Fast Converging Anytime Model Counting[*]

Yong Lai [1], **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

Model counting is a fundamental problem which has been influential in many applications, from artificial intelligence to formal verification. Due to the intrinsic hardness of model counting, approximate techniques have been developed to solve real-world instances of model counting. This paper designs a new anytime approach called PartialKC for approximate model counting. The idea is a form of partial knowledge compilation to provide an unbiased estimate of the model count which can converge to the exact count. Our empirical analysis demonstrates that PartialKC achieves significant scalability and accuracy over prior state-of-the-art approximate counters, including satss and STS. Interestingly, the empirical results show that PartialKC reaches convergence for many instances and therefore provides exact model counting performance comparable to state-of-the-art exact counters.

1 Introduction

Given a propositional formula φ, the model counting problem (#SAT) is to compute the number of satisfying assignments of φ. Model counting is a fundamental problem in computer science which has a wide variety of applications in practice, ranging from probabilistic inference (Roth 1996; Chavira and Darwiche 2008), probabilistic databases (Van den Broeck and Suciu 2017), probabilistic programming (Fierens et al. 2015), neural network verification (Baluta et al. 2019), network reliability (Dueñas-Osorio et al. 2017), computational biology (Sashittal and El-Kebir 2019), and the like. The applications benefit significantly from efficient propositional model counters.

In his seminal work, Valiant (1979) showed that model counting is #P-complete, where #P is the set of counting problems associated with NP decision problems. Theoretical investigations of #P have led to the discovery of strong evidence for its hardness. In particular, Toda (1989) showed that PH $\subseteq P^{\#P}$, i.e. each problem in the polynomial hierarchy can be solved with just one call to a #P oracle. Although there has been large improvements in the scalability of practical exact model counters, the issue of hardness is intrinsic.

As a result, researchers have studied approximate techniques to solve real-world instances of model counting.

The current state-of-the-art approximate counting techniques can be categorized into three classes based on the guarantees over estimates (Chakraborty, Meel, and Vardi 2013). Let φ be a formula with Z models. A counter in the first class is parameterized by (ε, δ), and computes a model count of φ that lies in the interval $[(1+\varepsilon)^{-1}Z, (1+\varepsilon)Z]$ with confidence at least $1 - \delta$. ApproxMC (Chakraborty, Meel, and Vardi 2013, 2016) is a well-known counter in the first class. A counter in the second class is parameterized by δ, and computes a lower (or upper) bound of Z with confidence at least $1 - \delta$ including tools such as MBound (Gomes, Sabharwal, and Selman 2006) and SampleCount (Gomes et al. 2007). Counters in the third class provide weaker guarantees, but offer relatively accurate approximations in practice and state-of-the-art counters include satss (Gogate and Dechter 2011) and STS (Ermon, Gomes, and Selman 2012). We remark that some counters in the third class can be converted into a counter in the second class. Despite significant efforts in the development of approximate counters over the past decade, scalability remains a major challenge.

In this paper, we focus on the third class of counters to achieve scalability. To this end, it is worth remarking that a well-known problem for this class of approximate model counters is slow convergence. Indeed, in our experiments, we found satss and STS do not converge in more than one hour of CPU time for many instances, whereas an exact model counter can solve those instances in several minutes of CPU time. We seek to remedy this. Firstly, we notice that each sample generated by a model counter in the third class represents a set of models of the original CNF formula. Secondly, we make use of knowledge compilation techniques to accelerate the convergences. Since knowledge compilation languages can often be seen as a compact representation of the model set of the original formula; we infer the convergence of the approximate counting by observing whether a full compilation is reached. By storing the existing samples into a compiled form, we can also speed up the subsequent sampling by querying the current stored representation.

We generalize a recently proposed language called CCDD (Lai, Meel, and Yap 2021), which was shown to support efficient exact model counting, to represent the existing samples. The new representation, partial CCDD, adds two new

---

types of nodes called known or unknown respectfully representing, (i) a sub-formula whose model count is known; and (ii) a sub-formula which is not explored yet. We present an algorithm, PartialKC, to generate a random partial CCDD that provides an unbiased estimate of the true model count. PartialKC has two desirable properties for an approximate counter: (i) it is an anytime algorithm; (ii) it can eventually converge to the exact count. Our empirical analysis demonstrates that PartialKC achieves significant scalability as well as accuracy over prior state of the art approximate counters, including satss and STS.

The rest of the paper is organized as follows: We present notations, preliminaries, and related work in Sections 2–3. We introduce partial CCDD in Section 4. In Section 5, we present the model counter, PartialKC. Section 6 gives detailed experiments, and Section 7 concludes.

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 the negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), and equality ($\leftrightarrow$) operators. A clause C is a set of literals representing their disjunction. 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 with 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 $true$. Given a formula φ, we use $Z(\varphi)$ to denote the number of models, and the problem of model counting is to compute $Z(\varphi)$.

Sampling-Based Approximate Model Counting Due to the hardness of exact model counting, sampling is a useful technique to estimate the count of a given formula. Since it is often hard to directly sample from the distribution over the model set of the given formula, we can use importance sampling to estimate the model count (Gogate and Dechter 2011, 2012). The main idea of importance sampling is to generate samples from another easy-to-simulate distribution Q called the proposal distribution. Let Q be a proposal distribution over $Vars(\varphi)$ satisfying that $Q(\omega) > 0$ for each model ω of φ. Assume that 0 divided by 0 equals 0, and therefore $Z(\varphi) = \mathbb{E}_Q\left[\frac{Z(\varphi|_\omega)}{Q(\omega)}\right]$. For a set of samples $\omega_1, \ldots, \omega_N$, $\widehat{Z_N} = \frac{1}{N}\sum_{i=1}^{N} \frac{Z(\varphi|_{\omega_i})}{Q(\omega_i)}$ is an *unbiased* estimator of $Z(\varphi)$; that is, $\mathbb{E}_Q[\widehat{Z_N}] = Z(\varphi)$. Similarly, a function $\widetilde{Z_N}$ is an *asymptotically* unbiased estimator of $Z(\varphi)$ if $\lim_{N \to \infty} \mathbb{E}_Q[\widetilde{Z_N}] = Z(\varphi)$. It is obvious that each unbiased estimator is asymptotically unbiased.

Knowledge 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, we use $sym(v)$ to denote the labeled symbol, and $Ch(v)$ to denote the set of its children. We also use $\vartheta(v)$ 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.

Recently, a new type of conjunctive nodes called *kernelized* was introduced (Lai, Meel, and Yap 2021). 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 the 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 $\prec$.

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:

(a) 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.

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

We use $\wedge_d$ and $\wedge_k$ to denote decomposed and kernelized conjunctions respectively. A *Constrained Conjunction*

& Decision Diagram (CCDD) consists of decision nodes, conjunction nodes, and leaf nodes where each decision variable appears at most once on each path from the root to a leaf, and each conjunction node is either decomposed or kernelized. Figure 1 depicts a CCDD. Lai et al. (2021) showed that CCDD supports model counting in linear time.
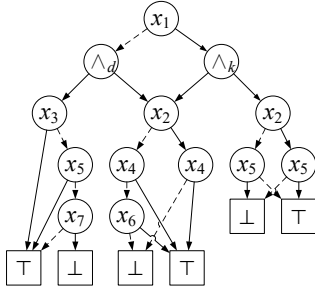

Figure 1: An illustrated CCDD

3 Related Work

The related work can be viewed along two directions: (1) work related to importance sampling for graphical models; and (2) work related to approximate compilation for propositional formula. While this work focuses on anytime approximate model counting, we highlight a line of work in the first category, namely, the design of efficient hashing-based approximate model counters that seek to provide 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).

The most related work in the first direction is Sample-Search (Gogate and Dechter 2011, 2012). For many KC (knowledge compilation) languages, each model of a formula can be seen a particle of the corresponding compiled result. SampleSearch used the generated models (i.e., samples) to construct an AND/OR sample graph, which can be used to estimate the model count of a CNF formula. Each AND/OR sample graph can be treated as a partial compiled result in AOBDD (binary version of AND/OR Multi-Valued Decision Diagram (Mateescu, Dechter, and Marinescu 2008)). They showed that the estimate variance of the partial AOBDD is smaller than that of the mean of samples. Our PartialKC approach has three main differences from SampleSearch. First, the SampleSearch approach envisages an independent generation of each sample, while the KC technologies used in PartialKC can accelerate the sampling (and thus the convergence), which we experimentally verified. Second, the decomposition used by the partial AOBDD in SampleSearch is static, while the one in PartialKC is dynamic. Results from the KC literature generally suggest that dynamic decomposition is more effective than a static one (Muise et al. 2012; Lagniez and Marquis 2017). Finally, our KC approach allows to determine if convergence is reached; but SampleSearch does not.

The related work in the second direction is referred to as approximate KC. Given a propositional formula and a range of weighting functions, Venturini and Provan (Venturini and Provan 2008) proposed two incremental approximate algorithms respectively for prime implicants and DNNF, which selectively compile all solutions exceeding a particular threshold. Their empirical analysis showed that these algorithms enable space reductions of several orders-of-magnitude over the full compilation. Intrinsically, partial KC is a type of approximate KC that still admits exactly reasoning to some extent (see Proposition 1). The output of PartialKC can be used to compute an unbiased estimate of model count, while the approximate DNNF compilation algorithm from Venturini and Provan only computes a lower bound of the model count. Some bottom-up compilation algorithms of OBDD (Bryant 1986) and SDD (Darwiche 2011) also perform compilation in an incremental fashion by using the operator APPLY. However, the OBDD and SDD packages (Somenzi 2015; Choi and Darwiche 2013) do not overcome the size explosion problem of full KC, because the sizes of intermediate results can be significantly larger than the final compilation result (Huang and Darwiche 2004). Thus, Friedman and Van den Broeck (2018) proposed an approximate inference algorithm, collapsed compilation, for discrete probabilistic graphical models. The differences between PartialKC and collapsed compilation are as follows: (i) collapsed compilation works in a bottom-up fashion while PartialKC works top-down; (ii) collapsed compilation is asymptotically unbiased while PartialKC is unbiased; and (iii) collapsed compilation does not support model counting so far.

4 Partial CCDD

In this section, we will define a new representation called *partial* CCDD, used for approximate model counting. For convenience, we call the standard CCDD *full*.

Definition 4 (Partial CCDD). Partial CCDD is a generalization of full CCDD, adding two new types of leaf nodes labeled with '?' or a number, which are the *unknown* and *known* nodes, respectively. Each edge from a decision node v is labeled by a pair $\langle p_b(v), f_b(v) \rangle$ giving the estimated marginal probability and visit frequency, where $b = 0$ (resp. 1) means the edge is dashed (resp. solid). For a decision node v, $p_0(v) + p_1(v) = 1$; $p_b(v) = 0$ iff $sym(ch_b(v)) = \bot$; and $f_b(v) = 0$ iff $sym(ch_b(v)) = ?$.

Hereafter, we use $\langle ? \rangle$ to denote an unknown node. For convenience, we require that each conjunctive node cannot have any unknown child. For simplicity, we sometimes use $f(w)$ and $p(w)$ to denote $\langle f_0(w), f_1(w) \rangle$ and $\langle p_0(w), p_1(w) \rangle$, respectively, for each decision node w. For a partial CCDD node v, we denote the DAG rooted at v by $\mathcal{D}_v$. We now establish a part-whole relationship between partial and full CCDDs:

Definition 5. Let u and u' be partial and full CCDD nodes, respectively, from the same formula. $\mathcal{D}_u$ is a *part* of $\mathcal{D}_{u'}$ iff one of the following conditions holds:
(a) u is an unknown node;
(b) $u = \langle \bot \rangle$ or $\langle \top \rangle$, and $u' = u$;

(c) u is a known node, and $sym(u)$ equals the model count of u';

(d) $u = \langle x, ch_0(u), ch_1(u) \rangle$, $sym(u') = x$, and the partial CCDDs rooted at $ch_0(u)$ and $ch_1(u)$ are parts of the full CCDDs rooted at $ch_0(u')$ and $ch_1(u')$, respectively;

(e) $u = \langle \wedge_d, Ch(u') \rangle$, $sym(u') = \wedge_d$, $|Ch(u)| = |Ch(u')|$, and each partial CCDD rooted at a child of u is exactly a part of one full CCDD rooted at a child of u'; or

(f) $u = \langle \wedge_k, ch_{core}(u), Ch_{rem}(u) \rangle$, $sym(u') = \wedge_k$, $Ch_{rem}(u) = Ch_{rem}(u')$, and the partial CCDD rooted at $ch_{core}(u)$ is a part of the full CCDD rooted at $ch_{core}(u')$.

Figure 2 shows two different partial CCDDs from the full CCDD in Figure 1 which can be generated by MICROKC given later in Algorithm 2. Given a partial CCDD rooted at u that is a part of full CCDD rooted at u', the above definition establishes a mapping from the nodes of $\mathcal{D}_u$ to those of $\mathcal{D}_{u'}$.
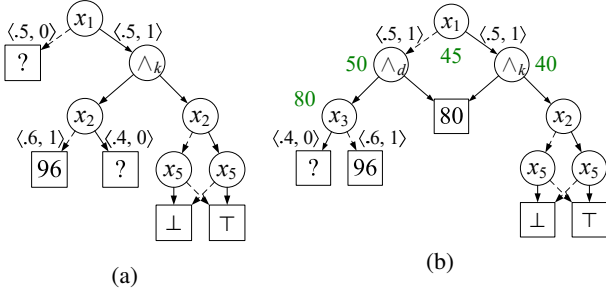

Figure 2: two partial CCDDs

A full CCDD can be seen as a compact representation for the model set corresponding to the original knowledge base. We can use a part of the full CCDD to estimate its model count. Firstly, a partial CCDD can be used to compute deterministic lower and upper bounds of the model count, respectively:

Proposition 1. *Let u and u' be, respectively, a partial CCDD node and a full CCDD node over X such that $\mathcal{D}_u$ is a part of $\mathcal{D}_{u'}$. For each unknown node v in $\mathcal{D}_u$ corresponding to v' in $\mathcal{D}_{u'}$ under the part-whole mapping, we assume that we have computed an estimate $\widetilde{Z}(v)$ that is a lower (resp. upper) bound $Z(v')$. A lower (resp. upper) bound of $Z(u')$ can be recursively computed in linear time:*

$$
\widetilde{Z}(u) = \begin{cases}
0 & sym(u) = \bot \\
2^{|X|} & sym(u) = \top \\
sym(u) & sym(u) \in \mathbb{N} \\
c^{-1} \cdot \prod_{v \in Ch(u)} \widetilde{Z}(v) & sym(u) = \wedge_d \\
\dfrac{\widetilde{Z}(ch_{core}(u))}{2^{|Ch(u)|-1}} & sym(u) = \wedge_k \\
\dfrac{\widetilde{Z}(ch_0(u)) + \widetilde{Z}(ch_1(u))}{2} & sym(u) \in X
\end{cases} \quad (1)
$$

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

We remark that we must compute lower or upper bound for each unknown node before applying Eq. (1). In practice, for example, we can compute lower and upper bounds of the model count of an unknown node as 0 and $2^{|X|}$, respectively. However, we mainly aim at computing an unbiased estimate of the model count. We will use a randomly generated partial CCDD to compute an unbiased estimate of the model count of the corresponding full CCDD. The main difference between the new computation and the one in Eq. (1) is at the estimation on decision nodes. Given a randomly generated partial CCDD rooted at u, the new estimate of the model count can be computed recursively in linear time:

$$
\widehat{Z}(u) = \begin{cases}
0 & sym(u) = \bot \\
2^{|X|} & sym(u) = \top \\
sym(u) & sym(u) \in \mathbb{N} \\
c^{-1} \cdot \prod_{v \in Ch(u)} \widehat{Z}(v) & sym(u) = \wedge_d \\
\dfrac{\widehat{Z}(ch_{core}(u))}{2^{|Ch(u)|-1}} & sym(u) = \wedge_k \\
\dfrac{\widehat{Z}(ch_0(u)) \cdot f_0(u)}{2 p_0(u) \cdot (f_0(u) + f_1(u))} + & \\
\dfrac{\widehat{Z}(ch_1(u)) \cdot f_1(u)}{2 p_1(u) \cdot (f_0(u) + f_1(u))} & sym(u) \in X
\end{cases}
$$
$$(2)$$

where $c = 2^{(|Ch(u)|-1) \cdot |X|}$. We remark that for each decision node u with one unknown child v, the visit frequency $f_b(u)$ on the edge from u to v is 0. Thus, $\widehat{Z}(v) \cdot f_b(u)$ always equals zero in the decision case of Eq. (2).

Example 1. *Consider the partial CCDD in Figure 2b. We denote the root by u and the decision child of $ch_0(u)$ by v.*

$$
\widehat{Z}(v) = \frac{? \times 0}{2 \times 0.4 \times 1} + \frac{96 \times 1}{2 \times 0.6 \times 1} = 80;
$$
$$
\widehat{Z}(ch_0(u)) = 2^{-1 \times 7} \times \widehat{Z}(v) \times 80 = 50;
$$
$$
\widehat{Z}(ch_1(u)) = \frac{\widehat{Z}(ch_{core}(ch_0(u)))}{2^1} = 40;
$$
$$
\widehat{Z}(u) = \frac{\widehat{Z}(ch_0(u)) \times 1}{2 \times 0.5 \times 2} + \frac{\widehat{Z}(ch_1(u)) \times 1}{2 \times 0.5 \times 2} = 45.
$$

5 PartialKC: An Anytime Model Counter

We aim to estimate model counts for CNF formulas that cannot be solved within time and space limits for exact model counters. Our approach is to directly generate a randomly partial CCDD formula from the CNF formula rather than from an equivalent full CCDD. This is achieved with PartialKC given in Algorithm 1, which compiles a CNF formula into a partial CCDD.

PartialKC calls MICROKC in Algorithm 2 multiple times in a given timeout t. We use a hash table called $Cache$ to store the current compiled result implicitly. Each call of MICROKC updates $Cache$, implicitly enlarging the current compiled result rooted at $Cache(\varphi)$. PartialKC reaches convergence in line 5 when the root of the resulting partial CCDD is fully computed, making the count exact. In lines

6–11, we will restart the compilation if we are close to memory limit. Thus, PartialKC is an anytime algorithm. Consider that the execution of PartialKC restarts compilation k times, so PartialKC will generate $k+1$ Partial CCDDs with roots $v_0, \dots, v_k$. Let $\widehat{Z}_0, \dots, \widehat{Z}_k$ be the counts computed on the roots. We also assume that we call MicroKC N_i times in PartialKC for generating the Partial CCDD rooted at v_i. Then $\frac{(N_0 \times Z_0) + \cdots + (N_k \times Z_k)}{N_0 + \cdots + N_k}$ is a proper estimate of the true count. Then in line 7 in Algorithm 1, $N = N_0 + \cdots + N_i$, $M = N_0 + \cdots + N_{i-1}$, and $\widehat{Z} = \frac{(N_0 \times \widehat{Z}_0) + \cdots + (N_k \times \widehat{Z}_{k-1})}{M}$. After line 8, $\widehat{Z} = \frac{(N_0 \times Z_0) + \cdots + (N_k \times \widehat{Z}_k)}{N}$.

Algorithm 1: PartialKC(φ, t)

1   $N \leftarrow M \leftarrow Z \leftarrow 0$
2   **while** *running time does not exceed to t* **do**
3     $N \leftarrow N + 1$
4   MICROKC(φ)
5   **if** $Cache(\varphi)$ *is known* **then return** $sym(Cache(\varphi))$
6   **if** *close to memory limit* **then**
7       $v \leftarrow Cache(\varphi)$
8       $Z = \frac{M \cdot Z + (N - M) \cdot \widehat{Z}(v)}{N}$
9       $M \leftarrow N$
10   Clear $Cache$
11   **end**
12 **end**
13 $v \leftarrow Cache(\varphi)$
14 **return** $\frac{M \cdot Z + (N - M) \cdot \widehat{Z}(v)}{N}$

We estimate the hardness of the input CNF formula in line 3 in MICROKC, and if it is easy, we will obtain a known node by calling an exact model counter, ExactMC (Lai, Meel, and Yap 2021) which uses a full CCDD. In lines 4–29, we deal with the case of the initial call of MICROKC on φ. We try to kernelize in lines 5–13. Otherwise, we decompose φ into a set of sub-formulas without shared variables in line 14. In lines 16–17, we deal with the case where φ is decomposable, and call MICROKC recursively for each sub-formula.

Otherwise, we deal with the case where φ is not decomposable in lines 19–27. We introduce a decision node u labeled with a variable x from $Vars(\varphi)$. We estimate the marginal probability of φ over x in line 20 and sample a Boolean value b with this probability in the next line. We remark that the variance of our model counting method depends on the accuracy of the marginal probability estimate, discussed further in Section 5.1. Then we generate the children of u, updating the probability and frequency.

We deal with the case of repeated calls of MICROKC on φ in lines 30–44. The inverse function $Cache^{-1}(v)$ of $Cache$ is used for finding formula φ such that $Cache(\varphi) = v$.

Example 2. *We run* PartialKC *on the formula* $\varphi = (x_1 \vee x_3 \vee x_5 \vee \neg x_7) \wedge (x_4 \vee x_6) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee \neg x_5)$. *For simplicity, we assume that* PICK-GOODVAR *chooses the variable with the minimum subscript and* EASYINSTANCE *returns true when the formula has less than three variables. For the first call of* MICROKC(φ), *the*

Algorithm 2: MICROKC(φ)

1   **if** $\varphi = false$ **then return** $\langle \bot \rangle$
2   **if** $\varphi = true$ **then return** $\langle \top \rangle$
3   **if** EASYINSTANCE(φ) **then return** $\langle \text{ExactMC}(\varphi) \rangle$
4   **if** $Cache(\varphi) = nil$ **then**
5   **if** SHOULDKERNELIZE(φ) **then**
6     $E \leftarrow$ DETECTLITEQU(φ)
7   **if** $||E|| > 0$ **then**
8      $\psi \leftarrow$ CONSTRUCTCORE$(\varphi, \lfloor E \rfloor)$
9      $v \leftarrow$ MICROKC(ψ)
10      $V \leftarrow \{\langle x \leftrightarrow l \rangle \mid x \leftrightarrow l \in \lfloor E \rfloor\}$
11   **return** $Cache(\varphi) \leftarrow \langle \wedge_k, \{v\} \cup V \rangle$
12   **end**
13   **end**
14    $\Psi \leftarrow$ DECOMPOSE(φ)
15   **if** $|\Psi| > 1$ **then**
16     $V \leftarrow \{$MICROKC$(\psi) \mid \psi \in \Psi\}$
17   **return** $Cache(\varphi) \leftarrow \langle \wedge_d, V \rangle$
18   **else**
19     $x \leftarrow$ PICKGOODVAR(φ)
20     $p \leftarrow$ MARGPROB(φ, x)
21     $b \sim Bernoulli(p)$
22   Create a decision node u with $sym(u) = x$
23     $ch_b(u) \leftarrow$ MICROKC$(\varphi[x \mapsto b])$
24     $ch_{1-b}(u) \leftarrow \langle ? \rangle$
25     $p_0(u) \leftarrow 1 - p; p_1(u) \leftarrow p$
26     $f_b(u) \leftarrow 1; f_{1-b}(u) \leftarrow 0$
27   **return** $Cache(\varphi) \leftarrow u$
28   **end**
29   **end**
30 $v \leftarrow Cache(\varphi)$
31 **if** v *is a known node* **then return** v
32 **else if** v *is kernelized* **then**
33   $ch_{core}(v) \leftarrow$ MICROKC$(Cache^{-1}(ch_{core}(v)))$
34 **else if** v *is decomposed* **then**
35   $Ch(v) \leftarrow \{$MICROKC$(Cache^{-1}(w)) \mid w \in Ch(v)\}$
36 **else**
37   $b \sim Bernoulli(p_1(v))$
38   $f_b(v) \leftarrow f_b(v) + 1$
39   $ch_b(v) \leftarrow$ MICROKC$(\varphi[sym(v) \mapsto b])$
40 **end**
41 **if** v *has no unknown descendants* **then**
42   Let c be the model count of v
43   **return** $Cache(\varphi) \leftarrow \langle c \rangle$
44 **else return** $Cache(\varphi) \leftarrow v$

---

condition in line 4 is satisfied. We assume that the marginal probability of φ over x_1 is estimated as 0.5 and 1 is sampled in line 21. Then MICROKC(φ_1) *is recursively called, where* $\varphi_1 = (x_4 \vee x_6) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee x_5) \wedge (x_2 \vee \neg x_5)$. *We kernelize φ_1 as* $\varphi_2 = (x_4 \vee x_6) \wedge (\neg x_2 \vee x_4)$ *and then invoke* MICROKC(φ_2). *Similarly, the condition in 4 is satisfied. We assume that the estimated marginal probability of φ_2 over x_2 is 0.4 and 0 is sampled in line 21. Then* MICROKC(φ_3) *is recursively called, where $\varphi_3 = x_4 \vee x_6$, and we call ExactMC(φ_3) to get a count 96. Finally, the partial CCDD in Figure 2a is returned. For the second call of* MICROKC(φ), *the condition in line 4 is not satisfied. We get the stored marginal probability of φ over x_1 and assume that 0 is sampled in line 37. Then* MICROKC *is recursively*

called on $\varphi_4 = (x_3 \vee x_5 \vee \neg x_7) \wedge (x_4 \vee x_6) \wedge (\neg x_2 \vee x_4)$, and then MICROKC is recursively called on $\varphi_5 = x_3 \vee x_5 \vee \neg x_7$ and $\varphi_2 = (x_4 \vee x_6) \wedge (\neg x_2 \vee x_4)$. Finally, we generate the partial CCDD in Figure 2b.

Proposition 2. *Given a CNF formula φ and a timeout setting t, the output of PartialKC(φ, t) is an unbiased estimate of the model count of φ.*

Proof. If PartialKC restarts the compilation, it finally returns the average estimate. Thus, we just need to show when the compilation is not restarted, PartialKC outputs an unbiased estimate of the true count.

Given the input φ, we denote all of the inputs of recursively calls of MICROKC as a sequence $S = (\varphi_1, \ldots, \varphi_n = \varphi)$ in a bottom-up way in the call of PartialKC(φ, t). Let N be the final number of calls of MICROKC on φ, and let $Z(\varphi_i)$ be the true model count φ_i.

We first prove the case $N = 1$ by induction with the hypothesis that the call of MICROKC(ψ) returns an unbiased estimate of the model count of ψ if $|Vars(\psi)| < |Vars(\varphi_n)|$. Thus, the calls of MICROKC on $\varphi_1, \ldots, \varphi_{n-1}$ return unbiased estimates of the model counts, and the results are stored in $Cache$. We denote the output of MICROKC(φ_i) by u_i. The cases when MICROKC returns a leaf node is obvious. We proceed by a case analysis (the equations about true counts can be found in the proofs of Propositions 1–2 in (Lai, Meel, and Yap 2021)):

(a) $E \neq \emptyset$ in line 6: The input of the recursive call is φ_{n-1}. According to the induction hypothesis, $\mathbb{E}[\widehat{Z}(v)] = Z(\varphi_{n-1})$. Thus, $\mathbb{E}[\widehat{Z}(u_n)] = \dfrac{Z(\varphi_{n-1})}{2^{|Ch(u)|-1}} = Z(\varphi_n)$.

(b) $|\Psi| = m > 1$ in line 15: The input of the recursive calls are $\varphi_{n-m}, \ldots, \varphi_{n-1}$. According to the induction hypothesis, $\mathbb{E}[\widehat{Z}(u_i)] = Z(\varphi_i)$ ($n - m \leq i \leq n - 1$). Due to the conditional independence, $\mathbb{E}[\widehat{Z}(u_n)] = c^{-1} \cdot \prod_{i=n-m}^{n-1} \mathbb{E}[\widehat{Z}(u_i)] = c^{-1} \cdot \prod_{i=n-m}^{n-1} Z(\varphi_i) = Z(\varphi_n)$.

(c) $|\Psi| = 1$ in line 15. The input of the recursive call is $\varphi_{n-1} = \varphi_n[x \mapsto false]$ with the probability $p_0(u)$, and $\varphi_{n-1} = \varphi_n[x \mapsto true]$ with the probability $p_1(u)$. $Z(\varphi_n) = \frac{1}{2} \cdot (Z(\varphi_n[x \mapsto false]) + Z(\varphi_n[x \mapsto true]))$.

$$
\begin{aligned}
\mathbb{E}[\widehat{Z}(u)] &= \frac{\mathbb{E}[\widehat{Z}(ch_0(u))] \cdot f_0(u)}{2p_0(u) \cdot (f_0(u) + f_1(u))} \cdot p_0(u) + \\
&\quad \frac{\mathbb{E}[\widehat{Z}(ch_1(u))] \cdot f_1(u)}{2p_1(u) \cdot (f_0(u) + f_1(u))} \cdot p_1(u) \\
&= \frac{1}{2} \cdot \left(\mathbb{E}[\widehat{Z}(ch_0(u))] + \mathbb{E}[\widehat{Z}(ch_1(u))] \right) \\
&= Z(\varphi_n).
\end{aligned}
$$

We can also prove the case $N > 1$ by induction. We call MICROKC on the same formula at most N times; in other words, each formula in S at most N times. It is assumed that the M-th call of MICROKC(ψ) returns an unbiased estimate of the model count of ψ if $|Vars(\psi)| < |Vars(\varphi_n)|$ or $M < N$. Then the proof for this case is similar to the case $N = 1$ except the decision case and the known case. When it returns a known node in lines 42–43, we can get

the exact count. Now we prove the decision case. For convenience, we denote the returned node by u. The value of $f(u)$ is independent from what are the children of u. We have $\mathbb{E}[\frac{f_0(u)}{N}] = p_0(u)$ and $\mathbb{E}[\frac{f_1(u)}{N}] = p_1(u)$. From the decision case of Eq. (2), we have the following equation:

$$
\begin{aligned}
\mathbb{E}[\widehat{Z}(u)] &= \frac{\mathbb{E}[\widehat{Z}(ch_0(u))] \cdot \mathbb{E}[f_0(u)]}{2p_0(u) \cdot N} + \\
&\quad \frac{\mathbb{E}[\widehat{Z}(ch_1(u))] \cdot \mathbb{E}[f_1(v)]}{2p_1(u) \cdot N} + \\
&= \frac{1}{2} \cdot \left(\mathbb{E}[\widehat{Z}(ch_0(u))] + \mathbb{E}[\widehat{Z}(ch_1(u))] \right) \\
&= Z(\varphi_n).
\end{aligned}
$$

$\square$

As we get an unbiased estimate of the exact count, probabilistic lower bounds can be obtained by Markov's inequality (Wei and Selman 2005; Gomes et al. 2007).

5.1 Implementation

We implemented PartialKC in the toolbox KCBox.[1] For the details of functions DECOMPOSE, PICKGOODVAR, SHOULDKERNELIZE, CONSTRUCTCORE and ExactMC, we refer the reader to ExactMC (Lai, Meel, and Yap 2021). In the function EASYINSTANCE, we rely on the number of variables as a proxy for the hardness of a formula, in particular at each level of recursion, we classify a formula φ to be easy if $|Vars(\varphi)| \leq easy\_bound$. We define $easy\_bound$ as the minimum of 512 and $easy\_param$:

$$
easy\_param = \begin{cases} \frac{3}{4} \cdot \#NonUnitVars & width \leq 32 \\ \frac{2}{3} \cdot \#NonUnitVars & 32 < width \leq 64 \\ \frac{1}{2} \cdot \#NonUnitVars & otherwise \end{cases}
$$

where $\#NonUnitVars$ is the number of variables appearing in the non-unit clauses of the original formula, and $width$ is the minfill treewidth (Darwiche 2009).

MICROKC can be seen as a sampling procedure enhanced with KC technologies, and the variance of the estimated count depends on three main factors. First, the variance depends on the quality of predicting marginal probability. Second, the variance of a single calling of MICROKC depends on the number of sampling Boolean values in lines 21 and 37. The fewer the samples from the Bernoulli distributions, the smaller the variance. Finally, the variance depends on the number of MICROKC calls when fixing their total time. We sketch four KC technologies for reducing the variance.

The first technology is how to implement MARGPROB. Without consideration of the dynamic decomposition and kernelization, each call of MICROKC can be seen as a process of importance sampling, where the resulting partial CCDD is treated as the proposal distribution. Similar to importance sampling, it is easy to see that the variance of using PartialKC to estimate model count depends on the quality of estimating the marginal probability in line 20. If the estimated marginal probability equals the true one, PartialKC

---

[1]KCBox is available at: https://github.com/meelgroup/KCBox

Algorithm 3: PROJECTEDKC(φ, P)

1 **if** $\varphi = false$ **then return** $\langle \bot \rangle$
2 **if** $\varphi = true$ **then return** $\langle \top \rangle$
3 **if** $ProjCache(\varphi) \neq nil$ **then return** $ProjCache(\varphi)$
4 $\Psi \leftarrow$ DECOMPOSE(φ)
5 **if** $|\Psi| > 1$ **then**
6 $W \leftarrow \{$PROJECTEDKC(ψ, P) $\mid \psi \in \Psi\}$
7   **return** $ProjCache(\varphi) \leftarrow \langle \wedge_d, W \rangle$
8 **else**
9   **if** $Vars(\varphi) \cap P = \emptyset$ **then**
10   **if** φ **is** *satisfiable* **then return** $\langle \top \rangle$
11   **else return** $\langle \bot \rangle$
12   **end**
13 $x \leftarrow$ PICKGOODVAR(φ, P)
14 $w_0 \leftarrow$ PROJECTEDKC($\varphi[x \mapsto false], P$)
15 $w_1 \leftarrow$ PROJECTEDKC($\varphi[x \mapsto true], P$)
16   **return** $ProjCache(\varphi) \leftarrow \langle x, w_0, w_1 \rangle$
17 **end**

will yield an optimal (zero variance) estimate. In principle, the exact marginal probability can be computed from an equivalent full CCDD. However, this full compilation is impractical. Rather, MARGPROB estimates the marginal probability via compiling the formula into full CCDDs on a small set P of projected variables by PROJECTEDKC in Algorithm 3. In detail, we first perform two projected compilations by calling PROJECTEDKC($\varphi[x \mapsto false], P$) and PROJECTEDKC($\varphi[x \mapsto true], P$) with the outputs u and v, and then use the compiled results to compute the marginal probability that is equal to $\frac{Z(v)}{Z(u)+Z(v)}$.

The second technology is dynamic decomposition in line 14. We employ a SAT solver to compute the implied literals of a formula, and use these implied literals to simplify the formula. Then we decompose the residual formula according to the corresponding primal graph. We can reduce the variance based on the following: (a) the sampling is backtrack-free, this remedies the rejection problem of sampling; (b) we reduce the variance by sampling from a subset of the variables, also known as Rao-Blackwellization (Casella and Robert 1997), and its effect is strengthened by decomposition; (c) after decomposing, more virtual samples can be provided in contrast to the original samples (Gogate and Dechter 2012).

The third technology is kernelization, which can simplify a CNF formula. After kernelization, we can reduce the number of sampling Boolean values in lines 21 and 37. It can also save time for computing implied literals in the dynamic decomposition as kernelization often simplifies the formula.

The fourth technology is the component caching implemented using hash table $Cache$. In different calls of MICROKC, the same sub-formula may need to be processed multiple times. Component caching, can save the time of dynamic decomposition, and accelerate the sampling. It can also reduce the variance by merging the calls to MICROKC on the same sub-formula. Consider Example 2 again. We call MICROKC on φ_2 twice, and obtain a known node. The corresponding variance is then smaller than that of a single

call of MICROKC. Our implementation uses the component caching scheme in sharpSAT (Thurley 2006).

6 Experiments

We evaluated PartialKC on a comprehensive set of benchmarks: (i) 1114 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; and (ii) 100 public instances adopted in the Model Counting Competition 2022. We remark that the 1114 instances have been employed in the past to evaluate model counting and knowledge compilation techniques (Lagniez and Marquis 2017; Lai, Liu, and Yin 2017; Fremont, Rabe, and Seshia 2017; Soos and Meel 2019; Lai, Meel, and Yap 2021, 2022). The experiments were run on a cluster (HPC cluster with job queue) 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 5000 seconds and 8GB memory.

We compared exact counters D4 (Lagniez and Marquis 2017), sharpSAT-td (Korhonen and Järvisalo 2021), and ExactMC (Lai, Meel, and Yap 2021), and approximate counters satss (SampleSearch, (Gogate and Dechter 2011)), STS (SearchTreeSampler, (Ermon, Gomes, and Selman 2012)), and the latest version (from the developers) of ApproxMC (Chakraborty, Meel, and Vardi 2013; Soos and Meel 2021) that combines the independent support computation technique, Arjun, with ApproxMC. We remark that both satss and STS are anytime. ApproxMC was run with $\varepsilon = 4$ and $\delta = 0.2$. We used the pre-processing tool B+E (Lagniez, Lonca, and Marquis 2016) for all the instances, which was shown very powerful in model counting. Consistent with recent studies, we excluded the pre-processing time from the solving time for each tool as pre-processed instances were used on all solvers.

Table 1 shows the performance of the seven counters. The results show that PartialKC has the best scalability as it can solve approximately many more instances than the other six tools. We emphasize that there are 123 instances in 1103 that was not solved by D4, sharpSAT-td, and ExactMC. The results also show that PartialKC can get convergence on 933 instances, i.e., it gets the exact counts for those instances. It is surprising that PartialKC, due to the anytime and sampling nature of the algorithm which entails some additional costs, can still outperform state-of-the-art exact counters D4 and sharpSAT-td.

We evaluate the accuracy of PartialKC from two aspects. First, we consider each estimated count that falls into the interval $[(1+\varepsilon)^{-1}Z, (1+\varepsilon)Z]$ of the true count Z. We say an estimate is *qualified* if it falls into this interval. Note that this is only for the instances where the true count is known. We choose $\varepsilon = 4$. If the estimate falls into $[0.2Z, 5Z]$, this comparison considers it the same order of magnitude as the true count. The results show that PartialKC computed the most qualified estimates. We highlight that there are 835 instances where PartialKC converged in one minute of CPU time. This number is much greater than the numbers of qualified

domain (#, #known)	exact			approximate							
	D4	sharpSAT-td	ExactMC	PartialKC			satss		STS		ApproxMC
				#approx	#conv	$\varepsilon = 4$	#approx	$\varepsilon = 4$	#approx	$\varepsilon = 4$	
BayesianNetworks (201, 186)	179	186	186	**195**	186	*186*	18	17	157	148	172
BlastedSMT (200, 183)	162	163	169	**200**	168	177	150	129	**200**	178	*197*
Circuit (56, 51)	50	50	51	**54**	50	*50*	15	13	50	44	46
Configuration (35, 35)	34	32	31	**35**	29	*33*	33	31	35	13	15
InductiveInference (41, 23)	18	21	22	37	21	*23*	40	19	**41**	*23*	21
ModelChecking (78, 78)	75	**78**	**78**	**78**	77	*78*	11	7	10	8	***78***
Planning (243, 219)	208	215	213	**240**	209	*213*	169	126	238	103	147
ProgramSynthesis (220, 119)	91	78	108	**135**	100	113	81	64	125	108	*115*
QIF (40, 33)	29	30	32	**40**	24	32	15	13	27	25	***40***
MC2022_public (100, 85)	72	76	76	**89**	69	*79*	55	45	88	77	61
Total (1214, 1012)	918	929	966	**1103**	933	*984*	585	463	971	727	892

Table 1: Comparative counting performance between D4, sharpSAT-td, ExactMC, PartialKC, satss, STS, and ApproxMC. Each cell below D4, sharpSAT-td, ExactMC, #approx of PartialKC, satss, STS, and the ApproxMC column refers to the number of solved instances, and the maximum numbers are marked in bold. #known denotes the number of instances that solved by D4, sharpSAT-td, or ExactMC with a longer timeout of four hours. #conv refers to the number of instances where convergence was reached. $\varepsilon = 4$ refers to the number of instances where the reported count falls into $[(1 + \varepsilon)^{-1}Z, (1 + \varepsilon)Z]$ with the true count Z. We remark that each count reported by ApproxMC falled into this interval. The maximum numbers of estimates falling into the interval are marked in italics.

(a) inductive-inference_ii32c2

(b) ModelChecking_bmc-ibm-7

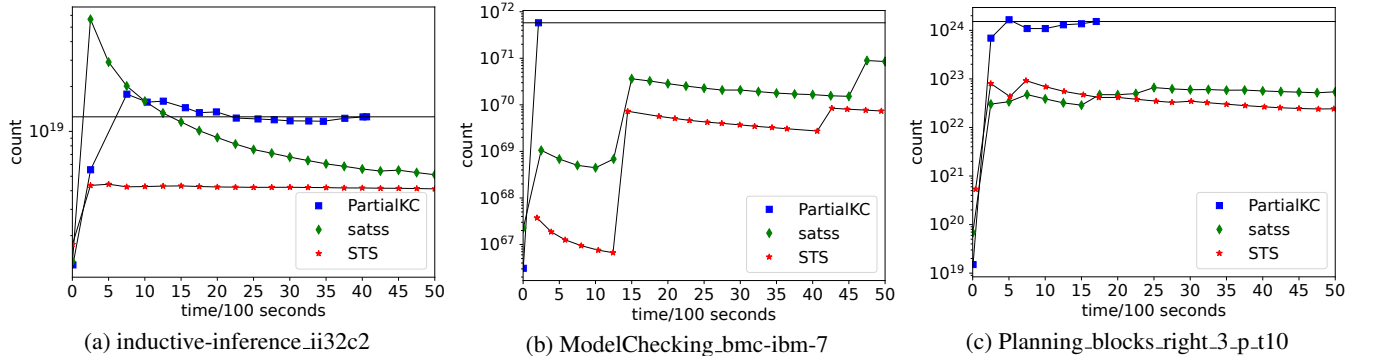(c) Planning_blocks_right_3_p_t10

Figure 3: Convergence results for PartialKC, SampleSearch and STS on three instances with exact counts depicted by straight horizontal lines. (Best viewed in color)

solved instances of satss and STS. We remark that convergence is stricter than the requirement that an estimate falls in $[(1+\varepsilon)^{-1}Z, (1+\varepsilon)Z]$. Second, we compare the accuracy of PartialKC, satss, and STS in terms of the average log relative error (Gogate and Dechter 2012). Given an estimated count $\widehat{Z}$, the log-relative error is defined as $\left| \frac{\log(Z) - \log(\widehat{Z})}{\log(Z)} \right|$. For fairness, we only consider the instances that are approximately solved by all of PartialKC, satss, and STS. Our results show that the average log relative errors of PartialKC, satss, and STS are 0.0075, 0.0081, and 0.0677, respectively; that is, PartialKC has the best accuracy.

The convergence results of PartialKC shows that it can give very accurate estimates. For instance, if we evaluate with $\varepsilon = 0.1$ rather than $\varepsilon = 4$ in Table 1, PartialKC still can get qualified estimates on 940 (44 fewer) instances, while the two other anytime tools satss and STS give qualified estimates on 337 (126 fewer) and 575 (157 fewer) instances, re-

spectively. We compare the convergence of PartialKC, satss, and STS further on three selected instances depicted in Figure 3. The results show that PartialKC converges quickly to the exact counts, while neither satss nor STS converge.

7 Conclusion

Model counting is intrinsically hard, hence, approximate techniques have been developed to scale beyond what exact counters can do. We propose a new approximate counter, PartialKC, based on our partial CCDD KC form. It is anytime and able to converge to the exact count. We present many techniques exploiting partial CCDD to achieve better scalability. Our experimental results show that PartialKC is more accurate than existing anytime approximate counters satss and STS, and scales better. Surprisingly, PartialKC is able to outperform recent state-of-art exact counters by reaching convergence on many instances.

Acknowledgments

References

Baluta, T.; Shen, S.; Shinde, S.; Meel, K. S.; and Saxena, P. 2019. Quantitative Verification of Neural Networks and Its Security Applications. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*, 1249–1264.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8): 677–691.

Casella, G.; and Robert, C. P. 1997. Rao-Blackwellisation of sampling schemes. *Biometrika*, 83(1): 81–094.

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*, 3569–3576.

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

Choi, A.; and Darwiche, A. 2013. Dynamic Minimization of Sentential Decision Diagrams. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI-13)*, 187–194.

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

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

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 819–826.

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

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

Ermon, S.; Gomes, C. P.; and Selman, B. 2012. Uniform Solution Sampling Using a Constraint Solver As an Oracle. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, 255–264.

Fierens, D.; den Broeck, G. V.; Renkens, J.; Shterionov, D. S.; Gutmann, B.; Thon, I.; Janssens, G.; and Raedt, L. D. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *TPLP*, 15(3): 358–401.

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.

Friedman, T.; and den Broeck, G. V. 2018. Approximate Knowledge Compilation by Online Collapsed Importance Sampling. In *Proc. of NeurIPS*, 8035–8045.

Gogate, V.; and Dechter, R. 2011. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence*, 175: 694–729.

Gogate, V.; and Dechter, R. 2012. Importance sampling-based estimation over AND/OR search spaces for graphical models. *Artificial Intelligence*, 184-185: 38–77.

Gomes, C. P.; Hoffmann, J.; Sabharwal, A.; and Selman, B. 2007. From Sampling to Model Counting. In Veloso, M. M., ed., *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2293–2299.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2006. Model Counting: A New Strategy for Obtaining Good Bounds. In *Proc. of AAAI*, 54–61.

Huang, J.; and Darwiche, A. 2004. Using DPLL for efficient OBDD construction. In *Proc. of SAT*, 157–172.

Korhonen, T.; and Järvisalo, M. 2021. Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, 8:1–8:11.

Lagniez, J.; Lonca, E.; and Marquis, P. 2016. Improving Model Counting by Leveraging Definability. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)*, 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 Yin, M. 2017. New Canonical Representations by Augmenting OBDDs with Conjunctive Decomposition. *Journal of Artificial Intelligence Research*, 58: 453–521.

Lai, Y.; Meel, K. S.; and Yap, R. H. C. 2021. The Power of Literal Equivalence in Model Counting. In *Proceedings of Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI-21)*, 3851–3859. AAAI Press.

Lai, Y.; Meel, K. S.; and Yap, R. H. C. 2022. CCDD: A Tractable Representation for Model Counting and Uniform Sampling. *CoRR*, abs/2202.10025.

Mateescu, R.; Dechter, R.; and Marinescu, R. 2008. AND/OR Multi-Valued Decision Diagrams (AOMDDs) for Graphical Models. *Journal of Artificial Intelligence Research*, 33: 465–519.

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.

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

Sashittal, P.; and El-Kebir, M. 2019. SharpTNI: Counting and Sampling Parsimonious Transmission Networks under a Weak Bottleneck. In *Proc. of RECOMB Comparative Genomics*.

Somenzi, F. 2015. CUDD: CU decision diagram package - release 3.0.0. Technical report, University of Colorado at Boulder.

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*, 463–484.

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*, 1592–1599.

Soos, M.; and Meel, K. S. 2021. Arjun: An Efficient Independent Support Computation Technique and its Applications to Counting and Sampling. *CoRR*, abs/2110.09026.

Stockmeyer, L. J. 1983. The Complexity of Approximate Counting. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, 118–126.

Thurley, M. 2006. sharpSAT — Counting Models with Advanced Component Caching and Implicit BCP. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, 424–429.

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

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

Van den Broeck, G.; and Suciu, D. 2017. Query Processing on Probabilistic Data: A Survey. *Foundations and Trends in Databases*, 7(3-4): 197–341.

Venturini, A.; and Provan, G. M. 2008. Incremental Algorithms for Approximate Compilation. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 1495–1498.

Wei, W.; and Selman, B. 2005. A New Approach to Model Counting. In *Proc. of SAT*, 324–339.