Towards Projected and Incremental Pseudo-Boolean Model Counting

Suwei Yang[1,2,4]**, Kuldeep S. Meel**[3,5]

[1]Grabtaxi Holdings
[2]Grab-NUS AI Lab
[3]Georgia Institute of Technology
[4]National University of Singapore
[5]University of Toronto

Abstract

Model counting is a fundamental task that involves determining the number of satisfying assignments to a logical formula, typically in conjunctive normal form (CNF). While CNF model counting has received extensive attention over recent decades, interest in Pseudo-Boolean (PB) model counting is just emerging partly due to the greater flexibility of PB formulas. As such, we observed feature gaps in existing PB counters such as a lack of support for projected and incremental settings, which could hinder adoption.

In this work, our main contribution is the introduction of the PB model counter PBCount2, the first exact PB model counter with support for projected and incremental model counting. Our counter, PBCount2, uses our Least Occurrence Weighted Min Degree (LOW-MD) computation ordering heuristic to support projected model counting and a cache mechanism to enable incremental model counting. In our evaluations, PBCount2 completed at least $1.40\times$ the number of benchmarks of competing methods for projected model counting and at least $1.18\times$ of competing methods in incremental model counting.

Paper with appendix — https://arxiv.org/abs/2412.14485

1 Introduction

Pseudo-Boolean (PB) model counting involves determining the number of satisfying assignments of a given pseudo-boolean formula, which differs from typical model counting tasks where the input is a Boolean formula in conjunctive normal form (CNF). In recent years, there has been increasing interest from the community in PB model counting and PB formulas in general, in part due to their succinctness and flexibility over CNF formulas (Sinz 2005). In particular, an arbitrary CNF clause can always be converted to a single PB constraint, but the converse is not true (Le Berre et al. 2018). The emerging interests take the form of new PB model counting tools such as ApproxMCPB and PBCount (Yang and Meel 2021, 2024) as well as the emergence of applications that include knapsack problems, neural network verification, and budgeted sensor placement (Pisinger 2005; Yang and Meel 2021; Latour, Sen, and Meel 2023).

In contrast, CNF model counting and solving (SAT solving) are well-established fields, benefiting from many decades of research advancements that have led to numerous feature-rich counters and solvers. The availability of tools for CNF model counting and solving has led to an ever-increasing number of applications across a wide range of domains such as software design, explainable machine learning, planning, and probabilistic reasoning (Bacchus, Dalmao, and Pitassi 2003; Narodytska et al. 2019; Jackson 2019; Fan, Miller, and Mitra 2020). In turn, the wide range of applications drives demand for better tools, from both performance and feature perspectives. Subsequently, better tools would lead to more adoption and applications. Eventually, a positive self-reinforcing loop is formed within the CNF model counting and solving community. On the other hand, we have yet to observe such a positive cycle in the PB model counting community, in part due to a lack of features such as exact projected counting and incremental setting support in existing PB model counting tools.

In this work, our main contribution is the PB model counter, PBCount2, which supports projected and incremental model counting features. We focus on these two aforementioned features due to their importance to the CNF model counting and SAT solving community. Projected model counting involves determining the number of partial assignments over a given set of variables, known as the projection set, that can be extended to a satisfying assignment over all variables of the formula. Projected model counting has been applied in areas such as planning, verification, and reliability estimation (Aziz et al. 2015; Klebanov, Manthey, and Muise 2013; Dueñas-Osorio et al. 2017). Similarly, incremental SAT solving has led to applications such as combinatorial testing, circuit design, and solving string constraints (Yamada et al. 2015; Yu et al. 2017; Lotz et al. 2023). In this work, we focus on the incremental setting whereby a given PB formula undergoes incremental modifications, more specifically addition and removal of PB constraints, while having its model count computed along each modification step. In order to support projected PB model counting, we introduce a new computation ordering heuristic that is compatible with the ordering requirements for projected model counting. Additionally, we introduce a new caching mechanism that is at the core of incremental counting. PBCount2 is to the best of our knowledge both the first

exact projected PB counter as well as the first incremental model counter.

We performed extensive evaluations on benchmark instances inspired by various application settings, such as sensor placement, multi-dimension knapsack, and combinatorial auction benchmarks (Gens and Levner 1980; Blumrosen and Nisan 2007; Latour, Sen, and Meel 2023). Our evaluations highlighted the efficacy of PBCount2 at projected PB model counting and incremental model counting compared to the baseline techniques. In particular, PBCount2 is able to successfully count 1957 projected model counting instances while the state-of-the-art CNF-based projected model counter could only count 1398 instances. Incremental benchmarks involve multiple modifications of the initially provided PB formula, with model counts computed after each modification step. Our experiments showed that PBCount2 is able to complete 1618 instances for incremental benchmarks involving 5 counts with 1 count for the initial PB formula and 4 counts for modification steps. In comparison, the state-of-the-art PB model counter, PBCount, completed only 1371 instances, demonstrating a significant performance advantage of PBCount2 at incremental counting. Moreover, PBCount2 also demonstrates superior performance at incremental settings compared to the state-of-the-art CNF model counters D4 and GPMC, both of which completed less than 1000 incremental benchmarks. Given that it is the early days of PB model counting, we hope the new capabilities introduced in this work will attract more interest and applications of PB model counting, leading to a positive self-reinforcing cycle within the PB model counting community.

2 Notations and Preliminaries

Pseudo-Boolean Formula A PB formula F consists of one or more PB constraints, each of which is either equality or inequality. A PB constraint takes the form $\sum_{i=1}^{n} a_i x_i \square k$ where $x_1, ..., x_n$ are Boolean literals, $a_1, ..., a_n$, and k are integers, and $\square$ is one of $\{\geq, =, \leq\}$. We refer to $a_1, ..., a_n$ as term coefficients in the PB constraint, where each term is of the form $a_i x_i$. F is satisfiable if there exists an assignment τ of all variables of F such that all its PB constraints evaluate to *true*. PB model counting refers to the computation of the number of satisfying assignments of F.

Projected Model Counting Let F be a formula and $\mathsf{Var}(F)$ be the set of variables of F. Let X and Y be disjoint subsets of $\mathsf{Var}(F)$ such that $X \cap Y = \emptyset$ and $X \cup Y = \mathsf{Var}(F)$. (F, X, Y) is a projected model counting instance. The projected model count is $\sum_{\beta \in 2^X} (\max_{\alpha \in 2^Y} [F](\alpha \cup \beta))$ (Dudek, Phan, and Vardi 2021). Where $[F](\alpha \cup \beta)$ is the evaluation of F with the variable assignment $\alpha \cup \beta$, and returns 1 if F is satisfied and 0 otherwise. In other words, the projected model count of F on X is the number of assignments of all variables in X such that there exists an assignment of variables in Y that makes F evaluate to *true* (Aziz et al. 2015). Non-projected model count is a special case of projected model counting where all variables are in the projection set, i.e. $X = \mathsf{Var}(F)$, and $Y = \emptyset$. With reference to Figure 2, sup-
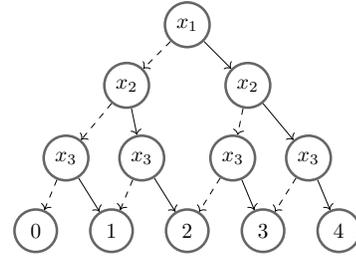

Figure 1: An ADD representing $2x_1 + x_2 + x_3$ with ordering x_1, x_2, x_3
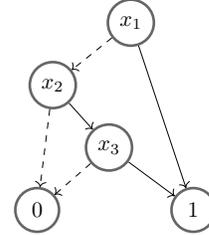

Figure 2: An ADD representing $2x_1 + x_2 + x_3 \geq 2$ with ordering x_1, x_2, x_3

pose the PB formula has only the single PB constraint $2x_1 + x_2 + x_3 \geq 2$, then the 5 satisfying assignments are $(x_1, x_2, x_3), (x_1, x_2, \bar{x}_3), (x_1, \bar{x}_2, x_3), (x_1, \bar{x}_2, \bar{x}_3)$, and $(\bar{x}_1, x_2, x_3)$. If the projection set is x_1 then the corresponding projected model count is 2, as both partial assignments involving x_1 can be extended to a satisfying assignment.

Projections Let $f : 2^X \to \mathbb{R}$ be a function defined over a set of boolean variables X. The Σ-projection of f with respect to a variable $x \in X$ for all σ is given by $f(\sigma \wedge x) + f(\sigma \wedge \bar{x})$ where σ refers to assignments of all variables excluding x. Similarly, the $\exists$-projection with respect to x is given by $\max(f(\sigma \wedge x), f(\sigma \wedge \bar{x}))$. In the case that f maps to 1 or 0, i.e. $f : 2^X \to \{0, 1\}$, then $\exists$-projection can be written as $f(\sigma \wedge x) \vee f(\sigma \wedge \bar{x})$. In this work, we use the two types of projections to compute the projected model count in Algorithm 1 which we detail in Section 4.2.

Algebraic Decision Diagram An algebraic decision diagram (ADD) (Bahar et al. 1993) is a directed acyclic graph (DAG) representation of a function $f : 2^{\mathsf{Var}(F)} \to S$ where $\mathsf{Var}(F)$ is the set of Boolean variables that f is defined over, and S is known as the carrier set. We denote the function represented by an ADD ψ as $\mathsf{Func}(\psi)$. The internal nodes of ADD represent decisions on variables $x \in \mathsf{Var}(F)$ and the leaf nodes represent $s \in S$. The order of variables represented by decision nodes, from root to leaf of ADD, is known as the variable ordering. In this work, S is a set of integers. As example, an ADD representing $2x_1 + x_2 + x_3$ is shown in Figure 1, and $2x_1 + x_2 + x_3 \geq 2$ in Figure 2. The dashed arrows of internal nodes represent when the corresponding variables are set to *false*, and solid arrows represent when they are set to *true*.

In this work, we make use of the Apply operation on

ADDs (Bryant 1986; Bahar et al. 1993). The Apply operation takes as input a binary operator $\bowtie$, two ADDs ψ_1, ψ_2, and outputs an ADD ψ_3 such that the $\mathsf{Func}(\psi_3) = \mathsf{Func}(\psi_1) \bowtie \mathsf{Func}(\psi_2)$. It is worth noting that the input ADDs ψ_1, ψ_2 are required to have the same ADD variable orderings i.e. the ordering of variables being represented by internal nodes of ADDs from root to leaf nodes. In this work, we use the term *merge* to refer to the usage of Apply with $\times$ operator on two ADDs. Referring to the explanation of Apply, we say ψ_3 represents PB constraints $\{c_1, c_2\}$ if ψ_1 represents $\{c_1\}$ and ψ_2 represents $\{c_2\}$.

Project-Join Tree Let F be a formula. A project-join tree (Dudek, Phan, and Vardi 2020b) of F is a tuple $\mathcal{T} = (T, r, \gamma, \pi)$ where T is a tree with nodes $\mathcal{V}(T)$ and rooted at node r. γ is a bijection from leaves $\mathcal{L}(T)$ of T to constraints of F and π is a labelling function $\mathcal{V}(T) \backslash \mathcal{L}(T) \to 2^{\mathsf{Var}(F)}$ on internal nodes $\mathcal{V}(T) \setminus \mathcal{L}(T)$ of T. Additionally, $\mathcal{T}$ satisfies the following:

1. The set $\{\pi(n) : n \in \mathcal{V}(T) \setminus \mathcal{L}(T)\}$ partitions $\mathsf{Var}(F)$

2. For an arbitrary internal node n. Let x be a variable in πn, and c be a PB constraint of F. If $x \in \mathsf{Var}(c)$, then leaf node $\gamma^{-1}(c)$ is a descendant of n.

X,Y-Graded Project-Join Tree Let F be a formula with project join tree $\mathcal{T}$, and variable sets X, Y being partitions of $\mathsf{Var}(F)$. The project join tree $\mathcal{T}$ is an X, Y graded project-join tree (Dudek, Phan, and Vardi 2021) if there exist grades $\mathcal{I}_X$ and $\mathcal{I}_Y$ such that:

1. The set $\{\mathcal{I}_X, \mathcal{I}_Y\}$ is a partition of internal nodes of $\mathcal{T}$

2. If node $n_X \in \mathcal{I}_X$, $\pi(n_X) \subseteq X$

3. If node $n_Y \in \mathcal{I}_Y$, $\pi(n_Y) \subseteq Y$

4. If $n_X \in \mathcal{I}_X$ and $n_Y \in \mathcal{I}_Y$, n_X is not a descendant of n_Y in tree T rooted at r.

Projected Model Counting with Project Join Trees Over a series of works (Dudek, Phan, and Vardi 2020a,b, 2021) Dudek, Phan, and Vardi established a CNF model counting framework with project join trees and extended the framework to projected CNF model counting by employing a specific type of project join tree, namely an X, Y-graded project join tree. Dudek, Phan, and Vardi showed that performing $\exists$-projection at $\mathcal{I}_Y$ nodes and Σ-projection at $\mathcal{I}_X$ according to $\mathcal{T}$ produces the correct projected CNF model count (Dudek, Phan, and Vardi 2021). We show that our projected PB model counting approach shares similarities with the project join tree framework in Section 4.2, providing the intuition for algorithm correctness.

3 Related Work

Search-Based Model Counters We can classify the numerous existing model counters into two main categories based on their underlying techniques – search-based model counters and decision diagram-based model counters. Search-based model counters iteratively assign values to variables, implicitly exploring a search tree of variable assignments while keeping count throughout the process. The counters typically employ component caching, so that

counts for each branch of the search tree are only computed once. Notable search-based model counters include GPMC, Ganak, and Sharpsat-TD (Ryosuke Suzuki and Sakai 2017; Sharma et al. 2019; Korhonen and Järvisalo 2021).

Decision Diagram-Based Model Counters The use of different types of decision diagrams, which are directed acyclic graph (DAG) representations of the assignment space of a given formula, is common among model counters, for both PB and CNF formulas. Recent decision diagram-based model counters include D4, ExactMC, ADDMC, and DPMC for CNF model counting and PBCount for PB model counting (Lagniez and Marquis 2017; Dudek, Phan, and Vardi 2020a,b; Lai, Meel, and Yap 2021; Yang and Meel 2024). D4 and ExactMC build their respective decision diagrams in a top-down manner, forming a single diagram representing the assignment space. In contrast, ADDMC, DPMC, and PBCount build decision diagrams in a bottom-up manner, starting from individual decision diagrams of clauses or constraints and subsequently combining the decision diagrams to represent the assignment space. Our counter, which we term PBCount2, follows a similar methodology and combines individual decision diagrams in a particular order to support projected model counting while also caching the intermediate diagrams during the process.

Pseudo-Boolean Model Counting There has been recent interest in PB model counting. Notable tools developed for PB model counting include the approximate PB model counter ApproxMCPB (Yang and Meel 2021) and the recently introduced exact PB model counter PBCount (Yang and Meel 2024). ApproxMCPB uses hash functions to evenly split the space of satisfying assignments and enumerates a partition of satisfying assignments to obtain an estimated count. On the other hand, PBCount follows the methodology introduced in ADDMC (Dudek, Phan, and Vardi 2020a) to exactly count the number of satisfying assignments of the input PB formula. However, PBCount does not support projected model counting or incremental counting features. In this work, we introduce the projected model counting and incremental counting features to the community via our counter PBCount2.

Incremental SAT Solving There have been numerous works in existing literature regarding incremental settings, more specifically for incremental satisfiability (SAT) solving (Nadel and Ryvchin 2012; Nadel, Ryvchin, and Strichman 2014; Fazekas, Biere, and Scholl 2019; Nadel 2022). Incremental SAT solving involves finding satisfying assignments to a user-provided Boolean formula, typically in CNF, under certain assumptions that users can incrementally specify. Specifically, users can add and remove CNF clauses to the initially specified CNF formula. Incremental SAT solving is also useful for cases where users have to solve a large number of similar CNF formulas and has led to a wide range of applications such as combinatorial testing, circuit design, and solving string constraints (Yamada et al. 2015; Yu et al. 2017; Lotz et al. 2023). In this work, we introduce the concept of incrementality to model counting tasks, by making use of a caching mechanism which we detail in Section 4.3.

4 Approach

In this section, we detail our Least Occurrence Weighted Min Degree (LOW-MD) computation ordering heuristic as well as the implementations of PBCount2 to support a) projected model counting and b) incremental model counting.

4.1 Least Occurrence Weighted Min Degree

The general methodology of performing PB model counting tasks with ADDs involves representing each constraint with its individual ADD and subsequently merging the ADDs and projecting away variables to produce the final model count. As implemented in existing work (Yang and Meel 2024), the ordering of merging ADDs and projecting away variables is determined by heuristics. However, the existing heuristics did not support projected model counting. To this end, we introduce a new ordering heuristic in PBCount2, which we term the *Least Occurrence Weighted Min Degree* heuristic (LOW-MD in short).

Algorithm 1: PBCount2 model counting with LOW-MD heuristic

Input : PB formula F, Projection set X,
           Non-projection set Y
Output: Model count

1   $F \leftarrow \mathsf{preprocess}(F)$
2   **for** all constraints c of F **do**
3   Compile ADD for c
4   **while** $Y \neq \emptyset$ **do**
5      $\psi \leftarrow \mathsf{ADD}(1)$ ▷ `ADD(1) returns ADD`
       `representing 1`
6      $y \leftarrow \mathsf{popNextVar}(Y)$
7   **foreach** ADD φ containing variable y **do**
        $\psi \leftarrow \psi \times \varphi$
8      $\psi \leftarrow \psi[y \mapsto 1] \vee \psi[y \mapsto 0]$ ▷ $\exists$−projection
9   **while** $X \neq \emptyset$ **do**
10     $\psi \leftarrow \mathsf{ADD}(1)$
11     $x \leftarrow \mathsf{popNextVar}(X)$
12   **foreach** ADD φ containing variable x **do**
        $\psi \leftarrow \psi \times \varphi$
13     $\psi \leftarrow \psi[x \mapsto 1] + \psi[x \mapsto 0]$ ▷ Σ−projection
14   $\psi \leftarrow \mathsf{ADD}(1)$
15   **foreach** remaining intermediate ADD φ **do**
     $\psi \leftarrow \psi \times \varphi$
16   **return** ψ.value

Our *Least Occurrence Weighted Min Degree* heuristic is as follows. Let G be an undirected bipartite graph, where the vertices either represent variables in PB formula F or a single PB constraint of F. A variable vertex v_x is connected to a constraint vertex v_c if the variable appears in that PB constraint. The LOW-MD heuristic entails picking the variable that has the corresponding variable vertex in G with the minimum degree. The heuristic is equivalent to a weighted version of the min-degree heuristic on Gaifman graphs, where weights correspond to the number of PB constraints in which two variables appear together.

The intuition behind LOW-MD stems from the observation that the algorithmic complexity of merging two ADDs of size m, n is on the order of $\mathcal{O}(mn)$. As such, we would like to reduce the size of operand ADDs as much as possible, especially when the overall model counting algorithm involves many such ADD merging operations. In the computation process, the size of an ADD reduces when a variable is projected away. To ensure correctness, a variable can only be projected away when all ADDs involving it have been merged (Dudek, Phan, and Vardi 2020a). Hence, we designed our LOW-MD heuristic to pick the least frequently occurring variable to project away, as it involves merging the fewest number of ADDs before projecting away the variable.

4.2 Projected Model Counting

Recall that in projected model counting, there are two non-overlapping sets of variables X, Y where X is the projection set and Y is the non-projection set. The key idea to support projected model counting is the different way variables in X and Y are projected away. For all variables $x \in X$, we project x away from an ADD ψ using $\psi \leftarrow \psi[x \mapsto 1] + \psi[x \mapsto 0]$, also referred to as Σ-projection (Dudek, Phan, and Vardi 2021). In contrast, for all variables $y \in Y$, we project away y from ψ using $\psi \leftarrow \psi[y \mapsto 1] \vee \psi[y \mapsto 0]$, also referred to as $\exists$-projection (Dudek, Phan, and Vardi 2021). In addition, all variables in Y must be projected away before any variable in X because the different projection operations are not commutative across variables in X and Y (Dudek, Phan, and Vardi 2021). To this end, we introduce the LOW-MD ordering which is compatible with the projected model counting ordering requirements. The overall flow of PBCount2 is shown in Figure 3, and the pseudocode is shown in Algorithm 1.

In Algorithm 1, we employ the same preprocessing (line 1) and individual constraint compilation techniques (line 3) as the original PBCount. Next, we process each variable y in non-projection set Y by merging all ADDs containing y and projecting away y from the merged ADD (lines 4-8). In lines 9-13, we do the same for variables in projection set X. In each iteration of the merge and project process, we select a variable using our LOW-MD heuristic, indicated by popNextVar($\cdot$) on lines 6 and 11, and remove it from X or Y respectively. As discussed previously, the LOW-MD ordering heuristic entails picking the variable that has the least occurrence in the ADDs at that moment of the computation.

Algorithm Correctness   The algorithm correctness of projected model counting of PBCount2 follows prior work on projected CNF model counting with ADDs (Dudek, Phan, and Vardi 2021). Dudek, Phan, and Vardi showed that for projected CNF model counting correctness, the computations should be performed according to an X, Y-graded project join tree $\mathcal{T}$. In particular, performing $\exists$-projection at $\mathcal{I}_Y$ nodes and Σ-projection at $\mathcal{I}_X$ nodes of $\mathcal{T}$ produces the correct projected model count. PBCount2's algorithm correctness for projected PB model counting comes from the fact that the computation in Algorithm 1 with LOW-MD heuristic implicitly follows an X, Y-graded project join tree, and therefore produces the correct count.
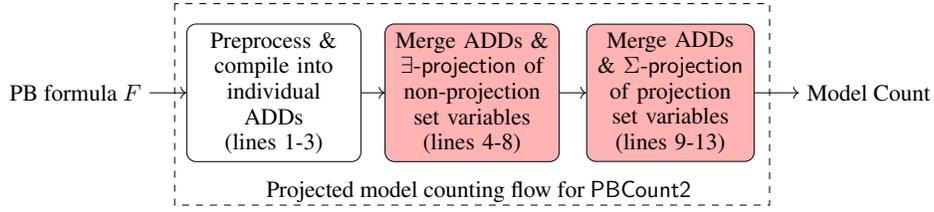
Figure 3: Overall flow of our projected model counter PBCount2. Shaded boxes indicate our contributions and white box indicates techniques adapted from existing works. Line numbers correspond to lines in Algorithm 1.

Theorem 1. *Let F be a formula defined over $X \cup Y$ such that X is the projection set, and Y is the set of variables not in projection set, then given an instance (F, X, Y), Algorithm 1 returns c such that $c = \sum_{\beta \in 2^X} (\max_{\alpha \in 2^Y} [F](\alpha \cup \beta))$*

Proof. The proof is deferred to the Appendix. □

An X, Y-graded project join tree $\mathcal{T}$ has two sets of disjoint variables, in Algorithm 1 this corresponds to variables in non-projection set Y and projection set X. The initial individual ADDs produced at line 3 of Algorithm 1 each corresponds to a leaf node in $\mathcal{T}$. Each of the intermediate ADDs during the merge and project iterations of variables in non-projection set Y (lines 4 to 8) corresponds to an internal node of $\mathcal{T}$, in grade $\mathcal{I}_Y$. Similarly, each intermediate ADD during the merge and project process of projection set variables X (lines 9 to 13) corresponds to an internal node of $\mathcal{T}$ in grade $\mathcal{I}_X$. Realize that no nodes of $\mathcal{T}$ in $\mathcal{I}_X$ are descendants of any node in $\mathcal{I}_Y$, satisfying the definition of X, Y-graded project join tree. In addition, we are performing $\exists$-projection at $\mathcal{I}_Y$ nodes and Σ-projection at $\mathcal{I}_X$ nodes. As such the computation process in Algorithm 1 can be cast under the graded project join tree framework and would therefore follow the same proof as prior work (Dudek, Phan, and Vardi 2021).

4.3 Incremental Counting

PBCount2 supports incremental PB model counting via the caching of intermediate ADDs during the handling of model count queries. In particular, PBCount2 supports the removal and addition of constraints in the PB formula. With reference to Algorithm 1, we cache the ADDs ψ at lines 8 and 13 respectively. In order to store the compute state associated with an ADD, for cache retrieval purposes, we store 3 pieces of information: 1) the set of constraints in PB formula that the ADD represents 2) the projection set variables of ADD that have been projected away and 3) the non-projection set variables of ADD that have been projected away.

The cache retrieval mechanism is shown in Algorithm 2. When given PB formula F', modified from F by adding or removing constraints, the core idea is to loop through the ADDs in the cache and retrieve the compatible ADDs, replacing the initial ADDs at line 3 of Algorithm 1. An ADD ψ is compatible with F' if the set of constraints that ψ represents is a subset of the constraints of F'. In addition, the

---

Algorithm 2: Cache retrieval of PBCount2

Input : PB formula F'
Output: Compute state - set of ADDs retrieved from cache
1   $C' \leftarrow \text{getConstraintSet}(F')$
2   $\mathcal{A}, \mathcal{B} \leftarrow \emptyset$ ▷ Initialize 2 empty sets
3   **for** each ADD ψ in cache **do**
4   **if** $\psi.\text{constraints} \subset C'$ and CheckNoExtraVar(ψ, C') **then** insert ψ into $\mathcal{A}$
5   **for** each ADD $\psi_{\mathcal{A}} \in \mathcal{A}$ **do**
6   conflicts $\leftarrow false$
7   **for** each ADD $\psi_{\mathcal{B}} \in \mathcal{B}$ **do**
8   **if** there exists x which is projected away in $\psi_{\mathcal{A}}$ and not $\psi_{\mathcal{B}}$, or vice versa **then**
9   conflicts $\leftarrow true$
10   **if** not conflicts **then** insert $\psi_{\mathcal{A}}$ into $\mathcal{B}$
11   **for** all $c \in C'$ **do**
12   **if** c not represented in $\mathcal{B}$ **then** insert constraint ADD of c into $\mathcal{B}$
13   **return** $\mathcal{B}$

---

variables that have been projected out from ψ must not appear in any constraint of F' that ψ does not represent, this is handled by CheckNoExtraVar$(\cdot)$ in Algorithm 2 line 4. Subsequently, from lines 5-10, we verify that each ADD that we retrieved is compatible with all other already retrieved ADD candidates in $\mathcal{B}$. Finally, for all constraints that are not represented by an ADD in $\mathcal{B}$, we insert an ADD representing each constraint into $\mathcal{B}$. Cache retrieval replaces lines 1 to 3 of Algorithm 1. It is worth noting that caching ADDs requires us to disable preprocessing currently, as there is a need to maintain a unique id for each constraint and also a fixed variable-to-constraint relation. The restriction arises from the fact that we have to maintain ADD to variable mapping for ADDs in the cache to perform retrieval compatibility checks in Algorithm 2. Preprocessing might remove variables and modify constraints, thus invalidating cached ADDs. As such, preprocessing is disabled when handling incremental counting.

5 Experiments

In this section, we detail the extensive evaluations conducted to investigate the performance of PBCount2's new features, namely projected PB model counting and incremental PB model counting. Specifically, we investigate the following:
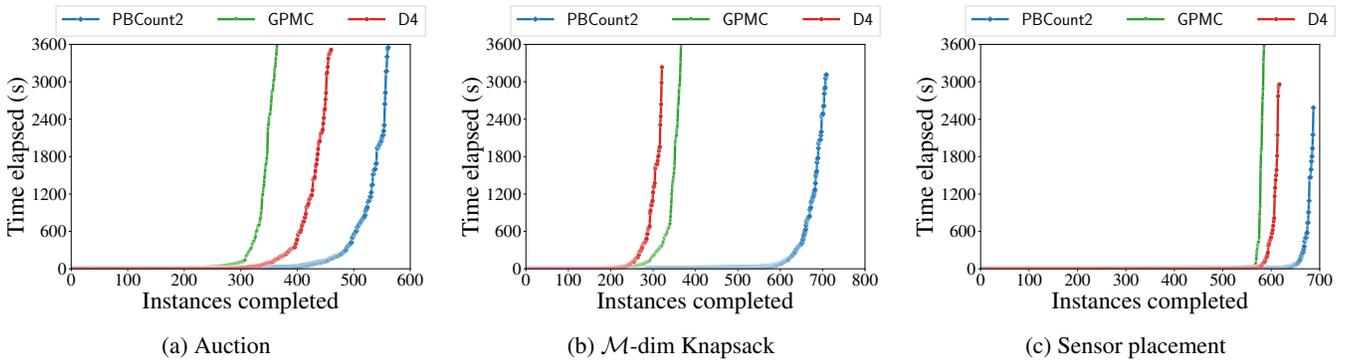
(a) Auction   (b) $\mathcal{M}$-dim Knapsack   (c) Sensor placement

Figure 4: Runtime cactus plots of PBCount2 and competing methods for each benchmark set, for projected model counting.

RQ 1 How does PBCount2 perform on projected PB model counting?

RQ 2 How does PBCount2 perform under incremental settings in comparison to the state-of-the-art PBCount and CNF counters?

5.1 Experiment Setup

In our experiments, we derived synthetic benchmarks from prior work (Yang and Meel 2024), which has 3 benchmark sets - *auction*, *multi-dimension knapsack*, and *sensor placement*. We defer the benchmark statistics to the appendix. The benchmarks are as follows:

RQ 1 Benchmarks: Projected benchmarks are adapted from prior work (Yang and Meel 2024), with the projection set being randomly selected from the original variables. The CNF counters use benchmarks converted using PBLib (Philipp and Steinke 2015), with the same projection set.

RQ 2 Benchmarks: Incremental benchmarks are also adapted from prior work (Yang and Meel 2024). Each benchmark involves computing the model count for a given PB formula (step 1) and each of the 2 (4) subsequent modification steps for 3-step (5-step) configurations. Each modification step involves modifying an existing constraint. The modifications for the auction benchmarks correspond to updates to utility values, i.e. when preferences change. The modifications for the knapsack benchmarks correspond to changes to constraints of a particular dimension. Finally, the modifications to the sensor placement benchmarks correspond to additional redundancy requirements for sensors at important locations in the graph. We provided each step of the incremental benchmarks as separate instances to competing approaches, as none supported incremental counting.

In the experiments, we ran each benchmark instance using 1 core of an AMD EPYC 7713 processor, 16GB memory, and a timeout of 3600 seconds. We implemented PBCount2 in C++, using CUDD library (Somenzi 2015) with double precision and the same ADD variable ordering (*MCS*) as PBCount and ADDMC (Dudek, Phan, and Vardi 2020a; Yang and Meel 2024). We compared PBCount2 against state-of-the-art projected CNF model counters D4

and GPMC[1] for **RQ1**, with the help of PB to CNF conversion tool PBLib. It is worth noting that the converted CNF instances have to be projected onto the original set of PB variables, as auxilary variables are introduced in the conversion process. For incremental benchmarks of **RQ2**, we compared PBCount2 against PBCount as well as CNF counters D4 and GPMC.

5.2 RQ 1: Projected PB Model Counting Performance

Benchmarks	D4	GPMC	PBCount2
Auction	460	364	**561**
$\mathcal{M}$-dim knapsack	321	366	**709**
Sensor placement	617	585	**687**
Total	1398	1315	**1957**

Table 1: Number of projected benchmark instances completed by D4, GPMC and PBCount2 in 3600s. A higher number is better.

We conducted extensive evaluations to understand the performance of PBCount2 compared to state-of-the-art projected CNF model counters D4 and GPMC (Lagniez and Marquis 2017; Ryosuke Suzuki and Sakai 2017). We show the results in Table 1 and cactus plots in Figure 4. PBCount2 is able to complete 1957 instances, demonstrating a substantial lead over the 1398 instances of D4 and the 1315 instances of GPMC. Overall, PBCount2 solves around $1.40\times$ the number of instances of D4 and $1.49\times$ that of GPMC, highlighting the efficacy of PBCount2 in projected PB model counting tasks.

5.3 RQ 2: Incremental Counting Performance

We conducted experiments to analyze the performance of PBCount2's incremental mode against the state-of-the-art PBCount and CNF counters. In our experiments, we looked at the 3-step and 5-step benchmark configurations. The experiments were run with a total timeout of 3600s for the two benchmark configurations. The results are shown in Table 2.

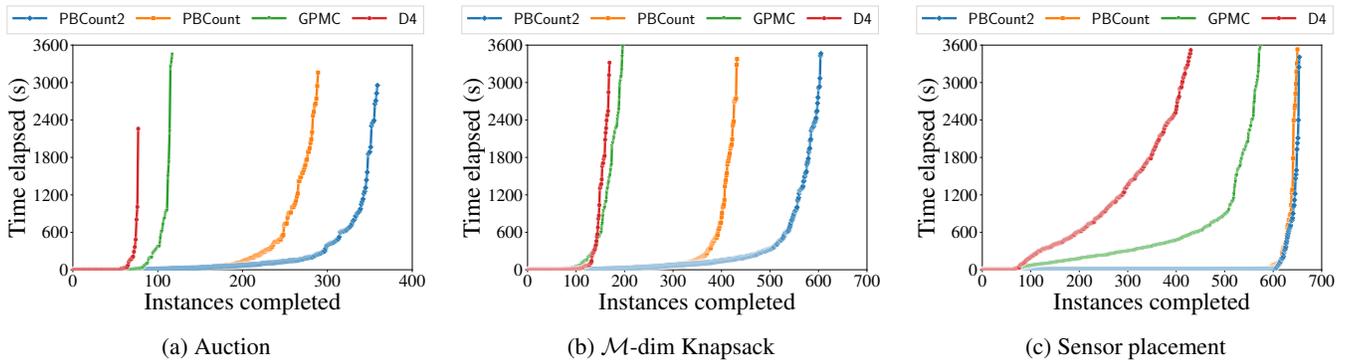[1] Winners of PMC track of Model Counting Competition 2023

Figure 5: Runtime cactus plots of PBCount2 and competing methods, for incremental counting with 5 steps.

Experiment	Counter	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
3-step	D4	85	179	536	800
	GPMC	130	216	586	932
	PBCount	312	458	657	1427
	PBCount2	**369**	**629**	**660**	**1658**
5-step	D4	77	169	430	676
	GPMC	117	196	573	886
	PBCount	289	432	650	1371
	PBCount2	**359**	**605**	**654**	**1618**

Table 2: Number of incremental benchmark instances completed by PBCount2 and competing methods in 3600s. '3-step' indicates results of incremental PBCount2 with 3 counting steps, and '5-step' indicates that with 5 counting steps.

In the 3-step benchmarks, PBCount2 shows a considerable performance advantage over D4, GPMC, and PBCount. More specifically, PBCount2 completed $2.07\times$ the number of benchmarks completed by D4, $1.78\times$ that of GPMC, and $1.16\times$ that of PBCount respectively. Across all 3 incremental benchmark sets for 3-step experiments, PBCount2 performs the same as or better than PBCount. As we move to 5-step benchmarks, we see PBCount2 having a more significant lead over the competing methods. Overall, PBCount2 completes $2.39\times$ the number of instances of D4, $1.83\times$ that of GPMC, and $1.18\times$ that of the original PBCount. Notably when moving from 3-step benchmarks to 5-step benchmarks, the drop in the number of completed incremental benchmarks of PBCount2 is only 40, compared to 56 for PBCount, 46 for GPMC, and 124 for D4. Additionally, we show the cactus plots of each set of incremental benchmarks under 5-step benchmark configuration in Figure 5. The plot further highlights the runtime advantages of PBCount2 over competing approaches. Overall, the evaluations demonstrate the efficacy of PBCount2 at incremental PB model counting, as opposed to existing approaches that can only treat each incremental step as a separate model counting instance.

5.4 Discussion

Through the extensive evaluations, we highlighted PBCount2's superior performance in both projected PB model counting as well as incremental PB model counting benchmarks. To the best of our knowledge, PBCount2 is the first counter to support exact projected PB model counting, and also the first counter to support incremental counting. Despite the practical trade-offs required to support incremental counting, such as the lack of preprocessing and overhead of searching the cache, PBCount2 still demonstrated promising runtime advantages over competing methods. Additionally, PBCount2 also demonstrated the performance benefits of natively handling PB formulas for projected PB model counting, rather than using the convert-and-count approach with state-of-the-art projected CNF counters.

6 Conclusion

In this work, we introduced PBCount2, the first exact PB model counter that supports a) projected model counting and b) incremental model counting, using our LOW-MD heuristic and caching mechanism. Our evaluations highlight PBCount2's superior runtime performance in both projected model counting and incremental counting settings, completing $1.40\times$ and $1.18\times$ the number of instances of the best competing approaches respectively. While PBCount2 supports incremental counting, it requires preprocessing techniques to be disabled to maintain coherent cached ADD to constraint and variable mappings. It would be of interest to overcome the preprocessing restriction in future works, perhaps by introducing the preprocessing techniques as inprocessing steps or by storing additional metadata about the preprocessing process. We hope the newly introduced capabilities of projection and incrementality lead to wider adoption and increased interest in PB model counting in general.

Acknowledgments

References

Aziz, R. A.; Chu, G.; Muise, C.; and Stuckey, P. J. 2015. #∃SAT: Projected Model Counting. In *International Conference on Theory and Applications of Satisfiability Testing*.

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. Algorithms and complexity results for #SAT and Bayesian inference. *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1993. Algebraic decision diagrams and their applications. In *International Conference on Computer Aided Design*.

Blumrosen, L.; and Nisan, N. 2007. *Algorithmic Game Theory*, chapter 11, 267–300. Cambridge University Press.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8): 677–691.

Dudek, J. M.; Phan, V. H. N.; and Vardi, M. Y. 2020a. ADDMC: Weighted Model Counting with Algebraic Decision Diagrams. In *AAAI Conference on Artificial Intelligence*.

Dudek, J. M.; Phan, V. H. N.; and Vardi, M. Y. 2020b. DPMC: Weighted Model Counting by Dynamic Programming on Project-Join Trees. In *International Conference on Principles and Practice of Constraint Programming*.

Dudek, J. M.; Phan, V. H. N.; and Vardi, M. Y. 2021. ProCount: Weighted Projected Model Counting with Graded Project-Join Trees. In *International Conference on Theory and Applications of Satisfiability Testing*.

Dueñas-Osorio, L.; Meel, K. S.; Paredes, R.; and Vardi, M. Y. 2017. Counting-Based Reliability Estimation for Power-Transmission Grids. In *AAAI Conference on Artificial Intelligence*.

Fan, C.; Miller, K.; and Mitra, S. 2020. Fast and Guaranteed Safe Controller Synthesis for Nonlinear Vehicle Models. *Computer Aided Verification*, 12224: 629 – 652.

Fazekas, K.; Biere, A.; and Scholl, C. 2019. Incremental Inprocessing in SAT Solving. In *International Conference on Theory and Applications of Satisfiability Testing*.

Gens, G.; and Levner, E. 1980. Complexity of approximation algorithms for combinatorial problems: a survey. *SIGACT News*, 12: 52–65.

Jackson, D. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62.

Klebanov, V.; Manthey, N.; and Muise, C. 2013. SAT-Based Analysis and Quantification of Information Flow in Programs. In *International Conference on Quantitative Evaluation of Systems*.

Korhonen, T.; and Järvisalo, M. 2021. Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters. In *International Conference on Principles and Practice of Constraint Programming*.

Lagniez, J.-M.; and Marquis, P. 2017. An Improved Decision-DNNF Compiler. In *International Joint Conference on Artificial Intelligence*.

Lai, Y.; Meel, K. S.; and Yap, R. H. C. 2021. The Power of Literal Equivalence in Model Counting. In *AAAI Conference on Artificial Intelligence*.

Latour, A. L. D.; Sen, A.; and Meel, K. S. 2023. Solving the Identifying Code Set Problem with Grouped Independent Support. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence*.

Le Berre, D.; Marquis, P.; Mengel, S.; and Wallon, R. 2018. Pseudo-Boolean Constraints from a Knowledge Representation Perspective. In *International Joint Conference on Artificial Intelligence*.

Lotz, K.; Goel, A.; Dutertre, B.; Kiesl-Reiter, B.; Kong, S.; Majumdar, R.; and Nowotka, D. 2023. Solving String Constraints Using SAT. In *International Conference on Computer Aided Verification*.

Nadel, A. 2022. Introducing Intel(R) SAT Solver. In *International Conference on Theory and Applications of Satisfiability Testing*.

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

Nadel, A.; Ryvchin, V.; and Strichman, O. 2014. Ultimately Incremental SAT. In *International Conference on Theory and Applications of Satisfiability Testing*.

Narodytska, N.; Shrotri, A. A.; Meel, K. S.; Ignatiev, A.; and Marques-Silva, J. 2019. Assessing Heuristic Machine Learning Explanations with Model Counting. In *International Conference on Theory and Applications of Satisfiability Testing*.

Philipp, T.; and Steinke, P. 2015. PBLib - A Library for Encoding Pseudo-Boolean Constraints into CNF. In *International Conference on Theory and Applications of Satisfiability Testing*.

Pisinger, D. 2005. Where are the hard knapsack problems? *Computers & Operations Research*, 32: 2271–2284.

Ryosuke Suzuki, K. H.; and Sakai, M. 2017. Improvement of Projected Model-Counting Solver with Component Decomposition Using SAT Solving in Components. In *JSAI Technical Report*.

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

Sinz, C. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *International Conference on Principles and Practice of Constraint Programming*.

Somenzi, F. 2015. CUDD: CU decision diagram package - release 3.0.0.

Yamada, A.; Kitamura, T.; Artho, C.; Choi, E.-H.; Oiwa, Y.; and Biere, A. 2015. Optimization of Combinatorial Testing by Incremental SAT Solving. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 1–10.

Yang, J.; and Meel, K. S. 2021. Engineering an Efficient PB-XOR Solver. In *International Conference on Principles and Practice of Constraint Programming*.

Yang, S.; and Meel, K. S. 2024. Engineering an Exact Pseudo-Boolean Model Counter. In *Proceedings of the 38th Annual AAAI Conference on Artificial Intelligence*.

Yu, C.; Zhang, X.; Liu, D.; Ciesielski, M. J.; and Holcomb, D. E. 2017. Incremental SAT-Based Reverse Engineering of Camouflaged Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36: 1647–1659.