Chapter 26

Approximate Model Counting

Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi

26.1. Introduction

Model counting, or counting the solutions to a set of constraints, is a fundamental computational problem with applications in diverse areas spanning probabilistic inference, network reliability estimation, statistical physics, explainable AI, security vulnerability analysis, and the like [Rot96, DH07, XCD12, ABB15, DOMPV17, LSS17, BEH$^+$18, ZQRZ18, NSM$^+$19, BSS$^+$19]. While exact counting is known to be computationally hard (see [GSS20] for more details on exact counting), a large class of applications can benefit even if approximate model counts are available. Approximate counting algorithms have therefore been studied in great depth by both theoreticians and practitioners. In this chapter, we discuss some of the key ideas behind modern approximate model counters for propositional constraints.

We begin with some notation and terminology. Let φ be a propositional formula. We use $|\varphi|$ to denote the size of φ presented as a string to a Turing machine, and call the set of all variables in φ the *support* of φ, or $\mathsf{Sup}(\varphi)$. Following standard terminology, a variable or its negation is called a *literal*, a *clause* is a disjunction of literals, and a *cube* is a conjunction of literals. We say φ is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. Similarly, we say φ is in *disjunctive normal form* (DNF) if it is a disjunction of cubes. A satisfying assignment, solution, or model of φ is an assignment of truth values to all variables in $\mathsf{Sup}(\varphi)$ such that φ evaluates to True. We use $\mathsf{Sol}(\varphi)$ to denote the set of all models of φ. The *model counting* problem asks: *Given φ, find* $|\mathsf{Sol}(\varphi)|$.

26.1.1. A historical perspective

Approximate model counting techniques, at least for CNF formulas, have mostly evolved hand-in-hand with techniques for exact model counting, except in the last decade or so, when a few innovations triggered a large body of work in the broad area of approximate counting and sampling. The real value-addition of approximate counters has always been in cases that lie beyond the reach of exact counters, given computational resource constraints. Scalability has therefore been a key consideration throughout the history of the development of approximate

model counters. It was widely believed until recently (circa 2006) that this scalability can be achieved only by weakening or sacrificing approximation guarantees. As a consequence, most applications used approximate counting techniques with weak or no guarantees at all. An example of such usage is that of Monte Carlo estimators for model counting where the number of Monte Carlo steps is truncated heuristically before the theory-mandated mixing of states happen [WS05]. Fortunately, work done over the last decade has convincingly demonstrated that by striking a fine balance between theoretical and practical considerations, it is indeed possible to design approximate model counters that provide strong guarantees while also scaling to problem instances that are orders of magnitude larger than what was possible earlier. To get a sense of where we stand in 2019, state-of-the-art approximate model counters can solve problem instances with half a million propositional variables in less than 2 hours on a single high-end computing node, yielding estimates that are provably within a factor of 1.8 of the true count with at least 99% confidence [SM19].

Some of the key theoretical results pertaining to approximate counting were already known as early as the mid 1980s [Sto83, JVV86]. In a seminal paper [Sto83], Stockmeyer used universal hash functions [CW77] to solve the approximate counting problem using polynomially many NP oracle calls. While theoretically elegant, the proposed technique suffered from the requirement of a prohibitively large number of invocations of NP oracle to perform symbolic reasoning on formulas with very large numbers of variables. Jerrum, Valiant, and Vazirani demonstrated inter-reducibility of approximate counting and almost-uniform sampling in [JVV86]. Almost a decade later, Bellare, Goldreich, and Petrank reported an almost uniform generator of models using hash-functions of high degree of universality [BGP00]. Technically, this could be used to approximately count models using Jerrum et al's result [JVV86]. Unfortunately, our attempt in this direction [CMV13a] showed that this approach did not scale beyond formulas with more than 20 propositional variables. The major stumbling block was the requirement of inverting a hash function with a high degree of universality.

After a long period of relatively incremental improvements in approximate counting techniques, in 2006, Gomes, Sabharwal and Selman [GSS06] reported a parameterized approximate counting algorithm that used random parity constraints as universal hash functions and harnessed the power of a state-of-the-art CNF satisfiability solver in the backend. They showed that if the right combination of parameters was chosen by the user (this is not always easy), this approach could solve problems much larger than what was possible earlier, and yet provide guarantees on the quality of the estimate. Subsequently, this work was developed further in [GHSS07b, GHSS07a]. In 2013, two independent papers [CMV13b, EGSS13b] reported highly scalable approximate counting algorithms in two different settings, guaranteeing $(1 + \varepsilon)$-factor approximation with confidence at least $1 - \delta$ for any user-provided values of ε and δ. The dependence on the user to provide the right combination of parameters was removed in these algorithms and their implementations benefitted from the then newly-proposed SAT solver CryptoMiniSat that could handle parity constraints natively [SNC09]. Since then, there has been a large body of work in this area, as detailed later in

the chapter.

Interestingly, the history of development of approximate model counters for DNF formulas followed a different trajectory. As early as 1983, Karp and Luby showed that it is possible to approximate the model count of DNF formulas efficiently and with strong guarantees using Monte Carlo sampling [KL83]. While there have been several follow-up works that built and improved on this algorithm, the success of these techniques could not be lifted to the case of CNF formulas due to key steps that exploit the fact that the input formula is in DNF. Monte Carlo techniques have dominated the landscape of approximate counters for DNF formulas since Karp and Luby's early work. It is only recently that an alternative approach using universal hash functions has been found to achieve the same guarantees and the same asymptotic time complexity [CMV16, MSV17, MSV19] as the best-known Monte Carlo algorithm. As observed, however, in [MSV19], the empirically observed runtime of approximate DNF counting algorithms is much more nuanced and hard to predict.

26.1.2. An overview of the complexity landscape

Exact model counting, and even some variants of approximate counting are computationally hard in a formal sense. To appreciate this better, we review a few concepts from complexity theory. The reader is assumed to be familiar with the notions of Turing machines, as well as the complexity classes P, NP, coNP and their generalization to the polynomial hierarchy (PH). A detailed exposition on these can be found in [AB09]. Given an instance of a problem in NP, the corresponding counting problem asks how many accepting paths exist in the non-deterministic Turing machine that solves the problem. The complexity class $\#\mathsf{P}$ is defined to be the set of counting problems associated with all decision problems in NP. Toda showed that a single invocation of a $\#\mathsf{P}$-oracle suffices to solve every problem in the polynomial hierarchy efficiently [Tod89]. Formally, $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$, where $\mathcal{C}_1^{\mathcal{C}_2}$ denotes the class of problems that can be solved by a Turing machine for a problem in $\mathcal{C}_1$ when equipped with an oracle for a complete problem in $\mathcal{C}_2$. A consequence of Toda's theorem is that problems in $\#\mathsf{P}$ (viz. exact model counting) are likely much harder than those in PH.

A *probabilistic Turing machine* is a non-deterministic Turing machine with specially designated "coin-tossing" states. When the machine reaches one of these states, it can choose between alternative transitions based on a probability distribution of coin tosses. For our purposes, the probability distribution is assumed to be uniform. Computing with probabilistic Turing machines gives rise to probabilistic complexity classes like RP, BPP etc. The reader is referred to [AB09] for more details on these classes. In a functional problem like model counting, for every input x, we seek an answer $f(x) \in \mathbb{N}$ of size polynomial in $|x|$. A *fully polynomial randomized approximation scheme* (FPRAS) for such a problem is a probabilistic Turing machine that takes as input x and a parameter ε (> 0). It then generates an estimate c that lies within $f(x) \cdot (1+\varepsilon)^{-1}$ and $f(x) \cdot (1+\varepsilon)$ with probability strictly greater than $\frac{1}{2}$, while taking time polynomial in $|x|$ and $\frac{1}{\varepsilon}$. Given an FPRAS for a functional problem, the probability of generating an estimate outside the tolerance window can be reduced below any desired confidence

threshold δ ($0 < \delta \leq 1$) by running the FPRAS $\mathcal{O}\left(\log \frac{1}{\delta}\right)$ times independently and by choosing the median estimate.

Valiant showed that exact model counting is #P-complete for both CNF and DNF formulas [Val79]. Therefore, it is unlikely that any efficient algorithm exists for exact model counting. This has motivated significant research on different variants of approximate model counting over the past four decades. To have a unified view of these variants, we introduce some notation. Let Z be a random variable with a specified probability distribution. Let $\bowtie \in \{=, \leq, <, >, \geq\}$ be a relational operator and let a be a value in the range of Z. We use $\mathsf{Pr}[Z \bowtie a]$ to denote the probability that $Z \bowtie a$ holds. We also use $\mathsf{E}[Z]$ and $\mathsf{Var}[Z]$ to denote the expectation and variance, respectively, of Z. An approximate model counter takes as input a formula φ (along with possibly tolerance and confidence parameters) and returns an estimate c that approximates $|\mathsf{Sol}(\varphi)|$. Depending on whether the algorithm is deterministic or randomized, and depending on the type of approximation used, we can identify several variants of approximate model counters. In the following discussion, unless specified otherwise, ε (> 0) represents a (additive or multiplicative) tolerance used in approximating $|\mathsf{Sol}(\varphi)|$, and δ ($0 < \delta \leq 1$) represents a confidence bound for a randomized algorithm.

A *deterministic additive approximate model counter* takes a formula φ and an additive tolerance ε (> 0) as inputs, and returns an estimate c (≥ 0) such that $(|\mathsf{Sol}(\varphi)| - \varepsilon) \leq c \leq (|\mathsf{Sol}(\varphi)| + \varepsilon)$. A *probabilistic additive approximate model counter* takes an additional confidence parameter δ ($0 < \delta \leq 1$) as input, and finds a random estimate c (≥ 0) such that $\mathsf{Pr}\left[(|\mathsf{Sol}(\varphi)| - \varepsilon) \leq c \leq (|\mathsf{Sol}(\varphi)| + \varepsilon)\right] \geq 1 - \delta$. Unfortunately, if $\varepsilon \leq 2^{\frac{|\mathsf{Sup}(\varphi)|}{2} - 2}$, both versions of additive approximate model counting are computationally hard. Specifically, the deterministic version continues to be #P-complete, while the probabilistic version lies beyond PH, unless the entire PH collapses [CMV19]. Thus, additive approximations do not simplify the problem of counting from a worst-case complexity perspective, and we choose not to elaborate further on this class of counters. It is worth mentioning, however, that approximate counters with additive tolerance have been used in some applications, viz. [SVP+16] and [FHO13].

Counters with multiplicative approximations have been much more extensively studied. A *deterministic multiplicative approximate model counter* takes as inputs a formula φ and a multiplicative tolerance ε (> 0) and returns an estimate c (≥ 0) such that $|\mathsf{Sol}(\varphi)| \cdot (1 + \varepsilon)^{-1} \leq c \leq |\mathsf{Sol}(\varphi)| \cdot (1 + \varepsilon)$. A *probabilisitc multiplicative approximate model counter*, also called a *probably approximately correct (PAC)* model counter, takes an additional confidence parameter δ ($0 < \delta \leq 1$) as input and returns a random estimate c (≥ 0) such that $\mathsf{Pr}\left[|\mathsf{Sol}(\varphi)| \cdot (1 + \varepsilon)^{-1} \leq c \leq |\mathsf{Sol}(\varphi)| \cdot (1 + \varepsilon)\right] \geq 1 - \delta$. In a seminal paper, Stockmeyer showed that approximate counting with multiplicative tolerance can be solved by a deterministic algorithm that makes polynomially many calls to a Σ_2^{P} oracle [Sto83]. Therefore, practically efficient deterministic algorithms for approximate counting can be designed if we have access to Σ_2^{P} solvers (also called *2QBF* solvers) that can solve large problem instances efficiently in practice. Unfortunately, despite significant advances in 2QBF solving, this has not yet yielded practically efficient deterministic approximate counters yet. Building on Stockmeyer's work, Jerrum et al. [JVV86] showed that approximate counting

with PAC guarantees can be achieved by a probabilistic Turing machine in polynomial time with access to an NP oracle. Given the spectacular improvement in performance of propositional SAT solvers over the past few decades, research in practically useful approximate model counting has largely focused on variants of approximate counters with PAC guarantees.

The variants of approximate counting discussed above require the estimate to lie within a specified tolerance window on *either* side of the exact count. The required approximation guarantees are therefore two-sided. In some applications this may not be needed, and one-sided approximation guarantees (viz. $c \geq |\mathsf{Sol}(\varphi)| \cdot (1+\varepsilon)^{-1}$) may be sufficient. Therefore, approximate counting with one-sided guarantees has also been studied in the literature. In this chapter, however, we focus on approximate counters with two-sided guarantees, since they provide the strongest approximation guarantees while still scaling to large problem instances in practice. The interested reader is referred to [GSS20] for more details on counters with one-sided approximation guarantees and guarantee-less approximate counters.

A brief examination of the differences between approximate model counting for CNF and DNF formulas is in order here. It was Karp and Luby [KL83] who first showed that DNF model counting admits an FPRAS. Their algorithm and analysis were subsequently improved in [KLM89]. Recently, it was re-proved in [CMV16, MSV17] that DNF counting admits an FPRAS using techniques different from those used in [KL83, KLM89]. Unfortunately, as observed in [KL83, KL85], CNF model counting cannot admit an FPRAS unless $\mathsf{NP} = \mathsf{RP}$ – a long-standing unresolved question in complexity theory. The best we know at present is that approximate counting for CNF formulas with PAC guarantees can be solved by a probabilistic Turing machine *with access to an* NP *oracle* in time polynomial in $|\varphi|$, $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$. Therefore, approximate model counting for CNF and DNF formulas present different sets of challenges and we discuss them separately.

The problems of approximate counting with PAC guarantees and *almost uniform generation* of models are intimately related to each other. Given φ and a tolerance ε (≥ 0), let $\alpha(\varphi)$ be a real number in $(0, 1]$ that depends only on φ. An almost uniform generator is a probabilistic Turing machine that takes as inputs a formula φ and a tolerance ε (> 0), and outputs a random assignment π to variables in $\mathsf{Sup}(\varphi)$ such that $\alpha(\varphi) \cdot (1+\varepsilon)^{-1} \leq \mathsf{Pr}[\pi \text{ is output}] \leq \alpha(\varphi) \cdot (1+\varepsilon)$ if π is a solution of φ, and $\mathsf{Pr}[\pi \text{ is output}] = 0$ otherwise. Jerrum et al. [JVV86] showed that approximate counting with PAC guarantees and almost uniform generation are polynomially inter-reducible. Thus, an almost uniform generator can be used as a sub-routine to obtain a approximate counter for φ and vice versa. Furthermore, the time-complexity of each problem is within a polynomial factor of that of the other. It turns out, however, that all practically efficient almost uniform generators developed so far already use an approximate counter (or some variant of it) internally in an intermediate step. Hence, using such a sampler to obtain an approximate counter again is not very meaningful.

26.2. Approximate Model Counting for CNF

In this section, we discuss algorithms for computing a PAC estimate of $|\mathsf{Sol}(\varphi)|$, where φ is a CNF formula. Significantly, all state-of-the-art algorithms for this problem make use of probabilistic hash functions. Therefore, we first present a generic framework for approximate model counting using probabilistic hash functions. We then show how different algorithms developed over the years [Sto83, GSS06, CMV13b, IMMV15, AD16, CMV16, ZCSE16, AT17, AHT18, SM19] can be viewed as instantiations and optimizations of this framework.

26.2.1. Probabilistic hash-based counting

A probabilistic hash function maps elements of a given domain, viz. $\{0,1\}^n$, to random buckets or "cells" from a finite range, viz. $\{0,1\}^i$ where $0 \leq i \leq n$. This is achieved by choosing a hash function uniformly at random from a family of functions, each of which maps $\{0,1\}^n$ to $\{0,1\}^i$. In probabilistic hash-based counting, the solution space of φ is randomly partitioned into approximately equal-sized "small" cells using probabilistic hash functions. Suppose each hash function in a family has a range of 2^i cells, and we choose one of these at random. By carefully designing the hash family, we can ensure that the count of solutions of φ in an arbitrary cell of the induced partitioning of $\mathsf{Sol}(\varphi)$ is an unbiased estimator of $\frac{|\mathsf{Sol}(\varphi)|}{2^i}$ with small variance. Therefore, if we count the solutions of φ in a single cell and scale it by 2^i, we obtain an estimate of $|\mathsf{Sol}(\varphi)|$.

In order to formalize the above intuition, we need some additional notation. Let $\{x_1, \ldots x_n\}$ be the support of φ. Let h_i denote a hash function that maps elements of $\{0,1\}^n$ to $\{0,1\}^i$, where $0 \leq i \leq n$[1]. Let $\mathcal{H}_i$ be a (sub-)family of such hash functions mapping $\{0,1\}^n$ to $\{0,1\}^i$. We use $\mathcal{H}$ to denote the family of hash functions obtained by taking the union of all $\mathcal{H}_i$ for $i \in \{0, \ldots n\}$.

The general structure of a probabilistic hash-based approximate counter is shown in Algorithm 1. The algorithm takes as inputs a formula φ, a tolerance bound ε (> 0) and a confidence bound δ ($0 < \delta \leq 1$). It returns an estimate c such that $\mathsf{Pr}\left[|\mathsf{Sol}(\varphi)| \cdot (1+\varepsilon)^{-1} \leq c \leq |\mathsf{Sol}(\varphi)| \cdot (1+\varepsilon)\right] \geq 1 - \delta$. Let us ignore lines 1 through 4 of Algorithm 1 for now. In line 5, we commit to using a family of hash functions, say $\mathcal{H}$. In general, $\mathcal{H}$ has sub-families $\mathcal{H}_i$ as discussed above. In line 8, we choose $n+1$ hash functions, one from each sub-family $\mathcal{H}_i$ in $\mathcal{H}$. Let these hash functions be $h_i : \{0,1\}^n \to \{0,1\}^i$ for $i \in \{0, \ldots n\}$. Depending on the counter, each h_i may be chosen independently of h_j (for $j \neq i$), or the choice of h_i may be correlated with the choice of h_j. Note that each h_i partitions $\mathsf{Sol}(\varphi)$ into 2^i cells. Furthermore, for every $\mathbf{a} \in \{0,1\}^i$, the set $\mathsf{Sol}(\varphi \wedge h_i^{-1}(\mathbf{a}))$ consists of exactly those solutions of φ that are mapped to the cell $\mathbf{a}$ by h_i.

The core of Algorithm 1 consists of the loop in lines 10–15. This loop searches for the smallest index i in $\{1, \ldots n\}$ such that the partitioning induced by h_i yields "small" cells with only a few solutions of φ per cell, with high probability. Since the choice of a cell to check for the count of solutions is arbitrary, we focus on the cell labeled all 0s, i.e. $\mathbf{0}^i$. The maximum number of solutions in a cell for it to be considered "small" is defined by a parameter thresh that is determined

[1] We assume h_0 maps elements of $\{0,1\}^n$ to a singleton set, chosen to be $\{0\}$.

Algorithm 1 HashCounter$(\varphi, \varepsilon, \delta)$

1: thresh $\leftarrow$ FindSmallCellSize(ε); ▷ *Size threshold for a cell to be considered "small"*
2: smc $\leftarrow$ SaturatingCounter$(\varphi,$ thresh$)$; ▷ *Saturated model count of* φ
3: **if** (smc $\neq \top$) **then return** smc; ▷ *Easy case: Solution space of* φ *itself is "small"*
4: itercount $\leftarrow$ FindRepetitionCount(δ); ▷ *# repetitions to amplify confidence*
5: $\mathcal{H} \leftarrow$ ChooseHashFamily();
6: CountEst $\leftarrow$ EmptyList; ▷ *List of model count estimates from different iterations*
7: **repeat** itercount times
8: $(h_0, \ldots h_n) \leftarrow$ ChooseHashFunctions$(\mathcal{H}, n)$; ▷ n *is # variables in* φ
9: InitializeArray(CellSize$, n)$; ▷ CellSize *is an array of size* $n+1$
▷ CellSize$[i]$ *records (saturated) model count in cell* $\mathbf{0}^i$ *induced by* h_i*;* CellSize$[0] = \top$
10: **repeat** until AllIndicesAssigned(CellSize)
11: $i \leftarrow$ ChooseUnassignedIndex(CellSize); ▷ *Choose* i *s.t.* CellSize$[i] = \perp$
12: CellSize$[i] \leftarrow$ SaturatingCounter$(\varphi \wedge h_i^{-1}(\mathbf{0}^i),$ thresh$)$;
13: **if** (CellSize$[i] <$ thresh $\wedge$ CellSize$[i-1] = \top$) **then** ▷ *Found right "small" cell*
14: Append CellSize$[i] \times 2^i$ to CountEst; ▷ *Current model count estimate*
15: **go to line** 7; ▷ *Repeat to amplify confidence*
16: **return** ComputeMedian(CountEst);

in line 1 of Algorithm 1. In general, thresh depends polynomially on $\frac{1}{\varepsilon}$, where ε is the tolerance bound of our PAC estimate. We assume access to a sub-routine SaturatingCounter that counts the solutions of a formula, but only upto a specified bound. Specifically, SaturatingCounter(ψ, c) returns a special symbol $\top$ (denoting saturation of count) if ψ has c or more solutions. Otherwise, it returns the exact count of solutions of ψ. Thus, SaturatingCounter(ψ, c) always returns a value in $\{0, c-1\} \cup \{\top\}$. We call this a *saturated count* of the solutions of ψ. Algorithm 1 uses SaturatingCounter at two places. In lines 2 through 3, we check if the saturated count of solutions of φ is itself within thresh-1; if so, we return this count directly. Otherwise, we use SaturatingCounter in line 12 to find the saturated count of solutions of φ in cell $\mathbf{0}^i$ induced by h_i for an appropriately chosen $i \in \{1, \ldots n\}$. This count is then recorded in the i^{th} element of an array named CellSize. The sub-routine InitializeArray is used to initialize CellSize prior to starting the iterations of the loop in lines 10–15. Specifically, InitializeArray sets CellSize$[0]$ to $\top$ (since the saturated model count of φ is $\top$ if the check in line 3 fails) and CellSize$[i]$ to $\perp$, representing a value that has not been assigned yet, for $1 \leq i \leq n$.

Recall that the count of solutions of φ in a cell induced by h_i is an unbiased estimator of $\frac{|\mathsf{Sol}(\varphi)|}{2^i}$. Therefore, this count reduces (in expectation) as i increases. In order to find the smallest i where this drops below thresh, we check in line 13 if the count induced by h_i is less than thresh, whereas that induced by h_{i-1} is thresh or more (i.e. saturated at $\top$). If so, we identify i as the smallest index for which the partitioning of $\mathsf{Sol}(\varphi)$ by h_i yields "small" cells. We also record the value of CellSize$[i] \times 2^i$ as the current estimate of $|\mathsf{Sol}(\varphi)|$ in a list of estimates named CountEst. If, however, the check in line 13 fails, we conclude that the cells induced by h_i are either too small (CellSize$[i-1] \neq \top$) or too large (CellSize$[i] = \top$). In this case, we continue the search by choosing another index $i \in \{1, \ldots n\}$ such that CellSize$[i]$ has not been assigned yet, i.e. has the value $\perp$. Different hash-based model counters use different strategies for choosing the next unassigned index. This is represented by an invocation of sub-routine ChooseUnassignedIndex

in line 11 of Algorithm 1.

While the loop in lines 10–15 of Algorithm 1 provides an estimate of $|\mathsf{Sol}(\varphi)|$ within a specified tolerance, the confidence may not be as high as required. The loop in lines 7–15 followed by computation of the median of all estimates in the list CountEst serves to amplify the probability of success to the desired level. Every iteration of this loop is required to be independent of all previous and subsequent iterations. The number of iterations needed to achieve the desired confidence is a linear function of $\log \frac{1}{\delta}$ and is calculated in line 4.

In the following subsections, we discuss how different choices of $\mathcal{H}$ and of the subroutines used in Algorithm 1 affect the performance and guarantees of various probabilistic hash-based counting algorithms. We start with a brief overview of the theoretical tools and techniques necessary to analyze different choices.

26.2.2. Theoretical tools and analysis

Let $\mathsf{Cell}_{\langle\varphi,h_i\rangle}$ denote the random set $\{y \in \mathsf{Sol}(\varphi) \mid h_i(y) = \mathbf{0}^i\}$. Now consider an arbitrary iteration of the loop in lines 7–15 of Algorithm 1. We say that the iteration *fails* if either no estimate of the model count is appended to CountEst in line 14, or if the estimate that is appended does not lie in the interval $\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon}, |\mathsf{Sol}(\varphi)| \cdot (1+\varepsilon)|\right]$. Let IterError denote the event that the iteration fails in the above sense. Furthermore, let T_i denote the event $\left(|\mathsf{Cell}_{\langle\varphi,h_i\rangle}| < \mathsf{thresh}\right)$, and let L_i and U_i denote the events $\left(|\mathsf{Cell}_{\langle\varphi,h_i\rangle}| \times 2^i < \frac{|\mathsf{Sol}(\varphi)|}{(1+\varepsilon)}\right)$ and $\left(|\mathsf{Cell}_{\langle\varphi,h_i\rangle}| > |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right)$, respectively. It now follows from the definition of IterError that $\mathsf{Pr}\left[\mathsf{IterError}\right] = \mathsf{Pr}\left[\bigcap_{j=0}^{n} \overline{T_j} \ \cup \ \bigcup_{i=1}^{n} \left(\overline{T_{i-1}} \cap T_i \cap (L_i \cup U_i)\right)\right]$. A key component of the analysis of Algorithm 1 concerns bounding $\mathsf{Pr}\left[\mathsf{IterError}\right]$ from above by a constant strictly less than $\frac{1}{2}$. Once this bound is established, standard arguments for probability amplification via median of independent random variables can be used to bound $\mathsf{Pr}\left[\mathsf{Error}\right]$, where Error denotes the event that Algorithm 1 doesn't yield a count within the prescribed tolerance bound ε.

To see how probability amplification happens, suppose the loop in lines 7–15 is iterated r independent times. Since we compute the median in line 16, the event Error can happen only if IterError happens for at least $(r/2)+1$ iterations, where we have assumed r to be even for simplicity. For notational simplicity, let ρ ($< \frac{1}{2}$) denote $\mathsf{Pr}\left[\mathsf{IterError}\right]$. Then, $\mathsf{Pr}\left[\mathsf{Error}\right] = \sum_{i=(r/2)+1}^{r} \binom{r}{i} \rho^i (1-\rho)^{r-i}$. Since $\binom{r}{i} \leq \binom{r}{r/2} \leq 2^r$, algebraic simplification yields $\mathsf{Pr}\left[\mathsf{Error}\right] < \left(\frac{1-\rho}{1-2\rho}\right) \times \left(2\sqrt{\rho(1-\rho)}\right)^r$. Therefore, in order to ensure $\mathsf{Pr}\left[\mathsf{Error}\right] \leq \delta$, we need $r \geq C(\rho) \times (\log \frac{1}{\delta} + D(\rho))$, where $C(\rho)$ and $D(\rho)$ depend only on ρ. This is what sub-routine FindRepetitionCount computes in line 4 of Algorithm 1.

To obtain a good upper bound of ρ, i.e. $\mathsf{Pr}\left[\mathsf{IterError}\right]$, we need to find good bounds of probabilities of the events T_i, L_i, and U_i. In general, different techniques are used to compute these bounds, depending on the choice of $\mathcal{H}$, and of sub-routines ChooseHashFamily and ChooseHashFunctions. The following three standard probability bounds are particularly useful in reasoning about $|\mathsf{Cell}_{\langle\varphi,h_i\rangle}|$, where $0 < \beta < 1$, and μ_i and σ_i denote the mean and standard

deviation respectively, of $|\mathsf{Cell}_{\langle\varphi,h_i\rangle}|$.

- (Chebyshev) $\mathsf{Pr}\left[\,|\,|\mathsf{Cell}_{\langle\varphi,h_i\rangle}| - \mu_i\,| \geq \beta\mu_i\right] \leq \frac{(1+\beta)^2\sigma_i^2}{\beta^2\mu_i^2}$
- (Payley-Zygmund) $\mathsf{Pr}\left[|\mathsf{Cell}_{\langle\varphi,h_i\rangle}| \leq \beta\mu_i\right] \leq \frac{\sigma_i^2}{\sigma_i^2+(1-\beta)^2\mu_i^2}$
- (Markov) $\mathsf{Pr}\left[|\mathsf{Cell}_{\langle\varphi,h_i\rangle}| \geq \frac{\mu_i}{\beta}\right] \leq \beta$

The recipe for obtaining good upper bounds of the probabilities of T_i, L_i, and U_i is to use appropriate values of β in the above inequalities, and to use hash-family specific properties to obtain expressions for σ_i^2 and μ_i. Existing techniques typically rewrite the expression for $\mathsf{Pr}\left[\mathsf{IterError}\right]$ so that the above bounds can be used for $\mu_i > 1$. Significantly, the choice of thresh informs the need for bounding either the coefficient of variation, i.e., $\frac{\sigma_i^2}{\mu_i^2}$, or the dispersion index, $\frac{\sigma_i^2}{\mu_i}$, as discussed below.

26.2.3. Choice of threshold

For an algorithm focused on a problem as hard as (approximately) counting the number of solutions of a CNF formula, it is pleasing to note that except the SaturatingCounter query in line 12, Algorithm 1 is fairly simple to implement and execute. Therefore, it is no surprise that practical implementations of the algorithm spend most of their time in SaturatingCounter queries in line 12 [SM19]. A straightforward way of implementing SaturatingCounter with modern SAT solvers is to enumerate solutions one by one, using blocking clauses for already found solutions, until either thresh solutions are generated or there are no more solutions. A more generic approach to implementing SaturatingCounter is described in Section 26.2.6. In all of these approaches, the number of times an underlying SAT solver has to be invoked is linear in thresh. Therefore, it is important to ensure that thresh grows polynomially in n and $\frac{1}{\varepsilon}$.

There is a fundamental tradeoff between thresh and the tolerance bound ε that every approximate counter obtained as an instance of Algorithm 1 must respect. To understand this, suppose a hypothetical algorithm $\mathbf{M}$ computes a multiplicative $(1+\varepsilon)$-approximation of a model count that lies in the interval $[1, 2^n]$. The minimum cardinality of the set of possible output values of $\mathbf{M}$ has an inverse logarithmic relation with $(1+\varepsilon)$. Specifically, in order to minimize the count of output values of $\mathbf{M}$, we must divide the interval $[1, 2^n]$ into non-overlapping sub-intervals such that the ratio of the upper bound to the lower bound of each sub-interval is $(1+\varepsilon)^2$. Since the model count being estimated can be anywhere in $[1, 2^n]$, algorithm $\mathbf{M}$ must have the ability to output at least one value from each of the above sub-intervals. If t denotes the number of sub-intervals, then we must have $\frac{2^n}{\left((1+\varepsilon)^2\right)^t} < 1$, i.e. $t > \frac{n}{2\log(1+\varepsilon)}$. This gives the minimum count of possible estimates that algorithm $\mathbf{M}$ must return. Notice that the estimated count appended to $\mathsf{CountEst}$ in the loop in lines 10–15 of Algorithm 1 can be viewed as $c \times 2^i$ for some $1 \leq c < \mathsf{thresh}$ and $i \in [0, n]$. Therefore, the cardinality of the set of possible values returned by Algorithm 1 is $n \times \mathsf{thresh}$. It follows that $n \times \mathsf{thresh} \geq \frac{n}{2\log_2(1+\varepsilon)}$, i.e., $\mathsf{thresh} \geq \frac{1}{2\log_2(1+\varepsilon)}$.

The lower bound of thresh derived above may not be achievable in a given instantiation of Algorithm 1, and typically larger values of thresh are used. For example, Chakraborty et al [CMV13b, CMV16] have used thresh in $\mathcal{O}(1/\varepsilon^2)$ to ensure a multiplicative $(1+\varepsilon)$-approximation. Ermon et al [EGSS14], Asteris and Dimakis [AD16], and Achlioptas and Theodoropoulos [AT17] have, however, used thresh in $\mathcal{O}(1/\varepsilon)$ for the same purpose. Not surprisingly, the choice of thresh affects the choice of hash functions that can be used in Algorithm HashCounter. Specifically, with $\mathsf{thresh} \in \mathcal{O}(1/\varepsilon^2)$, the dispersion index of $|\mathsf{Cell}_{\langle \varphi, h_i \rangle}|$, i.e. $\frac{\sigma_i^2}{\mu_i}$, must be bounded above by a constant. In contrast, when $\mathsf{thresh} \in \mathcal{O}(1/\varepsilon)$, the coefficient of variation of $|\mathsf{Cell}_{\langle \varphi, h_i \rangle}|$, i.e. $\frac{\sigma_i^2}{\mu_i^2}$, must be bounded by a constant.

26.2.3.1. Multiplicative approximation amplification

The above discussion implies that as ε decreases, thresh must increase if we aspire to obtain a multiplicative $(1+\varepsilon)$-factor approximation. If thresh becomes too large, the running time of SaturatingCounter may become a bottleneck. It is therefore interesting to ask if we can avoid increasing thresh at the cost of increasing the size of the formula fed as input to SaturatingCounter, while still obtaining a $(1+\varepsilon)$-factor approximation. In 1983, Stockmeyer [Sto83] observed that this can indeed be done: one can transform an algorithm that provides a constant-factor approximation into one that gives a multiplicative $(1+\varepsilon)$-approximation. To see how this can be achieved, suppose an algorithm $\mathbf{M}$ computes model counts with a 2-approximation. Given an input formula φ over the set of variables X, suppose we make two copies of the formula with disjoint sets of variables, i.e. $\psi(X,Y) = \varphi(X) \wedge \varphi(Y)$ where $\varphi(Y)$ is the same formula as $\varphi(X)$, but with variables X replaced by a new set of variables Y. It is easy to see that $|\mathsf{Sol}(\psi)| = |\mathsf{Sol}(\varphi)|^2$. Furthermore, if $\frac{|\mathsf{Sol}(\psi)|}{4} \leq c \leq 4|\mathsf{Sol}(\psi)|$, then $\frac{|\mathsf{Sol}(\varphi)|}{2} \leq \sqrt{c} \leq 2|\mathsf{Sol}(\varphi)|$. Thus, we have effectively reduced the constant multiplicative factor in approximating $|\mathsf{Sol}(\varphi)|$. Extending this argument, if we want an $(1+\varepsilon)$-approximation of $|\mathsf{Sol}(\varphi)|$ for small values of ε, we can construct ψ by conjoining $k = \mathcal{O}(1/\varepsilon)$ copies of φ, each with a disjoint set of variables, and compute the k-th root of the estimate of $|\mathsf{Sol}(\psi)|$ returned by algorithm $\mathbf{M}$. Therefore, thresh can be set to a constant when computing an $(1+\varepsilon)$-approximation of $|\mathsf{Sol}(\varphi)|$ if we are willing to invoke Algorithm 1 over a new formula obtained by conjoining $\mathcal{O}(1/\varepsilon)$ copies of φ.

The need for multiple copies of the given formula, however, leads to larger queries, to which SAT solvers are highly sensitive. For classes of formulas not closed under conjunction, the query SaturatingCounter$\big(\psi \wedge h_i^{-1}(\mathbf{0}^i), \mathsf{thresh}\big)$ may not even lie in the same complexity class as SaturatingCounter$\big(\varphi \wedge h_i^{-1}(\mathbf{0}^i), \mathsf{thresh}\big)$. As an example, the class of DNF formulas is not closed under conjunction. If φ is a DNF formula and ψ is a conjunction of DNF formulas, the query SaturatingCounter$\big(\varphi \wedge h_i^{-1}(\mathbf{0}^i), \mathsf{thresh}\big)$ can be answered in polynomial time, while the query SaturatingCounter$\big(\psi \wedge h_i^{-1}(\mathbf{0}^i), \mathsf{thresh}\big)$ is NP-hard in general. This is a crucial consideration for the design of hashing-based FPRAS techniques, which are discussed in detail in Section 26.4.

26.2.4. Choice of hash family

Extending the notation introduced earlier, let $h_i \xleftarrow{R} \mathcal{H}_i$ denote the probability space induced by choosing a function $h_i : \{0,1\}^n \to \{0,1\}^i$ uniformly at random from the sub-family $\mathcal{H}_i$.

Definition 26.2.1. A sub-family of hash functions $\mathcal{H}_i$ is said to be *uniform* if for all $x \in \{0,1\}^n$, $\alpha \in \{0,1\}^i$ and $h_i \xleftarrow{R} \mathcal{H}_i$, $\mathsf{Pr}[h_i(x) = \alpha] = \frac{1}{2^i}$.

Definition 26.2.2. [CW77] A sub-family of hash functions $\mathcal{H}_i$ is said to be *strongly 2-universal* if for all $x, y \in \{0,1\}^n$ and $h_i \xleftarrow{R} \mathcal{H}_i$, $\mathsf{Pr}[h_i(x) = h_i(y)] = \frac{1}{2^i}$

Uniform and strongly 2-universal hash functions play a central role in the design and analysis of probabilistic hash-based approximate counters.

Early work on probabilistic hash-based counting, viz. [GSS06, GHSS07b, GHSS07a], made the important observation that properties of strongly 2-universal hash families allow one to design approximate counters with PAC guarantees. Specifically, using strongly 2-universal hash families ensures that the dispersion index of $|\mathsf{Cell}_{\langle F,h_i\rangle}|$, i.e., $\frac{\sigma_i^2}{\mu_i}$, is bounded above by a constant. Note that if $\mu_i > 1$, this also implies that the coefficient of variation is bounded above by a constant. Furthermore, it was observed that there exist strongly 2-universal hash families for which the problem of finding a $y \in h_i^{-1}(\mathbf{0}^i)$ is in NP, and hence can be solved using SAT solvers. Among various hash families that have been studied over the years, two have featured predominately in the literature on hashing-based model counting. These are the families of random XOR-based hash functions, denoted $\mathcal{H}_{xor}$, and Toeplitz matrix [Gra06] based hash functions, denoted $\mathcal{H}_{\mathcal{T}}$. If we view the variables $x_1, x_2, \ldots x_n$ in the support of the formula φ as a vector X of dimension $n \times 1$, a hash function $h_i : \{0,1\}^n \mapsto \{0,1\}^i$ chosen from either $\mathcal{H}_{xor}$ or $\mathcal{H}_{\mathcal{T}}$ can be represented as $h_i(X) = \boldsymbol{A}X + \boldsymbol{b}$, where $\boldsymbol{A}$ is an $m \times n$ 0-1 matrix, $\boldsymbol{b}$ is $m \times 1$ 0-1 vector and all operations are in $\mathbf{GF_2}$. Regardless of whether we use $\mathcal{H}_{xor}$ or $\mathcal{H}_{\mathcal{T}}$, the vector $\boldsymbol{b}$ is chosen uniformly at random from the space of all $m \times 1$ 0-1 vectors. The way in which $\boldsymbol{A}$ is chosen, however, differs depending on whether we are using the family $\mathcal{H}_{xor}$ or $\mathcal{H}_{\mathcal{T}}$. In case we are using $\mathcal{H}_{xor}$, $\boldsymbol{A}$ is chosen uniformly at random from the space of all $m \times n$ 0-1 matrices. If, on the other hand, we are using $\mathcal{H}_{\mathcal{T}}$, $\boldsymbol{A}$ is chosen uniformly at random from the space of all $m \times n$ Toeplitz 0-1 matrices. Note that Toeplitz 0-1 matrices form a very small subset of all 0-1 matrices. While no significant performance difference has been reported for probabilistic hash-based approximate counters using these two families of hash functions, the family $\mathcal{H}_{xor}$ seems to be the hash family of choice in the existing literature.

When using the family $\mathcal{H}_{xor}$, the random selection of h_i can be achieved by choosing each entry of $\boldsymbol{A}$ to be 1 with probability $p = 1/2$. On an average, this gives $\frac{n}{2}$ 1's in each row of $\boldsymbol{A}$. The invocation of SaturatingCounter in Algorithm 1 necessitates usage of a SAT solver to solve the formula $\varphi \wedge (\boldsymbol{A}X + \boldsymbol{b} = 0)$. Thus, the SAT solver needs to reason about formulas that are presented as a conjunction of usual (i.e., OR) clauses and XOR clauses, and each XOR clause has an average size of $\frac{n}{2}$. Gomes et al [GHSS07b] have observed that the performance of SAT solvers, however, degrades with an increase in the size of XOR clauses.

Therefore recent efforts have focused on the design of sparse hash functions where the count of 1's in every row is $\ll \frac{n}{2}$ [EGSS14, IMMV15, AD16, AT17, AHT18].

Building on the classical notion of definability due to Beth [Bet56], one can define the notion of an *independent support* of a Boolean formula. Specifically, $\mathcal{I} \subseteq X$ is an independent support of φ if whenever two solutions π_1 and π_2 of φ agree on $\mathcal{I}$, then $\pi_1 = \pi_2$ [LM08, CMV14, LLM16]. Chakraborty et al. [CMV14] observed that strongly 2-universal hash functions defined only over $\mathcal{I}$ (instead of X) lend themselves to exactly the same reasoning as strongly 2-universal hash functions defined over the whole of X, when used to partition the solution space of φ. The importance of this observation comes from the fact that for many important classes of problems, the size of $\mathcal{I}$ can be one to two orders of magnitude smaller than that of X. This in turn leads to XOR clauses that are one to two orders of magnitude smaller than that obtained by using strongly 2-universal hash functions defined over X [IMMV15]. Ivrii et al. have proposed an algorithmic technique, called MIS, to determine the independent support of a given formula via a reduction to the Group-oriented Minimal Unsatisfiable Subformula (GMUS) problem [LS08, Nad10]. Given the hardness of GMUS, the proposed technique MIS can scale to a few tens of thousands of variables, and designing scalable techniques for GMUS is an active area of research.

Gomes et al [GHSS07b] observed that for some classes of formulas, setting the expected fraction p of 1's in each row of the matrix $\boldsymbol{A}$ to be significantly smaller than $1/2$ can still provide counts close to those obtained using $p = 1/2$. A straightforward calculation shows that if we choose $h_i(X) = \boldsymbol{A}X + \boldsymbol{b}$, with every entry in $\boldsymbol{A}$ set to 1 with probability p, then for any two distinct vectors $x, y \in \{0,1\}^n$, $\mathsf{Pr}[h_i(x) = h_i(y)] = (1/2+1/2(1-2p)^w)^i = q(w)$ where w is the Hamming weight of $x - y$, i.e., the number of ones in $x - y$. Note that for $p < 1/2$, such a family is not strongly 2-universal. Ermon et al [EGSS14] observed, however, that using a strongly 2-universal hash family is not a necessary requirement to bound the coefficient of variation from above by a constant. Furthermore, for a given $x \in \{0,1\}^n$ and for any $k \in \{1, \ldots n\}$, the count of $y \in \{0,1\}^n$ such that $\mathsf{Pr}[h(x) = h(y)] = q(k)$ is bounded from above by $\binom{n}{k}$. This allowed Ermon et al. to analytically compute values of p that are less than $\frac{1}{2}$, although obtaining a closed-from expression remained elusive. The gap between analytical bounds and closed form bounds was bridged by Zhao et al [ZCSE16] and Asteris and Dimakis [AD16], who showed that having $p \in \mathcal{O}(\frac{\log i}{i})$ suffices to upper bound the coefficient of variation when $i \in \Theta(n)$. Observe that the smaller the value of p, the sparser the matrix $\boldsymbol{A}$.

Motivated by the use of sparse matrices in the design of efficient error-correcting codes, Achlioptas and Theodoropoulos [AT17] showed that one could randomly choose $\boldsymbol{A}$ from an ensemble of Low-Density Parity Check (LDPC) matrices. Achlioptas, Hammoudeh, and Theodoropoulos [AHT18] put forth the observation that an approximate counter strives to compute an estimate c that satisfies two bounds: (i) a lower bound: $c \leq |\mathsf{Sol}(\varphi)| \cdot (1 + \varepsilon)$ and (ii) an upper bound: $c \geq \frac{\mathsf{Sol}(\varphi)}{1+\varepsilon}$. Achlioptas et al [AT17, AHT18] and Gomes et al [GHSS07b] further observed that obtaining good lower bounds requires the hash family $\mathcal{H}$ to be uniform, while obtaining good upper bounds requires $\mathcal{H}$ to ensure that the coefficient of variation of $|\mathsf{Cell}_{\langle \varphi, h_i \rangle}|$ is bounded from above by a constant.

There are a few issues that arise when translating the above ideas to a hash-based approximate counter with PAC guarantees that also scales in practice. First, the bounds achieved with $p \in \mathcal{O}(\frac{\log i}{i})$ hold only for $i \in \Theta(n)$, while for most practical applications $i \ll n$. Second, bounding the coefficient of variation suffices to provide constant-factor approximation guarantees. In order to guarantee $(1+\varepsilon)$-approximation, Stockmeyer's technique [Sto83] of reducing the tolerance must be used, which in turn leads to large formulas being fed to the SAT solver underlying SaturatingCounter.

In a departure from earlier works [EGSS14, AD16, ZCSE16] where the focus was to use analytical methods to obtain upper bound on the variance of $|\mathsf{Cell}_{\langle\varphi,h_i\rangle}|$, Meel(r)Akshay [MA20] focused on searching for the set $\mathsf{Sol}(F)$ that would achieve the maximum variance of $|\mathsf{Cell}_{\langle\varphi,h_i\rangle}|$. This allowed them to observe an elegant connection between the maximization of variance as well as dispersion index of $|\mathsf{Cell}_{\langle\varphi,h_i\rangle}|$ and minimization of the "t-boundary" (the number of pairs with Hamming distance at most t) of sets of points on the Boolean hypercube on n dimensions. Utilizing this connection, they introduced a new family of hash functions, denoted by $\mathcal{H}_{Rennes}$, which consists of hash functions of the form $\boldsymbol{A}y + b$, where every entry of $\boldsymbol{A}[i]$ is set to 1 with $p_i = \mathcal{O}(\frac{\log_2 i}{i})$. The construction of the new family marks a significant departure from prior families in that the density of 1's in the matrix $\boldsymbol{A}$ is dependent on the row index of the matrix. Meel(r)Akshay demonstrated that usage of $\mathcal{H}_{Rennes}$ can lead to significant speedup in runtime while preserving PAC guarantees [MA20].

26.2.5. Using dependent hash functions

Recall that in line 8 of Algorithm 1, the vector of hash functions $(h_0, \ldots h_n)$ is chosen such that h_i maps $\{0,1\}^n$ to $\{0,1\}^i$. As already discussed in Section 26.2.2, an analysis and proof of correctness of an instantiation of Algorithm 1 requires us to bound $\mathsf{Pr}\left[\mathsf{IterError}\right]$, which in turn involves analyzing expressions involving L_i, U_i, and T_i for different values of i. Since, the dependence of L_i, U_i, and T_i across different values of i stems from the choice of $(h_0, \ldots h_n)$, a natural alternative is to choose each h_i uniformly randomly from the sub-family $\mathcal{H}_i$, but independently of the choice of h_j for all $j \neq i$. This minimizes the dependence among events that correspond to different values of i. Not surprisingly, early work on probabilistic hash-based counting relied on this independence of the chosen hash functions for theoretical analysis. Interestingly, independence in the choice of h_i's necessitates a linear search among the indices to find the smallest i that satisfies the check in line 13. This is because with independently chosen h_i's, one cannot rule out the possibility that $\mathsf{CellSize}[i-1] < \mathsf{thresh}$ but $\mathsf{CellSize}[i] = \top$. Independence in the choice of h_i's, therefore, requires that sub-routine ChooseUnassignedIndex chooses indices in sequential order from 1 through n. Indeed, it is not known whether we can do better than linear search when h_i's are chosen independently.

At this point, one may ask if there is anything to be gained from dependence among hash functions. In this context, the *prefix hash family*, used in [CMV16], is worth examining. Let $\alpha \in \{0,1\}^i$ be a vector. For $1 \leq j \leq i$, we use $\alpha[j]$ to denote the j^{th} component of α, and $\alpha[1..j]$ to denote the projection of α on the first j dimensions, i.e $(\alpha[1], \ldots \alpha[j])$. For every hash function $h_i : \{0,1\}^n \to \{0,1\}^i$, we

now define the j^{th} *prefix-slice* of h_i, denoted $h_i[1..j]$ as a map from $\{0,1\}^n$ to $\{0,1\}^j$ such that $h_i[1..j](x) = h_i(x)[1..j]$, for all $x \in \{0,1\}^n$.

Definition 26.2.3. [CMV16] A family of hash functions $\{\mathcal{H}_1, \ldots \mathcal{H}_n\}$ is called a *prefix-family* if for all $h_i \in \mathcal{H}_i$, the j^{th} prefix-slice of h_i is in $\mathcal{H}_j$ for all $j \in \{1, \ldots i\}$,. Thus, for all $h_i \in \mathcal{H}_i$, there exists $h_{i-1} \in \mathcal{H}_{i-1}$ such that for every $x \in \{0,1\}^n$, $h_i(x)$ is simply $h_{i-1}(x)$ augmented with the i^{th} component of $h_i(x)$.

Suppose the hash functions $(h_0, \ldots h_n)$ are chosen from a prefix family such that $h_i = h_n[1..i]$ for $1 \leq i \leq n$ ($h_0 : \{0,1\}^n \to \{0\}$ leaves no choice in design). It is easy to see that with this choice of hash functions, $|\mathsf{Cell}_{\langle \varphi, h_i \rangle}| \leq |\mathsf{Cell}_{\langle \varphi, h_{i-1} \rangle}|$ for all $i \in \{1, \ldots n\}$. Therefore, one can perform a binary search to find the smallest (and unique) i such that $\mathsf{CellSize}[i] < \mathsf{thresh}$ but $\mathsf{CellSize}[i-1] = \top$. The runtime of known implementations of SaturatingCounter($\varphi \wedge h_i^{-1}(\mathbf{0}^i)$) has, however, been empirically observed to significantly increase with i. Therefore, a galloping search performs significantly better than binary search. Furthermore, notice that if C is the set of clauses learnt (as by a CDCL SAT solver) while solving $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$, the dependence among different hash functions facilitates incremental SAT solving. In other words, the clauses learnt while solving ($\varphi \wedge h_i^{-1}(\mathbf{0}^i)$) are sound (and hence, can be re-used) for solving ($\varphi \wedge h_j^{-1}(\mathbf{0}^j)$) for $j > i$. While this sounds promising, one needs to re-think theoretical proofs of PAC guarantees, when using such dependent hash functions. Indeed, the dependence among different events may force usage of union bounds over linear (in n) terms when computing $\mathsf{Pr}[\mathsf{IterError}]$, leading to overall weaker bounds. By a carefully constructed sequence of arguments, Chakraborty et al showed in [CMV16] that one can indeed use dependence to one's advantage even for the theoretical analysis of PAC guarantees. In particular, they showed that when h_i's are chosen from a prefix family as described above, $\mathsf{Pr}[\mathsf{IterError}] \leq 0.36$. The work of Achlioptas and Theodoropoulos [AT17, AHT18], in which LDPC matrices were used to define hash functions, also provides another example where dependence between h_i's have been used to advantage in designing better hash-based approximate model counters.

26.2.6. Counting up to a threshold

Finally, we discuss the key sub-routine SaturatingCounter used in Algorithm 1. Recall that SaturatingCounter is used to find the saturated count (upto thresh) of models of $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$. Computationally, this is the most expensive step of Algorithm 1. We now show that if satisfiability checking of $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$ is *self-reducible*, then SaturatingCounter can be implemented with at most $\mathcal{O}(n \cdot \mathsf{thresh})$ calls to a satisfiability checker for $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$. Fortunately, satisfiability checking is indeed self-reducible for a large class of formulas φ and hash functions h_i.

While a general definition of self-reducibility can be found in [Bal88], our requirement can be stated simply as follows. Let $\mathcal{C}$ be a class of propositional formulas, viz. CNF, DNF, CNF+XOR, DNF+XOR, etc. and let ψ be a formula in $\mathcal{C}$. As an example, if φ is in CNF and $h_i^{-1}(\mathbf{0}^i)$ is a conjunction of XOR clauses, then $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$ is a formula in the CNF+XOR class. Let $x_1, \ldots x_n$ be the

variables in ψ and let $(\pi_1, \ldots \pi_n) \in \{0,1\}^n$ be an assignment of these variables. Let $\psi|_{x_1 \ldots x_i = \pi_1 \ldots \pi_i}$ denote the formula obtained by substituting π_j for x_j in ψ, for all $j \in \{1, \ldots i\}$. We say that satisfiability checking is *self-reducible* for the class $\mathcal{C}$ if for every formula $\psi \in \mathcal{C}$, the following hold: (i) $\psi|_{x_1=\pi_1}$ is in $\mathcal{C}$, and (ii) $(\pi_1, \ldots \pi_n) \models \psi \;\Leftrightarrow\; (\pi_2, \ldots \pi_n) \models \psi|_{x_1=\pi_1}$. Continuing with our earlier example, let $\mathcal{C}$ denote the CNF+XOR class and ψ denote the formula $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$ in class $\mathcal{C}$. It is easy to see that (i) $\psi|_{x_1=\pi_1}$ continues to be a CNF+XOR formula, and (ii) $(\pi_1, \ldots \pi_n) \models \psi \;\Leftrightarrow\; (\pi_2, \ldots \pi_n) \models \psi|_{x_1=\pi_1}$. Hence, satisfiability checking is self-reducible for the CNF+XOR class.

Suppose we have access to a satisfiability checker for formulas in $\mathcal{C}$. If ψ is unsatisfiable, a single invocation of the checker suffices to detect it. Otherwise, using a standard self-reducibility argument, a solution of ψ can be obtained using at most $\mathcal{O}(n)$ calls to the satisfiability checker. Let $\pi = (\pi_1, \ldots \pi_n)$ be such a solution. For $1 \leq i \leq n$, let $\widetilde{\pi}_i$ denote $(\pi_1, \ldots \pi_{i-1}, \overline{\pi_i})$, where $\overline{1} = 0$ and $\overline{0} = 1$. Following a technique used by Murty [Mur68], we can now partition $\mathsf{Sol}(\psi)$ into $n+1$ (possibly empty) sets $Z_0, \ldots Z_n$, where $Z_0 = \{\pi\}$ and $Z_i = \{\widetilde{\pi}_i \tau \mid \tau \in \mathsf{Sol}(\psi|_{x_1 \ldots x_i = \widetilde{\pi}_i})\}$ for $1 \leq i \leq n-1$, with $\widetilde{\pi}_i \tau$ denoting the assignment obtained by concatenating $\widetilde{\pi}_i$ and τ. Finally, $Z_n = \{\widetilde{\pi}_n\}$ if $\widetilde{\pi}_n \models \psi$ and $Z_n = \emptyset$ otherwise. As an example, if $n = 3$ and $\pi = (0,1,1)$, then $\mathsf{Sol}(\psi)$ is partitioned into four sets: (i) $Z_0 = \{(0,1,1)\}$, (ii) $Z_1 = \{(1, \tau_1, \tau_2) \mid (\tau_1, \tau_2) \in \mathsf{Sol}(\psi|_{x_1=1})$, (iii) $Z_2 = \{(0,0,\tau_1') \mid \tau_1' \in \mathsf{Sol}(\psi|_{x_1,x_2=0,0})$, and (iv) $Z_3 = \{(0,1,0)\}$ assuming $(0,1,0) \models \psi$. Observe that starting with a formula ψ with n variables, the above step yields one solution (viz., π) and n self-reduced problem instances (viz. $\psi|_{x_1 \ldots x_i = \widetilde{\pi}_i}$ for $1 \leq i \leq n$), each with strictly fewer than n variables in their support.

In order to implement SaturatingCounter, i.e. count the solutions of ψ upto thresh, we apply the same step as above to each of the self-reduced problem instances. We stop when a total of thresh solutions of ψ have been generated or when all remaining self-reduced problem instances are unsatisfiable. Observe that whenever a new satisfying assignment is found, $\mathcal{O}(n)$ self-reduced problem instances are generated. Since at most thresh solutions of ψ are generated (and counted) by SaturatingCounter, only $\mathcal{O}(n \cdot \mathsf{thresh})$ self-reduced problem instances can be generated. For each of these, we either detect unsatisfiability using a single satisfiability check or obtain a solution using $\mathcal{O}(n)$ satisfiability checks. The latter can, however, happen at most thresh times. Therefore, the total number of calls to the satisfiability checker is in $\mathcal{O}(n \cdot \mathsf{thresh})$.

The above technique of implementing SaturatingCounter using at most $\mathcal{O}(n \cdot \mathsf{thresh})$ invocations of a satisfiability checker works for all class of problems for which satisfiability checking is self-reducible. This includes cases like DNF+XOR formulas, perfect matchings in bi-partite graphs etc.

If $\mathcal{C}$ is a class of propositional formulas for which satisfiability checking is NP-complete, then checking satisfiability of $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$, where $\varphi \in \mathcal{C}$, is also NP-complete. In such cases, we are often interested in the query complexity of SaturatingCounter, given access to an NP oracle. Interestingly, the technique of implementing SaturatingCounter as discussed above may not yield optimal query complexity in such cases. Observe that the above technique not only finds the saturated count of solutions but also generates the corresponding

set of solutions. Bellare, Goldreich, and Petrank [BGP00] observed that if we are interested only in the saturated count, then SaturatingCounter can be implemented with $\mathcal{O}(\mathsf{thresh})$ queries to an NP oracle. Bellare et al's technique, however, constructs queries of size $\mathcal{O}(|\varphi| \cdot \mathsf{thresh})$. While this is a polynomial blow-up in the size of the formula fed to the oracle, the performance of modern SAT solvers can be significantly impacted by such a blow up.

It is worth noting here that while an NP oracle simply provides a Yes/No answer to a decision query, modern SAT solvers either provide a satisfying assignment for an input formula or report unsatisfiability of the formula. Given this difference, it is interesting to ask how many invocations of a SAT solver would be needed in order to implement SaturatingCounter. To formalize this question, we introduce the notion of a *SAT oracle* that takes a propositional formula φ as input, and returns a solution σ if φ is satisfiable and $\perp$ is φ is unsatisfiable. We can now implement SaturatingCounter by simply enumerating the satisfying assignments of an input formula φ using a SAT oracle, until the oracle either returns $\perp$ or we have already found thresh solutions. It is not hard to see that this requires $\mathcal{O}(\mathsf{thresh})$ invocations of the SAT oracle, and the largest size of a formula used in any invocation is $\mathcal{O}(|\varphi| + \mathsf{thresh} \cdot |\mathsf{Sup}(\varphi)|)$.

ApproxMC: A specific hash-based approximate counter

As discussed earlier, state-of-the-art probabilistic hash-based approximate counters (for CNF formulas) can be viewed as (optimized) instances of the generic Algorithm 1. One such counter is ApproxMC [CMV16], obtained with the following specific choices in Algorithm 1.

- In line 1 of the algorithm, thresh is set to $\mathcal{O}(1/\varepsilon^2)$.
- In line 4, $\mathsf{itercount}$ is set to $\mathcal{O}(\log_2 \delta^{-1})$.
- In line 5, the prefix hash family $\mathcal{H}_{xor}$ is chosen.
- In line 8, individual hash functions are chosen such $h_i = h_n[1..i]$ for $1 \leq i \leq n$, with $h_0 : \{0,1\}^n \to \{0\}$.
- SaturatingCounter$(\varphi, \mathsf{thresh})$ is implemented using $\mathcal{O}(\mathsf{thresh})$ invocations of a SAT oracle (a SAT solver, CryptoMiniSat, in practice).

It is shown in [CMV16] that the resulting algorithm provides the following PAC guarantee.

Theorem 26.2.1. Given a formula φ, tolerance ε (> 0), and confidence parameter δ ($0 \leq \delta < 1$), invocation of ApproxMC$(\varphi, \varepsilon, \delta)$ returns a count c such that $\Pr[|\mathsf{Sol}(\varphi)| \cdot (1+\varepsilon)^{-1} \leq c \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)] \geq 1 - \delta$. Furthermore, ApproxMC makes $\mathcal{O}\big(\log n \cdot \varepsilon^{-2} \cdot \log \delta^{-1}\big)$ calls to a SAT oracle.

26.3. Handling CNF+XOR Constraints

Given the extensive reliance on XOR clauses for partitioning the solution space of φ, it is desirable that the underlying SAT solver in SaturatingCounter have native support for conjunctions of XOR clauses. While a system of only XOR clauses can be solved in polynomial-time via Gauss-Jordan Elimination (GJE),

predominant CDCL SAT solvers are known to have poor performance in their ability to handle XORs [BM00]. Given the importance and usage of XOR clauses in cryptanalysis, there is a rich and long history of building hybrid solvers that perform CDCL and/or lookahead reasoning for CNF clauses and Gauss-Jordan elimination for XOR clauses [BM00, HDVZVM04, Che09, SNC09, Soo10, LJN10, LJN11, LJN12].

Soos, Nohl and Catelluccia [SNC09] proposed an elegant architecture, best viewed as an instance of CDCL(XOR), that keeps CNF and XOR clauses separately. It performs Gauss-Jordan Elimination on XORs while performing CDCL reasoning on CNF constraints, with support for sharing of the learnt clauses from XOR-based reasoning to CNF-based reasoning. This architecture formed the basis of the widely used SAT solver CryptoMiniSat. Han and Jiang [HJ12] observed that performing Gauss-Jordan Elimination ensures that the matrix representing XORs is in row echelon form, which allows for lazy and incremental matrix updates. While the separation of CNF and XOR clauses shares similarities to the architecture of modern SMT solvers that separate different theory clauses, there are important differences as well. The primary difference stems from the fact that SMT solvers reason about clauses over different theories with disjoint signatures, while CNF and XOR clauses are defined over the same set of variables. Han and Jiang [HJ12] observed that partial interpolant generation can still be achieved in the presence of mixed CNF and XOR clauses, and the generated interpolants improve the runtime performance of the solver. Laitinen, Junttila, and Niemelä [LJN12] observed that XOR reasoning can be significantly improved by splitting XORs into different *components* that are connected to each other only by *cut variables*. The decomposition ensures that full propagation for each of the components guarantee propagation for the entire set of XORs. Such a decomposition has been empirically shown to improve the memory usage when solving a conjunction of mixed CNF and XOR clauses.

The architecture of separating XOR and CNF clauses, however, comes at the cost of disabling execution of in-processing steps. Upon closer inspection, the separation of CNF and XOR clauses does not fundamentally imply lack of soundness of in-processing steps. Nevertheless, a sound implementation of in-processing steps is a daunting task, given the need for extensive studies into the effect of every in-processing step on XOR clauses. The lack of usage of pre- and in-processing steps significantly hurts the performance of the backend SAT solver in SaturatingCounter since these techniques have been shown to be crucial to the performance of state-of-the-art SAT solvers.

To allow seamless integration of pre- and in-processing steps, it is important that the solver has access to XOR clauses in CNF form while ensuring native support for XORs to perform Gauss-Jordan elimination. To achieve such integration, Soos and Meel [SM19] recently proposed a new architecture called BIRD (an acronym for Blast, In-process, Recover, and Destroy). A high-level pseudocode for BIRD is shown in Algorithm 2.

Note that Algorithm BIRD exits its main loop as soon as it finds a solution or proves that the formula is unsatisfiable. Furthermore, if the benchmarks have XOR clauses encoded in CNF, BIRD can efficiently recover such XORs and use Gauss-Jordan elimination on such recovered XORs. The primary challenge for

Algorithm 2 BIRD(φ) ▷ *φ has a mix of CNF and XOR clauses*

1: **Blast** XOR clauses into normal CNF clauses
2: **In-process** (and pre-process) over CNF clauses
3: **Recover** simplified XOR clauses
4: Perform CDCL on CNF clauses with on-the-fly Gauss-Jordan Elimination on XOR clauses until: (a) in-processing is scheduled, (b) a solution is found, or (c) formula is found to be unsatisfiable
5: **Destroy** XOR clauses and goto line 2 if conditions (b) or (c) above don't hold. Otherwise, **return** solution or report unsatisfiable.

BIRD is to ensure that line 3 can be executed efficiently. Towards this end, the core idea of the recovery algorithm proposed by Soos and Meel is to choose a clause, say base_cl, and recover the unique XOR defined exactly over the variables in base_cl, if such an XOR exists. For example, if base_cl is $(x_1 \vee x_2 \vee x_3)$, then the only XOR defined over x_1, x_2, x_3 in which base_cl can possibly participate is $x_1 \oplus x_2 \oplus x_3 = 1$. Observe that the right hand side of the XOR clause (i.e., 0 or 1) can be obtained by computing the parity of the variables in base_cl. The key idea behind the search for XORs over variables in base_cl, say of size t, is to perform a linear pass and check whether there exists a subset of CNF clauses that would imply the 2^{t-1} combination of CNF clauses over t variables that corresponds to the desired XOR clause over these t variables.

Note that a clause may imply multiple CNF clauses over t variables. For example, let base_cl $:= (x_1 \vee x_2 \vee x_3)$, then a clause cl $:= (x_1)$ would imply 4 clauses over $\{x_1, x_2, x_3\}$, i.e. $\{(x_1 \vee x_2 \vee \neg x_3), (x_1 \vee x_2 \vee x_3), (x_1 \vee \neg x_2 \vee x_3), (x_1 \vee \neg x_2 \vee \neg x_3)\}$. To this end, Soos and Meel maintain an array of possible combinations of size 2^t. They update the entry, indexed by the binary representation of the clause for a fixed ordering of variables, corresponding to a clause cl$'$ to 1 if cl $\rightarrow$ cl$'$. Similar to other aspects of SAT solving, efficient data structures are vital to perform the above mentioned checks and updates efficiently. The interested reader is referred to [SM19, SGM20] for a detailed discussion.

26.4. Approximate Model Counting for DNF

We now turn our attention to computing a PAC estimate for $|\mathsf{Sol}(\varphi)|$, where φ is a DNF formula. While the problem of exact counting for DNF formulas is #P-complete, Karp and Luby showed in their seminal work [KL83] that there exists an FPRAS for the problem. Specifically, their work used the Monte Carlo framework to arrive at an FPRAS for DNF counting. Subsequently, there have been several follow-up works based on Monte Carlo counting techniques [KLM89, DKLR00, Vaz13, Hub17, HJ19]. These have yielded an improved theoretical understanding of DNF counting and also algorithms that have improved performance in practice. Recently, Chakraborty et al. [CMV16] and Meel, Shrotri, and Vardi [MSV17] showed that the hashing-based framework discussed in Section 26.2.1 could also be adapted to yield an FPRAS for DNF counting. Thus, we have two different approaches for obtaining an FPRAS for DNF counting. These are discussed in detail below.

We fix some notation before delving into the details. A DNF formula φ is

given as a disjunction of cubes. We assume the cubes are indexed by natural numbers, and we denote the i^{th}cube by φ^i. Thus $\varphi = \varphi^1 \vee \ldots \vee \varphi^m$. We use n and m to denote the number of variables and cubes, respectively, in the input DNF formula. The *width* of a cube φ^i refers to the number of literals in φ^i and is denoted by $\mathsf{width}(\varphi^i)$. We use w to denote the minimum width, minimized over all cubes of the formula, i.e. $\mathsf{w} = \min_i \mathsf{width}(\varphi^i)$.

26.4.1. DNF counting via Monte Carlo framework

Approximate counting algorithms designed using the Monte Carlo approach effectively determine the value of an estimator through multiple independent random samples. The seminal work of Karp and Luby [KL83] can be viewed as a Monte Carlo algorithm using a 0-1 estimator, as shown in Algorithm 3. This algorithm

Algorithm 3 Monte-Carlo-Count($\mathcal{A}$,$\mathcal{U}$) ▷ $\mathcal{A} \subseteq \mathcal{U}$

1: $Y \leftarrow 0$
2: **repeat** N times
3: Select an element $t \in \mathcal{U}$ uniformly at random
4: **if** $t \in \mathcal{A}$ **then**
5: $Y \leftarrow Y + 1$
6: **until**
7: $Z \leftarrow \frac{Y}{N} \times |\mathcal{U}|$
8: **return** Z

estimates the cardinality of a set $\mathcal{A}$ in the universe $\mathcal{U}$, given access to a sub-routine that samples uniformly from $\mathcal{U}$ and a sub-routine that tests if a randomly chosen element is in the set $\mathcal{A}$. Here, $\frac{Y}{N}$ is an unbiased estimator for $\rho = \frac{|\mathcal{A}|}{|\mathcal{U}|}$ and Z is an unbiased estimator for $|\mathcal{A}|$. It can be shown [Ŕ70] that if $N \geq \frac{1}{\rho\varepsilon^2} \cdot \ln(\frac{2}{\delta})$, then $\Pr[(1-\varepsilon) \cdot |\mathcal{A}| \leq Z \leq (1+\varepsilon) \cdot |\mathcal{A}|] \geq 1 - \delta$. Algorithm 3 is an FPRAS if the number of samples N, and the times taken by the sampler in line 3 and by the inclusion checker in line 4, are polynomial in the size of the input.

In the context of counting solutions of DNF formulas, the set $\mathcal{U}$ is the set of all assignments over n variables, and $\mathcal{A} = \mathsf{Sol}(\varphi)$. In this case, the ratio ρ is also called the *density of solutions*. A trivial lower bound on $|\mathsf{Sol}(\varphi)|$ is $2^{n-\mathsf{w}}$. Thus, $\frac{1}{\rho} \leq 2^{\mathsf{w}}$. If w is bounded above by a logarithmic function of n and m, then $\frac{1}{\rho}$ is polynomial in n and m and we need polynomially many samples. Nevertheless, since the above requirement may not always hold, this straightforward algorithm does not give an FPRAS.

Karp and Luby [KL83] observed that the dependence of N on w can be avoided if, instead of using $\mathcal{A} = \mathsf{Sol}(\varphi)$ and $\mathcal{U} = \{0,1\}^n$, we use suitably defined alternate sets $\widehat{\mathcal{A}}$ and $\widehat{\mathcal{U}}$, while ensuring that $|\mathsf{Sol}(\varphi)| = |\widehat{\mathcal{A}}|$ and $\frac{1}{\widehat{\rho}} = |\widehat{\mathcal{U}}|/|\widehat{\mathcal{A}}|$ is bounded polynomially in m and n. This is a powerful technique and researchers have used this idea and its variants over the years to achieve improvement in runtime performance and quality of approximations of Monte Carlo DNF counters. We mention below three variants of Monte Carlo approximate DNF counting algorithms.

Karp and Luby's counter Karp and Luby [KL83] developed the first FPRAS for DNF counting, which we refer to as KL Counter. They defined a new universe

$\widehat{\mathcal{U}} = \{(\sigma, \varphi^i) \mid \sigma \models \varphi^i\}$, and the corresponding solution space $\widehat{\mathcal{A}} = \{(\sigma, \varphi^i) \mid \sigma \models \varphi^i$ and $\forall j < i, \sigma \not\models \varphi^j\}$ for a fixed ordering of the cubes. They showed that $|\mathsf{Sol}(\varphi)| = |\widehat{\mathcal{A}}|$ and the ratio $\frac{|\widehat{\mathcal{U}}|}{|\widehat{\mathcal{A}}|} \leq m$. Furthermore, executing Step 3 of Algorithm 3 takes $\mathcal{O}(n)$ time, while executing Step 4 takes $\mathcal{O}(mn)$ time. Consequently, the time complexity of KL Counter is in $\mathcal{O}(\frac{m^2n^2}{\varepsilon^2} \cdot \log(\frac{1}{\delta}))$. Karp and Luby showed that by designing a slightly different unbiased estimator, this complexity can indeed be reduced to $\mathcal{O}(\frac{m^2n}{\varepsilon^2} \log(\frac{1}{\delta}))$.

Karp, Luby and Madras' counter Karp, Luby and Madras [KLM89] proposed an improvement of KL Counter by employing a non 0-1 estimator. Towards this end, the concept of *coverage* of an assignment σ in $\mathcal{U}$ was introduced as $\mathsf{cover}(\sigma) = \{j | \sigma \models \varphi^j\}$. The first key insight of Karp et al. was that $|\widehat{\mathcal{A}}| = \sum_{(\sigma,\varphi^i) \in \mathcal{U}} \frac{1}{|\mathsf{cover}(\sigma)|}$. Their second key insight was to define an estimator for $1/|\mathsf{cover}(\sigma)|$ using the geometric distribution. They were then able to show that the time complexity of the resulting Monte-Carlo algorithm, which we call KLM Counter, is in $\mathcal{O}(\frac{mn}{\varepsilon^2} \cdot \log(\frac{1}{\delta}))$ – an improvement over KL Counter.

Vazirani's counter A variant of KLM Counter, called Vazirani Counter, was described in Vazirani [Vaz13], where $|\mathsf{cover}(\sigma)|$ is computed exactly by iterating over all cubes, avoiding the use of the geometric distribution in [KLM89]. The advantage of Vazirani Counter is that it is able to utilize ideas for optimal Monte Carlo estimation proposed in [DKLR00]. Consequently, Vazirani Counter requires fewer samples than KL Counter to achieve the same error bounds. The time for generating a sample, however, can be considerably more since the check for $\sigma \models \varphi^j$ has to be performed for all cubes.

26.4.2. DNF Counting via Hashing-based Approach

While Monte Carlo techniques have been the predominant paradigm for approximate DNF counting, it turns out that the universal hashing based approach also yields an FPRAS for DNF counting. Recall that Algorithm HashCounter, described in Section 26.2.1 is a randomized approximation scheme that makes polynomially many invocations to SaturatingCounter. It has been observed in [CMV16, MSV17] that SaturatingCounter can be designed to run in time polynomial in the size of the input formula and the threshold, if the input formula is in DNF. To see why this is true, observe that satisfiability checking of $\varphi \wedge h_i^{-1}(0^i)$ is self-reducible if $h_i^{-1}(0^i)$ is a conjunction of XOR constraints and φ is a DNF formula. Furthermore, the satisfiability of $\varphi \wedge h_i^{-1}(0^i)$ can be checked in polynomial time by iterating over all the cubes of the input formula, substituting the forced assignments induced by each cube into the XOR constraints separately, and using Gauss-Jordan Elimination to check satisfiability of the simplified XOR constraints. Interestingly, we can even avoid appealing to the self-reducibility of satisfiability checking of $\varphi \wedge h_i^{-1}(0^i)$ by simply enumerating solutions of the simplified XOR constraints. Note that at no step, one would have to enumerate more than thresh solutions. Concretely, this leads to an FPRAS, called DNFApproxMC in [MSV17], for DNF formulas with complexity $\mathcal{O}((mn^3 + mn^2/\varepsilon^2) \log n \log(1/\delta))$.

The existence of universal hashing-based FPRAS for DNF counting leads to an obvious question: *can these algorithms match the asymptotic complexity of Monte Carlo based FPRAS for DNF counting?* Towards this end, Meel et al. [MSV17] sought to avoid the need for Gauss-Jordan elimination originating due to the usage of $\mathcal{H}_{xor}$ or $\mathcal{H}_{\mathcal{T}}$. Specifically, they proposed a new class of hash functions $\mathcal{H}_{\text{Rex}}$ such that every hash function $h \in \mathcal{H}_{\text{Rex}}$ that maps $\{0,1\}^n \mapsto \{0,1\}^m$ can be represented as $\boldsymbol{A}x + \boldsymbol{b}$, where $\boldsymbol{A}$ is a 0-1 random matrix of dimension $m \times n$ in row-echelon form and $\boldsymbol{b}$ is a random 0-1 matrix of dimension $m \times 1$. In particular, we can represent $\boldsymbol{A}$ as $\boldsymbol{A} = [\boldsymbol{I} : \boldsymbol{D}]$, where $\boldsymbol{I}$ is the identity matrix of size $m \times m$ and $\boldsymbol{D}$ is a random 0-1 matrix of size $m \times (n-m)$. In follow-up work, Meel et al. [MSV19] sought to avoid the complexity bottleneck presented by the requirement of having to enumerate up to thresh solutions in every call to SaturatingCounter. Towards this end, they proposed the search for the right number (m) of cells in reverse order starting from $m = n$ instead of from $m = 0$. This helps in ensuring that only $\mathcal{O}(\mathsf{thresh})$ solutions are enumerated during the entire iteration of the loop in lines 7–15 of Algorithm HashCounter (see Algorithm 1).

In a series of papers [MSV17, MSV19], Meel et al. proposed several other improvements, eventually resulting in a universal hashing-based DNF counting algorithm, called SymbolicDNFApproxMC, that runs in time $\widetilde{O}(\frac{mn}{\varepsilon^2}\log(1/\delta))$. Note that this matches the asymptotic complexity bound for Monte Carlo techniques discussed above. One would expect that the asymptotic complexity of different FPRAS correlates positively with their empirical run-time behavior when comparing the performance of different FPRAS on a large set of benchmarks. In a rather surprising twist, however, the landscape of empirically observed run-time behavior turns out to be far more nuanced than that captured by worst-case analysis, as shown in [MSV19]. In particular, there is no single algorithm that outperforms all others for all classes of formulas and input parameters. Interestingly, Meel et al. observed that the algorithm, DNFApproxMC, with one of the worst-time complexities ended up solving the largest number of benchmarks.

It is worth noting that there has been a long gap of 34 years between Karp and Luby's Monte Carlo-based FPRAS for DNF counting, and the discovery of hashing-based FPRAS for DNF counting. This is despite the fact that hashing techniques for CNF counting have been known at least since Stockmeyer's seminal work in 1983. Interestingly, Stockmeyer's technique for transforming an algorithm with a constant-factor approximation to one with an $(1+\varepsilon)$-factor approximation does not work for DNF counting. This is because conjoining a set of DNF formulas (a key component of Stockmeyer's technique) does not always yield a formula representable in DNF without an exponential blow-up. In contrast, in hashing-based approximate counting (ref. Algorithm HashCounter), a $(1+\varepsilon)$-factor approximation is easily achieved by choosing thresh to be in $\mathcal{O}(1/\varepsilon^2)$.

26.5. Weighted Counting

A natural extension of model counting is to augment the formula φ with a weight function ρ that assigns a non-negative weight to every assignment of values to variables. The problem of weighted model counting, also known as discrete in-

tegration, seeks to compute the weight of φ, defined as the sum of the weight of all its solutions. Note that if ρ assigns weight 1 to each assignment, then the corresponding problem is simply model counting.

Formally, let $\rho : \{0,1\}^n \to \mathbb{Q}_+ \cap [0,1]$ be a *weight function* mapping each truth assignment to a non-negative rational number in $[0,1]$ such that (i) $\forall \sigma \in \{0,1\}^n$, $\rho(\sigma)$ is computable in polynomial time, and (ii) $\forall \sigma \in \{0,1\}^n$, $\rho(\sigma)$ is written in binary representation with p bits. We extend the weight function to sets of truth assignments and Boolean formulas in the obvious way. If Y is a subset of $\{0,1\}^n$, the weight of Y is defined as the cumulative weight of the truth assignments in Y: $\rho(Y) = \sum_{\sigma \in Y} \rho(\sigma)$. By definition, the weight of the empty set is 0. The weight of a formula φ is defined as the cumulative weight of its solutions, i.e., $\rho(\varphi) = \sum_{\sigma \models F} \rho(\sigma)$. Given ρ and φ, the problem of weighted model counting seeks to compute $\rho(\varphi)$. The polynomial-time computability of $\rho(\sigma)$ implies that the problem of weighted counting is #P-complete.

The recent success of approximate (unweighted) model counters has inspired several attempts for approximate weighted model counting as well [EGSS13a, EGSS13b, CFM+14, EGSS14, AJ15, AD16, dCM19]. Despite impressive advances made by these attempts, the inherent hardness of weighted model counting has repeatedly manifested itself as a scalability challenge. This highlights the need and opportunity for both algorithmic and systems-oriented research in this area.

In order to present a unified view of different algorithms proposed for weighted model counting, we follow the treatment of Ermon et al [EGSS13b] and de Colnet and Meel [dCM19], and introduce two additional notions: the *tail function* and *effective weight function*. The tail function τ maps the space of weights (i.e. $\mathbb{Q}_+ \cap [0,1]$) to $\mathbb{N}$. Informally, $\tau(u)$ counts the number of models of φ with weight at least as large as u. Formally, $\tau(u) = \left|\{\sigma \in \{0,1\}^n \mid \sigma \models \varphi \text{ and } \rho(\sigma) \geq u\}\right|$. The effective weight function w, which is the dual of the tail function, can be intuitively thought of as mapping non-zero natural numbers to weights. Informally, $w(t)$ gives the largest weight of a model σ of φ that has at least t models of larger or equal weight. For technical reasons, we formally define w as a mapping from the positive reals (instead of positive natural numbers) to the space of weights as follows: $w(t) = \max_\sigma\{w(\sigma) \mid \sigma \models \varphi \text{ and } \tau(\rho(\sigma)) \geq t\}$. Note that both the effective weight function and the tail function are non-increasing functions of their argument. It can be shown [EGSS13b, dCM19] that the weighted model count $\rho(\varphi)$ is exactly the area under the $\tau(u)$ (plotted against u) curve, as well as that under the $w(t)$ (plotted against t) curve. In other words, $\rho(\varphi) = \int \tau(u)du = \int w(t)dt$. Both of these are integrals of non-increasing functions defined over $\mathbb{R}_+$ and are of finite support.

Recent work on approximate weighted model counting can now be broadly described as following a two-step strategy. In the first step, the task of weighted counting is reduced to an integration problem for a suitable real-valued non-increasing function. In the second step, methods based on upper and lower rectangle approximations or Monte Carlo integrators are used to approximate the integral of a real function. We describe below two different approaches that follow this strategy, effectively reducing weighted counting to optimization and unweighted counting respectively.

26.5.1. From weighted counting to optimization

Ermon et al. [EGSS13a, EGSS13b] used the rectangle approximation of $\int \tau(u)du$ to reduce the problem of weighted counting to one of optimization. The first step in this process is the partitioning of the weight (i.e., u) axis into $\mathcal{O}(n)$ intervals, where $n = |\mathsf{Sup}(\varphi)|$. Specifically, the axis is split at the weights $q_0, q_1 \ldots, q_n$, where q_i is the maximal weight such that $\tau(q_i) \geq 2^i$; we call these q_i's *quantile weights*. Observe that the quantile weights are all well-defined, and form a non-increasing sequence. Furthermore, for each positive quantile weight q_i, there exists some truth assignment such that $q_i = \rho(\sigma)$.

The partitioning of the u axis in $\int \tau(u)du$ at the quantile weights gives $\rho(\varphi) = q_n 2^n + \sum_{i=1}^{n} \int_{q_i}^{q_{i-1}} \tau(u)du$ where $\int_{q_i}^{q_{i-1}}$ represents the integral on $(q_i, q_{i-1}]$. Note that if u is in $(q_i, q_{i-1}]$, then we have $2^{i-1} \leq \tau(u) \leq 2^i$. Therefore, $2^{i-1}\,(q_{i-1} - q_i) \leq \int_{q_i}^{q_{i-1}} \tau(u)du \leq 2^i\,(q_{i-1} - q_i)$. Summing all bounds together and rearranging the terms, we obtain $q_0 + \sum_{i=0}^{n-1} q_{i+1}2^i \leq \rho(\varphi) \leq q_0 + \sum_{i=0}^{n-1} q_i 2^i$. Thus, if W_1 denotes $q_0 + \sum_{i=0}^{n-1} q_{i+1}2^i$, we have $W_1 \leq \rho(\varphi) \leq 2W_1$.

Given $q_0, \cdots, q_n$, the estimate W_1 can be computed in polynomial time. Recall that for all i, the weight q_i is defined as $\max\left\{\rho(\sigma) \mid \sigma \models \varphi \text{ and } \tau(\rho(\sigma)) \geq 2^i\right\}$. Therefore the task of approximating the weighted model count $\rho(\varphi)$ has effectively been transformed to $n+1$ optimization problems. While computing q_i exactly is intractable, we can obtain a good approximation of q_i via the usage of 2-universal hash functions. Similar to hashing-based counting, the core technical idea is to employ 2-universal hash functions to partition the space of solutions, viz. $\{0,1\}^n$, into approximately equal-sized 2^i cells. We then choose an arbitrary cell and use an optimization query (viz. MPE query in [EGSS13b] or MaxSAT query in [dCM19]) to find a solution of φ with maximum weight in the chosen cell. Let us call this σ_{max}. The 2-universality property of the hash family ensures that $\rho(\sigma_{max}) \in [q_{i-2}, q_{i+2}]$ with probability greater than $\frac{1}{2}$. Therefore, we can choose $\rho(\sigma_{max})$ as an estimate $\widehat{q_i}$ of q_i. As in the case of counting, the above process of randomly partitioning the space of solutions and finding an estimate of q_i can be repeated an appropriate number (linear in $\log \frac{1}{\delta}$ and $\log n$) of times, and the median of the estimates used with high enough confidence. The estimate $\widehat{W_1}$ of W_1 computed using the median estimates of q_is can be shown to be a 16-factor approximation with confidence at least $1 - \delta$ [EGSS13b, dCM19]. To obtain a $(1+\varepsilon)$-factor approximation, Stockmeyer's technique, as outlined in Section 26.2.3.1, can be applied.

26.5.2. From weighted counting to counting

We now discuss techniques that effectively use rectangular approximations of $\int w(t)dt$ to estimate a weighted model count. The work of Chakraborty et al [CFM$^+$14] and de Colnet and Meel [dCM19] belong to this category. As in the previous subsection, the first step is to partition the tail (i.e., t) axis. Towards this end, we choose a non-decreasing sequence $\tau_0 \leq \tau_1 \leq \ldots \leq \tau_p$ such that $\tau_i = \tau(1/2^i)$ and p is the number of bits in binary representation of $\rho(\sigma)$ as mentioned in Section 26.5. We call these τ_i's *splitting tails*.

The partitioning of the integral $\int w(t)dt$ at the splitting tails gives $\rho(\varphi) = \tau_0 + \sum_{i=0}^{p-1} \int_{\tau_i}^{\tau_{i+1}} w(t)dt$, where $\int_{\tau_i}^{\tau_{i+1}}$ represents the integral on $(\tau_i, \tau_{i+1}]$. If t is in $(\tau_i, \tau_{i+1}]$, then $2^{-i-1} \leq w(t) \leq 2^{-i}$. So we bound the tails in each interval $(\tau_i, \tau_{i+1}]$ within a factor of 2 as $2^{-i-1}(\tau_{i+1} - \tau_i) \leq \int_{\tau_i}^{\tau_{i+1}} w(t)dt \leq 2^{-i}(\tau_{i+1} - \tau_i)$ Note that the bound holds even when $(\tau_i, \tau_{i+1}]$ is empty (i.e. $\tau_i = \tau_{i+1}$). Summing all bounds together and rearranging the terms, we obtain $\tau_p 2^{-p} + \sum_{i=0}^{p-1} \tau_i 2^{-(i+1)} \leq \rho(\varphi) \leq \tau_p 2^{-p} + \sum_{i=0}^{p-1} \tau_{i+1} 2^{-(i+1)}$. Let W_2 denote $\tau_p 2^{-p} + \sum_{i=0}^{p-1} \tau_i 2^{-(i+1)}$. Then, we have $W_2 \leq \rho(\varphi) \leq 2W_2$. As in the previous subsection, if we are given $\{\tau_0, \cdots, \tau_p\}$, the estimate W_2 can be computed in polynomial time. Furthermore, recall that the tail τ_i is defined to be $\left|\{\sigma \mid \sigma \models \varphi \text{ and } \rho(\sigma) \geq 2^{-i}\}\right|$ Therefore, the problem of estimating $\rho(\varphi)$ is now transformed to $p+1$ counting sub-problems. Note that each sub-problem requires us to count not just models of φ, but models with a specified minimum weight. In special cases, where the weight of an assignment is the product or sum of weights of corresponding literals, the set of models of φ with weight at least 2^{-i} can be represented using a Pseudo-Boolean constraint, as has been done in [CFM+14]. Estimating the model count of such a constraint can be accomplished using the same hashing-based technique discussed earlier, by partitioning the solution space using random XOR constraints. Let $\widehat{\tau_i}$ denote the estimate of τ_i obtained in this manner. Chakraborty et al [CFM+14] and de Colnet and Meel [dCM19] have shown that the estimate $\widehat{W_2}$ of W_2 obtained by using $\widehat{\tau_i}$ instead of τ_i is a constant-factor approximation with confidence at least 1-δ [dCM19]. As before, Stockmeyer's technique can be used to obtain a $(1+\varepsilon)$-factor approximation of $\rho(\varphi)$.

In a separate line of work, Chakraborty et al. [CFMV15] and Dudek, Fried, and Meel [DFM] investigated reduction from weighted model counting to unweighted model counting for a specific class of weight functions called *literal weight functions*. In this case, weights are assigned to individual literals, and the weight of an assignment is simply the product of weights of its literals. For a variable x_i in the support of φ and a weight function $\rho(\cdot)$, let $\rho(x_i^1)$ and $\rho(x_i^0)$ denote the weights of the positive and negative literals respectively, of x_i. Without loss of generality, we assume that $\rho(x_i^1) + \rho(x_i^0) = 1$, and that each $\rho(x_i^1) = \frac{k_i}{\ell_i}$, where $k_i, \ell_i \in \mathbb{N}$ and $k_i \leq \ell_i$. The key idea in [CFMV15, DFM] is to transform a given formula φ into another formula $\widehat{\varphi}$ such that $\rho(\varphi) = C_\rho \cdot |\mathsf{Sol}(\widehat{\varphi})|$, where C_ρ depends only on ℓ_i's, and *not* on φ. Once the above transformation is achieved, the problem of estimating $\rho(\varphi)$ reduces to that of counting models of $\widehat{\varphi}$. To achieve the transformation, Chakraborty et al used a gadget called *chain formulas* [CFMV15]. The chain formula ψ_{k_i,ℓ_i} represents a Boolean formula with k_i solutions over $\lceil \log_2 \ell_i \rceil$ fresh variables. The transformed formula $\widehat{\varphi}$ is then obtained as $\varphi \wedge \bigwedge_{i=1}^{n} ((x_i \rightarrow \psi_{k_i,\ell_i}) \wedge (\neg x_i \rightarrow \psi_{\ell_i - k_i,\ell_i}))$. An approximate counting technique, viz. Algorithm HashCounter, can now be invoked to obtain a $(1+\varepsilon)$-approximation of $|\mathsf{Sol}(\widehat{\varphi})|$ with confidence $1-\delta$. Since C_ρ is polynomially computable from the ℓ_i's, this gives us a PAC estimate of $\rho(\varphi) = C_\rho \cdot |\mathsf{Sol}(\widehat{\varphi})|$.

26.5.3. Scalability challenges

Both the algorithmic approaches to weighted counting outlined above provide rigorous theoretical guarantees on the quality of computed estimates. They have

been implemented and demonstrated to scale to problems involving several thousand propositional variables. Nevertheless, there are factors that have impeded scaling them up to even larger problem sizes. Some of these factors include:

1. The lack of highly efficient MaxSAT or MPE solvers capable of handling constraints with XOR clauses.
2. The lack of solvers capable of handling formulas expressed as a conjunction of CNF, Pseudo-Boolean constraints, and XOR constraints [PJM19].
3. The reduction proposed by Chakraborty et al. [CFMV15] suffers from a linear increase in the number of variables. This results in large XOR constraints when using hash-based approximate counting techniques. Large XOR constraints are known to present scalability hurdles to state-of-the-art satisfiability solvers.

Given these bottlenecks, the spectacular progress witnessed in recent years in MaxSAT, MPE and pseudo-Boolean constraint solving present significant opportunities for pushing the frontiers of the weighted model counting. In addition, the use of short XOR clauses without weakening theoretical guarantees and the integration of Gaussian elimination in MaxSAT, MPE and pseudo-Boolean solvers (akin to what has been done for CNF solvers) present promising research directions to pursue.

26.6. Conclusion

Model counting is a fundamental computational problem with a wide variety of applications. It is not surprising therefore that it has attracted the attention of theoreticians and practitioners for the past four decades. This has resulted in a rich bouquet of algorithms with differing characteristics in terms of guarantees on the quality of estimates and runtime performance. Among them, universal hashing-based approximate counting has emerged as the dominant paradigm over the past decade. The success of hashing-based counters can be attributed to two primary reasons: (i) properties of universal hash functions that allow these algorithms to provide (ε, δ) guarantees, and (ii) the spectacular advances in modern satisfiability solving techniques that allow counting algorithms to scale to constraints involving hundreds of thousands of variables.

While we focused on counting all models of a formula φ in this chapter, the hashing-based techniques described in the preceding sections also generalize to the related problem of projected model counting, wherein given $X \subseteq \mathsf{Sup}(\varphi)$, the task is to compute $|\mathsf{Sol}(\exists X \varphi)|$ [SM19]. Given that usage of the existential quantifier ($\exists$) can give exponentially succinct formulas vis-a-vis equivalent formulas written without quantifiers, the problem of projected model counting has been employed in several applications ranging from network (un)reliability to neural network verification [DOMPV17, BSS$^+$19, NSM$^+$19].

The promise exhibited by state-of-the-art approximate model counters has led to the study of several extensions and generalizations: maximum model counting [FRS17], weighted model integration [BPVdB15, KMS$^+$18], stochastic satisfiability [LWJ17], and counting over string and bit-vector constraints [LSSD14, ABB15, CMMV16, CDM17]. Each of these generalizations have necessitated the

development of new techniques and have attracted newer applications. Furthermore, approximate model counting and almost-uniform sampling are known to be inter-reducible [JVV86]. The close relationship has led to a long line of fruitful work extending several hashing-based techniques discussed in this chapter to the context of uniform sampling [CMV13a, CMV14, CFM+14, CFM+15, MVC+16, Mee17]

The promise and progress in the development of approximate model counters has opened several new directions of research, that is evident from the recent flurry of activity in the research community studying approximate counting and its applications. The availability of implementations that both scale to large inputs and also provide strong approximation guarantees has ignited interest among practitioners in seeking new applications for counting. Therefore, we look optimistically to the future where the groundwork laid by hashing-based techniques will lead to the development of new paradigms and practical applications of approximate model counting.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge Univ. Press, 2009.

[ABB15] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *Proc. of CAV*, pages 255–272. Springer, 2015.

[AD16] Megasthenis Asteris and Alexandros G Dimakis. LDPC codes for discrete integration. Technical report, Technical report, UT Austin, 2016.

[AHT18] Dimitris Achlioptas, Zayd Hammoudeh, and Panos Theodoropoulos. Fast and flexible probabilistic model counting. In *Proc. of SAT*, pages 148–164. Springer, 2018.

[AJ15] Dimitris Achlioptas and Pei Jiang. Stochastic integration via error-correcting codes. In *UAI*, pages 22–31, 2015.

[AT17] Dimitris Achlioptas and Panos Theodoropoulos. Probabilistic model counting with short XORs. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 3–19. Springer, 2017.

[Bal88] José Luis Balcázar. *Self-reducibility structures and solutions of NP problems*. Departament de Llenguatges i Sistemes Informàtics [de la] Universitat, 1988.

[BEH+18] Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf. Scalable approximation of quantitative information flow in programs. In *Proc. of VMCAI*, 2018.

[Bet56] Evert W Beth. On Padoa's method in the theory of definition. *Journal of Symbolic Logic*, 21(2):194–195, 1956.

[BGP00] Mihir Bellare, Oded Goldreich, and Erez Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 2000.

[BM00] Peter Baumgartner and Fabio Massacci. The taming of the (X)OR. In *Proc. of CL*, pages 508–522. Springer, 2000.

[BPVdB15] Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *Proc. of IJCAI*, pages 2770–2776, 2015.

[BSS+19] Teodora Baluta, Shiqi Shen, Shweta Shine, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In *Proc. of CCS*, 2019.

[CDM17] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. *Acta Informatica*, 54(8):729–764, 2017.

[CFM+14] Supratik Chakraborty, Daniel J. Fremont, Kuldep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In *Proc. of AAAI*, pages 1722–1730, 2014.

[CFM+15] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. On parallel scalable uniform sat witness generation. In *Proc. of TACAS*, pages 304–319, 2015.

[CFMV15] Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. From weighted to unweighted model counting. In *Proc. of AAAI*, pages 689–695, 2015.

[Che09] Jingchao Chen. Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques. In *Proc. of SAT*, pages 298–311. Springer, 2009.

[CMMV16] Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. Approximate probabilistic inference via word-level counting. In *Proc. of AAAI*, 2016.

[CMV13a] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable and nearly uniform generator of SAT witnesses. In *Proc. of CAV*, pages 608–623, 2013.

[CMV13b] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Proc. of CP*, pages 200–216, 2013.

[CMV14] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, pages 1–6, 2014.

[CMV16] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proc. of IJCAI*, 2016.

[CMV19] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. On the hardness of probabilistic inference relaxations. In *Proc. of AAAI*, 2019.

[CW77] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proc. of STOC*, pages 106–112. ACM, 1977.

[dCM19] Alexis de Colnet and Kuldeep S. Meel. Dual hashing-based algorithms for discrete integration. In *Proc. of CP*, 10 2019.

[DFM] Jeffrey Dudek, Dror Fried, and Kuldeep S. Meel. Taming discrete integration via the boon of dimensionality. In *Proc. of NeurIPS*.

[DH07] Carmen Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30(1):565–620, 2007.

[DKLR00] Paul Dagum, Richard M. Karp, Michael Luby, and Sheldon Ross. An optimal algorithm for Monte Carlo estimation. *SIAM Journal on Computing*, 29(5):1484–1496, 2000.

[DOMPV17] Leonardo Duenas-Osorio, Kuldeep S Meel, Roger Paredes, and Moshe Y Vardi. Counting-based reliability estimation for power-transmission grids. In *Proc. of AAAI*, pages 4488–4494, 2017.

[EGSS13a] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Optimization with parity constraints: From binary codes to discrete integration. In *Proc. of UAI*, 2013.

[EGSS13b] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proc. of ICML*, pages 334–342, 2013.

[EGSS14] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Low-density parity constraints for hashing-based discrete integration. In *Proc. of ICML*, pages 271–279, 2014.

[FHO13] Robert Fink, Jiewen Huang, and Dan Olteanu. Anytime approximation in probabilistic databases. *The VLDB Journal*, 22(6):823–848, Dec 2013.

[FRS17] Daniel J Fremont, Markus N Rabe, and Sanjit A Seshia. Maximum model counting. In *Proc. of AAAI*, 2017.

[GHSS07a] Carla P Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *Proc. of IJCAI*, pages 2293–2299, 2007.

[GHSS07b] Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. Short XORs for model counting: From theory to practice. In *Proc. of SAT*, pages 100–106, 2007.

[Gra06] Robert M. Gray. *Toeplitz And Circulant Matrices: A Review (Foundations and Trends(R) in Communications and Information Theory)*. Now Publishers Inc., Hanover, MA, USA, 2006.

[GSS06] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proc. of AAAI*, volume 21, pages 54–61, 2006.

[GSS20] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of Satisfiability*. 2020.

[HDVZVM04] Marijn Heule, Mark Dufour, Joris Van Zwieten, and Hans Van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead sat solver. In *Proc. of SAT*, pages 345–359. Springer, 2004.

[HJ12] Cheng-Shen Han and Jie-Hong Roland Jiang. When Boolean satisfiability meets Gaussian elimination in a simplex way. In *Proc. of CAV*, pages 410–426. Springer, 2012.

[HJ19] Mark Huber and Bo Jones. Faster estimates of the mean of

bounded random variables. *Mathematics and Computers in Simulation*, 161:93–101, 2019.

[Hub17] Mark Huber. A Bernoulli mean estimate with known relative error distribution. *Random Structures and Algorithms*, 50(2):173–182, 2017.

[IMMV15] Alexander Ivrii, Sharad Malik, Kuldeep S. Meel, and Moshe Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, pages 1–18, 2015.

[JVV86] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43(2-3):169–188, 1986.

[KL83] Richard M. Karp and Michael Luby. Monte-carlo algorithms for enumeration and reliability problems. *Proc. of FOCS*, 1983.

[KL85] Richard M. Karp and Michael Luby. Monte-carlo algorithms for the planar multiterminal network reliability problem. *Journal of Complexity*, 1(1):45–64, 1985.

[KLM89] Richard M. Karp, Michael Luby, and Neal Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429 – 448, 1989.

[KMS+18] Samuel Kolb, Martin Mladenov, Scott Sanner, Vaishak Belle, and Kristian Kersting. Efficient symbolic integration for probabilistic inference. In *Proc. of IJCAI*, pages 5031–5037, 2018.

[LJN10] Tero Laitinen, Tommi A Junttila, and Ilkka Niemelä. Extending clause learning DPLL with parity reasoning. In *Proc. of ECAI*, volume 2010, pages 21–26, 2010.

[LJN11] Tero Laitinen, Tommi Junttila, and Ilkka Niemela. Equivalence class based parity reasoning with DPLL (XOR). In *Prof. of ICTAI*, pages 649–658. IEEE, 2011.

[LJN12] Tero Laitinen, Tommi Junttila, and Ilkka Niemelä. Conflict-driven XOR-clause learning. In *Proc. of SAT*, pages 383–396. Springer, 2012.

[LLM16] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving model counting by leveraging definability. In *Proc. of IJCAI*, pages 751–757, 2016.

[LM08] Jérôme Lang and Pierre Marquis. On propositional definability. *Artificial Intelligence*, 172(8-9):991–1017, 2008.

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[LSS17] Jingcheng Liu, Alistair Sinclair, and Piyush Srivastava. The Ising partition function: Zeros and deterministic approximation. *CoRR*, abs/1704.06493, 2017.

[LSSD14] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. *ACM SIGPLAN Notices*, 49(6):565–576, 2014.

[LWJ17] Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R Jiang. Solving stochastic Boolean satisfiability under Random-Exist quantifica-

tion. In *Proc. of IJCAI*, pages 688–694, 2017.

[MA20] Kuldeep S Meel and S Akshay. Sparse hashing for scalable approximate model counting: Theory and practice. In *Proc. of LICS*, pages 728–741, 2020.

[Mee17] Kuldeep S. Meel. *Constrained Counting and Sampling: Bridging the Gap between Theory and Practice*. PhD thesis, Rice University, 2017.

[MSV17] Kuldeep S. Meel, Aditya A. Shrotri, and Moshe Y. Vardi. On hashing-based approaches to approximate DNF-counting. In *Proc. of FSTTCS*, 12 2017.

[MSV19] Kuldeep S. Meel, Aditya A. Shrotri, and Moshe Y. Vardi. Not all FPRASs are equal: Demystifying FPRASs for DNF-counting. *Constraints*, 24(3-4):211–233, 2019.

[Mur68] Katta G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

[MVC+16] Kuldeep S Meel, Moshe Vardi, Supratik Chakraborty, Daniel J Fremont, Sanjit A Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. Constrained sampling and counting: Universal hashing meets SAT solving. In *Proc. of Beyond NP Workshop*, 2016.

[Nad10] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In *Proc. of FMCAD*, pages 221–229, 2010.

[NSM+19] Nina Narodytska, Aditya A. Shrotri, Kuldeep S. Meel, Alexey Ignatiev, and Joao Marques-Silva. Assessing heuristic machine learning explanations with model counting. In *Proc. of SAT*, pages 267–278, 2019.

[PJM19] Yash Pote, Saurabh Joshi, and Kuldeep S. Meel. Phase transition behavior of cardinality and XOR constraints. In *Proc. of IJCAI*, 8 2019.

[R70] Alfréd Rényi. *Probability Theory*. North Holland, Amsterdam, 1970.

[Rot96] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1):273–302, 1996.

[SGM20] Mate Soos, Stephan Gocht, and Kuldeep S Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Proc. of CAV*, pages 463–484. Springer, 2020.

[SM19] Mate Soos and Kuldeep S Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proc. of AAAI*, 2019.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT*, pages 244–257, 2009.

[Soo10] Mate Soos. Enhanced Gaussian elimination in DPLL-based SAT solvers. In *Proc. of Pragmatics of SAT Workshop*, pages 2–14, 2010.

[Sto83] Larry Stockmeyer. The complexity of approximate counting. In *Proc. of STOC*, pages 118–126, 1983.

[SVP+16] Somdeb Sarkhel, Deepak Venugopal, Tuan Anh Pham, Parag Singla, and Vibhav Gogate. Scalable training of Markov logic networks using approximate counting. In *Proc. of AAAI*, pages 1067–1073, 2016.

[Tod89] Seinosuke Toda. On the computational power of PP and (+)P. In *Proc. of FOCS*, pages 514–519. IEEE, 1989.

[Val79] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[Vaz13] Vijay V. Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[WS05] Wei Wei and Bart Selman. A new approach to model counting. In *Proc. of SAT*, pages 2293–2299. Springer, 2005.

[XCD12] Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. In *Proc. of AAAI*, 2012.

[ZCSE16] Shengjia Zhao, Sorathan Chaturapruek, Ashish Sabharwal, and Stefano Ermon. Closing the gap between short and long XORs for model counting. In *Proc. of AAAI*, 2016.

[ZQRZ18] Ziqiao Zhou, Zhiyun Qian, Michael K Reiter, and Yinqian Zhang. Static evaluation of noninterference using approximate model counting. In *Proc. of IEEE Symposium on Security and Privacy*, pages 514–528. IEEE, 2018.