The Power of Literal Equivalence in Model Counting[*]

Yong Lai,[1, 2] **Kuldeep S. Meel,**[2] **Roland H. C. Yap**[2]

[1] Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, China
[2] School of Computing, National University of Singapore

Abstract

The past two decades have seen the significant improvements of the scalability of practical model counters, which have been quite influential in many applications from artificial intelligence to formal verification. While most of exact counters fall into two categories, search-based and compilation-based, Huang and Darwiche's remarkable observation ties these two categories: the trace of a search-based exact model counter corresponds to a Decision-DNNF formula (Huang and Darwiche 2007). Taking advantage of literal equivalences, this paper designs an efficient model counting technique such that its trace is a generalization of Decision-DNNF. We first propose a generalization of Decision-DNNF, called CCDD, to capture literal equivalences, then show that CCDD supports model counting in linear time, and finally design a model counter, called ExactMC, whose trace corresponds to CCDD. We perform an extensive experimental evaluation over a comprehensive set of benchmarks and conduct performance comparison of ExactMC vis-a-vis the state of the art counters, c2d, miniC2D, D4, ADDMC, and Ganak. Our empirical evaluation demonstrates ExactMC can solve 885 instances while the prior state of the art could solve only 843 instances, representing a significant improvement of 42 instances.

1 Introduction

Given a propositional formula φ, the problem of model counting (#SAT), seeks to compute the number of satisfying assignments of φ. Model counting is a fundamental problem with a wide variety of applications ranging from probabilistic inference (Roth 1996; Chavira and Darwiche 2008), neural network verification (Baluta et al. 2019), network reliability (Duenas-Osorio et al. 2017), computational biology, and the like. Given the fundamental nature of the problem, there has been a sustained investigation from theoreticians and practitioners alike over the past three decades. Valiant (1979) showed that the problem of model counting is #P-complete, which was subsequently followed by Toda's seminal work showing that PH $\subseteq P^{\#P}$ (Toda 1989).

From the practitioner's perspective, the earliest approaches to propositional model counting focused on extensions of the DPLL framework via *smarter* enumeration of partial solutions (Birnbaum and Lozinskii 1999). Subsequently, Bayardo and Pehoushek (2000) introduced the notion of component caching to capture the observation that if a formula φ can be partitioned into a subset of clauses, called components, $\{C_1, C_2, \ldots, C_n\}$, such that each of the components is defined over a mutually exclusive set of variables, then the solutions of the φ is simply the product of the solutions of the individual components C_i. Bayardo and Pehsouhek (2000) observed that components reappear in different parts of the search space, and therefore, one can rely on the caching of components to achieve efficiency. Since the early 2000s, three major approaches to the design of scalable model counters have emerged: (1) search-based, (2) compilation-based, and (3) variable elimination-based methods.

The *search-based* techniques predominately focus on the combination of component caching with Conflict Driven clause learning, an idea that was pioneered by Sang et al (2004; 2005), in their model counter Cachet. Subsequently, Thurley (2006) proposed improved component encoding schemes along with enhanced decision heuristics in his model counter, sharpSAT. Recently, Sharma et al. (2019) in their counter Ganak further improved upon sharpSAT via probabilistic caching, improved decision heuristics aided by the utilization of independent support.

The *compilation-based* techniques rely on the paradigm of knowledge compilation, which focuses on the compilation of models represented in an input language to a target language such that the resulting target language supports the desired range of queries such as model counting efficiently. A target language of interest is deterministic Decomposable Negation Normal Form (d-DNNF), which supports model counting in polynomial time (in the compiled size). In practice, *binary decision* is an important property to impose determinism in the design of a compiler (see e.g., D4 (Lagniez and Marquis 2017)), and the resulting subset of d-DNNF is called Decision-DNNF (Oztok and Darwiche 2014). The state of the art model counters based on knowledge compilation focus on compilation to Decision-DNNF and then compute the model count.

Recently, *variable elimination-based* techniques have been proposed that seek to combine the techniques of Bouquet's method and bucket elimination with compact rep-

resentation offered by Algebraic Decision Diagrams, and a scalable counter called ADDMC was implemented by Dudek et al. (2020).

While the development of search-based model counters and knowledge compilation techniques emerged independently to a large extent, Huang and Darwiche's remarkable observation ties the two approaches (Huang and Darwiche 2007). In particular, they observed that the trace of a search-based exact model counter corresponds to d-DNNF (in detail, Decision-DNNF). Huang and Darwiche's observation motivated Muise et al. (2012) to design a state of the art Decision-DNNF compiler based on the exact model counter, sharpSAT. The starting point of our work is to investigate the following natural question: *Can we design an efficient model counting technique such that its trace is a generalization of Decision-DNNF?*

The primary contribution of our work is an affirmative answer to the above question. As a first step, we observe that the widely employed restrictions, in the context of knowledge compilation, on the internal nodes, decomposability, and determinism, are not expressive enough to capture literal equivalences.

Indeed, pre-/in-processing techniques are an important step in modern SAT solvers (Marques-Silva, Lynce, and Malik 2009). We then first propose a generalization of Decision-DNNF, called CCDD, to capture literal equivalence, and show that CCDD supports model counting in linear time. Guided by our motivation, we now design a model counter, called ExactMC, whose trace corresponds to CCDD. To empirically measure the effectiveness of ExactMC, we perform an extensive experimental evaluation over a comprehensive set of benchmarks and conduct performance comparison of ExactMC vis-a-vis the state of the art counters, c2d (Darwiche 2004), miniC2D (Oztok and Darwiche 2015), D4 (Lagniez and Marquis 2017), ADDMC (Dudek, Phan, and Vardi 2020), and Ganak (Sharma et al. 2019). Our empirical evaluation demonstrates while the most number of instances solved among the prior state of the art techniques is 843 (Ganak), ExactMC solves 885, representing a significant improvement of 42 instances. Since the developments in model counting techniques have demonstrated the significance of engineering improvements, we believe that the significant performance improvements of ExactMC open up directions of future research in the improvement of decision heuristics, caching schemes, and the like for counters whose trace corresponds to CCDD.

The rest of the paper is organized as follows: We present notations and preliminaries in Section 2. We introduce CCDD in Section 3 to capture literal equivalence. In Section 4, we present the model counter, ExactMC, whose trace corresponds to CCDD. Next, we present detailed empirical evaluation in Section 5. Finally, we conclude in Section 7.

2 Notations and Background

In a formula or the representations discussed, x denotes a propositional variable, and literal l is a variable x or its negation $\neg x$, where $var(l)$ denotes the variable. $PV = \{x_0, x_1, \ldots, x_n, \ldots\}$ denotes a set of propositional variables. A formula is constructed from constants $true$, $false$

and propositional variables using negation operator $\neg$, conjunction operator $\wedge$, disjunction operator $\vee$, and equality operator $\leftrightarrow$. A clause C (resp. term T) is a set of literals representing their disjunction (resp. conjunction). A formula in conjunctive normal form (CNF) is a set of clauses representing their conjunction. Given a formula φ, a variable x, and a constant b, a substitution $\varphi[x \mapsto b]$ is a transformed formula by replacing x by b in φ. An assignment ω over a variable set X is a mapping from X to $\{true, false\}$. The set of all assignments over X is denoted by 2^X. A model of φ is an assignment over $Vars(\varphi)$ that satisfies φ; that is, the substitution of φ on the model equals to $true$. Let $\mathsf{sol}(\varphi) \subseteq 2^X$ represent the set of models of φ, and $\varphi \models \psi$ iff $\mathsf{sol}(\varphi) \subseteq \mathsf{sol}(\psi)$. Given a formula φ, the problem of model counting is to compute $|\mathsf{sol}(\varphi)|$.

Compilation In this work, we will concern ourselves with the subsets of Negation Normal Form (NNF) wherein the internal nodes are labeled with conjunction ($\wedge$) or disjunction ($\vee$) while the leaf nodes are labeled with $\bot$ ($false$), $\top$ ($true$), or a literal. For a node v, let $\vartheta(v)$ and $Vars(v)$ denote the formula represented by the DAG rooted at v, and the variables that label the descendants of v, respectively. We define the well-known decomposed conjunction (Darwiche and Marquis 2002) as follows:

Definition 1. A conjunction node v is called a *decomposed conjunction* if its children (also known as conjuncts of v) do not share variables. That is, for each pair of children w and w' of v, we have $Vars(w) \cap Vars(w') = \emptyset$.

If each conjunction node is decomposed, we say the formula is in *Decomposable* NNF (DNNF) (Darwiche 2001). DNNF does not support tractable model counting, but the following subset does:

Definition 2. A disjunction node v is called *deterministic* if each two disjuncts of v are logically contradictory. That is, any two different children w and w' of v satisfy that $\vartheta(w) \wedge \vartheta(w') \models false$.

If each disjunction node of a DNNF formula is deterministic, we say the formula is in deterministic DNNF (d-DNNF). *Binary decision* is a practical property to impose determinism in the design of a compiler (see e.g., D4 (Lagniez and Marquis 2017)). Essentially, each decision node with one variable x and two children is equivalent to a disjunction node of the form $(\neg x \wedge \varphi) \vee (x \wedge \psi)$, where φ, ψ represent the formulas corresponding to the children. If each disjunction node is a decision node, the formula is in Decision-DNNF. Each Decision-DNNF formula satisfies the read-once property: each decision variable appears at most once on a path from the root to a leaf.

Search-Based Counters and Compilation As mentioned in Section 1, we focus on the design of search-based model counters. To this end, we first present the skeleton of a general search-based model counter in Algorithm 1.[1] The

---

[1] To improve readability, we slightly modified the fashion of calculating the current count to be consistent with our ExactMC algorithm. X is the set of variables in the original formula.

Algorithm 1: SearchCounter(φ)

---
1 **if** $\varphi = false$ **then return** 0
2 **if** $\varphi = true$ **then return** $2^{|X|}$
3 **if** $Cache(\varphi) \neq nil$ **then return** $Cache(\varphi)$
4 $\Psi \leftarrow$ DECOMPOSE(φ)
5 **if** $|\Psi| > 1$ **then**
6 $\quad c \leftarrow \prod_{\psi \in \Psi}$ SearchCounter(ψ)
7 $\quad$ **return** $Cache(\varphi) \leftarrow \frac{c}{2^{(|\Psi|-1)\cdot|X|}}$
8 **else**
9 $\quad x \leftarrow$ PICKGOODVAR(φ)
10 $\quad c_0 \leftarrow$ SearchCounter($\varphi[x \mapsto false]$)
11 $\quad c_1 \leftarrow$ SearchCounter($\varphi[x \mapsto true]$)
12 $\quad$ **return** $Cache(\varphi) \leftarrow \frac{c_0+c_1}{2}$
13 **end**

---

algorithms often maintain a cache that stores the residual sub-formulas along with their corresponding model count. The component-based decomposition, represented in line 4, seeks to partition the φ into sub-formulas, referred to as components, such that each of the components is defined over a mutually disjoint set of variables. Else, we pick a variable in line 9 and recursively compute the exact model count. Huang and Darwiche observed that the trace of the execution of such a model counter could be viewed to correspond to a Decision-DNNF, a negation normal form that has been well studied in the knowledge compilation community. In this context, it is worth emphasizing that Decision-DNNF supports linear time model counting, which is reflected in simple constant time computations in lines 7 and 12 during each step of the recursions wherein every step of the recursion would correspond to a node in Decision-DNNF capturing the trace of the execution of SearchCounter.

Remark on Approximate Model Counting While this work focuses on exact model counting, it is worth remarking that there has been a long line of work in the design of efficient hashing-based approximate model counters that seek to provide (ε, δ)-guarantees (Stockmeyer 1983; Gomes, Sabharwal, and Selman 2006; Chakraborty, Meel, and Vardi 2013, 2016; Soos and Meel 2019; Soos, Gocht, and Meel 2020).

3 Capturing Literal Equivalences by CCDD

To seek an answer to the natural question of designing a counter whose trace is a generalization of Decision-DNNF, we first investigate appropriate generalizations of Decision-DNNF. To this end, we turn to the literal equivalences, a powerful technique in SAT solving, and we design a new representation language that seeks to utilize literal equivalences. We first discuss how to capture literal equivalence from the knowledge compilation perspective, which is then manifested into a corresponding new tractable language, called CCDD. We finally show that CCDD supports linear model counting, which serves as motivation for us to design a counter whose trace corresponds to CCDD.

3.1 Capturing Literal Equivalences

Given two literals l and l', we use $l \leftrightarrow l'$ to denote literal equivalence of l and l'. Given a set of literal equivalences E, let $E' = \{l \leftrightarrow l', \neg l \leftrightarrow \neg l' \mid l \leftrightarrow l' \in E\}$; and then we define semantic closure of E, denoted by $\lceil E \rceil$, as equivalence closure of E'. Now for every literal l under $\lceil E \rceil$, let $[l]$ denote the equivalence class of l. Given E, a unique equivalent representation of E, denoted by $\lfloor E \rfloor$ and called *prime literal equivalences*, is defined as follows:

$$\lfloor E \rfloor = \bigcup_{x \in PV, \min_\prec[x]=x} \{x \leftrightarrow l \mid l \in [x], l \neq x\}$$

where $\min_\prec[x]$ is the minimum variable appearing in $[x]$ over the lexicographic order $\prec$. It can be shown that $\lceil E \rceil = \lceil \lfloor E \rfloor \rceil$.

Let φ be a formula and let E be a set of prime literal equivalences implied by φ. We can obtain another formula φ' by performing a *literal-substitution*: replace each l (resp. $\neg l$) in φ with x (resp. $\neg x$) for each $x \leftrightarrow l \in E$. Note that, $\varphi \equiv \varphi' \wedge \bigwedge_{x \leftrightarrow l \in E} x \leftrightarrow l$.

Example 1. *Given* $E = \{\neg x_1 \leftrightarrow x_3, \neg x_4 \leftrightarrow x_3, \neg x_2 \leftrightarrow \neg x_6, x_5 \leftrightarrow x_5\}$, *we have* $\lfloor E \rfloor = \{x_1 \leftrightarrow \neg x_3, x_1 \leftrightarrow x_4, x_2 \leftrightarrow x_6\}$. *Given* $\varphi = (x_1 \vee \neg x_3 \vee x_4 \vee x_7) \wedge (x_1 \vee x_3 \vee x_5) \wedge (\neg x_1 \leftrightarrow x_3) \wedge (\neg x_4 \leftrightarrow x_3) \wedge (\neg x_2 \leftrightarrow \neg x_6) \wedge (x_5 \leftrightarrow x_5)$, *each literal equivalence in* $\lfloor E \rfloor$ *is implied. We can use* $\lfloor E \rfloor$ *to perform a literal-substitution to simplify* φ *as* $(x_1 \vee x_7) \wedge \bigwedge \lfloor E \rfloor$.

We propose a new notion on conjunction nodes to represent literal equivalences:

Definition 3. A *kernelized conjunction node* v is a conjunction node consisting of a distinguished child, we call the *core* child, denoted by $ch_{core}(v)$, and a set of remaining children which define equivalences, denoted by $Ch_{rem}(v)$, such that:

1. Every $w_i \in Ch_{rem}(v)$ describes a literal equivalence, i.e., $w_i = \langle x \leftrightarrow l \rangle$ and the union of $\vartheta(w_i)$, denoted by E_v, represents a set of prime literal equivalences.

2. For each literal equivalence $x \leftrightarrow l \in E_v$, $var(l) \notin Vars(ch_{core}(v))$.

We now show how the model count of a kernelization of formula is related to its core. For simplicity, we use a sightly more general definition for model in Propositions 1–2. Given a formula φ and a set of variables $X \supseteq Vars(\varphi)$, a model of φ over X is an assignment over X that satisfies φ. In practice, when we want to count models for φ, we only need to make $X = Vars(\varphi)$.

Proposition 1. *For a kernelized conjunction v over X, if $\vartheta(ch_{core}(v))$ has m models over X, then $\vartheta(v)$ has $\frac{m}{2^{|Ch_{rem}(v)|}}$ models over X.*

Proof. Given each kernelized conjunction $\varphi \wedge (x_{i_1} \leftrightarrow l_{i_1}) \wedge \cdots \wedge (x_{i_m} \leftrightarrow l_{i_m})$, we can rewrite it as a recursive form $\left[[[\varphi \wedge (x_{i_1} \leftrightarrow l_{i_1})] \wedge (\wedge x_{i_2} \leftrightarrow l_{i_2})] \wedge \cdots \right] \wedge (x_{i_m} \leftrightarrow l_{i_m})$. Next we show given a kernelized conjunction $\varphi = \psi \wedge (x \leftrightarrow l)$ over X, if ψ has m models over X, then φ has $\frac{m}{2}$ models over X. By induction, we get Proposition 1. Without loss of generality, assume $l = x'$. As this is a kernelized conjuction,

$x' \notin \mathit{Vars}(\psi)$. Let $\omega \cup \{x' = \mathit{false}\}$ and $\omega \cup \{x' = \mathit{true}\}$ be two assignments over X, where ω is a model of ψ over $X \setminus \{x'\}$. Since $x \leftrightarrow x'$, exactly one of the two assignments can be a model of φ, so half of the models of ψ are the models of φ. $\qquad\square$

3.2 Defining CCDD

We begin with the widely used idea of augmenting decision diagram with conjunction in knowledge compilation (Fargier and Marquis 2006; Oztok and Darwiche 2014; Bart et al. 2014; Lai, Liu, and Yin 2017). This idea is restated in a general form, Conjunction & Decision Diagram, to cover our kernelization-integrated languages:

Definition 4. A Conjunction & Decision Diagram (CDD) is a rooted DAG wherein each node v is labeled with a symbol $sym(v)$. If v is a leaf, $sym(v) = \bot$ or $\top$. Otherwise, $sym(v)$ is a variable (v is called a *decision node*) or operator $\wedge$ (called a *conjunction node*). Each internal node v has a set of children $Ch(v)$. For a decision node, $Ch(v) = \{lo(u), hi(u)\}$, where $lo(u)$ ($hi(u)$) is connected by a dashed (solid) edge. The formula represented by a CDD rooted at u is defined as follows:

$$\vartheta(u) = \begin{cases} \mathit{false} & sym(u) = \bot \\ \mathit{true} & sym(u) = \top \\ \bigwedge_{v \in Ch(u)} \vartheta(v) & sym(u) = \wedge \quad (1) \\ [\neg sym(u) \wedge \vartheta(lo(u))] \vee \\ \quad [sym(u) \wedge \vartheta(hi(u))] & \text{otherwise} \end{cases}$$

Hereafter we denote a leaf node by $\langle\bot\rangle$ or $\langle\top\rangle$, an internal node by $\langle sym(v), Ch(v)\rangle$; and sometimes a decision node is denoted by $\langle sym(v), lo(v), hi(v)\rangle$. Given a CDD rooted at v (denoted by $\mathcal{D}_v$), its size $|\mathcal{D}_v|$ is defined as the number of its edges, similar to other languages in the knowledge compilation literature. If we admit only read-once decisions and decomposed conjunctions, then the subset of CDD is Decision-DNNF. We are now ready to describe an extension of Decision-DNNF that captures literal equivalence, by imposing a different constraint on conjunction:

Definition 5 (Constrained CDD, CCDD)**.** A CDD is called *constrained* if each decision node u and its decision descendant v satisfy $sym(u) \neq sym(v)$, and each conjunction node v is either: (i) decomposed; or (ii) kernelized. The language of all constrained CDDs is called CCDD.

We use $\wedge_d$ and $\wedge_k$ to denote decomposed and kernelized conjunctions respectively. Figure 1 depicts a CCDD. Since Decision-DNNF is a subset of CCDD and is known to be complete, we obtain the following result on the completeness of CCDD:

Theorem 1. *Given a formula, there is at least one CCDD to represent it.*

3.3 Linear Model Counting

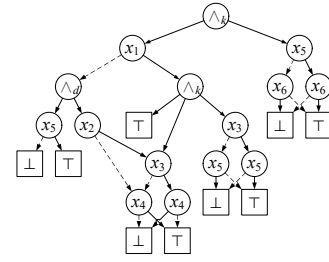Now we are ready to show how CCDD supports model counting in linear time:

Figure 1: A diagram in CCDD representing $(x_5 \leftrightarrow x_6) \wedge \Big[\big[\neg x_1 \wedge x_5 \wedge [(\neg x_2 \wedge x_4) \vee (x_2 \wedge (x_3 \leftrightarrow \neg x_4))]\big] \vee \big[x_1 \wedge (x_3 \leftrightarrow \neg x_4) \wedge (x_3 \leftrightarrow x_5)\big]\Big]$, where the core child of the root is the child on the left hand side

Proposition 2. *Given a node u in* CCDD *with $\mathit{Vars}(u) \subseteq X$ and a node v in $\mathcal{D}_u$, we use $CT(v)$ to denote the model count of $\vartheta(v)$ over X. Then $CT(u)$ can be recursively computed in linear time in $|\mathcal{D}_u|$:*

$$CT(u) = \begin{cases} 0 \text{ or } 2^{|X|} & u \text{ is a leaf} \\ c^{-1} \cdot \prod_{v \in Ch(u)} CT(v) & u \text{ is a } \wedge_d\text{-node} \\ \dfrac{CT(ch_{core}(u))}{2^{|Ch(u)|-1}} & u \text{ is a } \wedge_k\text{-node} \\ \dfrac{CT(lo(u)) + CT(hi(u))}{2} & \text{otherwise} \end{cases}$$

where $c = 2^{(|Ch(u)|-1) \cdot |X|}$.

Proof. It is easy to see the case for the leaf nodes. The case for kernelized conjunctions was discussed in Proposition 1. For a decision node u, we can see that there are only half of the models over X of its low (resp. high) child satisfying $\neg sym(u) \wedge \vartheta(u)$ (resp. $sym(u) \wedge \vartheta(u)$), since $sym(u)$ does not appear in $\vartheta(lo(u))$ (resp. $\vartheta(hi(u))$). Now we discuss the case for decomposed conjunctions. Given a decomposed conjunction u, we show that this proposition holds when $|Ch(u)| = 2$. For the cases $|Ch(u)| > 2$, we only need to iteratively use the conclusion of the case $|Ch(u)| = 2$. Assume that $Ch(u) = \{v, w\}$. We can divide X into three disjoint sets $X_1 = \mathit{Vars}(v)$, $X_2 = \mathit{Vars}(w)$, and $X_3 = X \setminus (X_1 \cup X_2)$. Assume that $\vartheta(v)$ and $\vartheta(w)$ have m_1 and m_2 models over X_1 and X_2, respectively. Then $\vartheta(v)$ and $\vartheta(w)$ have $m_1 \cdot 2^{|X_2|+|X_3|}$ and $m_2 \cdot 2^{|X_1|+|X_3|}$ models over X, respectively. $\vartheta(u)$ has $m_1 \cdot m_2$ models over $X_1 \cup X_2$, and has $m_1 \cdot m_2 \cdot 2^{|X_3|}$ models over X. It is easy to see the following equation:

$$m_1 \cdot m_2 \cdot 2^{|X_3|} = \frac{m_1 \cdot 2^{|X_2|+|X_3|} \cdot m_2 \cdot 2^{|X_1|+|X_3|}}{2^{|X|}}$$

$\qquad\square$

4 ExactMC: A Scalable Model Counter

As discussed in previous section, CCDD has two key properties: CCDD is complete, i.e., every formula can be represented using CCDD and it supports linear model counting. These two properties motivate us to design a model counter,

ExactMC, whose trace corresponds to CCDD. Algorithm ExactMC takes in a CNF formula φ and the set of variables X (initialized to $Vars(\varphi)$), and returns $|\mathsf{sol}(\varphi)|$. ExactMC is based on the architecture of search-based model counters, as shown in Algorithm 1. We remark that in the context of knowledge compilation, there are some other languages that are generalizations of Decision-DNNF (see e.g., Sym-DDG (Bart et al. 2014)). As far as we know, however, there are no scalable model counters reported, based on these languages.

We first handle the base cases lines 1–2 corresponding to the first case in Proposition 2. Since we are interested in computing the number of satisfying assignments over X, we return $2^{|X|}$ in line 2 in case φ is $true$. We then turn to the discovery and usage of literal equivalences in the formula to perform model counting as presented in lines 4–11. We use a heuristic, SHOULDKERNELIZE, to determine whether we should spend time in detecting and using literal equivalence because those steps are themselves possibly costly. We discuss SHOULDKERNELIZE further in Section 4.1. When SHOULDKERNELIZE returns $true$, we turn to DETECTLITEQU to discover literal equivalences in the formula in line 5 and if a non-trivial literal equivalence is discovered, we proceed to perform exact model counting with respect to kernelized conjunction in lines 7–9 (corresponding to the third case in Proposition 2, where $\|\lfloor E \rfloor\|$ is equal to the number of children minus one). In particular, we first invoke CONSTRUCTCORE to perform literal-substitution (see Section 3.1) to obtain the formula, $\hat{\varphi}$, corresponding to the core child, and then recursively call ExactMC over $\hat{\varphi}$.

If no non-trivial literal equivalence is found in line 5, then the rest of the algorithm follows the template of search-based model counters. We first invoke DECOMPOSE in line 12 to determine if the formula φ can be decomposed into components. If such a decomposition is not found, we pick a variable x and recursively invoke ExactMC on the residual formulas $\varphi[x \mapsto false]$ and $\varphi[x \mapsto true]$. We remark that lines 14–15 and lines 17–20 correspond to the second and fourth cases in Proposition 2, respectively.

We now employ a simple example to show how kernelization helps us to reduce the search space. For simplicity, we assume PICKGOODVAR gives variables in the lexicographic order, and SHOULDKERNELIZE always returns $true$ or always returns $false$.

Example 2. *Consider the CNF formula φ:*

$$\varphi = (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$
$$\wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_4)$$
$$\wedge (x_1 \vee x_4) \wedge (\neg x_2 \vee \neg x_5) \wedge (x_2 \vee x_5)$$

with $X = \{x_1, \ldots, x_5\}$. Now, there are two cases:

Without Kernelization *If* SHOULDKERNELIZE *is false, the trace of* ExactMC *will correspond to the CCDD in Figure 2a.*

With Kernelization *If* SHOULDKERNELIZE *is true, we can detect two literal equivalences $x_1 \leftrightarrow \neg x_4$ and $x_2 \leftrightarrow \neg x_5$, and thus the residual sub-formula is equivalent to $(x_1 \oplus x_2 \oplus x_3 = 1)$. After running lines 18–20, we have two other literal equivalences $x_2 \leftrightarrow \neg x_3$ and $x_2 \leftrightarrow x_3$.*

Algorithm 2: ExactMC(φ, X)

1 **if** $\varphi = false$ **then return** 0
2 **if** $\varphi = true$ **then return** $2^{|X|}$
3 **if** $Cache(\varphi) \neq nil$ **then return** $Cache(\varphi)$
4 **if** SHOULDKERNELIZE(φ) **then**
5      $E \leftarrow$ DETECTLITEQU(φ)
6   **if** $\|\lfloor E \rfloor\| > 0$ **then**
7          $\hat{\varphi} \leftarrow$ CONSTRUCTCORE($\varphi, \lfloor E \rfloor$)
8          $c \leftarrow$ ExactMC($\hat{\varphi}, X$)
9   **return** $Cache(\varphi) \leftarrow \frac{c}{2^{\|\lfloor E \rfloor\|}}$
10   **end**
11 **end**
12 $\Psi \leftarrow$ DECOMPOSE(φ)
13 **if** $|\Psi| > 1$ **then**
14      $c \leftarrow \prod_{\psi \in \Psi}\{$ExactMC($\psi, X$)$\}$
15   **return** $Cache(\varphi) \leftarrow \frac{c}{2^{(|\Psi|-1) \cdot |X|}}$
16 **else**
17      $x \leftarrow$ PICKGOODVAR(φ)
18      $c_0 \leftarrow$ ExactMC($\varphi[x \mapsto false], X$)
19      $c_1 \leftarrow$ ExactMC($\varphi[x \mapsto true], X$)
20   **return** $Cache(\varphi) \leftarrow \frac{c_0 + c_1}{2}$
21 **end**

The trace corresponds to the CCDD in Figure 2b. When backtracking to the call corresponding to the root, we know the core child has 16 models over $\{x_1, \ldots, x_5\}$, then φ has $\frac{16}{2^2} = 4$ models.

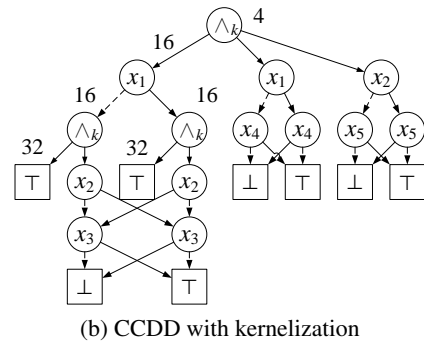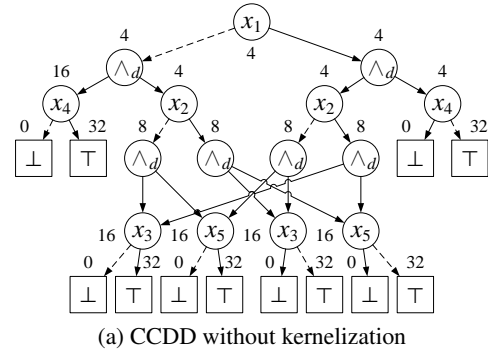(a) CCDD without kernelization

(b) CCDD with kernelization

Figure 2: DAGs corresponding to counting of φ. The number labeled beside a node denotes its model count on $\{x_1, \ldots, x_5\}$

4.1 Implementation

Since the core contribution of our work lies in the on-the-fly construction and usage of kernelized conjunction nodes, we now discuss the implementation details that are crucial for runtime efficiency. As is the case for most heuristics in SAT solving and related communities, we selected parameters empirically and detailed analysis for different choices of parameters is left to future work. Given the original formula φ, we will use $\#NonUnitVars$ to denote the number of variables appearing in the non-unit clauses of φ.

SHOULDKERNELIZE As mentioned earlier, the detection and usage of literal equivalences can be significantly advantageous but our preliminary experiments indicated the need for caution. In particular, we observed that the implicit construction of kernelized conjunction node over the trace was not helpful for *easy* instances. To this end, we rely on the number of variables as a proxy for the hardness of a formula, in particular at every level of recursion, we classify a formula φ to be easy if $|Vars(\varphi)| \leq easy\_bound$, where $easy\_bound$ is defined by $\min(128, \#NonUnitVars/2)$. If the formula φ is classified as easy, then SHOULDKERNELIZE returns $false$. Else, we consider the search path from the last kernelization (if no kernelization, then the root) to the current node. If the number of unit clauses on the path is greater than 48 and also greater than twice the number of decisions on the path, SHOULDKERNELIZE returns $true$. The intuition behind the usage of unit clauses is that unit clauses are often useful to simplify the current sub-formula and thus possibly lead to many literal equivalences. In the other cases, SHOULDKERNELIZE returns $false$. We empirically determine the heuristic to have good performance.

DETECTLITEQU Recall, we need to check for a chosen pair of literals l_1 and l_2, whether $l_1 \leftrightarrow l_2$ is a literal equivalence implied by φ at line 5 in DETECTLITEQU. For an efficient check, we rely on using implicit Boolean Constraint Propagation (i-BCP) for the assignments $l_1 \wedge \neg l_2$ and $\neg l_1 \wedge l_2$. The usage of i-BCP in model counting dates back to sharpSAT (Thurley 2006).

Prime Literal Equivalences We employ union-find sets to represent prime literal equivalences, which allows us to efficiently compute prime literal equivalences from a set of literal equivalences.

Decision Heuristics We combine the widely used heuristic minfill (Darwiche 2009) and a new dynamic ordering, which we call *dynamic combined largest product* (DLCP) to pick good variables. Given a variable, the DLCP value is the product of the weighted sum of negative appearances and positive appearances of the variable. Given an appearance, the heuristic considers the following cases: (i) if it is in an original binary clause, the weight is 2; (ii) if it is in a learnt binary clause, the weight is 1; (iii) if it is in an original non-binary clause with m literals, the weight is $\frac{1}{m}$; otherwise, (iv) the weight is 0. If the minfill treewidth is greater than a crossover constant $\min(128, \#NonUnitVars/7)$, we use DLCP, otherwise, minfill.

We observed in the experiments that for an instance with high treewidth, DLCP is often useful to lead to a sub-formula with many literal equivalences after assigning some variables.

5 Experiments

We implemented a prototype of ExactMC in C++ and evaluated it on a comprehensive set of 1114 benchmarks (Benchmarks) from a wide range of application areas, including automated planning, Bayesian networks, configuration, combinatorial circuits, inductive inference, model checking, program synthesis, and quantitative information flow (QIF) analysis. These instances have been employed in the past to evaluate model counting and knowledge compilation techniques (Lagniez and Marquis 2017; Lai, Liu, and Wang 2013; Lai, Liu, and Yin 2017; Soos and Meel 2019; Fremont, Rabe, and Seshia 2017). The experiments were run on a cluster[2] where each node has 2xE5-2690v3 CPUs with 24 cores and 96GB of RAM. Each instance was run on a single core with a timeout of 3600 seconds and 4GB memory.

We compared ExactMC with state-of-the-art from exact counters from each of the three paradigms: compilation-based, search-based or variable elimination-based. Compilation-based counters used the following target languages: (i) miniC2D on SDD (Oztok and Darwiche 2015); (ii) c2d (Darwiche 2004) and D4 (Lagniez and Marquis 2017) on d-DNNF. For search-based counters, we compared with a state of the art tool called Ganak (Sharma et al. 2019), which is a recent probabilistic exact model counter that implicitly combines Decision-DNNF approach with probabilistic hashing to provide exact model count with a given confidence $1 - \delta$ (we used the default $\delta = 0.05$). Note that probabilistic exact is a stronger notion than another related notion of probabilistic approximate counting (Chakraborty, Meel, and Vardi 2019). Also, perhaps it is worth remarking that Ganak builds on and was shown to significantly improve upon the prior state of the art search-based counter, sharpSAT. For variable elimination-based counters, we compared with ADDMC (Dudek, Phan, and Vardi 2020).

We used the widely employed pre-processing tool B+E (Lagniez, Lonca, and Marquis 2016) for all the instances, which was shown very powerful in model counting (Lagniez, Lonca, and Marquis 2016; Sharma et al. 2019). We remark that B+E can often simplify almost all of literal equivalences in the original formula detected by i-BCP. We emphasize that the literal equivalences in ExactMC is a "in-processing technology", and since B+E is already used, the literal equivalences used in ExactMC are basically the ones appearing in the sub-formulas. Consistent with recent studies, we excluded the preprocessing time from the solving time for each tool as preprocessed instances were used on all solvers. We emphasize that the usage of pre-processing favors other competing tools than ExactMC. In detail, Ganak, c2d, miniC2D, D4, ADDMC, and ExactMC solved 173, 117, 170, 95, 283, and 54 less instances without

---

[2]The cluster is a typical HPC cluster where jobs are run through a job queue.
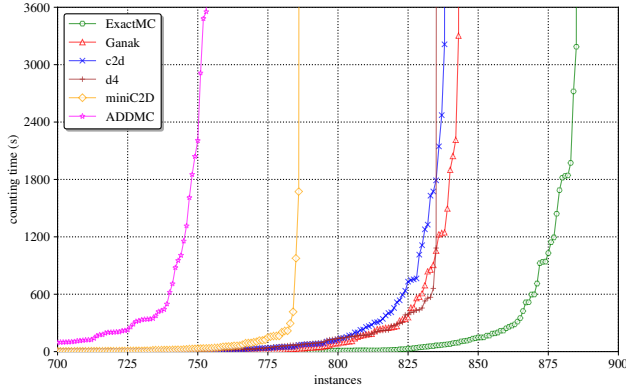
Figure 3: Cactus plot comparing the solving time of different counters. (Best viewed in color)

the pre-processing, respectively. We employed the minfill heuristic for variable ordering in miniC2D and c2d, which has been shown to significantly improve runtime and space performance (Muise et al. 2012; Lai, Liu, and Yin 2017). D4 and Ganak employ their own custom variable ordering heuristics, which were shown to improve their performance (Lagniez and Marquis 2017; Sharma et al. 2019).

Table 1 shows the performance of the six counters. Overall, ExactMC solved 42, 47, 50, 99, and 132 more instances than Ganak, c2d, D4, miniC2D, and ADDMC, respectively. Upon closer inspection of the performance of various tools across different domains, we observe that ExactMC performed the best on seven out of nine domains. Figure 3 shows the cactus plot for runtime for all the six tools. The x-axis gives the number of benchmarks; and the y-axis is running time, i.e., a point (x, y) in Figure 3 shows that x benchmarks took less than or equal to y seconds to solving. The results show that ExactMC can improve the state-of-the-art model counting across all three paradigms.

We remark that all of c2d, miniC2D, D4, and Ganak perform searches with respect to Decision-DNNF. In order to show the effect of kernelization, we compared ExactMC with the virtual best solver of c2d, miniC2D, D4, and Ganak (VBS-DecDNNF). We found that even in such an extreme case, ExactMC solved 9 more instances than VBS-DecDNNF.

We present the effect of kernelization on some selected instances and solving times in Table 2. The experimental results show that for some instances (e.g., blasted_case_2_ptb_1), even a small number of kernelizations are very useful to accelerate solving. Furthermore, it is worth noticing that we are able to perform a large number of kernelizations in the benchmarks, showing that substantial literal equivalence can occur in sub-formulas despite the use of pre-processing, e.g. sygus_09A-1 (Program-Synthesis). We also conducted experiments where kernelization was disabled in ExactMC (without lines 4–11 in Algorithm 2). We found that the resulting counter solved 17 less instances than the original version of ExactMC, and the

average PAR-2 score increased to 1613 from 1509.[3]

We analyze the running time for individual instances by omitting the easy instances (solved by at least five tools in under 2 seconds) and hard instances (not solved by any tool). ExactMC performed the best on 39.5% instances, while Ganak, c2d, miniC2D, D4, and ADDMC performed the best on 22.3%, 2.5%, 3.9%, 16.1%, and 15.7%, respectively. Compared with state-of-art counters, we see that ExactMC is able to solve more instances with improved runtime.

6 Discussion on Tractability of CCDD

Encouraged by the significant performance improvement attained by ExactMC, we delve into deeper investigation of the underlying language, CCDD. To this end, we study CCDD from a knowledge compilation perspective and seek to characterize the tractability of CCDD. In this paper, our focus is on improving the scalability of model counters. We refer the reader to Darwiche and Marquis's seminal work (Darwiche and Marquis 2002) for definitions of different standard operations in the literature. We focus on the five queries: implicant check, model counting, consistency check, validity check, and model enumeration.

We first show that CCDD supports tractable implicant check:

Proposition 3. *Given a consistent term T and a CCDD node u, we use $IM(T, u)$ to denote whether $T \models \vartheta(u)$. Then $IM(T, u)$ can be recursively performed in linear time:*

$$IM(T, u) = \begin{cases} false & sym(u) = \bot \\ true & sym(u) = \top \\ IM(T, lo(u)) & \neg sym(u) \in T \\ IM(T, hi(u)) & sym(u) \in T \\ \bigwedge_{v \in Ch(u)} IM(T, v) & otherwise \end{cases}$$

Proof. The constant, and decomposed and kernelized conjunction cases are obvious, and thus we focus on the decision case. Note that a literal equivalence is a special decision node. For the case where $\neg sym(u) \in T$, each model of T is not a model of $sym(u) \wedge \vartheta(hi(u))$, and thus $T \models \vartheta(u)$ iff $T \models \vartheta(lo(u))$. The case where $sym(u) \in T$ is similar. Otherwise, $T \models \vartheta(u)$ iff $\neg sym(u) \wedge T \models \neg sym(u) \wedge \vartheta(lo(u))$ and $sym(u) \wedge T \models sym(u) \wedge \vartheta(hi(u))$ iff $T \models \vartheta(lo(u))$ and $T \models \vartheta(hi(u))$. $\square$

Since CCDD supports model counting in linear time, we obtain that CCDD supports consistency check, validity check, and model enumeration in polynomial time.

Theorem 2. CCDD *supports model counting, consistency check, validity check, and implicant check in time polynomial in the DAG size, and supports model enumeration in time polynomial in both the DAG size and model count.*

According to the notations in the knowledge compilation map (Darwiche and Marquis 2002), we know that CCDD

---

[3]The average PAR-2 scoring scheme gives a penalized average runtime, assigning a runtime of two times the time limit (instead of a "unsolved" status) for each benchmark not solved by a tool.

Table 1: Comparative counting performance between Ganak, c2d, miniC2D, D4, ADDMC, and ExactMC, where each cell below tool refers to the number of solved instances

domain (#)	probabilistic exact counter	exact counter				
	Ganak	c2d	miniC2D	D4	ADDMC	ExactMC
Bayesian-Networks (201)	170	183	183	179	**191**	186
BlastedSMT (200)	163	160	155	162	166	**169**
Circuit (56)	49	50	48	49	45	**51**
Configuration (35)	**35**	**35**	28	33	21	31
Inductive-Inference (41)	18	19	15	18	3	**22**
Model-Checking (78)	73	**74**	71	72	64	**74**
Planning (243)	207	209	201	206	187	**212**
Program-Synthesis (220)	96	76	68	90	52	**108**
QIF (40)	**32**	**32**	17	26	24	**32**
Total (1114)	843	838	786	835	753	**885**

Table 2: Counting statistics on selected instances, where "–" denotes timeout or out of memory, "kdepth" denotes the maximum number of kernelizations appearing in each path, "#kers" denotes the total number of kernelizations, and the other columns are about solving time in seconds

domain	instance	Ganak	c2d	miniC2D	D4	ADDMC	ExactMC		
							time	kdepth	#kers
Bayesian-Networks	Grids_11	1239.5	–	–	–	0.3	941.1	0	0
BlastedSMT	blasted_case_2_ptb_1	–	–	–	–	–	4.4	4	383
Circuit	2bitadd_11	–	–	–	–	–	2721.4	2	11580
Configuration	C168_FW	338.6	14.0	23.1	68.3	–	–	–	–
Inductive-Inference	ii32d2	–	–	–	–	–	285.7	2	559
Model-Checking	bmc-galileo-8	1.3	2145.9	–	–	–	1.7	6	34
Planning	logistics.c	214.4	536.7	–	173.5	–	30.8	5	7579
Program-Synthesis	sygus_09A-1	–	–	–	–	–	150.0	18	21851
QIF	min-2s	61.3	0.3	20.1	125.4	0.1	10.3	1	8

satisfies **CT**, **CO**, **VA**, **IM**, and **ME**, respectively. We mention that if we restrict the number of $\wedge_k$-nodes in each path from the root to a leaf, to be a constant t, we can obtain a subset of CCDD. This subset is still a superset of Decision-DNNF, and supports the same tractable operations as Decision-DNNF. We remark that another representation in the knowledge compilation literature called EADT (Koriche et al. 2013) uses a generalization of literal equivalence; however, EADT is not a generalization of Decision-DNNF.

7 Conclusion

In this paper, we proposed a new model counting method ExactMC whose trace corresponds to CCDD, which is a generalization of Decision-DNNF. To this end, we introduced a new notion of kernelization to capture literal equivalence. Experimental results show that the performance of ExactMC is significantly better than the state-of-art counters Ganak, c2d, miniC2D, D4, and ADDMC. We believe that ExactMC opens up new directions of future research in improvement of decision heuristics, caching schemes, and the like for counters whose trace corresponds to CCDD. Since the notion of kernelization is orthogonal to other notions such as determinism, decomposability in knowledge compilation, we also expect kernelization will help the knowledge compilation community to identify more interesting languages and develop more efficient compilers.

References

Baluta, T.; Shen, S.; Shinde, S.; Meel, K. S.; and Saxena, P. 2019. Quantitative verification of neural networks and its security applications. In *Proc. of CCS*, 1249–1264.

Bart, A.; Koriche, F.; Lagniez, J.; and Marquis, P. 2014. Symmetry-Driven Decision Diagrams for Knowledge Compilation. In *ECAI*, volume 263, 51–56.

Bayardo Jr, R. J.; and Pehoushek, J. D. 2000. Counting models using connected components. In *AAAI/IAAI*, 157–162.

Benchmarks. 2020. https://www.cril.univ-artois.fr/KC/benchmarks.html, https://github.com/meelgroup/sampling-

benchmarks, https://github.com/dfremont/counting-benchmarks, https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html. [Online; accessed 21-Jan-2020].

Birnbaum, E.; and Lozinskii, E. L. 1999. The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research* 10: 457–477.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2013. A Scalable Approximate Model Counter. In *Proc. of CP*, 200–216.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proc. of IJCAI*.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2019. On the Hardness of Probabilistic Inference Relaxations. In *Proc. of AAAI*, 7785–7792. AAAI Press.

Chavira, M.; and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172(6–7): 772–799.

Darwiche, A. 2001. Decomposable negation normal form. *Journal of the ACM* 48(4): 608–647.

Darwiche, A. 2004. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *Proc. of ECAI*, 328–332.

Darwiche, A. 2009. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press.

Darwiche, A.; and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17: 229–264.

Dudek, J. M.; Phan, V.; and Vardi, M. Y. 2020. ADDMC: Weighted Model Counting with Algebraic Decision Diagrams. In *Proc. of AAAI*, 1468–1476.

Duenas-Osorio, L.; Meel, K. S.; Paredes, R.; and Vardi, M. Y. 2017. Counting-Based Reliability Estimation for Power-Transmission Grids. In *Proc. of AAAI*.

Fargier, H.; and Marquis, P. 2006. On the use of partially ordered decision graphs in knowledge compilation and quantified Boolean formulae. In *Proc. of AAAI*, 42–47.

Fremont, D. J.; Rabe, M. N.; and Seshia, S. A. 2017. Maximum Model Counting. In Singh, S. P.; and Markovitch, S., eds., *Proc. of AAAI*, 3885–3892. AAAI Press.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2006. Model counting: A new strategy for obtaining good bounds. In *Proc. of AAAI*, volume 21, 54–61.

Huang, J.; and Darwiche, A. 2007. The Language of Search. *Journal of Artificial Intelligence Research* 29: 191–219.

Koriche, F.; Lagniez, J.; Marquis, P.; and Thomas, S. 2013. Knowledge Compilation for Model Counting: Affine Decision Trees. In *Proc. of IJCAI*, 947–953.

Lagniez, J.-M.; Lonca, E.; and Marquis, P. 2016. Improving Model Counting by Leveraging Definability. In *IJCAI*, 751–757.

Lagniez, J.-M.; and Marquis, P. 2017. An Improved Decision-DNNF Compiler. In *Proc. of IJCAI*, 667–673.

Lai, Y.; Liu, D.; and Wang, S. 2013. Reduced Ordered Binary Decision Diagram with Implied Literals: A New Knowledge Compilation Approach. *Knowledge and Information Systems* 35(3): 665–712.

Lai, Y.; Liu, D.; and Yin, M. 2017. New Canonical Representations by Augmenting OBDDs with Conjunctive Decomposition. *Journal of Artificial Intelligence Research* 58: 453–521.

Marques-Silva, J. P.; Lynce, I.; and Malik, S. 2009. Conflict-Driven Clause Learning SAT Solvers. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 131–153. IOS Press.

Muise, C. J.; McIlraith, S. A.; Beck, J. C.; and Hsu, E. I. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT. In *Proceedings of the 25th Canadian Conference on Artificial Intelligence*, 356–361.

Oztok, U.; and Darwiche, A. 2014. On compiling CNF into Decision-DNNF. In *Proc. of CP*, 42–57.

Oztok, U.; and Darwiche, A. 2015. A Top-Down Compiler for Sentential Decision Diagrams. In *Proc. of AAAI*, 3141–3148.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82: 273–302.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H.; and Pitassi, T. 2004. Combining component caching and clause learning for effective model counting. In *Proc. of SAT*.

Sang, T.; Beame, P.; and Kautz, H. 2005. Performing Bayesian inference by weighted model counting. In *Prof. of AAAI*, 475–481.

Sharma, S.; Roy, S.; Soos, M.; and Meel, K. S. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *IJCAI*, 1169–1176.

Soos, M.; Gocht, S.; and Meel, K. S. 2020. Tinted, Detached, and Lazy CNF-XOR solving and its Applications to Counting and Sampling. In *Proc. of CAV*.

Soos, M.; and Meel, K. S. 2019. BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting. In *Proc. of AAAI*.

Stockmeyer, L. 1983. The complexity of approximate counting. In *Proc. of STOC*, 118–126.

Thurley, M. 2006. SharpSAT: counting models with advanced component caching and implicit BCP. In *Proc. of SAT*, 424–429.

Toda, S. 1989. On the computational power of PP and (+)P. In *Proc. of FOCS*, 514–519. IEEE.

Valiant, L. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8(3): 410–421.