On Top-Down Pseudo-Boolean Model Counting

Suwei Yang
Grabtaxi Holdings, Singapore
National University of Singapore, Singapore

Yong Lai
Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun, China

Kuldeep S. Meel
Georgia Institute of Technology, Atlanta, GA, USA
University of Toronto, Canada

Abstract

Pseudo-Boolean model counting involves computing the number of satisfying assignments of a given pseudo-Boolean (PB) formula. In recent years, PB model counting has seen increased interest partly owing to the succinctness of PB formulas over typical propositional Boolean formulas in conjunctive normal form (CNF) at describing problem constraints. In particular, the research community has developed tools to tackle exact PB model counting. These recently developed counters follow one of the two existing major designs for model counters, namely the bottom-up model counter design. A natural question would be whether the other major design, the top-down model counter paradigm, would be effective at PB model counting, especially when the top-down design offered superior performance in CNF model counting literature.

In this work, we investigate the aforementioned top-down design for PB model counting and introduce the first exact top-down PB model counter, PBMC. PBMC is a top-down search-based counter for PB formulas, with a new variable decision heuristic that considers variable coefficients. Through our evaluations, we highlight the superior performance of PBMC at PB model counting compared to the existing state-of-the-art counters PBCount, PBCounter, and Ganak. In particular, PBMC could count for 1849 instances while the next-best competing method, PBCount, could only count for 1773 instances, demonstrating the potential of a top-down PB counter design.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Pseudo-Boolean, Model Counting, Constraint Satisfiability

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.31

Related Version *Full Version*: https://arxiv.org/abs/2506.05232

Funding This work was supported in part by the Grab-NUS AI Lab, a joint collaboration between GrabTaxi Holdings Pte. Ltd. and National University of Singapore, and the Industrial Postgraduate Program (Grant: S18-1198-IPP-II), funded by the Economic Development Board of Singapore. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2024-05956], Jilin Provincial Natural Science Foundation [20240101378JC], and Jilin Provincial Education Department Research Project [JJKH20241286KJ].

Acknowledgements The authors thank the reviewers for their feedback. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore.

1 Introduction

Model counting refers to the task of computing the number of satisfying assignments of a given logical formula, typically in the form of a propositional Boolean formula in conjunctive normal form (CNF). Model counting has witnessed sustained attention from the research community in the past decades, owing to its relevance as a technique for tackling problems in

various domains such as circuit vulnerability analysis, network reliability, and probabilistic inference [8, 6, 16]. While model counting techniques are applied across various domains, a potential barrier to their wider adoption is the requirement for users to express the domain problem as a CNF formula before passing it to model counters. As a result, there has been recent interest in the research community in developing model counting tools for alternative input logic formats, such as PBCount [19, 20], PBCounter [13], and ApproxMCPB [18] for pseudo-Boolean formulas and sharpASP [9] and ApproxASP [10] for answer set programming. The motivation for exploring alternative input formats is in part due to the fact that these inputs are more expressive and might be natural for certain application domains.

One notable emerging alternative input format for the model counting task is pseudo-Boolean (PB) formulas. PB formulas and related tools have seen an increase in attention from the research community in recent years, partly owing to the succinctness of PB formulas over typical CNF formulas [14]. There are typically two major paradigms when designing model counters – bottom-up and top-down designs. Whilst both designs have been used in exact CNF model counters, the existing exact PB model counters, namely PBCount and PBCounter, are of the bottom-up design. In particular, both counters make use of algebraic decision diagrams (ADDs) [1] as building blocks to perform counting in a bottom-up manner, starting with ADDs which represent individual constraints of the input PB formula and combining the individual ADDs in the computation process. The bottom-up design enables PBCount and PBCounter to reuse some of the decision diagram techniques found in existing ADD-based CNF model counters, namely ADDMC and DPMC [3, 4]. A natural question one would wonder is *"How would a top-down counter design perform in comparison to existing counters for PB model counting?"*.

In this work, we address the aforementioned question via a prototype top-down exact PB model counter, which we term PBMC. PBMC uses a top-down search-based approach along with techniques similar to conflict-driven clause learning (CDCL), adapted for PB formulas, to perform model counting. In particular, PBMC builds on top of methodologies adapted from RoundingSat PB solver [7], GPMC CNF model counter [15], and PBCount PB counter [19]. In addition, we developed a new variable decision ordering heuristic, VCIS, and a new component caching scheme tailored for PB formulas. In our evaluations, PBMC demonstrates superior performance to state-of-the-art PB counters PBCount and PBCounter, and CNF counter Ganak [19, 13, 17]. More specifically PBMC is able to return counts for 1849 instances while PBCount could only return for 1773 instances, PBCounter for 1508 instances, and Ganak for 1164 instances. In addition, we show that our VCIS variable decision heuristic brings performance benefits, enabling PBMC to improve from 1772 instances using prior heuristics to 1849 instances with VCIS.

2 Background and Preliminaries

2.1 Pseudo-Boolean Formula

A pseudo-Boolean (PB) formula F consists of a set Ω of one or more pseudo-Boolean constraints. Each PB constraint $\omega \in \Omega$ is of the form $\sum_{i=1}^{n} a_i x_i \triangleright k$, where $x_1, ..., x_n$ are Boolean literals, $a_1, ..., a_n$, and k are integers, and $\triangleright$ is one of $\{\geq, =, \leq\}$. $a_1, ..., a_n$ are referred to as term coefficients in the PB constraint, where each term is of the form $a_i x_i$ and k is known as the degree of a PB constraint. An assignment σ is a satisfying assignment if it assigns values to all variables of F such that all the PB constraints in Ω evaluate to *true*. PB model counting refers to the task of determining the number of satisfying assignments of F.

Without loss of generality, the PB constraints in this work are all of the "$\geq$" type unless otherwise stated as "$\leq$" type constraints can be converted by multiplying -1 on both sides, and "$=$" type constraints can be replaced with a pair of "$\geq$" and "$\leq$" constraints. In addition, all coefficients are positive unless stated otherwise as the signs of coefficients can be changed using $\bar{x} = 1 - x$ and updating the degree accordingly. We use the term *gap s* to denote the satisfiability gap of a PB constraint ω under assignment σ. A gap value of 0 or less indicates ω is always satisfied under σ, otherwise ω is yet to be satisfied.

▶ **Definition 1.** *Let ω be a PB constraint $\sum a_i \ell_i \geq k$ and σ be a partial assignment. The gap s of ω under σ is given by $k - (\sum_{j:\sigma(\ell_j)=1} a_j)$, where $j : \sigma(\ell_j) = 1$ indicates all literals ℓ that evaluates to true under σ.*

2.2 Bottom-Up Model Counting

Model counters that are of the bottom-up design typically make use of decision diagrams, which are directed acyclic graph (DAG) representations of functions, in the computation process. In particular, the counters make use of the Apply operation [2] to build up the representation of the input logical formula in a bottom-up manner. Notable bottom-up counters including ADDMC, DPMC, PBCounter, and PBCount [3, 4, 5, 13, 19, 20] employ algebraic decision diagrams (ADDs) [1] to perform model counting tasks. These counters use early projection of variables to reduce the size of the intermediate decision diagrams in practice, as the complexity of the Apply operation is on the order of the product of sizes of the two operand diagrams. The aforementioned counters first construct individual ADDs of the input formula components i.e. clauses in the case of CNF and constraints in the case of PB formula. The individual ADDs are then merged via the Apply operation with the "$\times$" operator to produce intermediate ADDs. Variables are projected out using the Apply operation with the "$+$" operator according to either an initially computed plan or via heuristics. Once all ADDs are merged and all variables are projected out, the final ADD has a single leaf node that indicates the model count.

2.3 Top-Down Model Counting

Existing top-down model counters, typically for CNF formulas, adopt similar search methodologies in solvers, which tend to be the Conflict-Driven Clause Learning (CDCL) algorithm, to compute the model count. Notable counters with the top-down component caching design include D4, GPMC, Ganak, and SharpSAT-TD [12, 15, 17, 11] which demonstrated superior performance as winners of recent model counting competitions. On a high level, top-down counters iteratively assign values to variables until all variables are assigned a value or a conflict is encountered. In the process, sub-components are created by either branching on a variable or splitting a component into smaller variable-disjoint components. The counters cache the components and their respective counts to avoid duplicate computation. When all variables are assigned, the counter reaches a satisfying assignment and will assign the current component the count 1 and backtrack to account for other branches. When the counter encounters a conflict i.e. the existing assigned variable already falsifies the formula, it learns the reason for the conflict and prunes the search space such that this sub-branch of the search will be avoided in the remaining computation process. A leaf component in the implicit search tree either has a count of 0 for conflicts and a count of 1 when all variables are assigned. The count of a component that has variable-disjoint sub-components is the product of the counts of its sub-components. The count of a component that branches on a variable is the sum of the counts from the two branches. After accounting for all search branches, the counter returns the final count of satisfying assignments.

3 Approach

In this section, we detail the general algorithm of our top-down PB model counter PBMC as well as design decisions that enable a performant PB counter.

3.1 Search-Based Algorithm

Algorithm 1 CountPBMC – Counting Algorithm of PBMC.

Input : PB formula F
Output : Model count
1 $F \leftarrow \mathsf{Preprocess}(F)$
2 component $\varphi \leftarrow F$; cache $\zeta \leftarrow \emptyset$; assignment $\sigma \leftarrow \emptyset$
3 $\mathsf{Count}(\varphi, \sigma, \zeta)$
4 **return** $\zeta[\varphi]$

The overall counting approach of PBMC is shown in Algorithms 1 and 2. CountPBMC starts with preprocessing the input PB formula F with techniques adapted from existing state-of-the-art counters [19]. Subsequently, Count sets the preprocessed F as the root component, and initializes an empty cache ζ and an empty assignment σ. CountPBMC calls Count to search the space of assignments and compute the model count. The main computation process of PBMC in Count comprises of the following steps – propagation (line 2), conflict handling (lines 3–7), variable decision (lines 11–13), and component decomposition (line 9). As shown in Count, PBMC starts with propagation so that variable decisions can be inferred and applied. If the propagation leads to a conflict, PBMC would perform conflict analysis and backjumping using similar techniques as RoundingSat [7] (lines 5–6). If the conflict is a top-level conflict, that is it cannot be resolved by backjumping, PBMC will terminate and return 0 as the formula is unsatisfiable (line 7). Otherwise, PBMC would learn a PB constraint that prunes the search space, backjump to the appropriate decision level d, and clear the components and recursive calls created after d.

If there are no conflicts, PBMC will attempt to split the current component into variable-disjoint child components, and the count of the current component would be the product of child component counts (lines 9–10). There are two cases of component splitting – (i) there are no new disjoint components and (ii) there are new disjoint components. In case (i) PBMC will proceed to branch on another variable, and the count of the current component would be the sum of counts from the two branches (lines 12–13). If there are no unassigned variables remaining, the count of the current component would be 1 (line 14). Finally, after completing the implicit search tree exploration PBMC returns the count of F in cache ζ.

3.2 VCIS Variable Decision Heuristic

In our top-down PB model counter PBMC, we introduce a variable decision ordering heuristic that we term *variable coefficient impact score* (VCIS). The intuition for a new variable decision heuristic for PB formulas came from the fact that the coefficient of each literal affects its impact on the overall counting process. Existing variable decision heuristics from CNF model counting literature do not have to consider the impact of coefficients, and therefore modifications are required to adapt to PB model counting. Additionally, tree decomposition may also be less effective when dealing with PB constraints that are relatively longer or constraints that share a lot of variables as in the case of multi-dimensional knapsack problems.

Algorithm 2 Count – helper function of CountPBMC.

```
   Input  : component φ, assignment σ, cache ζ
 1 if φ entry in ζ then return ζ[φ]
 2 isConf ← propagate(σ)
 3 if isConf then
 4     ζ[φ] ← 0
 5     success ← ConflictAnalysis(σ)
 6     if success then Backjump()    ▷ Clears related cache & recursive calls
 7     else ExitReturnZero()                  ▷ Unresolvable conflict, return 0
 8 else
 9     C ← SplitComponent(φ, σ)
10     if |C| > 1 then ζ[φ] ← ∏_{c∈C} Count(c, σ, ζ)
11     else if φ has unassigned variable then
12         l ← pickNextLit(φ, σ)
13         ζ[φ] ← Count(φ ∧ l, σ ∪ {l}, ζ) + Count(φ ∧ l̄, σ ∪ {l̄}, ζ)
14     else ζ[φ] ← 1
15 return ζ[φ]
```

In our VCIS heuristics, we add a coefficient impact score for each variable as an equal-weighted additional component to prior variable decision heuristics adopted from GPMC. The coefficient impact score for a variable x is given by $\left(\sum_{j\in\Omega_x} b_x^j/k_j\right) \div |\Omega_x|$ where Ω_x is the set of constraints that variable x appears in, b_x^j is the coefficient of x in constraint j, and k_j is the degree of constraint j. At the start of the counting process, the score for each variable is computed, and variables are decided in descending order of scores in the counting process. In addition, the phase preferences of variable decisions are set such that the term coefficient with the largest coefficient impact score is applied, this is reflected in line 12 of Algorithm 2. The intuition is that branching on a variable with a larger score would have a greater impact on satisfiability gaps than that on a variable with a smaller score.

3.3 Caching Scheme and Optimization

After computing the count of each component, we store it in the cache to avoid redundant computations when the same component is encountered in a different branch of the counting process. A component in our caching scheme contains the following information for unique identification – 1) the set of unassigned variables in the component, 2) the set of yet-to-be-satisfied PB constraints in the component, and 3) their current gaps. In our implementation, we stored variables and constraints of a component by their IDs.

▶ **Lemma 2.** *Given a PB formula F and partial assignments σ and σ'. Let component c be that of $F|_\sigma$ and c' be that of $F|_{\sigma'}$. If $VarIDs(c) = VarIDs(c')$, $CstrIDs(c) = CstrIDs(c')$, and each constraint ω in c satisfies $gap(\omega, \sigma) = gap(\omega, \sigma')$, then $c = c'$.*

Proof. It is clear that $Vars(c) = Vars(c')$. For each constraint ω with $CstrID(\omega) \in CstrIDs(c)$, ω_σ and $\omega_{\sigma'}$ appear in c and c', respectively. As $Vars(c) = Vars(c')$, the left-hand side of ω_σ is the same as that of $\omega_{\sigma'}$. Then we know $\omega_\sigma = \omega_{\sigma'}$ as $gap(\omega, \sigma) = gap(\omega, \sigma')$. ◀

In our implementation, we cache the variable and constraint IDs in ascending order, thereby only needing to record the first ID and the differences between each i^{th} and $i + 1^{th}$. Since a difference is often smaller than an ID, we can use fewer bits to record each component.

Since each gap s is positive, we can record it as $s-1$. Additionally, we do not record gap for a clausal constraint, that is a constraint with coefficients and degree 1. Such a scheme is particularly efficient for the constraints with a lot of literals.

As a cache entry optimization trick to increase the cache hit rate, we perform saturation on the gaps during the generation of component cache IDs. The intuition for cache gap saturation comes from existing literature on PB solving where saturation is performed on the coefficients of a PB constraint [7] i.e. a term ax in the PB constraint is modified to kx where k is the degree when $a > k$. Given a PB constraint $\sum_{j=1}^{m} a_j x_j \geq s$ where s is the gap, our caching scheme changes s in the cache ID to $\min a_j$ over all coefficients a_j of unassigned variables when $0 < s < \min a_j$. This is sound as assigning any of the remaining literals to true would lead to the constraint being satisfied which is the same outcome without modification of s i.e. the resulting gap being 0 or negative. As such the set of satisfying assignments in both cases, with and without modification of s, are the same.

4 Experiments

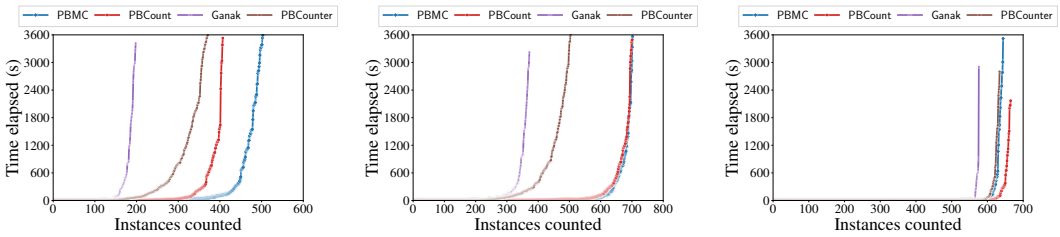
4.1 Setup

We implemented our top-down PB model counter, PBMC, in C++, building on methodologies introduced in the GPMC CNF model counter and RoundingSat PB solver. We performed comparisons to the existing state-of-the-art PB model counters PBCount [19] and PBCounter [13], and also model counting competition (MCC2024) winner Ganak [17]. For consistency, we employed the same benchmark suite as prior work [19]. The suite comprises of 3500 instances across three benchmark sets for PB model counting – *Auction* (1000 instances), *M-dim Knapsack* (1000 instances), and *Sensor placement* (1500 instances). We ran experiments on a cluster with AMD EPYC 7713 CPUs with 1 core and 16 GB memory for each instance and a timeout of 3600 seconds.

Through our evaluations, we seek to understand the performance of our top-down exact PB model counter PBMC. In particular, we investigate the following research questions:
RQ 1: What is the performance of PBMC compared to baselines?
RQ 2: What is the performance impact of our VCIS heuristic in Section 3.2?
RQ 3: What is the performance impact of our cache entry optimization in Section 3.3?

4.2 RQ 1: Performance of Top-down PB Counter PBMC

We compared PBMC against the state-of-the-art PB model counter PBCount, which has a bottom-up design, to understand how well a top-down design would perform. The results are shown in Table 1 and the cactus plot of runtime is shown in Figure 1.



(a) Auction. **(b)** $\mathcal{M}$-dim Knapsack. **(c)** Sensor placement.

Figure 1 Runtime cactus plots of Ganak, PBCount, PBCounter, and PBMC with 3600s timeout. A point on the plot indicates the respective counter could return counts for x instances in y time.

Table 1 Number of benchmark instances counted by Ganak, PBCount, PBCounter, and PBMC in 3600s, higher is better.

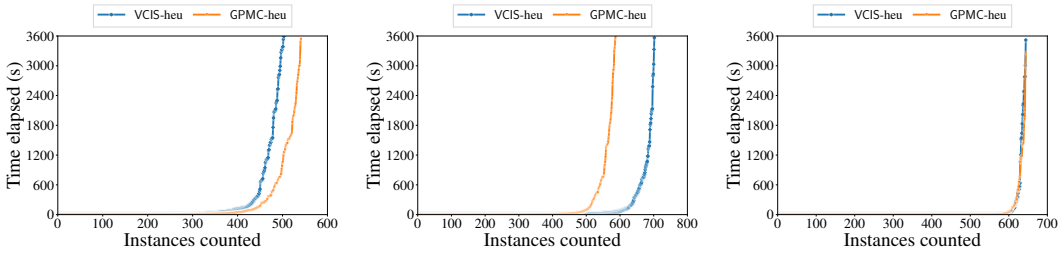
Counters	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
Ganak	198	372	576	1164
PBCount	407	701	**665**	1773
PBCounter	371	503	634	1508
PBMC	**503**	**702**	644	**1849**

Overall, PBMC is able to return counts for 1849 instances, 76 instances more than that of PBCount, 341 instances more than PBCounter, and 685 instances more than Ganak. More specifically, PBMC leads in *auction* and *M dim-knapsack* benchmark sets and came in second in the *sensor placement* benchmark set. In addition, there are 104 instances that could only be counted by PBMC.

It is worth noting that the *sensor placement* benchmark instances all have coefficients of 1 or -1, and that for each instance the degree of all but one PB constraint is 1. We also compared PBMC and PBCount on a modified version of *sensor placement*, whereby coefficients are not all 1 or -1 by accounting for potential redundancy requirements and varying costs associated with different sensor placement locations. On the cost-aware sensor placement instances PBMC outperforms PBCount by returning counts for 715 instances whereas PBCount could only return for 678 instances.

4.3 RQ 2: Performance Impact of VCIS Heuristics

We conducted further experiments to understand the impact of our VCIS heuristic as introduced in Section 3.2. In particular, we compared our implementation of PBMC with VCIS against a baseline version of PBMC that uses the existing variable decision heuristics from the GPMC model counter. The results are shown in Table 2 and the runtime cactus plot is shown in Figure 2.



(a) Auction. **(b)** $\mathcal{M}$-dim Knapsack. **(c)** Sensor placement.

Figure 2 Runtime cactus plots of PBMC with VCIS heuristic and GPMC variable decision heuristics (PBMC-GHeu) with 3600s timeout.

Table 2 Number of PB benchmark instances counted by PBMC, using VCIS and GPMC variable decision heuristics in 3600s.

Counters	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
GPMC-heu	**541**	587	644	1772
VCIS-heu	503	**702**	644	**1849**

In the evaluations, PBMC with our VCIS variable decision heuristic (VCIS-heu) is able to return counts for 1849 benchmark instances in total, substantially more than the 1772 benchmark instances when PBMC is coupled with the existing heuristics from GPMC (GPMC-heu). In particular, VCIS-heu returned counts for 115 more instances than GPMC-heu in $\mathcal{M}$-*dim knapsack* benchmarks, while losing out 38 instances in *Auction* benchmarks. Both heuristics performed the same in *sensor placement* benchmarks. This set of evaluations highlights the tremendous impact of variable decision ordering in the context of PB model counting and demonstrates the performance benefits of our VCIS variable decision heuristic.

4.4 RQ 3: Performance Impact of Cache Entry Optimization

We conducted additional experiments to understand the impact of our optimization for our caching scheme mentioned in Section 3.3, that is modifying the gap value to the smallest coefficient of remaining unassigned variables. We compared PBMC to a version with cache entry optimization disabled (denoted as PBMC-NCS). We show the number of benchmark instances each configuration could count in Table 3. The result shows that our cache entry optimization technique has a positive impact on the performance of PBMC, although less significant than VCIS heuristics in the previous set of experiments.

Table 3 Number of PB benchmark instances counted by PBMC with (PBMC) and without (PBMC-NCS) cache entry optimization in 3600s.

Counters	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
PBMC-NCS	500	700	644	1844
PBMC	**503**	**702**	644	**1849**

4.5 Summary

In **RQ 1**, PBMC demonstrated a significant lead over state-of-the-art competitors. In the process, we observed that PBMC performs better on PB instances with coefficients that are not just 1, as evident in the sensor placement benchmark set. In **RQ 2**, we showed the need for heuristics to consider PB-specific features such as coefficients through the significant performance improvement that VCIS enabled. We also observed that the top-down approach is rather sensitive to variable decision heuristics. Finally, in **RQ 3** we showed that by simply optimizing cache entries, one can get noticeable performance improvements.

5 Conclusion and Future Work

In this work, we explored the top-down design for model counters in the context of pseudo-Boolean model counting. In particular, we introduced our prototype PB model counter PBMC which made use of methodologies from PB solvers and CNF counters as well as our VCIS heuristics to demonstrate leading performance over the existing state-of-the-art PB counter PBCount. PBMC highlights the promising performance of the top-down counter design in the context of PB model counting. It is worth noting that the current performance lead of top-down PB counter PBMC over bottom-up PB counter (PBCount) is not as significant as witnessed in the case of top-down vs bottom-up CNF counters. This could be perhaps partly due to the fact that top-down counters are rather sensitive to variable decision ordering, especially in the context of PB formulas where coefficients and degrees have an impact on

overall performance, as shown in our evaluations. Therefore, it would be of interest to further study variable decision ordering heuristics in the context of PB formulas as future work, because existing heuristics from CNF model counting literature might not be ideal as they do not account for PB-specific features. In addition, it would also make sense to develop techniques to select the appropriate variable decision heuristic based on the given input. Another topic of interest would be preprocessing techniques for the top-down PB counter design, similar to CNF model counting where the top-down counters incorporated various preprocessing techniques which led to significant performance improvements over the years. We hope this study increases the visibility of PB model counting, inspires better counter designs, and more applications of PB model counting.

References

1 R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer Aided Design*, 1993.

2 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

3 Jeffrey M. Dudek, Vu Hoang Nguyen Phan, and Moshe Y. Vardi. Addmc: Weighted model counting with algebraic decision diagrams. In *AAAI Conference on Artificial Intelligence*, 2020.

4 Jeffrey M. Dudek, Vu Hoang Nguyen Phan, and Moshe Y. Vardi. Dpmc: Weighted model counting by dynamic programming on project-join trees. In *International Conference on Principles and Practice of Constraint Programming*, 2020.

5 Jeffrey M. Dudek, Vu Hoang Nguyen Phan, and Moshe Y. Vardi. Procount: Weighted projected model counting with graded project-join trees. In *International Conference on Theory and Applications of Satisfiability Testing*, 2021.

6 Leonardo Dueñas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In *AAAI Conference on Artificial Intelligence*, 2017.

7 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

8 Linus Feiten, Matthias Sauer, Tobias Schubert, Alexander Czutro, Eberhard Böhl, Ilia Polian, and Bernd Becker. #SAT-based vulnerability analysis of security components – A case study. *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 49–54, 2012. doi:10.1109/DFT.2012.6378198.

9 Mohimenul Kabir, Supratik Chakraborty, and Kuldeep S. Meel. Exact asp counting with compact encodings. *AAAI Conference on Artificial Intelligence*, 2024.

10 Mohimenul Kabir, Flavio Everardo, Ankit Shukla, Markus Hecher, Johannes Klaus Fichte, and Kuldeep S. Meel. Approxasp – A scalable approximate answer set counter. In *AAAI Conference on Artificial Intelligence*, 2022.

11 Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters. In *International Conference on Principles and Practice of Constraint Programming*, 2021.

12 Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In *International Joint Conference on Artificial Intelligence*, 2017.

13 Yong Lai, Zhenghang Xu, and Minghao Yin. Pbcounter: weighted model counting on pseudo-boolean formulas. *Frontiers of Computer Science*, 19:193402, 2024. doi:10.1007/S11704-024-3631-1.

14 Daniel Le Berre, Pierre Marquis, Stefan Mengel, and Romain Wallon. Pseudo-boolean constraints from a knowledge representation perspective. In *International Joint Conference on Artificial Intelligence*, 2018.

15 Kenji Hashimoto Ryosuke Suzuki and Masahiko Sakai. Improvement of projected model-counting solver with component decomposition using sat solving in components. In *JSAI Technical Report*, March 2017.

16 Tian Sang, Paul Beame, and Henry A. Kautz. Performing bayesian inference by weighted model counting. In *AAAI Conference on Artificial Intelligence*, 2005.

17 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. In *Proceedings of International Joint Conference on Artificial Intelligence*, August 2019.

18 Jiong Yang and Kuldeep S. Meel. Engineering an efficient pb-xor solver. In *International Conference on Principles and Practice of Constraint Programming*, 2021.

19 Suwei Yang and Kuldeep S. Meel. Engineering an exact pseudo-boolean model counter. In *Proceedings of the 38th Annual AAAI Conference on Artificial Intelligence*, 2024.

20 Suwei Yang and Kuldeep S. Meel. Towards projected and incremental pseudo-boolean model counting. In *Proceedings of the 39th Annual AAAI Conference on Artificial Intelligence*, 2025.