \cap P$ holds as well for any sets A, B, C, P .

As a result, $=_P$ is an equivalence relation. \square

Observation 3. *Let n be a positive integer, $X = \{1, \dots, n\}$, and X_1, X_2, \dots, X_n subsets of X . The number of elements in the intersection over all sets A_i is*

$$\begin{aligned} \left| \bigcap_{i \in X} X_i \right| = & \left| \bigcup_{j=1}^n X_j \right| - \sum_{\emptyset \subsetneq I \subsetneq X, |I|=1} \left| \bigcap_{i \in I} X_i \right| + \sum_{\emptyset \subsetneq I \subsetneq X, |I|=2} \left| \bigcap_{i \in I} X_i \right| - \dots \\ & + \sum_{\emptyset \subsetneq I \subsetneq X, |I|=n-1} (-1)^{|I|} \left| \bigcap_{i \in I} X_i \right|. \end{aligned}$$

It trivially works to count arbitrary sets.

Proof. We take the well-known inclusion-exclusion principle [46] and rearrange

the equation.

$$\begin{aligned}
|\bigcup_{j=1}^n X_j| &= \sum_{\emptyset \subsetneq I \subseteq X} (-1)^{|I|-1} |\bigcap_{i \in I} X_i| \\
|\bigcup_{j=1}^n X_j| &= \sum_{\emptyset \subsetneq I \subsetneq X} (-1)^{|I|-1} |\bigcap_{i \in I} X_i| + (-1)^{|X|-1} |\bigcap_{i \in X} X_i| \\
(-1)^{|X|-1} |\bigcap_{i \in X} X_i| &= |\bigcup_{j=1}^n X_j| - \sum_{\emptyset \subsetneq I \subsetneq X} (-1)^{|I|-1} |\bigcap_{i \in I} X_i| \\
|\bigcap_{i \in X} X_i| &= \left| |\bigcup_{j=1}^n X_j| - \sum_{\emptyset \subsetneq I \subsetneq X} (-1)^{|I|-1} |\bigcap_{i \in I} X_i| \right| \\
|\bigcap_{i \in X} X_i| &= \left| |\bigcup_{j=1}^n X_j| - \sum_{\emptyset \subsetneq I \subsetneq X, |I|=1} |\bigcap_{i \in I} X_i| \right. \\
&\quad + \sum_{\emptyset \subsetneq I \subsetneq X, |I|=2} |\bigcap_{i \in I} X_i| \\
&\quad - \dots \\
&\quad \left. + \sum_{\emptyset \subsetneq I \subsetneq X, |I|=n-1} (-1)^{|I|} |\bigcap_{i \in I} X_i| \right|
\end{aligned}$$

□