ProCount: Weighted Projected Model Counting with Graded Project-Join Trees*

Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi**

Rice University, Houston, TX, USA
{jmd11,vhp1,vardi}@rice.edu

Abstract. Recent work in weighted model counting proposed a unifying framework for dynamic-programming algorithms. The core of this framework is a project-join tree: an execution plan that specifies how Boolean variables are eliminated. We adapt this framework to compute exact literal-weighted projected model counts of propositional formulas in conjunctive normal form. Our key conceptual contribution is to define *gradedness* on project-join trees, a novel condition requiring irrelevant variables to be eliminated before relevant variables. We prove that building graded project-join trees can be reduced to building standard project-join trees and that graded project-join trees can be used to compute projected model counts. The resulting tool `ProCount` is competitive with the state-of-the-art tools `D4P`, `projMC`, and `reSSAT`, achieving the shortest solving time on 131 benchmarks of 390 benchmarks solved by at least one tool, from 849 benchmarks in total.

1 Introduction

Weighted projected model counting is a fundamental problem in artificial intelligence, with applications in planning [4], formal verification [34], and reliability estimation [20]. Counting is also closely connected to sampling [32], a problem of major interest in probabilistic reasoning [33]. The input is a set of constraints, whose variables are divided into *relevant variables* X and *irrelevant variables* Y. The goal is to compute the weighted number of assignments to X that, with some assignment to Y, satisfy the constraints. This problem is complete for the complexity class $\#\mathrm{P}^{\mathrm{NP}[1]}$ [63]. There are recent tools for weighted projected model counting [37, 38].

Dynamic programming is a powerful technique that has been applied across computer science [7]. The key idea is to solve a large problem by solving many smaller subproblems then combining partial solutions into the final result. Dynamic programming is a natural framework to solve problems defined on sets of constraints, as subproblems can be formed by partitioning the constraints.

* Work supported in part by NSF grants CCF-1704883, DMS-1547433, IIS-1527668, and IIS-1830549, DoD MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office. The final authenticated version is available online at `https://doi.org/10.1007/978-3-030-80223-3_11`.

** Authors sorted alphabetically by surnames.

This framework has been instantiated into algorithms for database-query optimization [41], SAT solving [3, 45, 59], QBF evaluation [10], model counting [5, 17, 25, 31, 49], and projected model counting [22, 29].

Recently, a unifying framework based on *project-join trees* for dynamic-programming algorithms was proposed [17]. The key idea is to consider project-join trees as *execution plans* and decompose dynamic-programming algorithms into two phases: a *planning phase*, where a project-join tree is constructed from an input problem instance, and an *execution phase*, where the project-join tree is used to compute the result. The project-join-tree-based model counter `DPMC` [17] was found to be competitive with state-of-the-art exact weighted model counters [12, 36, 44, 50]. Notably, `DPMC` subsumes `ADDMC` [18], which tied with `D4` [36] for first place in the weighted track of the 2020 Model Counting Competition [23].

We adapt this framework for weighted projected model counting. The central challenge is that there are two kinds of variables: relevant and irrelevant. This contrasts with model counting, where all variables are relevant and can be treated similarly. This challenge also occurs for other problems. For example, in Boolean functional synthesis [56], some variables are *free* and must not be projected out. Our solution is to model multiple types of variables by requiring the project-join tree to be *graded*, meaning that irrelevant variables must be projected before relevant variables. Our main theoretical contribution is a novel algorithm to construct graded project-join trees from standard project-join trees. This has two primary advantages.

The first advantage is that graded project-join trees can be constructed using existing tools for standard project-join trees [17] in a black-box way. Tools exist to construct standard project-join trees with tree decompositions [48] or with constraint-satisfaction heuristics [8, 13, 14, 35, 58]. We can thus easily leverage all current and future work in tree-decomposition solvers [2, 28, 57] and constraint-satisfaction heuristics to produce graded project-join trees. This is crucial for the practical success of our tool.

The second advantage of our approach is in the simplicity of the algorithm. Given a project-join tree, its gradedness can be easily verified. Moreover, the algorithm to compute the projected model count from a graded project-join tree is straightforward. This gives us confidence in the correctness of our implementation. During our experimental evaluation, we found correctness errors in $\texttt{D4}_\texttt{P}$ [37], `projMC` [37], and `reSSAT` [38]. We reported these issues to the authors, who then fixed the tools. We believe that this work is a step towards certificates for the verification of projected model counters, similar to certificates produced by SAT solvers [60].

The primary contribution of this work is a dynamic-programming framework for weighted projected model counting based on project-join trees. In particular:

1. We show that graded project-join trees can be used to compute weighted projected model counts.
2. We prove that building graded project-join trees and project-join trees with free variables can be reduced to building standard project-join trees.

3. We find that project-join-tree-based algorithms make a significant contribution to the portfolio of exact weighted projected model counters (`D4P`, `projMC`, and `reSSAT`). Our tool, `ProCount`, achieves the shortest solving time on 131 benchmarks of 390 benchmarks solved by at least one tool, from 849 benchmarks in total.

2 Preliminaries

Pseudo-Boolean Functions and Projections. A *pseudo-Boolean function* over a set X of variables is a function $f : 2^X \to \mathbb{R}$, where 2^X denotes the power set of X. A Boolean formula φ over variables X represents a pseudo-Boolean function over X, denoted $[\varphi] : 2^X \to \mathbb{R}$, where for all $\tau \in 2^X$, if τ satisfies φ then $[\varphi](\tau) \equiv 1$ else $[\varphi](\tau) \equiv 0$. Operations on pseudo-Boolean functions include *product* and *projections*. We define product as follows.

Definition 1 (Product). *Let X and Y be sets of Boolean variables. The* product *of functions $f : 2^X \to \mathbb{R}$ and $g : 2^Y \to \mathbb{R}$ is the function $f \cdot g : 2^{X \cup Y} \to \mathbb{R}$ defined for all $\tau \in 2^{X \cup Y}$ by $(f \cdot g)(\tau) \equiv f(\tau \cap X) \cdot g(\tau \cap Y)$.*

Product generalizes conjunction: if φ and ψ are propositional formulas, then $[\varphi] \cdot [\psi] = [\varphi \wedge \psi]$.

Definition 2 (Projections). *Let X be a set of Boolean variables, x be a variable in X, and $f : 2^X \to \mathbb{R}$ be a pseudo-Boolean function.*

- *The* Σ-projection *of f w.r.t. x is the function $\Sigma_x f : 2^{X \setminus \{x\}} \to \mathbb{R}$ defined for all $\tau \in 2^{X \setminus \{x\}}$ by $(\Sigma_x f)(\tau) \equiv f(\tau) + f(\tau \cup \{x\})$.*
- *The* $\exists$-projection *of f w.r.t. x is the function $\exists_x f : 2^{X \setminus \{x\}} \to \mathbb{R}$ defined for all $\tau \in 2^{X \setminus \{x\}}$ by $(\exists_x f)(\tau) \equiv \max(f(\tau), f(\tau \cup \{x\}))$.*

Σ-projection is also called *additive projection* or *marginalization*. $\exists$-projection is also called *disjunctive projection* and generalizes existential quantification: if φ is a Boolean formula and $x \in \texttt{Vars}(\varphi)$, then $\exists_x[\varphi] = [\exists x.\varphi]$.

Σ-projection and $\exists$-projection are each independently commutative. Formally, for all $x, y \in X$ and $f : 2^X \to \mathbb{R}$, we assert that $\Sigma_x \Sigma_y f = \Sigma_y \Sigma_x f$ and $\exists_x \exists_y f = \exists_y \exists_x f$. For all sets $X = \{x_1, \ldots, x_n\}$, we define $\Sigma_X f \equiv \Sigma_{x_1} \ldots \Sigma_{x_n} f$ and $\exists_X f \equiv \exists_{x_1} \ldots \exists_{x_n} f$. We also take the convention that $\Sigma_\varnothing f \equiv f$ and $\exists_\varnothing f \equiv f$.

In general, Σ-projection does not commute with $\exists$-projection. For example, if $f(x, y) = x \oplus y$ (XOR), then $\Sigma_x \exists_y f \neq \exists_y \Sigma_x f$.

Weighted Projected Model Counting. We compute the total weight, subject to a given weight function and a set of irrelevant variables, of all models of an input Boolean formula. A formal definition follows.

Definition 3. *Let φ be a Boolean formula, $\{X, Y\}$ be a partition of $\texttt{Vars}(\varphi)$, and $W : 2^X \to \mathbb{R}$ be a pseudo-Boolean function. We say that (X, Y, φ, W) is an instance of* weighted projected model counting. *The W-*weighted Y-projected model count *of φ is $\texttt{WPMC}(\varphi, W, Y) \equiv \sum_{\tau \in 2^X} (W(\tau) \cdot \max_{\alpha \in 2^Y} [\varphi](\tau \cup \alpha))$.*

Variables in X are called *relevant* or *additive*, and variables in Y are called *irrelevant* or *disjunctive*. For the special case of *unprojected model counting*, all variables are relevant, and the *W-weighted model count* is $\texttt{WPMC}(\varphi, W, \varnothing)$.

Weights are usually given by a *literal-weight function* $W \equiv \prod_{x \in X} W_x$, where the factors are functions $W_x : 2^{\{x\}} \to \mathbb{R}$. In detail, a positive literal x has weight $W_x(\{x\})$, and a negative literal $\neg x$ has weight $W_x(\varnothing)$.

Graphs. A *graph* G has a set $\mathcal{V}(G)$ of vertices, a set $\mathcal{E}(G)$ of undirected edges, a function $\delta_G : \mathcal{V}(G) \to 2^{\mathcal{E}(G)}$ that gives the set of edges incident to each vertex, and a function $\epsilon_G : \mathcal{E}(G) \to 2^{\mathcal{V}(G)}$ that gives the set of vertices incident to each edge. Each edge must be incident to exactly two vertices. A *tree* is a simple, connected, and acyclic graph. We often refer to a vertex of a tree as a *node*.

A *rooted tree* is a tree T together with a distinguished node $r \in \mathcal{V}(T)$ called the *root*. In a rooted tree (T, r), each node $n \in \mathcal{V}(T)$ has a (possibly empty) set of *children*, denoted $\mathcal{C}_{T,r}(n)$, which contains all nodes n' adjacent to n such that all paths from n' to r contain n. A *leaf* of a rooted tree T is a non-root node of degree one. We use $\mathcal{L}(T)$ to denote the set of leaves of T.

3 Using Project-Join Trees for Projected Model Counting

We first describe an existing framework for performing unprojected model counting [17]. We then adapt this framework for projected model counting.

3.1 Project-Join Trees for Model Counting

This framework leverages Boolean formulas given in a factored representation, *conjunctive normal form (CNF)*. A *clause* is a non-empty disjunction of literals, and a *CNF formula* is a non-empty set (conjunction) of clauses. The key idea is to represent the computation as a rooted tree, called a *project-join tree*, where leaves correspond to clauses, and internal nodes correspond to Σ-projections [17].

Definition 4 (Project-Join Tree). *Let φ be a CNF formula. A* project-join tree *of φ is a tuple $\mathcal{T} = (T, r, \gamma, \pi)$ where*

- *T is a tree with root $r \in \mathcal{V}(T)$,*
- *$\gamma : \mathcal{L}(T) \to \varphi$ is a bijection from the leaves of T to the clauses of φ, and*
- *$\pi : \mathcal{V}(T) \setminus \mathcal{L}(T) \to 2^{\texttt{Vars}(\varphi)}$ is a labeling function on internal nodes.*

Moreover, $\mathcal{T}$ must satisfy the following two properties.

1. *The set $\{\pi(n) : n \in \mathcal{V}(T) \setminus \mathcal{L}(T)\}$ is a partition of $\texttt{Vars}(\varphi)$.*
2. *Let $n \in \mathcal{V}(T)$ be an internal node, x be a variable in $\pi(n)$, and c be a clause of φ. If $x \in \texttt{Vars}(c)$, then the leaf node $\gamma^{-1}(c)$ is a descendant of n.*

A project-join tree of a CNF formula φ can be used to compute the weighted model count of φ. The algorithm traverses the project-join tree from leaves to root, multiplying clauses according to the tree structure and additively projecting out variables according to π. This is formalized with the following definition.

Definition 5. *Let $\mathcal{T} = (T, r, \gamma, \pi)$ be a project-join tree and W be a literal-weight function over X. The* W-valuation *of a node n, denoted f_n^W, is*

$$f_n^W \equiv \begin{cases} [\gamma(n)] & \text{if } n \in \mathcal{L}(T) \\ \sum_{\pi(n)} \left(\prod_{o \in \mathcal{C}_{T,r}(n)} f_o^W \cdot \prod_{x \in \pi(n)} W_x \right) & \text{if } n \in \mathcal{V}(T) \setminus \mathcal{L}(T) \end{cases}$$

where $[\gamma(n)]$ is the pseudo-Boolean function represented by the clause $\gamma(n) \in \varphi$.

This leads to a two-phase algorithm for computing the weighted model count of a CNF formula φ. First, the *planning phase* builds a project-join tree (T, r, γ, π) of φ. Second, the *execution phase* computes f_r^W according to Definition 5. The following theorem asserts that f_r^W is the weighted model count of φ.

Theorem 1 ([17]). *Let φ be a CNF formula, $\mathcal{T} = (T, r, \gamma, \pi)$ be a project-join tree of φ, and W be a literal-weight function over $\texttt{Vars}(\varphi)$. Then $f_r^W(\varnothing)$ is the W-weighted model count of φ.*

When computing a W-valuation, the number of variables appearing in the intermediate pseudo-Boolean functions significantly influences the runtime. These variables are actually independent of W. For a node $n \in \mathcal{V}(T)$, define $\texttt{Vars}(n)$ as follows.

$$\texttt{Vars}(n) \equiv \begin{cases} \texttt{Vars}(\gamma(n)) & \text{if } n \in \mathcal{L}(T) \\ \left(\bigcup_{o \in \mathcal{C}_{T,r}(n)} \texttt{Vars}(o) \right) \setminus \pi(n) & \text{if } n \in \mathcal{V}(T) \setminus \mathcal{L}(T) \end{cases}$$

The W-valuation of a node n is then a pseudo-Boolean function over variables $\texttt{Vars}(n)$. If $N \subseteq \mathcal{V}(T)$, for convenience, we define $\texttt{Vars}(N) \equiv \bigcup_{n \in N} \texttt{Vars}(n)$.

The difficulty of valuation scales with the maximum number of variables needed to compute each pseudo-Boolean function. The *size* of a node n, $\texttt{size}(n)$, is defined as $|\texttt{Vars}(n)|$ for leaf nodes and $|\texttt{Vars}(n) \cup \pi(n)|$ for internal nodes. The *width* of a project-join tree $\mathcal{T} = (T, r, \gamma, \pi)$ is $\texttt{width}(\mathcal{T}) \equiv \max_{n \in \mathcal{V}(T)} \texttt{size}(n)$.

Two algorithms have been proposed to construct project-join trees [17]. The first, `LG`, uses *tree decompositions* [48], following similar work in join-query optimization [11, 41]. The second, `HTB`, uses *bucket elimination* [13] and *Bouquet's Method* [8] with various constraint-satisfaction heuristics: *maximum-cardinality search* [58], *lexicographic search for perfect/minimal orders* [35], and *min-fill* [14].

3.2 Adaptations for Projected Model Counting

In order to adapt this framework for weighted projected model counting, we aim to modify the valuation of project-join trees to incorporate disjunctive as well as additive projections. In particular, we must perform $\exists$-projections with all disjunctive variables and Σ-projections with all additive variables.

The challenge is that Σ-projections do not commute with $\exists$-projections. Since the $\exists$-projections appear on the inside of the expression for projected counting,

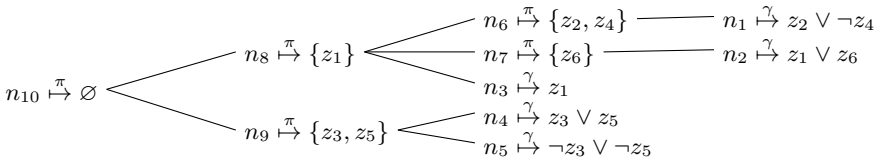

Fig. 1. A graded project-join tree $\mathcal{T} = (T, n_{10}, \gamma, \pi)$ of a CNF formula φ with relevant variables $X = \{z_1, z_3, z_5\}$ and irrelevant variables $Y = \{z_2, z_4, z_6\}$. Each leaf node corresponds to a clause of φ under γ. Each internal node is labeled by π with a set of variables of φ. Note that $\mathcal{T}$ is graded with grades $\mathcal{I}_X = \{n_8, n_9, n_{10}\}$ and $\mathcal{I}_Y = \{n_6, n_7\}$.

we must ensure that all $\exists$-projections occur before all Σ-projections while traversing the project-join tree. We formalize this by requiring the project-join tree to be *graded*.

Definition 6 (Graded Project-Join Tree). *Let φ be a CNF formula with project-join tree $\mathcal{T} = (T, r, \gamma, \pi)$, and let $\{X, Y\}$ be a partition of $\texttt{Vars}(\varphi)$. We say that $\mathcal{T}$ is* (X, Y)-graded *if there exist $\mathcal{I}_X, \mathcal{I}_Y \subseteq \mathcal{V}(T)$, called* grades*, that satisfy the following properties.*

1. *The set $\{\mathcal{I}_X, \mathcal{I}_Y\}$ is a partition of $\mathcal{V}(T) \setminus \mathcal{L}(T)$.*
2. *If $n_X \in \mathcal{I}_X$, then $\pi(n_X) \subseteq X$.*
3. *If $n_Y \in \mathcal{I}_Y$, then $\pi(n_Y) \subseteq Y$.*
4. *If $n_X \in \mathcal{I}_X$ and $n_Y \in \mathcal{I}_Y$, then n_X is not a descendant of n_Y in the rooted tree (T, r).*

Intuitively, a project-join tree is (X, Y)-graded if all X variables are projected (according to π) closer to the root than all Y variables in the tree. Figure 1 illustrates an exemplary graded project-join tree.

We now define a new valuation on graded project-join trees, which uses Σ-projections at nodes in $\mathcal{I}_X$ and $\exists$-projections at nodes in $\mathcal{I}_Y$.

Definition 7 (Projected Valuation). *Let (X, Y, φ, W) be a weighted projected model counting instance, and let $\mathcal{T} = (T, r, \gamma, \pi)$ be an (X, Y)-graded project-join tree of φ with grades $\mathcal{I}_X$ and $\mathcal{I}_Y$. The* W-projected-valuation *of each node $n \in \mathcal{V}(T)$, denoted g_n^W, is defined by*

$$g_n^W \equiv \begin{cases} [\gamma(n)] & \text{if } n \in \mathcal{L}(T) \\ \sum\limits_{\pi(n)} \left(\prod\limits_{o \in \mathcal{C}_{T,r}(n)} g_o^W \cdot \prod\limits_{x \in \pi(n)} W_x \right) & \text{if } n \in \mathcal{I}_X \\ \exists\limits_{\pi(n)} \left(\prod\limits_{o \in \mathcal{C}_{T,r}(n)} g_o^W \right) & \text{if } n \in \mathcal{I}_Y \end{cases}$$

where $[\gamma(n)]$ is the pseudo-Boolean function represented by the clause $\gamma(n) \in \varphi$.

If the project-join tree is graded, then the projected valuation of the root node is the weighted projected model count.

Theorem 2. *Let (X, Y, φ, W) be an instance of weighted projected model counting, and let $\mathcal{T}$ be a project-join tree of φ with root r. If $\mathcal{T}$ is (X, Y)-graded, then $g_r^W(\varnothing) = \texttt{WPMC}(\varphi, W, Y)$.*

In the next section, we show how to build graded project-join trees.

4 Building Graded Project-Join Trees

We now show how building graded project-join trees can be reduced to building ungraded project-join trees. This allows us to use prior work on ungraded project-join trees [17] to compute graded project-join trees.

As a building block, we first show how constructing project-join trees with free variables can be reduced to constructing ungraded project-join trees. This both illustrates the key ideas of our approach and appears as a subroutine in the larger graded reduction.

4.1 Reducing Free Project-Join Trees to Ungraded Project-Join Trees

Project-join trees project out every variable in the set of corresponding clauses. This is desirable for applications where all variables are processed in the same way, e.g., model counting. In many other applications, however, it is desirable to process a set of clauses while leaving specified *free variables* untouched.

We model free variables by ensuring that they are projected in the project-join tree as late as possible, at the root node. Thus free variables must be "kept alive" throughout the entire tree.

Definition 8. *Let F be a set of variables, and let $\mathcal{T} = (T, r, \gamma, \pi)$ be a project-join tree. We say that $\mathcal{T}$ is* F-free *if $F = \pi(r)$.*

Note that Definition 8 is a much stronger restriction than Definition 6. In particular, if a project-join tree $\mathcal{T}$ of a CNF formula φ is F-free, then $\mathcal{T}$ is also $(F, \texttt{Vars}(\varphi) \setminus F)$-graded.

We now reduce the problem of building F-free project-join trees to building ungraded project-join trees. One approach is to build a project-join tree while ignoring all variables in F, then insert the variables in F as projections at the root. However, building minimal-width project-join trees while ignoring variables may not produce minimal-width F-free project-join trees for the full formula.

Instead, we adapt a similar reduction in the context of tensor networks [16] for the context of project-join trees. The key idea is to add to φ a *virtual clause* that contains all variables in F. For a set Z of variables, let $\texttt{virtual}(Z)$ denote a fresh clause with variables Z. Project-join trees of $\varphi \cup \{\texttt{virtual}(F)\}$ can then be used to find F-free project-join trees of φ. This virtual clause can be viewed as a goal atom in `DataLog` [39].

This reduction is presented as Algorithm 1. The input $\mathcal{T}$ is a project-join tree of $\varphi \cup \{C_F\}$, where C_F is a virtual clause with variables F. On lines 2-6, we rotate $\mathcal{T}$ so that the leaf node s corresponding to C_F becomes the root node. This rotation does not increase the width. Projecting F at the new root s still does not increase the width. Thus we obtain an F-free project-join tree of φ.

Algorithm 1: Building an F-free project-join tree of a CNF formula

Input: φ: a CNF formula
Input: F: a subset of $\texttt{Vars}(\varphi)$
Input: $\mathcal{T} = (T, r, \gamma, \pi)$: a project-join tree of $\varphi \cup \{C_F\}$, where $C_F = \texttt{virtual}(F)$ is a fresh clause with variables F
Output: an F-free project-join tree of φ

1 $s \leftarrow \gamma^{-1}(C_F)$ // s will be the root node of the returned project-join tree
2 $\pi' \leftarrow$ a mapping where $\pi'(n) = \varnothing$ for all $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$ // π' will be the labeling function of the returned project-join tree
3 **for** $y \in \texttt{Vars}(\varphi) \setminus F$
4 $\quad \varphi_y = \{C \in \varphi : y \in \texttt{Vars}(C)\}$
5 $\quad i \leftarrow$ lowest common ancestor of $\left\{\gamma^{-1}(C) : C \in \varphi_y\right\}$ in the rooted tree (T, s)
6 $\quad \pi'(i) \leftarrow \pi'(i) \cup \{y\}$ // project out y at the lowest allowable node
7 $\pi'(s) \leftarrow F$ // project out variables in F at the new root s
8 $\gamma' \leftarrow \gamma \setminus \{s \mapsto C_F\}$ // γ' is the bijection γ without the pair (s, C_F)
9 **return** (T, s, γ', π')

We state the correctness of Algorithm 1 in the following theorem. In particular, the width of the output F-free project-join tree is no worse than the width of the unrestricted input tree.

Theorem 3. *Let φ be a CNF formula, and let $F \subseteq \texttt{Vars}(\varphi)$. If $\mathcal{T}$ is a project-join tree of $\varphi \cup \{\texttt{virtual}(F)\}$, then Algorithm 1 returns an F-free project-join tree of φ of width at most $\texttt{width}(\mathcal{T})$.*

We also prove that Algorithm 1 is optimal. That is, a minimal-width project-join tree for $\varphi \cup \{C_F\}$ produces a minimal-width F-free project-join tree for φ.

Theorem 4. *Let φ be a CNF formula, F be a subset of $\texttt{Vars}(\varphi)$, and w be a positive integer. If there is an F-free project-join tree of φ of width w, then there is a project-join tree of $\varphi \cup \{\texttt{virtual}(F)\}$ of width w.*

4.2 Reducing Graded Project-Join Trees to Free Project-Join Trees

In this section, we use free project-join trees as a building block to construct graded project-join trees. We present this framework as Algorithm 2. The key idea is to create a graded project-join tree by combining many free project-join trees for subformulas. We first combine clauses to remove Y variables, then we combine project-join-tree components to remove X variables.

In detail, on line 1, we partition the clauses of φ into blocks that share Y variables. On line 3, we find a project-join tree $\mathcal{T}_N$ for each block N. This tree must keep all X variables free, i.e., must be $(\texttt{Vars}(N) \cap X)$-free. The trees $\{\mathcal{T}_N\}$ collectively project out all Y variables. On line 6, we construct a project-join tree $\mathcal{T}$ that will guide the combination of all trees in $\{\mathcal{T}_N\}$ while projecting out all X variables, where each $\mathcal{T}_N$ is represented by the corresponding virtual clause C_N. On lines 7-9, we hook the trees in $\{\mathcal{T}_N\}$ together as indicated by $\mathcal{T}$.

Algorithm 2: Building a graded project-join tree of a CNF formula

Input: X: a set of Σ-variables
Input: Y: a set of $\exists$-variables where $X \cap Y = \varnothing$
Input: φ: a CNF formula where $\texttt{Vars}(\varphi) = X \cup Y$
Output: $\mathcal{T}$: an (X, Y)-graded project-join tree of φ

```
partition ← GroupBy(φ, Y) // group clauses that share Y variables
for N ∈ partition
    T_N ← BuildComponent(N, Vars(N) ∩ X) // build a (Vars(N) ∩ X)-free
          project-join tree of N
    T_N ← T_N with all projections at the root of T_N removed
    C_N ← virtual(Vars(N) ∩ X)
T ← BuildComponent({C_N : N ∈ partition}, ∅) // build a project-join tree
    from virtual clauses C_N
for N ∈ partition
    ℓ_N ← leaf of T corresponding to C_N
    T ← T with ℓ_N replaced by T_N
return T
```

The function $\texttt{GroupBy}(\varphi, Y)$ in Algorithm 2 partitions the clauses of φ so that every pair of clauses that share a variable from Y appear together in the same block of the partition. A formal definition follows.

Definition 9. *Let φ be a set of clauses and Y be a subset of $\texttt{Vars}(\varphi)$. Define $\sim_Y \subseteq \varphi \times \varphi$ to be the relation such that, for clauses $c, c' \in \varphi$, we have $c \sim_Y c'$ if and only if $\texttt{Vars}(c) \cap \texttt{Vars}(c') \cap Y \neq \varnothing$. Then $\texttt{GroupBy}(\varphi, Y)$ is the set of equivalence classes of the reflexive transitive closure of $\sim_Y$.*

The intuition is that two clauses in the same block in $\texttt{GroupBy}(\varphi, Y)$ must be combined to project out all variables in Y. Conversely, clauses that appear in separate blocks need not be combined in order to project out all variables in Y.

In Algorithm 2, each function call $\texttt{BuildComponent}(\alpha, F)$ returns an F-free project-join tree of α, where α is a set of clauses and $F \subseteq \texttt{Vars}(\alpha)$. `BuildComponent` can be implemented by implementing Algorithm 1 on top of an algorithm for building ungraded project-join trees. For example, in Section 5, we consider two implementations of Algorithm 2 built on top of the two algorithms to construct standard project-join trees [17] discussed at the end of Section 3.1.

We next state the correctness of Algorithm 2 and show that the width of the output graded project-join tree is no worse than the widths of the trees used for the components.

Theorem 5. *Let φ be a CNF formula, $\{X, Y\}$ be a partition of $\texttt{Vars}(\varphi)$, and w be a positive integer. Assume each call to $\texttt{BuildComponent}(\alpha, F)$ returns an F-free project-join tree for α of width at most w. Then Algorithm 2 returns an (X, Y)-graded project-join tree for φ of width at most w.*

Although Algorithm 2 constructs a sequence of small ungraded project-join trees, it is sufficient to compute a single ungraded project-join tree from which all smaller trees can be extracted. This is demonstrated by the following theorem.

Theorem 6. *Let φ be a CNF formula, $\{X, Y\}$ be a partition of* $\texttt{Vars}(\varphi)$*, and ψ be the CNF formula $\varphi \cup \{\texttt{virtual}(\texttt{Vars}(N) \cap X) : N \in \texttt{GroupBy}(\varphi, Y)\}$. For every positive integer w, if there is a project-join tree $\mathcal{T}'$ for ψ of width w, then there is an (X, Y)-graded project-join tree for φ of width at most w.*

The key idea of the proof is to answer every `BuildComponent` call in Algorithm 2 by extracting a subtree of $\mathcal{T}'$ and applying Theorem 3.

We show in the following theorem that this approach is optimal. Thus (X, Y)-graded project-join trees of φ are equivalent to project-join trees of ψ.

Theorem 7. *Let φ be a CNF formula, $\{X, Y\}$ be a partition of* $\texttt{Vars}(\varphi)$*, and ψ be the CNF formula $\varphi \cup \{\texttt{virtual}(\texttt{Vars}(N) \cap X) : N \in \texttt{GroupBy}(\varphi, Y)\}$. For every positive integer w, if there is an (X, Y)-graded project-join tree for φ of width w, then there is a project-join tree for ψ of width w.*

Note that requiring the project-join tree to be graded may significantly increase the width of available project-join trees. Theorems 5 and 7 indicate that our algorithm for constructing a graded project-join tree pays no additional cost in width beyond what is required by gradedness.

5 Experimental Evaluation

To implement our projected model counter `ProCount`, we modify the unprojected model counter `DPMC`, which is based on ungraded project-join trees [17]. The `DPMC` framework includes: (1) the `LG` planner that uses tree-decomposition techniques, (2) the `HTB` planner that uses constraint-satisfaction heuristics, and (3) the `DMC` executor that uses *algebraic decision diagrams (ADDs)*. We generalize these three components to support graded project-join trees and projected model counting.

We conduct three experiments to address the following research questions.

(RQ1) In the planning phase (for constructing project-join trees), how do tree-decomposition techniques compare to constraint-satisfaction heuristics?
(RQ2) In the execution phase, how do different ADD variable orders compare?
(RQ3) How does `ProCount` compare to other exact weighted projected counters?

To answer RQ1, in Experiment 1, we compare the planner `LG` (which uses tree decompositions) and the planner `HTB` (which uses constraint-satisfaction heuristics). `LG` uses the tree decomposers `FlowCutter` [28], `htd` [2], and `Tamaki` [57]. `HTB` implements four heuristics for variable ordering: maximal-cardinality search (`MCS`) [58], lexicographic search for perfect/minimal orders (`LP`/`LM`) [35], and min-fill (`MF`) [14]. `HTB` also implements two clause-ordering heuristics: bucket elimination (`BE`) [13] and Bouquet's Method (`BM`) [8].

To answer RQ2, in Experiment 2, we compare variable-ordering heuristics for the ADD-based executor `DMC`. An ADD [6] is a directed acyclic graph that compactly represents a pseudo-Boolean function. An ADD requires a variable order, which strongly influences the compactness of the ADD. `DMC` implements four aforementioned variable-ordering heuristics: `MCS`, `LP`, `LM`, and `MF`. Note that we use ADDs throughout the entire execution for consistency, although binary decision diagrams [9] or SAT solvers would suffice to valuate existential nodes.

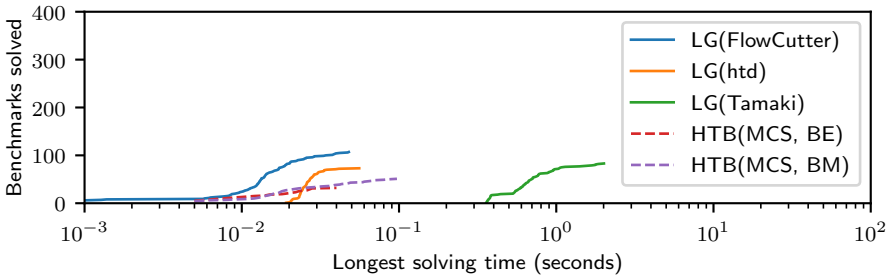

Fig. 2. Experiment 1 compares the tree-decomposition-based planner `LG` to the constraint-satisfaction-based planner `HTB`. A planner "solves" a benchmark when it finds a project-join tree of width 30 or lower. For `HTB`, we only show the variable-ordering heuristic `MCS`; the `LP`, `LM`, and `MF` curves are qualitatively similar.

To answer RQ3, in Experiment 3, we compare `ProCount` to state-of-the-art exact weighted projected model counters $\texttt{D4}_\texttt{P}$ [37], `projMC` [37], and `reSSAT` [38].

We use 849 CNF benchmarks gathered from two families. The first family contains 90 formulas and was used for weighted projected sampling [27]. For each benchmark in this family, a positive literal x has weight $0 < W_x(\{x\}) < 1$, and a negative literal $\neg x$ has weight $W(\varnothing) = 1 - W_x(\{x\})$. The second family contains 759 formulas and was used for unweighted projected model counting [52]. We add weights to this family by randomly assigning $W_x(\{x\}) = 0.4$ and $W_x(\varnothing) = 0.6$ or vice versa to each variable x. All 849 benchmarks are satisfiable, as verified by the SAT solver `CryptoMiniSat` [53]. We run all experiments on single CPU cores of a Linux cluster with Intel Xeon E5-2650v2 processors (2.60-GHz) and 30 GB of RAM. All code and data are available (`https://github.com/vardigroup/DPMC`).

5.1 Experiment 1: Comparing Planners

In this experiment, we run all configurations of the planners `LG` and `HTB` on each CNF benchmark with a timeout of 100 seconds. We present results in Figure 2. Each point (x, y) on a plotted curve indicates that: within x seconds, on each of y benchmarks, the first graded project-join tree produced by the corresponding planner has width at most 30. We choose 30 because previous work shows that executors do not handle larger project-join trees well [16, 17]. While `LG` is an *anytime* tool that produces several trees (of decreasing widths) for each benchmark, we only use the first tree. The tree-decomposition-based planner `LG` produces more low-width trees than the constraint-satisfaction-based planner `HTB`. Moreover, for `LG`, the tree decomposer `FlowCutter` is faster than `htd` and `Tamaki`. Thus we use `LG` with `FlowCutter` in `ProCount` for later experiments.

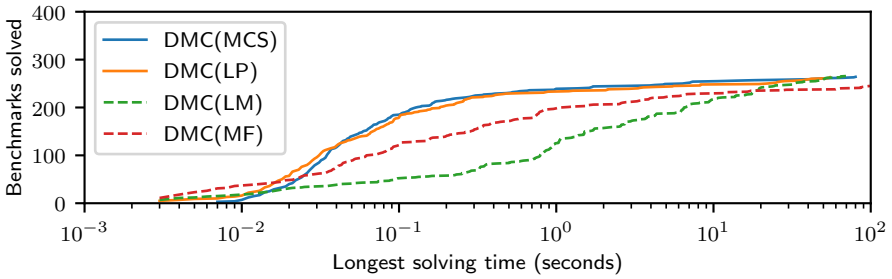

Fig. 3. Experiment 2 compares variable-ordering heuristics (`MCS`, `LP`, `LM`, and `MF`) for the ADD-based executor `DMC`. `MCS` and `LP` are significantly faster than `LM` and `MF`.

5.2 Experiment 2: Comparing Execution Heuristics

In this experiment, we take all 346 graded project-join trees produced by `LG` with `FlowCutter` in Experiment 1 and run `DMC` for 100 seconds with each ADD variable-ordering heuristic. We present the execution time of each heuristic (excluding planning time) in Figure 3. We observe that `MCS` and `LP` outperform `LM` and `MF`. We use `DMC` with `MCS` in `ProCount` for Experiment 3.

5.3 Experiment 3: Comparing Weighted Projected Model Counters

Informed by Experiments 1 and 2, we choose `LG` with `FlowCutter` as the planner and `DMC` with `MCS` as the executor for our framework `ProCount`. We compare `ProCount` with the weighted projected model counters $\texttt{D4}_\texttt{P}$, `projMC`, and `reSSAT`. Since all benchmarks are satisfiable with positive literal weights, the model counts must be positive. Thus, for all tools, we exclude outputs that are zero (possible floating-point underflow). We are confident that the remaining results are correct. Differences in model counts among tools are less than 10^{-6}.

Figure 4 shows the performance of `ProCount`, $\texttt{D4}_\texttt{P}$, `projMC`, and `reSSAT` with a 1000-second timeout. Additional statistics are given in Table 1. Of 849 benchmarks, 390 are solved by at least one of four tools. `ProCount` achieves the shortest solving time on 131 benchmarks, including 44 solved by none of the other three tools. Between the two *virtual best solvers* in Figure 4, `VBS1` (all four tools) is significantly faster than `VBS0` (three existing tools, without `ProCount`).

Project-Join Tree Width and Computation Time To identify which type of benchmarks can be solved efficiently by `ProCount`, we study how the performance of each projected model counter varies with the widths of graded project-join trees. In particular, for each benchmark, we consider the width of the first graded project-join tree produced by the planner `LG` (with `FlowCutter`) in Experiment 1. Figure 5 shows how these widths relate to mean PAR-2 scores of projected model counters. `ProCount` seems to be the fastest solver on instances for which there exist graded project-join trees of widths between 50 and 100.

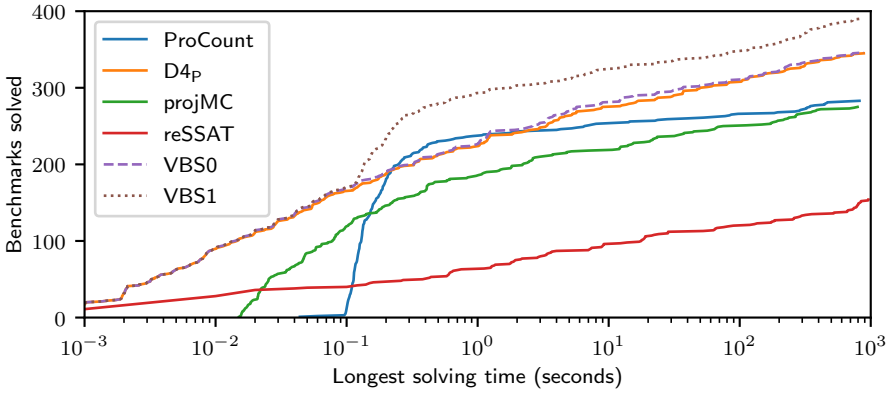

Fig. 4. Experiment 3 compares our framework `ProCount` to the state-of-the-art exact weighted projected model counters `D4P`, `projMC`, and `reSSAT`. `VBS0` is the virtual best solver of the three existing tools, excluding `ProCount`. `VBS1` includes all four tools. Adding `ProCount` significantly improves the portfolio of projected model counters.

6 Related Work

There are a number of recent tools for projected model counting. For example, `D4P` uses decision decomposable negation normal form [37], `projMC` leverages disjunctive decomposition [37], and `reSSAT` combines counting with SAT techniques [38]. While our focus in this work is on (deterministic) exact weighted projected model counting, it is worth mentioning that various relaxations have also been studied, e.g., probabilistic [51], approximate [21,26,52], or unweighted [4,29,42,63] projected model counting.

A recent framework for projected counting is `nestHDB` [29], a hybrid solver. Similar to our framework, `nestHDB` includes a planning phase (using the tree-decomposition tool `htd` [2]) and an execution phase (using the database engine `Postgres` [55] and the projected counter `projMC` [37], alongside other tools). We predict that `nestHDB` may benefit from switching the projected-counting component to `ProCount`, which was often faster than `projMC` in Experiment 3. While we were unable to run a full experimental comparison[1] with `nestHDB`, we evaluated `nestHDB` against `ProCount` on 90 benchmarks [27] (with weights removed) using a single CPU core of an Intel i7-7700HQ processor (2.80-GHz) with 30 GB of RAM. `ProCount` and `nestHDB` respectively solved 69 and 59 benchmarks, with a 100-second timeout. The mean PAR-2 scores for `ProCount` and `nestHDB` were 47 and 87. Further comparison is needed in future work.

Our proposed graded project-join trees can be seen as a specialization of *structure trees* [54] to the case of projected model counting. Sterns and Hunt [54] suggest constructing structure trees by manually modifying tree decomposers to

[1] `nestHDB` is an unweighted tool, but the benchmarks in Section 5 are weighted. Moreover, the cluster used in Section 5 does not support database management systems.

Table 1. Experiment 3 compares our framework `ProCount` to the state-of-the-art exact weighted projected model counters `D4P`, `projMC`, and `reSSAT`. There are 390 benchmarks solved by at least one of four tools. By including `ProCount`, the portfolio of tools solves 44 more benchmarks and achieves shorter solving time on 87 other benchmarks. For each tool-benchmark pair, the PAR-2 score is the runtime if the tool solves the benchmark (within time and space limits) or twice the 1000-second timeout otherwise.

Solver	Number of benchmarks solved (of 849)			Mean PAR-2 score
	Uniquely (solved by no other)	In shortest time	In total	
`ProCount`	44	131	283	1341
`D4P`	50	235	345	1203
`projMC`	0	8	275	1362
`reSSAT`	1	16	154	1659
`VBS0`	NA	NA	346	1199
`VBS1`	NA	NA	390	1099

consider only structure trees respecting the variable quantification order (i.e., to consider gradedness directly). In this work, we take a different approach by using existing tools for standard project-join trees (in particular, tree decomposers) in a black-box way. This is crucial for the practical success of our tool, as we can leverage continual progress in tree decomposition.

Projected model counting is also a special case of *functional aggregate queries (FAQs)* [1]. Our graded project-join trees can be seen as a specialization of FAQ variable orders. Theorem 7.5 of [1] gives an algorithm for constructing an FAQ variable order from a sequence of tree decompositions, which, in the context of projected model counting, is equivalent to the technique we discussed in Section 4.1 of ignoring relevant variables while planning to project irrelevant variables. In contrast, our approach may find lower-width graded project-join trees by incorporating relevant variables even when planning to project irrelevant variables. This improvement may be lifted to the FAQ framework in future work.

It is worth comparing our theoretical results to a different algorithm for projected counting [24], which runs on a formula φ in time $2^{2^{O(k)}} \cdot p(\varphi)$, where k is the *primal treewidth* [49] of φ, and p scales polynomially in the size of φ. Assuming the Exponential-Time Hypothesis [30], all FPT algorithms parameterized by primal treewidth must be double-exponential [24]. On the other hand, by Theorem 5 in [17] and Theorem 6 here, our algorithm, based on graded project-join trees, runs in time $2^{O(k')}$, where k' is the primal treewidth of ψ (which we call the $\{X, Y\}$-*graded treewidth* of φ). While k' is larger than k, we can see that k' is significantly smaller than 2^k on many benchmarks.

In some sense, projected model counting on Boolean formulas is a dual problem of *maximum a posteriori (MAP)* inference [40, 43, 62] on Bayesian networks [47]: a projected model count has the form $\sum_X \max_Y f(X, Y)$, while a MAP probability has the form $\max_Y \sum_X f(X, Y)$. Both problems can be solved using variable elimination, but an elimination order may not freely interleave X variables with Y variables. A valid variable order induces an *evaluation tree* (similar to a project-join tree) [46]. As mentioned in [46], exact MAP algorithms construct

Fig. 5. We plot mean PAR-2 scores (in seconds) against mean project-join tree widths. Each projected counter in Experiment 3 corresponds to a plotted curve, on which a point (x, y) indicates that: x is the central moving average of 10 consecutive project-join tree widths $1 \leq w_1 < w_2 < \ldots < w_{10} \leq 99$, and y is the average PAR-2 score of the benchmarks whose project-join trees have widths w s.t. $w_1 \leq w \leq w_{10}$. We observe that the performance of `ProCount` degrades as the project-join tree width increases. However, `ProCount` tends to be the fastest solver on benchmarks whose graded project-join trees have widths roughly between 50 and 100.

evaluation trees using constraint-satisfaction heuristics (similar to our planner `HTB`). Our work goes further by constructing low-width graded project-join trees using tree-decomposition techniques (with our planner `LG`) and by performing efficient computations using compact ADDs (with our executor `DMC`).

7 Discussion

We adapted an existing dynamic-programming framework [17] to perform projected model counting by requiring project-join trees to be graded. This framework decomposes projected model counting into two phases. First, the planning phase produces a graded project-join tree from a CNF formula. Second, the execution phase uses the this tree to guide the computation of the projected model count of the formula w.r.t. a literal-weight function. We proved that algorithms for building project-join trees can be used to build graded project-join trees. Our framework `ProCount` is competitive with the exact weighted projected model counters `D4P` [37], `projMC` [37], and `reSSAT` [38]. `ProCount` considerably improves the virtual best solver and thus is a valuable addition to the portfolio.

In future work, `ProCount` can be generalized for maximum model counting [26] and functional aggregate queries [1]. Another research direction is multicore programming. The planning tool `LG` can be improved to run tree decomposers in parallel [19] in a portfolio approach [61]. One can also make the execution tool `DMC` support multicore ADD packages (e.g., `Sylvan` [15]).

References

1. Abo Khamis, M., Ngo, H.Q., Rudra, A.: FAQ: questions asked frequently. In: PODS. pp. 13–28 (2016)
2. Abseher, M., Musliu, N., Woltran, S.: htd–a free, open-source framework for (customized) tree decompositions and beyond. In: CPAIOR. pp. 376–386 (2017). https://doi.org/10.1007/978-3-319-59776-8_30
3. Aguirre, A.S.M., Vardi, M.: Random 3-SAT and BDDs: the plot thickens further. In: CP. pp. 121–136 (2001). https://doi.org/10.1007/3-540-45578-7_9
4. Aziz, R.A., Chu, G., Muise, C., Stuckey, P.: Projected model counting. In: SAT. pp. 121–137 (2015)
5. Bacchus, F., Dalmao, S., Pitassi, T.: Solving #SAT and Bayesian inference with backtracking search. JAIR **34**, 391–442 (2009). https://doi.org/10.1613/jair.2648
6. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. Form Method Syst Des **10**(2-3), 171–206 (1997). https://doi.org/10.1023/A:1008699807402
7. Bellman, R.: Dynamic programming. Science **153**(3731), 34–37 (1966). https://doi.org/10.1126/science.153.3731.34
8. Bouquet, F.: Gestion de la dynamicité et énumération d'impliquants premiers: une approche fondée sur les Diagrammes de Décision Binaire. Ph.D. thesis, Aix-Marseille 1 (1999), `https://www.theses.fr/1999AIX11011`
9. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE TC **100**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
10. Charwat, G., Woltran, S.: BDD-based dynamic programming on tree decompositions. Tech. rep., Technische Universität Wien, Institut für Informationssysteme (2016), `https://dbai.tuwien.ac.at/research/report/dbai-tr-2016-95.pdf`
11. Dalmau, V., Kolaitis, P.G., Vardi, M.Y.: Constraint satisfaction, bounded treewidth, and finite-variable logics. In: CP. pp. 310–326 (2002). https://doi.org/10.1007/3-540-46135-3_21
12. Darwiche, A.: New advances in compiling CNF to decomposable negation normal form. In: ECAI. pp. 318–322 (2004), `https://dl.acm.org/doi/10.5555/3000001.3000069`
13. Dechter, R.: Bucket elimination: a unifying framework for reasoning. AIJ **113**(1-2), 41–85 (1999). https://doi.org/10.1016/S0004-3702(99)00059-4
14. Dechter, R.: Constraint processing. Morgan Kaufmann (2003). https://doi.org/10.1016/B978-1-55860-890-0.X5000-2
15. van Dijk, T., van de Pol, J.: Sylvan: multi-core decision diagrams. In: TACAS. pp. 677–691 (2015). https://doi.org/10.1007/978-3-662-46681-0_60
16. Dudek, J.M., Dueñas-Osorio, L., Vardi, M.Y.: Efficient contraction of large tensor networks for weighted model counting through graph decompositions. arXiv preprint arXiv:1908.04381 (2019), `https://arxiv.org/abs/1908.04381`
17. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: DPMC: weighted model counting by dynamic programming on project-join trees. In: CP. pp. 211–230 (2020), `arxiv.org/abs/2008.08748`
18. Dudek, J.M., Phan, V.H., Vardi, M.Y.: ADDMC: weighted model counting with algebraic decision diagrams. In: AAAI. vol. 34, pp. 1468–1476 (2020). https://doi.org/10.1609/aaai.v34i02.5505
19. Dudek, J.M., Vardi, M.Y.: Parallel weighted model counting with tensor networks. In: MCW (2020), `https://mccompetition.org/assets/files/2020/MCW_2020_paper_1.pdf`

20. Duenas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: AAAI. pp. 4488–4494 (2017)
21. Ermon, S., Gomes, C., Sabharwal, A., Selman, B.: Taming the curse of dimensionality: discrete integration by hashing and optimization. In: ICML. pp. 334–342 (2013)
22. Fichte, J.K., Hecher, M.: Counting with bounded treewidth: meta algorithm and runtime guarantees. In: NMR. pp. 9–18 (2020)
23. Fichte, J.K., Hecher, M., Hamiti, F.: The Model Counting Competition 2020. arXiv preprint arXiv:2012.01323 (2020)
24. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Exploiting treewidth for projected model counting and its limits. In: SAT. pp. 165–184 (2018)
25. Fichte, J.K., Hecher, M., Thier, P., Woltran, S.: Exploiting database management systems and treewidth for counting. In: PADL. pp. 151–167 (2020). https://doi.org/10.1007/978-3-030-39197-3_10
26. Fremont, D.J., Rabe, M.N., Seshia, S.A.: Maximum model counting. In: AAAI. pp. 3885–3892 (2017)
27. Gupta, R., Sharma, S., Roy, S., Meel, K.S.: WAPS: weighted and projected sampling. In: TACAS. pp. 59–76 (2019)
28. Hamann, M., Strasser, B.: Graph bisection with pareto optimization. JEA **23**, 1–34 (2018)
29. Hecher, M., Thier, P., Woltran, S.: Taming high treewidth with abstraction, nested dynamic programming, and database technology. In: SAT. pp. 343–360 (2020)
30. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity? JCSS **63**(4), 512–530 (2001)
31. Jégou, P., Kanso, H., Terrioux, C.: Improving exact solution counting for decomposition methods. In: ICTAI. pp. 327–334 (2016). https://doi.org/10.1109/ICTAI.2016.0057
32. Jerrum, M.R., Valiant, L.G., Vazirani, V.V.: Random generation of combinatorial structures from a uniform distribution. Theoretical computer science **43**, 169–188 (1986)
33. Kelly, C., Sarkhel, S., Venugopal, D.: Adaptive Rao-Blackwellisation in Gibbs sampling for probabilistic graphical models. In: AISTATS. pp. 2907–2915 (2019)
34. Klebanov, V., Manthey, N., Muise, C.: SAT-based analysis and quantification of information flow in programs. In: QEST. pp. 177–192 (2013). https://doi.org/10.1007/978-3-642-40196-1_16
35. Koster, A.M., Bodlaender, H.L., Van Hoesel, S.P.: Treewidth: computational experiments. Electron Notes Discrete Math **8**, 54–57 (2001). https://doi.org/10.1016/S1571-0653(05)80078-2
36. Lagniez, J.M., Marquis, P.: An improved decision-DNNF compiler. In: IJCAI. pp. 667–673 (2017). https://doi.org/10.24963/ijcai.2017/93
37. Lagniez, J.M., Marquis, P.: A recursive algorithm for projected model counting. In: AAAI. vol. 33, pp. 1536–1543 (2019)
38. Lee, N.Z., Wang, Y.S., Jiang, J.H.R.: Solving stochastic Boolean satisfiability under random-exist quantification. In: IJCAI. pp. 688–694 (2017)
39. Lloyd, J.W.: Foundations of logic programming. Springer Science & Business Media (2012)
40. Maua, D.D., de Campos, C.P., Cozman, F.G.: The complexity of MAP inference in Bayesian networks specified through logical languages. In: IJCAI. pp. 889–895 (2015)
41. McMahan, B.J., Pan, G., Porter, P., Vardi, M.Y.: Projection pushing revisited. In: EDBT. pp. 441–458 (2004). https://doi.org/10.1007/978-3-540-24741-8_26

42. Möhle, S., Biere, A.: Dualizing projected model counting. In: ICTAI. pp. 702–709 (2018)
43. Murphy, K.P.: Machine learning: a probabilistic perspective. MIT press (2012)
44. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: IJCAI. pp. 3141–3148 (2015), https://dl.acm.org/doi/10.5555/2832581.2832687
45. Pan, G., Vardi, M.Y.: Symbolic techniques in satisfiability solving. J Autom Reasoning **35**(1-3), 25–50 (2005). https://doi.org/10.1007/s10817-005-9009-7
46. Park, J.D., Darwiche, A.: Complexity results and approximation strategies for MAP explanations. JAIR **21**, 101–133 (2004)
47. Pearl, J.: Bayesian networks: A model cf self-activated memory for evidential reasoning. In: Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, CA, USA. pp. 15–17 (1985)
48. Robertson, N., Seymour, P.D.: Graph minors. X. Obstructions to tree-decomposition. J Combinatorial Theory B **52**(2), 153–190 (1991). https://doi.org/10.1016/0095-8956(91)90061-N
49. Samer, M., Szeider, S.: Algorithms for propositional model counting. J Discrete Algorithms **8**(1), 50–64 (2010). https://doi.org/10.1007/978-3-540-75560-9_35
50. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. SAT **4**, 20–28 (2004), http://www.satisfiability.org/SAT04/accepted/65.html
51. Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: a scalable probabilistic exact model counter. In: IJCAI. pp. 1169–1176 (2019)
52. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: AAAI. vol. 33, pp. 1592–1599 (2019)
53. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: SAT. pp. 244–257 (2009)
54. Stearns, R.E., Hunt III, H.B.: Exploiting structure in quantified formulas. Journal of Algorithms **43**(2), 220–263 (2002)
55. Stonebraker, M., Rowe, L.A.: The design of Postgres. ACM Sigmod Record **15**(2), 340–355 (1986)
56. Tabajara, L.M., Vardi, M.Y.: Factored Boolean functional synthesis. In: FMCAD. pp. 124–131 (2017), https://dl.acm.org/doi/10.5555/3168451.3168480
57. Tamaki, H.: Positive-instance-driven dynamic programming for treewidth. J Comb Optim **37**(4), 1283–1311 (2019). https://doi.org/10.1007/s10878-018-0353-z
58. Tarjan, R.E., Yannakakis, M.: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SICOMP **13**(3), 566–579 (1984). https://doi.org/10.1137/0213035
59. Uribe, T.E., Stickel, M.E.: Ordered binary decision diagrams and the Davis-Putnam procedure. In: CCL. pp. 34–49 (1994). https://doi.org/10.1007/BFb0016843
60. Wetzler, N., Heule, M.J., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: SAT. pp. 422–429. Springer (2014)
61. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. JAIR **32**, 565–606 (2008)
62. Xue, Y., Li, Z., Ermon, S., Gomes, C.P., Selman, B.: Solving marginal MAP problems with NP oracles and parity constraints. In: NIPS. pp. 1127–1135 (2016)
63. Zawadzki, E.P., Platzer, A., Gordon, G.J.: A generalization of SAT and #SAT for robust policy evaluation. In: IJCAI. pp. 2583–2589 (2013)

A Additional Experimental Evaluation

A.1 Experiment 1: Comparing Planners

Figure 6 illustrates how the planner `LG` compares to the planner `HTB` across several settings. `LG` is an *anytime* tool, which produces better and better project-join trees the longer it runs. In Experiment 1, within 100 seconds, each `LG` setting may produce several project-join trees (of decreasing widths) for a single benchmark. Figure 6 plots the time of the first such project-join tree of width at most 30. `HTB` is a *one-shot* tool, which produces only one project-join tree for each benchmark.

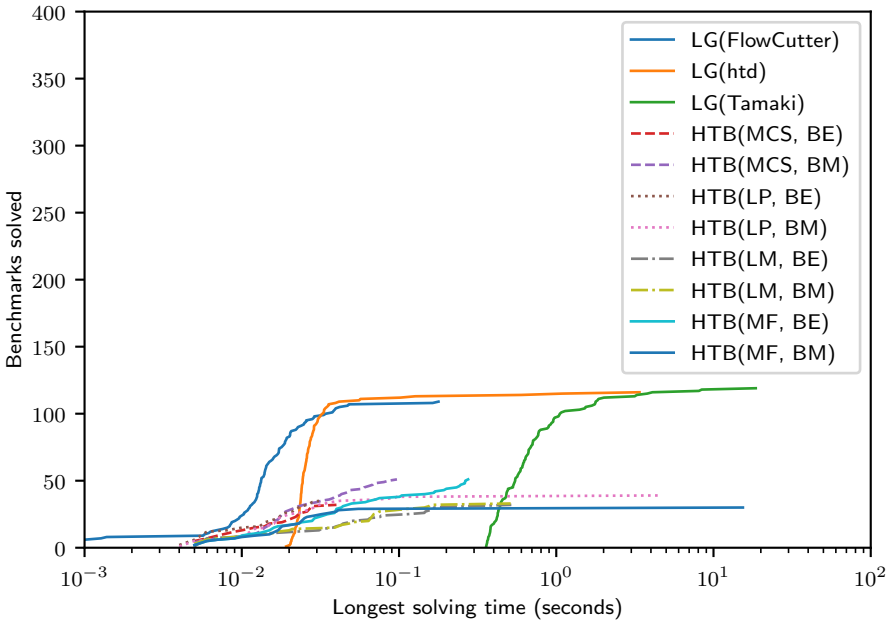

Fig. 6. Experiment 1 compares the tree-decomposition-based planner `LG` to the constraint-satisfaction-based planner `HTB`. `LG` can be used with a tree decomposer (`FlowCutter` [28], `htd` [2], or `Tamaki` [57]). `HTB` requires a variable-ordering heuristic (`MCS` [58], `LP`/`LM` [35], or `MF` [14]) and a clause-ordering heuristic (`BE` [13] or `BM` [8]). A planner "solves" a benchmark when it eventually finds a project-join tree of width 30 or lower. `LG` is an anytime tool that can output several trees (of decreasing widths) for each benchmark. On this plot, for each `LG` benchmark, we use the time of the first tree whose width is at most 30. In contrast, in Figure 2, we discard an `LG` benchmark when the first tree has width over 30, even if a later tree has width at most 30.

A.2 Experiment 2: Comparing Execution Heuristics

Figure 7 shows the performance of four ADD variable-ordering heuristics with the executor `DMC` in 100 seconds (execution time only, excluding planning time). The graded project-join trees here are taken from Experiment 1. Recall that `LG` is an anytime tool that may produce several project-join trees (of decreasing widths) for each benchmark. We measure the execution time using the first tree and the last tree produced within 100 seconds for each benchmark.

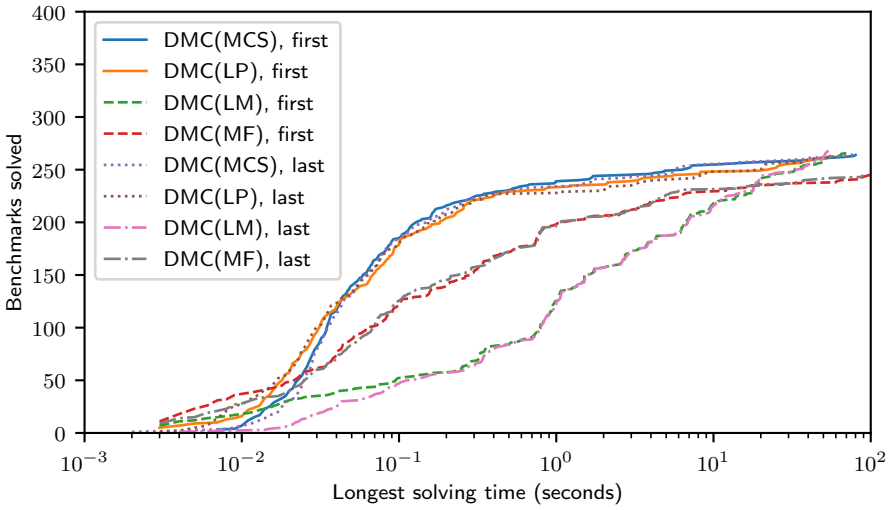

Fig. 7. Experiment 2 compares various variable-ordering heuristics (`MCS`, `LP`, `LM`, and `MF`) for the ADD-based executor `DMC`. The graded project-join trees here were produced by the planner `LG` with the tree decomposer `FlowCutter` from Experiment 1. `LG` is an anytime tool that may produce several trees of decreasing widths per benchmark. Nevertheless, across all four variable-ordering heuristics, there is little difference in execution time between using the first tree and using the last tree (planning time is excluded).

A.3 Experiment 3: Comparing Weighted Projected Model Counters

Figure 8 illustrates how six combinations of three `LG` tree-decomposition tools (`FlowCutter`, `htd`, and `Tamaki`) and two `DMC` variable-ordering heuristics (`MCS` and `LP`) compare in 1000 seconds. We exclude the planner `HTB` because it is slower than `LG` in Experiment 1. We also exclude the variable-ordering heuristics `LM` and `MF` because they are slower than `MCS` and `LP` in Experiment 2.

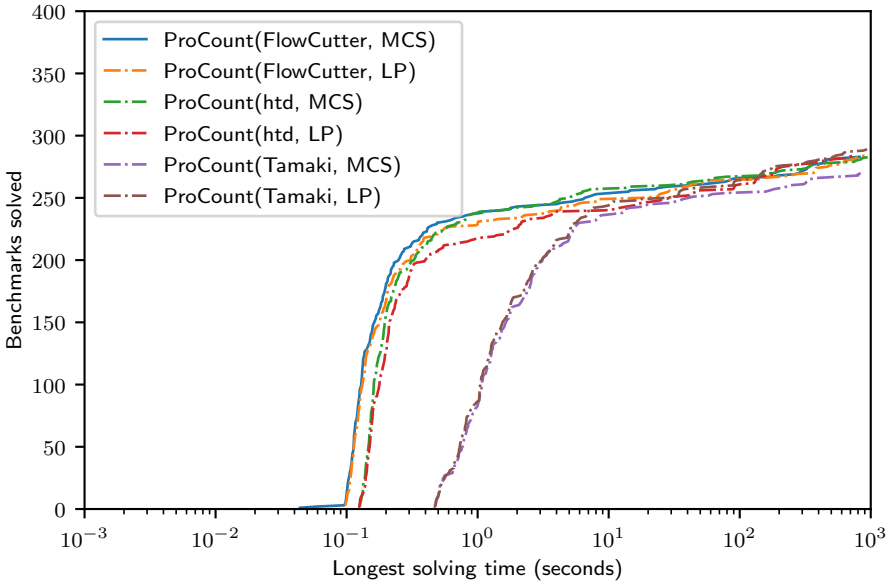

Fig. 8. This plot compares different combinations of an `LG` tree decomposer (`FlowCutter`, `htd`, or `Tamaki`) and a `DMC` variable-ordering heuristic (`MCS` or `LP`) for our framework `ProCount`. We choose `LG` with `FlowCutter` and `DMC` width `MCS` as the representative setting of `ProCount` to compete with existing projected model counters.

B Proofs

B.1 Proof of Theorem 2

When performing a product followed by a projection, it is often possible to perform the projection first. This is known as *early projection* [41], which forms the core of the proof of Theorem 2.

Theorem 8 (Early Projection). *Let X and Y be sets of variables. For all functions $f : 2^X \to \mathbb{R}$ and $g : 2^Y \to \mathbb{R}$, if $x \in X \setminus Y$, then $\Sigma_x(f \cdot g) = (\Sigma_x f) \cdot g$ and $\exists_x(f \cdot g) = (\exists_x f) \cdot g$.*

We ultimately prove Theorem 2 by structural induction. It is therefore helpful to have some additional notations for subtrees of project-join trees. Let (T, r, γ, π) be a project-join tree for φ, and let $n \in \mathcal{V}(T)$. Denote by $S(n) \subseteq \mathcal{V}(T)$ the set of all descendants of n in T (including n itself). Let $P(n) = \bigcup_{o \in S(n) \setminus \mathcal{L}(T)} \pi(o)$ be the set of all variables projected in $S(n)$, and let $\Phi(n) = \{\gamma(\ell) : \ell \in \mathcal{L}(T) \cap S(n)\}$ be the set of all clauses that appear as leaves in $S(n)$.

The key property of project-join trees is that variables projected in one branch of the tree cannot appear in sibling branches of the tree. Formally:

Lemma 1. *In a project-join tree (T, r, γ, π), let n be an internal node with children $o \neq q$. Then $P(o) \cap \texttt{Vars}(\Phi(q)) = \varnothing$.*

Proof. Let variable $x \in P(o)$. Notice that $x \in \pi(s)$ for some internal node s that is a descendant of o. Assume there is an arbitrary clause $c \in \varphi$ s.t. x appears in c. By the last property in Definition 4, the corresponding leaf $\gamma^{-1}(c)$ is a descendant of s and thus a descendant of o. So x appears in no descendant leaf of q (as q is a sibling of o in the tree T). Thus $x \notin \texttt{Vars}(\Phi(q))$. Since $x \in P(o)$ is arbitrary, we conclude that $P(o) \cap \texttt{Vars}(\Phi(q)) = \varnothing$.

Using Lemma 1, we can prove that projected-valuations have an equivalent, non-recursive definition:

Lemma 2. *Let (X, Y, φ, W) be an instance of weighted projected model counting, and let $\mathcal{T}$ be an (X, Y)-graded project-join tree for φ.*

Define, for every node n of T,

$$h_n^W = \left(\prod_{C \in \Phi(n)} [C]\right) \cdot \left(\prod_{x \in P(n) \cap X} W_x\right).$$

Then for every node n of T,

$$g_n^W = \sum_{P(n)\cap X} \exists\!\!\!\!\!\!\exists_{P(n)\cap Y} \, h_n^W. \tag{1}$$

Proof. We employ structural induction on $n \in \mathcal{V}(T)$. In the base case, n is a leaf. Then $P(n) = \varnothing$ and $\Phi(n) = \{\gamma(n)\}$. Thus $h_n^W = \left(\prod_{C \in \{\gamma(n)\}} [C]\right) \cdot \left(\prod_{x \in \varnothing} W_x\right) = [\gamma(n)]$, so the right-hand side of Equation (1) is $\sum_{\varnothing} \exists\!\!\!\!\!\!\exists_{\varnothing} \, h_n^W = h_n^W = [\gamma(n)]$, which is exactly g_n^W.

In the inductive case, n is an internal node of T and, for each $o \in \mathcal{C}_{T,r}(n)$, we have $g_o^W = \sum_{P(o)\cap X} \exists\!\!\!\!\!\!\exists_{P(o)\cap Y} \, h_o^W$.

Consider the product

$$\prod_{o \in \mathcal{C}_{T,r}(n)} g_o^W = \prod_{o \in \mathcal{C}_{T,r}(n)} \sum_{P(o)\cap X} \exists\!\!\!\!\!\!\exists_{P(o)\cap Y} \, h_o^W. \tag{2}$$

By Lemma 1, for distinct $o, q \in \mathcal{C}_{T,r}(n)$, we know $P(o) \cap \texttt{Vars}(\Phi(q)) = \varnothing$. Thus $P(o) \cap \texttt{Vars}(h_q^W) = \varnothing$ as well. We can therefore apply Theorem 8 to Equation (2) to get that

$$\prod_{o \in \mathcal{C}_{T,r}(n)} g_o^W = \sum_{A \cap X} \prod_{o \in \mathcal{C}_{T,r}(n)} \exists\!\!\!\!\!\!\exists_{P(o)\cap Y} \, h_o^W = \sum_{A\cap X} \exists\!\!\!\!\!\!\exists_{A \cap Y} \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \tag{3}$$

where $A = \bigcup_{o \in \mathcal{C}_{T,r}(n)} P(o)$.

Let $\mathcal{I}_X$ and $\mathcal{I}_Y$ be the grades of $\mathcal{T}$. By Definition 6, either $n \in \mathcal{I}_X$ or $n \in \mathcal{I}_Y$. We divide the inductive case further into these two cases.

Case: $n \in \mathcal{I}_Y$**.** Then for each $p \in S(n)$, by Definition 6, we have $p \in \mathcal{I}_Y$, so $\pi(p) \subseteq Y$. Thus $A \subseteq Y$. By Definition 7 and Equation (3), we have

$$g_n^W = \exists\!\!\!\!\!\!\sum_{\pi(n)} \prod_{o \in \mathcal{C}_{T,r}(n)} g_o^W = \exists\!\!\!\!\!\!\sum_{\pi(n)} \exists\!\!\!\!\!\!\sum_{A} \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W = \exists\!\!\!\!\!\!\sum_{P(n)} \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W.$$

We therefore conclude that

$$g_n^W = \exists\!\!\!\!\!\!\sum_{P(n)} \prod_{o \in \mathcal{C}_{T,r}(n)} \prod_{C \in \Phi(o)} [C] = \exists\!\!\!\!\!\!\sum_{P(n)} \prod_{C \in \Phi(n)} [C] = \exists\!\!\!\!\!\!\sum_{P(n)} h_n^W.$$

Case: $n \in \mathcal{I}_X$**.** Thus $\pi(n) \subseteq X$. By Definition 7 and Equation (3), we have

$$\begin{aligned} g_n^W &= \sum_{\pi(n)} \left(\prod_{o \in \mathcal{C}_{T,r}(n)} g_o^W \cdot \prod_{x \in \pi(n)} W_x \right) \\ &= \sum_{\pi(n)} \left(\left(\sum_{A \cap X} \exists\!\!\!\!\!\!\sum_{A \cap Y} \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \right) \cdot \prod_{x \in \pi(n)} W_x \right). \end{aligned}$$

Since $\pi(n) \cap A = \varnothing$, we can apply Theorem 8 (in the other direction, which undoes early projection) to get that

$$g_n^W = \sum_{\pi(n)} \left(\sum_{A \cap X} \exists\!\!\!\!\!\!\sum_{A \cap Y} \left(\prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \cdot \prod_{x \in \pi(n)} W_x \right) \right).$$

Finally, observe that $\pi(n) \cup A = P(n)$ and that $h_n^W = \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \cdot \prod_{x \in \pi(n)} W_x$. We therefore conclude that

$$g_n^W = \sum_{P(n) \cap X} \exists\!\!\!\!\!\!\sum_{P(n) \cap Y} \left(\prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \cdot \prod_{x \in \pi(n)} W_x \right) = \sum_{P(n) \cap X} \exists\!\!\!\!\!\!\sum_{P(n) \cap Y} h_n^W.$$

Moreover, this non-recursive definition is equivalent to the weighted projected model count at the root node.

Theorem 2 *Let (X, Y, φ, W) be an instance of weighted projected model counting, and let $\mathcal{T}$ be a project-join tree for φ with root r. If $\mathcal{T}$ is (X, Y)-graded, then $g_r^W(\varnothing) = \texttt{WPMC}(\varphi, W, Y)$.*

Proof. As r is the root of the project-join tree, $P(r) = X \cup Y$ and $\Phi(r) = \varphi$. By Lemma 2,

$$g_r^W = \sum_X \exists\!\!\!\!\!\!\sum_Y h_n^W = \sum_X \exists\!\!\!\!\!\!\sum_Y \left(\prod_{C \in \varphi} [C] \right) \cdot \left(\prod_{x \in X} W_x \right) = \sum_X \exists\!\!\!\!\!\!\sum_Y [\varphi] \cdot W.$$

Thus $g_r^W(\varnothing)$ is exactly the W-weighted Y-projected model count of φ.

B.2 Proof of Theorem 3

Theorem 3 *Let φ be a CNF formula, and let $F \subseteq \texttt{Vars}(\varphi)$. If $\mathcal{T}$ is a project-join tree for $\varphi \cup \{\texttt{virtual}(F)\}$, then Algorithm 1 returns an F-free project-join tree for φ of width at most $\texttt{width}(\mathcal{T})$.*

Proof. Let $C_F = \texttt{virtual}(F)$. Let $\mathcal{T} = (T, r, \gamma, \pi)$ be the input project-join tree for $\varphi \cup \{C_F\}$, and let $\mathcal{T}' = (T, s, \gamma', \pi')$ be the output of Algorithm 1. Moreover, let $\texttt{Vars}_{\mathcal{T}}(n)$ and $\texttt{Vars}_{\mathcal{T}'}(n)$ denote the sets of variables at a node n of $\mathcal{T}$ and $\mathcal{T}'$ respectively.

First, we prove that $\mathcal{T}'$ is a project-join tree. Since γ is a bijection onto $\varphi \cup \{C_F\}$ and we have removed both C_F and the leaf corresponding to C_F, γ' is indeed a bijection onto φ. Moreover, since every variable appears in exactly one set in the image of π', the first condition of Definition 4 is satisfied. Finally, each variable y is projected out at the lowest common ancestor of all leaves corresponding to clauses that contain y; thus the second condition of Definition 4 is satisfied. It follows that $\mathcal{T}'$ is a project-join tree.

Second, we prove that $\mathcal{T}'$ is F-free. Since s has degree 1, s is never the lowest common ancestor of a set of leaves of $\mathcal{T}'$. Thus $\pi'(s) \setminus F = \varnothing$. By line 7, it follows that $\pi'(s) = F$, so $\mathcal{T}'$ is F-free.

Finally, we prove that the width of $\mathcal{T}'$ is at most $\texttt{width}(\mathcal{T})$. To do this, if S is a project-join tree and n is a node of S, define $\texttt{rel}_S(n) \equiv \texttt{Vars}_S(n)$ for leaf nodes and $\texttt{rel}_S(n) \equiv \texttt{Vars}_S(n) \cup \pi(n)$ for internal nodes. Notice the size of n in S is exactly $|\texttt{rel}_S(n)|$, so the width of S is exactly the maximum size of $\texttt{rel}_S(n)$ across all nodes n.

Consider an arbitrary node $n \in \mathcal{V}(T) \setminus \{s\}$. Define:

$$A(n) = \{y : \exists \ell \in \mathcal{L}(T) \text{ s.t. } \ell \text{ is a descendant of } n \text{ in the rooted tree } (T, s) \text{ and } y \in \texttt{rel}_{\mathcal{T}'}(\gamma'(\ell))\}$$
$$B(n) = \{x \in \texttt{Vars}(\varphi) : \exists \ell, \ell' \in \mathcal{L}(T) \text{ s.t. } n \text{ is between } \ell, \ell' \text{ in } (T, r) \text{ and } x \in \texttt{rel}_{\mathcal{T}}(\gamma(\ell)) \cap \texttt{rel}_{\mathcal{T}}(\gamma(\ell'))\}$$
$$B'(n) = \{x \in \texttt{Vars}(\varphi) : \exists \ell, \ell' \in \mathcal{L}(T) \text{ s.t. } n \text{ is between } \ell, \ell' \text{ in } (T, s) \text{ and } x \in \texttt{rel}_{\mathcal{T}'}(\gamma'(\ell)) \cap \texttt{rel}_{\mathcal{T}'}(\gamma'(\ell'))\}$$

Note that a node n is *between* $\ell, \ell' \in \mathcal{V}(T)$ if n is on the unique shortest path between ℓ and ℓ'. There are several key relationships among $A(n)$, $B(n)$, $B'(n)$, $\texttt{rel}_{\mathcal{T}}(n)$, and $\texttt{rel}_{\mathcal{T}'}(n)$:

1. By the construction in Algorithm 1, we know $\texttt{rel}_{\mathcal{T}'}(n) \setminus F = B'(n) \setminus F$ and $\texttt{rel}_{\mathcal{T}'}(n) \cap F = A(n) \cap F$.
2. Since γ and γ' agree on all nodes of T except for s, we know $B(n) \setminus F = B'(n) \setminus F$.
3. Since $\texttt{Vars}(C_F) = F$, we know $B(n) \cap F = A(n) \cap F$.
4. By Property 2 of Definition 4, $B(n) \subseteq \texttt{rel}_{\mathcal{T}}(n)$.

Putting these relationships together, we observe that:

$$\begin{aligned}\texttt{rel}_{\mathcal{T}'}(n) &= (\texttt{rel}_{\mathcal{T}'}(n) \setminus F) \cup (\texttt{rel}_{\mathcal{T}'}(n) \cap F) \\ &= (B'(n) \setminus F) \cup (A(n) \cap F) \\ &= (B(n) \setminus F) \cup (B(n) \cap F) \\ &= B(n) \\ &\subseteq \texttt{rel}_{\mathcal{T}}(n)\end{aligned}$$

Finally, observe that $\texttt{rel}_{\mathcal{T}'}(s) = \texttt{rel}_{\mathcal{T}}(s) = F$. Hence the width of $\mathcal{T}'$ is indeed no larger than the width of $\mathcal{T}$, as desired.

B.3 Proof of Theorem 4

Theorem 4 *Let φ be a CNF formula. Let $F \subseteq \texttt{Vars}(\varphi)$, and let w be a positive integer. If there is an F-free project-join tree for φ of width w, then there is a project-join tree for $\varphi \cup \{\texttt{virtual}(F)\}$ of width w.*

Proof. Let $\mathcal{T}$ be an F-free project-join tree for φ of width at most w. Produce $\mathcal{T}'$ by attaching to the root of $\mathcal{T}$ a new leaf node corresponding to C_F. Then $\mathcal{T}'$ is a project-join tree for $\varphi \cup \{C_F\}$, and its width is identical to $\mathcal{T}$.

B.4 Proof of Theorem 5

Theorem 5 *Let φ be a CNF formula. Let $\{X, Y\}$ be a partition of $\texttt{Vars}(\varphi)$, and let w be a positive integer. Assume each call to $\texttt{BuildComponent}(\alpha, F)$ returns an F-free project-join tree for α of width at most w. Then Algorithm 2 returns an (X, Y)-graded project-join tree for φ of width at most w.*

Proof. Let $\mathcal{T}$ be the project-join tree produced on line 6, and let $\mathcal{T}' = (T, r, \gamma, \pi)$ be the project-join tree returned by Algorithm 2. By Definition 9, every $y \in Y$ is a variable of exactly one $N_y \in \texttt{GroupBy}(\varphi, Y)$. It follows that every $y \in Y$ is projected out at exactly one node of $\mathcal{T}'$, namely the node at which y is projected out in $\mathcal{T}_{N_y}$. Similarly, after the loop on line 2 completes, no variable from X is projected out across all of $\{\mathcal{T}_N : N \in \texttt{GroupBy}(\varphi, Y)\}$, since all X projections are removed on line 4. Thus every $x \in X$ is also projected out at exactly one node of $\mathcal{T}'$, namely the node at which x is projected out in $\mathcal{T}$. Thus $\mathcal{T}'$ satisfies the first property of Definition 4.

We prove the second property of Definition 4 by contrapositive. That is, assume that there is some $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$, variable $z \in \pi(n)$, and $c \in \varphi$ s.t. $z \in \texttt{Vars}(c)$ but $\gamma^{-1}(c)$ is not a descendant of n in $\mathcal{T}'$. Let N be the block of $\texttt{GroupBy}(\varphi, Y)$ that contains c. We split into two cases:

- *Case: $z \in Y$.* Let n' be the node in $\mathcal{T}_N$ that corresponds to n, where z is projected. Then $\gamma^{-1}(c)$ is not a descendant of n' in $\mathcal{T}_N$. It follows that $\mathcal{T}_N$ is not a project-join tree.
- *Case: $z \in X$.* Let n' be the node in $\mathcal{T}$ that corresponds to n, where z is projected. Since $\gamma^{-1}(c)$ is not a descendant of n in $\mathcal{T}$, the leaf corresponding

to C_N is not a descendant of n' in $\mathcal{T}$. But $z \in \texttt{Vars}(N) \cap X$, so $z \in \texttt{Vars}(C_N)$.

It follows that $\mathcal{T}$ is not a project-join tree.

We conclude that $\mathcal{T}'$ satisfies the second property of Definition 4 provided that `BuildComponent` always returns project-join trees.

Finally, we prove that the width of $\mathcal{T}'$ is at most w. To see this, we observe that the set of variables at each node of $\mathcal{T}'$ is exactly the set of variables appearing at the node of the corresponding component project-join tree. The width of $\mathcal{T}'$ is thus the maximum size that appears across all component project-join trees returned by `BuildComponent`.

B.5 Proof of Theorem 6

Theorem 6 *Let φ be a CNF formula, and let $\{X, Y\}$ be a partition of $\texttt{Vars}(\varphi)$. Let w be a positive integer, and let $\psi = \varphi \cup \{\texttt{virtual}(\texttt{Vars}(N) \cap X) : N \in \texttt{GroupBy}(\varphi, Y)\}$. If there is a project-join tree $\mathcal{T}'$ for ψ of width w, then there is an (X, Y)-graded project-join tree for φ of width at most w.*

Proof. We first show that, for each call to $\texttt{BuildComponent}(\alpha, F)$ in Algorithm 2, there is an F-free project-join tree for α of width at most w. There are two cases to consider: the calls on line 3 and the call on line 6.

- *Case: line 3.* Consider some $N \in \texttt{GroupBy}(\varphi, Y)$. Our goal is to find a $(\texttt{Vars}(N) \cap X)$-free project-join tree for N. Observe that $N \cup \{C_N\}$ is a subset of ψ. Thus let S_N be the smallest subtree of $\mathcal{T}'$ containing all leaves labeled by some element of $N \cup \{C_N\}$. S_N is a project-join tree for $N \cup \{C_N\}$ whose width is no more than w. By Theorem 3, there is a $(\texttt{Vars}(N) \cap X)$-free project-join tree for N of width no more than w.
- *Case: line 6.* Similarly, let $N' = \{C_N : N \in \texttt{GroupBy}(\varphi, Y)\}$ and observe that N' is a subset of ψ. Let S be the smallest subtree of $\mathcal{T}'$ containing all leaves labeled by some element of N'. Then S is a project-join tree for N' whose width is no more than w.

It then follows from Theorem 5 that there is an (X, Y)-graded project-join tree for φ of width at most w.

B.6 Proof of Theorem 7

Theorem 7 *Let φ be a CNF formula, and let $\{X, Y\}$ be a partition of $\texttt{Vars}(\varphi)$. Let w be a positive integer, and let $\psi = \varphi \cup \{\texttt{virtual}(\texttt{Vars}(N) \cap X) : N \in \texttt{GroupBy}(\varphi, Y)\}$. If there is an (X, Y)-graded project-join tree for φ of width w, then there is a project-join tree for ψ of width w.*

Proof. Let $\mathcal{T}$ be an (X, Y)-graded project-join tree for φ of width w, and let $\mathcal{I}_X, \mathcal{I}_Y$ be the grades of $\mathcal{T}$. We first aim to show that, for every $N \in \texttt{GroupBy}(\varphi, Y)$, there is some node n_N of $\mathcal{T}$ with $N \cap X \subseteq \texttt{Vars}(n_N)$.

Consider an arbitrary $N \in \texttt{GroupBy}(\varphi, Y)$. If $|N| = 1$, define n_N to be the node of $\mathcal{T}$ corresponding to the only element of N; thus $\texttt{Vars}(n_N) = \texttt{Vars}(N)$, so indeed $N \cap X \subseteq \texttt{Vars}(n_N)$. Otherwise, define $n_N \in \mathcal{V}(T)$ to be the lowest common ancestor of N in $\mathcal{T}$. Then there exist distinct clauses $A, B \in N$ s.t. n_N

is also the lowest common ancestor of the two leaves labeled by A and B. By Definition 9, since $A, B \in N$, there must be a sequence $C_1, C_2, \cdots, C_k \in N$ s.t. $C_1 = A$, $C_k = B$, and for each $1 \leq i \leq k$, we have $\texttt{Vars}(C_i) \cap \texttt{Vars}(C_{i+1}) \cap Y \neq \varnothing$. Since n_N is the lowest common ancestor of A and B, and because $\mathcal{T}$ is a tree, there must be some $1 \leq i \leq k$ s.t. n_N is also the lowest common ancestor of C_i and C_{i+1}. Thus $\texttt{Vars}(C_i) \cap \texttt{Vars}(C_{i+1}) \subseteq \texttt{Vars}(n_N)$. Since $\texttt{Vars}(C_i) \cap \texttt{Vars}(C_{i+1}) \cap Y \neq \varnothing$, it follows that $Y \cap \texttt{Vars}(n_N) \neq \varnothing$ as well. Thus $n_N \in \mathcal{I}_Y$. By Definition 6, this means that for all descendants o of n_N, $\pi(o) \cap X = \varnothing$. It follows that $\texttt{Vars}(n_N)$ must still contain all variables in N from X; that is, $\texttt{Vars}(N) \cap X \subseteq \texttt{Vars}(n_N)$.

Construct $\mathcal{T}'$ from $\mathcal{T}$ by, for each $N \in \texttt{GroupBy}(\varphi, Y)$, attaching a new leaf labeled by $\texttt{virtual}(\texttt{Vars}(N) \cap X)$ as a child of n_N. Since $N \cap X \subseteq \texttt{Vars}(n_N)$ in the initial tree, the width of $\mathcal{T}'$ is equal to the width of $\mathcal{T}$. Moreover, $\mathcal{T}'$ is now a project-join tree for ψ.