Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls[*]

Supratik Chakraborty
Indian Institute of Technology, Bombay

Kuldeep S. Meel
Rice University

Moshe Y. Vardi
Rice University

Abstract

Probabilistic inference via model counting has emerged as a scalable technique with strong formal guarantees, thanks to recent advances in hashing-based approximate counting. State-of-the-art hashing-based counting algorithms use an NP oracle (SAT solver in practice), such that the number of oracle invocations grows linearly in the number of variables n in the input constraint. We present a new approach to hashing-based approximate model counting in which the number of oracle invocations grows logarithmically in n, while still providing strong theoretical guarantees. We use this technique to design an algorithm for #CNF with *strongly probably approximately correct* (SPAC) guarantees, i.e. PAC guarantee plus expected return value matching the exact model count. Our experiments show that this algorithm outperforms state-of-the-art techniques for approximate counting by 1-2 orders of magnitude in running time. We also show that our algorithm can be easily adapted to give a new fully polynomial randomized approximation scheme (FPRAS) for #DNF.

1 Introduction

Probabilistic inference is increasingly being used to reason about large uncertain data sets arising from diverse applications including medical diagnostics, weather modeling, computer vision and the like [Bacchus *et al.*, 2003; Domshlak and Hoffmann, 2007; Sang *et al.*, 2004; Xue *et al.*, 2012]. Given a probabilistic model describing conditional dependencies between variables in a system, the problem of probabilistic inference requires us to determine the probability of an event of interest, given observed evidence. This problem has been the subject of intense investigations by both theoreticians and practitioners for more than three decades (see [Koller and Friedman, 2009] for a nice survey).

Exact probabilistic inference is intractable due to the curse of dimensionality [Cooper, 1990; Roth, 1996]. As a result, researchers have studied approximate techniques to solve real-world instances of this problem. Techniques based on Markov Chain Monte Carlo (MCMC) methods [Brooks *et al.*, 2011], variational approximations [Wainwright and Jordan, 2008], interval propagation [Tessem, 1992] and randomized branching choices in combinatorial reasoning algorithms [Gogate and Dechter, 2007] scale to large problem instances; however they fail to provide rigorous approximation guarantees in practice [Ermon *et al.*, 2014; Kitchen and Kuehlmann, 2007].

A promising alternative approach to probabilistic inference is to reduce the problem to discrete integration or constrained counting, in which we count the models of a given set of constraints [Roth, 1996; Chavira and Darwiche, 2008]. While constrained counting is known to be computationally hard, recent advances in hashing-based techniques for approximate counting have revived a lot of interest in this approach. The use of universal hash functions in counting problems was first studied in [Sipser, 1983; Stockmeyer, 1983]. However, the resulting algorithms do not scale well in practice [Meel, 2014]. This leaves open the question of whether it is possible to design algorithms that simultaneously scale to large problem instances *and* provide strong theoretical guarantees for approximate counting. An important step towards resolving this question was taken in [Chakraborty *et al.*, 2013b], wherein a scalable approximate counter with rigorous approximation guarantees, named ApproxMC, was reported. In subsequent work [Ermon *et al.*, 2013a; Chakraborty *et al.*, 2014a; Belle *et al.*, 2015], this approach has been extended to finite-domain discrete integration, with applications to probabilistic inference.

Given the promise of hashing-based counting techniques in bridging the gap between scalability and providing rigorous guarantees for probabilistic inference, there have been several recent efforts to design efficient universal hash functions [Ivrii *et al.*, 2015; Chakraborty *et al.*, 2016a]. While these efforts certainly help push the scalability frontier of hashing-based techniques for probabilistic inference, the structure of the underlying algorithms has so far escaped critical examination. For example, all recent approaches to probabilistic inference via hashing-based counting use a linear search to identify the right values of parameters for the hash functions. As a result, the number of calls to the NP oracle (SAT solver in practice) increases linearly in the number of

[*]The author list has been sorted alphabetically by last name; this should not be used to determine the extent of authors' contributions.

variables, n, in the input constraint. Since SAT solver calls are by far the computationally most expensive steps in these algorithms [Meel *et al.*, 2016], this motivates us to ask: *Can we design a hashing-based approximate counting algorithm that requires sub-linear (in n) calls to the* SAT *solver, while providing strong theoretical guarantees?*

The primary contribution of this paper is a positive answer to the above question. We present a new hashing-based approximate counting algorithm, called ApproxMC2, for CNF formulas, that reduces the number of SAT solver calls *from linear in n to logarithmic in n.* Our algorithm provides SPAC, *strongly probably approximately correct*, guarantees; i.e., it computes a model count within a prescribed tolerance ε of the exact count, and with a prescribed confidence of at least $1-\delta$, while also ensuring that the expected value of the returned count matches the exact model count. We also show that for DNF formulas, ApproxMC2 gives a fully polynomial randomized approximation scheme (FPRAS), which differs fundamentally from earlier work [Karp *et al.*, 1989].

Since the design of recent probabilistic inference algorithms via hashing-based approximate counting can be broadly viewed as adaptations of ApproxMC [Chakraborty *et al.*, 2013b], we focus on ApproxMC as a paradigmatic representative, and show how ApproxMC2 improves upon it. Extensive experiments demonstrate that ApproxMC2 outperforms ApproxMC by 1-2 orders of magnitude in running time, when using the same family of hash functions. We also discuss how the framework and analysis of ApproxMC2 can be lifted to other hashing-based probabilistic inference algorithms [Chakraborty *et al.*, 2014a; Belle *et al.*, 2015]. Significantly, the algorithmic improvements of ApproxMC2 are orthogonal to recent advances in the design of hash functions [Ivrii *et al.*, 2015], permitting the possibility of combining ApproxMC2-style algorithms with efficient hash functions to boost the performance of hashing-based probabilistic inference even further.

The remainder of the paper is organized as follows. We describe notation and preliminaries in Section 2. We discuss related work in Section 3. In Section 4, we present ApproxMC2 and its analysis. We discuss our experimental methodology and present experimental results in Section 5. Finally, we conclude in Section 6.

2 Notation and Preliminaries

Let F be a Boolean formula in conjunctive normal form (CNF), and let $\mathsf{Vars}(F)$ be the set of variables appearing in F. The set $\mathsf{Vars}(F)$ is also called the *support* of F. An assignment σ of truth values to the variables in $\mathsf{Vars}(F)$ is called a *satisfying assignment* or *witness* of F if it makes F evaluate to true. We denote the set of all witnesses of F by R_F. Given a set of variables $S \subseteq \mathsf{Vars}(F)$, we use $R_{F\downarrow S}$ to denote the projection of R_F on S. Furthermore, given a function $h : \{0,1\}^{|\mathsf{Vars}(F)|} \rightarrow \{0,1\}^m$ and an $\alpha \in \{0,1\}^m$, we use $R_{\langle F,h,\alpha\rangle\downarrow S}$ to denote the projection on S of the witnesses of F that are mapped to α by h, i.e. $R_{(F\wedge(h(Y)=\alpha))\downarrow S}$, where Y is a vector of support variables of F.

We write $\mathsf{Pr}\left[X : \mathcal{P}\right]$ to denote the probability of outcome X when sampling from a probability space $\mathcal{P}$. For brevity, we omit $\mathcal{P}$ when it is clear from the context. The expected value of X is denoted $\mathsf{E}\left[X\right]$ and its variance is denoted $\mathsf{V}\left[X\right]$.

The *constrained counting problem* is to compute $|R_{F\downarrow S}|$ for a given CNF formula F and sampling set $S \subseteq \mathsf{Vars}(F)$. A *probably approximately correct* (or PAC) counter is a probabilistic algorithm $\mathsf{ApproxCount}(\cdot,\cdot,\cdot,\cdot)$ that takes as inputs a formula F, a sampling set S, a tolerance $\varepsilon > 0$, and a confidence $1-\delta \in (0,1]$, and returns a count c such that $\mathsf{Pr}\Big[|R_{F\downarrow S}|/(1+\varepsilon) \le c \le (1+\varepsilon)|R_{F\downarrow S}|\Big] \ge 1-\delta$. The probabilistic guarantee provided by a PAC counter is also called an (ε,δ) guarantee, for obvious reasons. A PAC counter that additionally ensures that the expected value of the returned count equals $|R_{F\downarrow S}|$ is called *strongly probably approximately correct* (or SPAC). Intuitively, SPAC is a more useful notion of approximation than PAC in the context of counting, since the expectation of the returned count equals $|R_{F\downarrow S}|$ for a SPAC counter.

For positive integers n and m, a special family of 2-universal hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$, called $H_{xor}(n,m)$, plays a crucial role in our work. Let $y[i]$ denote the i^{th} component of a vector y. The family $H_{xor}(n,m)$ can then be defined as $\{h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n} a_{i,k} \cdot y[k]), a_{i,k} \in \{0,1\}, 1 \le i \le m, 0 \le k \le n\}$, where $\oplus$ denotes "xor" and $\cdot$ denotes "and". By choosing values of $a_{i,k}$ randomly and independently, we can effectively choose a random hash function from $H_{xor}(n,m)$. It was shown in [Gomes *et al.*, 2007b] that $H_{xor}(n,m)$ is 3-universal (and hence 2-universal too). We use $h \xleftarrow{U} H_{xor}(n,m)$ to denote the probability space obtained by choosing a function h uniformly at random from $H_{xor}(n,m)$. The property of 2-universality guarantees that for all $\alpha_1,\alpha_2 \in \{0,1\}^m$ and for all distinct $y_1,y_2 \in \{0,1\}^n$, $\mathsf{Pr}\left[\bigwedge_{i=1}^{2} h(y_i) = \alpha_i \ : h \xleftarrow{U} H_{xor}(n,m)\right] = 2^{-2m}$. Note that ApproxMC [Chakraborty *et al.*, 2013b] also uses the same family of hash functions.

3 Related Work

The deep connection between probabilistic inference and propositional model counting was established in the seminal work of [Cooper, 1990; Roth, 1996]. Subsequently, researchers have proposed various encodings to solve inferencing problems using model counting [Sang *et al.*, 2004; Chavira and Darwiche, 2008; Chakraborty *et al.*, 2014a; Belle *et al.*, 2015; Chakraborty *et al.*, 2015b]. What distinguishes this line of work from other inferencing techniques, like those based on Markov Chain Monte Carlo methods [Jerrum and Sinclair, 1996] or variational approximation techniques [Wainwright and Jordan, 2008], is that strong guarantees can be offered while scaling to large problem instances. This has been made possible largely due to significant advances in model counting technology.

Complexity theoretic studies of propositional model counting were initiated by Valiant, who showed that the problem is #P-complete [Valiant, 1979]. Despite advances in exact model counting over the years [Sang *et al.*, 2004; Thurley, 2006], the inherent complexity of the problem poses significant hurdles to scaling exact counting to large problem instances. The study of approximate model counting has therefore been an important topic of research for several decades. Approximate counting was shown to lie in the third level of the polynomial hierarchy in [Stockmeyer, 1983]. For DNF formulas, Karp, Luby and Madras gave a fully polynomial randomized approximation scheme for counting models [Karp *et al.*, 1989]. For the general case, one can build on [Stockmeyer, 1983] and design a hashing-based probably approximately correct counting algorithm that makes polynomially many calls to an NP oracle [Goldreich, 1999]. Unfortunately, this does not lend itself to a scalable implementation because every invocation of the NP oracle (a SAT solver in practice) must reason about a formula with significantly large, viz. $\mathcal{O}(n/\varepsilon)$, support.

In [Chakraborty *et al.*, 2013b], a new hashing-based strongly probably approximately correct counting algorithm, called ApproxMC, was shown to scale to formulas with hundreds of thousands of variables, while providing rigorous PAC-style (ε, δ) guarantees. The core idea of ApproxMC is to use 2-universal hash functions to randomly partition the solution space of the original formula into "small" enough cells. The sizes of sufficiently many randomly chosen cells are then determined using calls to a specialized SAT solver (CryptoMiniSAT [Soos *et al.*, 2009]), and a scaled median of these sizes is used to estimate the desired model count. Finding the right parameters for the hash functions is crucial to the success of this technique. ApproxMC uses a linear search for this purpose, where each search step invokes the specialized SAT solver, viz. CryptoMiniSAT, $\mathcal{O}(1/\varepsilon^2)$ times. Overall, ApproxMC makes a total of $\mathcal{O}(\frac{n\log(1/\delta)}{\varepsilon^2})$ calls to CryptoMiniSAT. Significantly, and unlike the algorithm in [Goldreich, 1999], each call of CryptoMiniSAT reasons about a formula with only n variables.

The works of [Ermon *et al.*, 2013b; Chakraborty *et al.*, 2014a; 2015a; Belle *et al.*, 2015] have subsequently extended the ApproxMC approach to finite domain discrete integration. Furthermore, approaches based on ApproxMC form the core of various sampling algorithms proposed recently [Ermon *et al.*, 2013a; Chakraborty *et al.*, 2014b; 2014a; 2015a]. Therefore, any improvement in the core algorithmic structure of ApproxMC can potentially benefit several other algorithms.

Prior work on improving the scalability of hashing-based approximate counting algorithms has largely focused on improving the efficiency of 2-universal linear (xor-based) hash functions. It is well-known that long xor-based constraints make SAT solving significantly hard in practice [Gomes *et al.*, 2007a]. Researchers have therefore investigated theoretical and practical aspects of using short xors [Gomes *et al.*, 2007a; Chakraborty *et al.*, 2014b; Ermon *et al.*, 2014; Zhao *et al.*, 2016].

Recently, Ermon et al. [Ermon *et al.*, 2014] and Zhao et al. [Zhao *et al.*, 2016] have shown how short xor constraints (even logarithmic in the number of variables) can be used for approximate counting with certain theoretical guarantees. The resulting algorithms, however, do not provide PAC-style (ε, δ) guarantees. In other work with (ε, δ) guarantees, techniques for identifying small independent supports have been developed [Ivrii *et al.*, 2015], and word-level hash functions have been used to count in the theory of bitvectors [Chakraborty *et al.*, 2016a]. A common aspect of all of these approaches is that a linear search is used to find the right parameters of the hash functions, where each search step involves multiple SAT solver calls. We target this weak link in this paper, and drastically cut down the number of steps required to identify the right parameters of hash functions. This, in turn, reduces the SAT solver calls, yielding a scalable counting algorithm.

Algorithm 1 ApproxMC2$(F, S, \varepsilon, \delta)$

```
1: thresh ← $1 + 9.84\left(1+\frac{\varepsilon}{1+\varepsilon}\right)\left(1+\frac{1}{\varepsilon}\right)^2$;
2: $Y$ ← BSAT($F$, thresh, $S$);
3: if (|$Y$| < thresh) then return |$Y$|;
4: $t \leftarrow \lceil 17\log_2(3/\delta)\rceil$;
5: nCells ← 2; $C$ ← emptyList; iter ← 0;
6: repeat
7:     iter ← iter + 1;
8:     (nCells, nSols) ← ApproxMC2Core($F$, $S$, thresh, nCells);
9:     if (nCells ≠ ⊥) then AddToList($C$, nSols × nCells);
10: until (iter < $t$);
11: finalEstimate ← FindMedian($C$);
12: return finalEstimate
```

4 From Linear to Logarithmic SAT Calls

We now present ApproxMC2, a hashing-based approximate counting algorithm, that is motivated by ApproxMC, but also differs from it in crucial ways.

4.1 The Algorithm

Algorithm 1 shows the pseudocode for ApproxMC2. It takes as inputs a formula F, a sampling set S, a tolerance ε (> 0), and a confidence $1 - \delta \in (0, 1]$. It returns an estimate of $|R_{F\downarrow S}|$ within tolerance ε, with confidence at least $1 - \delta$. Note that although ApproxMC2 draws on several ideas from ApproxMC, the original algorithm in [Chakraborty *et al.*, 2013b] computed an estimate of $|R_F|$ (and not of $|R_{F\downarrow S}|$). Nevertheless, the idea of using sampling sets, as described in [Chakraborty *et al.*, 2014b], can be trivially extended to ApproxMC. Therefore, whenever we refer to ApproxMC in this paper, we mean the algorithm in [Chakraborty *et al.*, 2013b] extended in the above manner.

There are several high-level similarities between ApproxMC2 and ApproxMC. Both algorithms start by checking if $|R_{F\downarrow S}|$ is smaller than a suitable threshold (called pivot in ApproxMC and thresh in ApproxMC2). This check is done using subroutine BSAT, that takes as inputs a formula F, a threshold thresh, and a sampling set S, and returns a subset Y of $R_{F\downarrow S}$, such that $|Y| = \min(\mathsf{thresh}, |R_{F\downarrow S}|)$. The thresholds used in invocations of BSAT lie in $O(1/\varepsilon^2)$ in both ApproxMC and ApproxMC2, although the exact values used are different. If $|Y|$ is found to be less than thresh, both algorithms return $|Y|$ for the size of $|R_{F\downarrow S}|$. Otherwise, a core subroutine, called ApproxMCCore in ApproxMC and

ApproxMC2Core in ApproxMC2, is invoked. This subroutine tries to randomly partition $R_{F\downarrow S}$ into "small" cells using hash functions from $H_{xor}(|S|, m)$, for suitable values of m. There is a small probability that this subroutine fails and returns $(\bot, \bot)$. Otherwise, it returns the number of cells, nCells, into which $R_{F\downarrow S}$ is partitioned, and the count of solutions, nSols, in a randomly chosen small cell. The value of $|R_{F\downarrow S}|$ is then estimated as nCells $\times$ nSols. In order to achieve the desired confidence of $(1-\delta)$, both ApproxMC2 and ApproxMC invoke their core subroutine repeatedly, collecting the resulting estimates in a list C. The number of such invocations lies in $O(\log(1/\delta))$ in both cases. Finally, both algorithms compute the median of the estimates in C to obtain the desired estimate of $|R_{F\downarrow S}|$.

Despite these high-level similarities, there are key differences in the ways ApproxMC and ApproxMC2 work. These differences stem from: (i) the use of dependent hash functions when searching for the "right" way of partitioning $R_{F\downarrow S}$ within an invocation of ApproxMC2Core, and (ii) the lack of independence between successive invocations of ApproxMC2Core. We discuss these differences in detail below.

Subroutine ApproxMC2Core lies at the heart of ApproxMC2. Functionally, ApproxMC2Core serves the same purpose as ApproxMCCore; however, it works differently. To understand this difference, we briefly review the working of ApproxMCCore. Given a formula F and a sampling set S, ApproxMCCore finds a triple (m, h_m, α_m), where m is an integer in $\{1, \ldots |S|-1\}$, h_m is a hash function chosen randomly from $H_{xor}(|S|, m)$, and α_m is a vector chosen randomly from $\{0,1\}^m$, such that $|R_{\langle F, h_m, \alpha_m\rangle\downarrow S}| <$ thresh and $|R_{\langle F, h_{m-1}, \alpha_{m-1}\rangle\downarrow S}| \geq$ thresh. In order to find such a triple, ApproxMCCore uses a linear search: it starts from $m = 1$, chooses h_m and α_m randomly and independently from $H_{xor}(|S|, m)$ and $\{0,1\}^m$ respectively, and checks if $|R_{\langle F, h_m, \alpha_m\rangle\downarrow S}| \geq$ thresh. If so, the partitioning is considered too coarse, h_m and α_m are discarded, and the process repeated with the next value of m; otherwise, the search stops. Let m^*, h_{m^*} and α_{m^*} denote the values of m, h_m and α_m, respectively, when the search stops. Then ApproxMCCore returns $|R_{\langle F, h_{m^*}, \alpha_{m^*}\rangle\downarrow S}| \times 2^{m^*}$ as the estimate of $|R_{F\downarrow S}|$. If the search fails to find m, h_m and α_m with the desired properties, we say that ApproxMCCore fails.

Every iteration of the linear search above invokes BSAT once to check if $|R_{\langle F, h_m, \alpha_m\rangle\downarrow S}| \geq$ thresh. A straightforward implementation of BSAT makes up to thresh calls to a SAT solver to answer this question. Therefore, an invocation of ApproxMCCore makes $\mathcal{O}(\text{thresh}.|S|)$ SAT solver calls. A key contribution of this paper is a new approach for choosing hash functions that allows ApproxMC2Core to make at most $\mathcal{O}(\text{thresh}.\log_2|S|)$ calls to a SAT solver. Significantly, the sizes of formulas fed to the solver remain the same as those used in ApproxMCCore; hence, the reduction in number of calls comes without adding complexity to the individual calls.

A salient feature of ApproxMCCore is that it randomly and independently chooses (h_m, α_m) pairs for different values of m, as it searches for the right partitioning of $R_{F\downarrow S}$. In contrast, in ApproxMC2Core, we randomly choose one function h from $H_{xor}(|S|, |S|-1)$, and one vector α from $\{0,1\}^{|S|-1}$. Thereafter, we use "prefix-slices" of h and α to obtain h_m and α_m for all other values of m. Formally, for every $m \in \{1, \ldots |S|-1\}$, the mth prefix-slice of h, denoted $h^{(m)}$, is a map from $\{0,1\}^{|S|}$ to $\{0,1\}^m$, such that $h^{(m)}(y)[i] = h(y)[i]$, for all $y \in \{0,1\}^{|S|}$ and for all $i \in \{1, \ldots m\}$. Similarly, the mth prefix-slice of α, denoted $\alpha^{(m)}$, is an element of $\{0,1\}^m$ such that $\alpha^{(m)}[i] = \alpha[i]$ for all $i \in \{1, \ldots m\}$. Once h and α are chosen randomly, ApproxMC2Core uses $h^{(m)}$ and $\alpha^{(m)}$ as choices of h_m and α_m, respectively. The randomness in the choices of h and α induces randomness in the choices of h_m and α_m. However, the (h_m, α_m) pairs chosen for different values of m are no longer independent. Specifically, $h_j(y)[i] = h_k(y)[i]$ and $\alpha_j[i] = \alpha_k[i]$ for $1 \leq j < k < |S|$ and for all $i \in \{1, \ldots j\}$. This lack of independence is a fundamental departure from ApproxMCCore.

Algorithm 2 ApproxMC2Core(F, S, thresh, prevNCells)

```
1: Choose $h$ at random from $H_{xor}(|S|, |S|-1)$;
2: Choose $\alpha$ at random from $\{0,1\}^{|S|-1}$;
3: $Y \leftarrow$ BSAT($F \wedge h(S) = \alpha$, thresh, $S$);
4: **if** ($|Y| \geq$ thresh) **then return** $(\bot, \bot)$;
5: mPrev $\leftarrow \log_2$ prevNCells;
6: $m \leftarrow$ LogSATSearch($F, S, h, \alpha$, thresh, mPrev);
7: nSols $\leftarrow$ |BSAT($F \wedge h^{(m)}(S) = \alpha^{(m)}$, thresh, $S$)|;
8: **return** ($2^m$, nSols);
```

Algorithm 2 shows the pseudo-code for ApproxMC2Core. After choosing h and α randomly, ApproxMC2Core checks if $|R_{\langle F, h, \alpha\rangle\downarrow S}| <$ thresh. If not, ApproxMC2Core fails and returns $(\bot, \bot)$. Otherwise, it invokes sub-routine LogSATSearch to find a value of m (and hence, of $h^{(m)}$ and $\alpha^{(m)}$) such that $|R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle\downarrow S}| <$ thresh and $|R_{\langle F, h^{(m-1)}, \alpha^{(m-1)}\rangle\downarrow S}| \geq$ thresh. This ensures that nSols computed in line 7 is $|R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle\downarrow S}|$. Finally, ApproxMC2Core returns (2^m, nSols), where 2^m gives the number of cells into which $R_{F\downarrow S}$ is partitioned by $h^{(m)}$.

An easy consequence of the definition of prefix-slices is that for all $m \in \{1, \ldots |S|-1\}$, we have $R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle\downarrow S} \subseteq R_{\langle F, h^{(m-1)}, \alpha^{(m-1)}\rangle\downarrow S}$. This linear ordering is exploited by sub-routine LogSATSearch (see Algorithm 3), which uses a galloping search to zoom down to the right value of m, $h^{(m)}$ and $\alpha^{(m)}$. LogSATSearch uses an array, BigCell, to remember values of m for which the cell $\alpha^{(m)}$ obtained after partitioning $R_{F\downarrow S}$ with $h^{(m)}$ is large, i.e. $|R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle\downarrow S}| \geq$ thresh. As boundary conditions, we set BigCell[0] to 1 and BigCell[$|S|-1$] to 0. These are justified because (i) if $R_{F\downarrow S}$ is partitioned into 2^0 (i.e. 1) cell, line 3 of Algorithm 1 ensures that the size of the cell (i.e. $|R_{F\downarrow S}|$) is at least thresh, and (ii) line 4 of Algorithm 2 ensures that $|R_{\langle F, h^{|S|-1}, \alpha^{|S|-1}\rangle\downarrow S}| <$ thresh. For every other i, BigCell[i] is initialized to $\bot$ (unknown value). Subsequently, we set BigCell[i] to 1 (0) whenever we find that $|R_{\langle F, h^{(i)}, \alpha^{(i)}\rangle\downarrow S}|$ is at least as large as (smaller than) thresh.

In the context of probabilistic hashing-based counting algorithms like ApproxMC, it has been observed [Meel, 2014]

Algorithm 3 $\mathsf{LogSATSearch}(F, S, h, \alpha, \mathsf{thresh}, \mathsf{mPrev})$

```
1: loIndex ← $0$; hiIndex ← $|S| - 1$; $m$ ← mPrev;
2: BigCell[$0$] ← $1$; BigCell[$|S| - 1$] ← $0$;
3: BigCell[$i$] ← $\perp$ for all $i$ other than 0 and $|S| - 1$;
4: while true do
5:     $Y \leftarrow \mathsf{BSAT}(F \wedge (h^{(m)}(S) = \alpha^{(m)}), \mathsf{thresh}, S)$;
6:     if ($|Y| \geq$ thresh) then
7:         if (BigCell[$m + 1$] $= 0$) then return $m + 1$;
8:         BigCell[$i$] ← $1$ for all $i \in \{1, \dots m\}$;
9:         loIndex ← $m$;
10:        if ($|m - \mathsf{mPrev}| < 3$) then $m \leftarrow m + 1$;
11:        else if ($2.m < |S|$) then $m \leftarrow 2.m$;
12:        else $m \leftarrow (\mathsf{hiIndex} + m)/2$;
13:    else
14:        if (BigCell[$m - 1$] $= 1$) then return $m$;
15:        BigCell[$i$] ← $0$ for all $i \in \{m, \dots |S|\}$;
16:        hiIndex ← $m$;
17:        if ($|m - \mathsf{mPrev}| < 3$) then $m \leftarrow m - 1$;
18:        else $m \leftarrow (m + \mathsf{loIndex})/2$;
```

that the "right" values of m, h_m and α_m for partitioning $R_{F\downarrow S}$ are often such that m is closer to 0 than to $|S|$. In addition, repeated invocations of a hashing-based probabilistic counting algorithm with the same input formula F often terminate with similar values of m. To optimize LogSATSearch using these observations, we provide mPrev, the value of m found in the last invocation of ApproxMC2Core, as an input to LogSATSearch. This is then used in LogSATSearch to linearly search a small neighborhood of mPrev, viz. when $|m - \mathsf{mPrev}| < 3$, before embarking on a galloping search. Specifically, if LogSATSearch finds that $|R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle \downarrow S}| \geq$ thresh after the linear search, it keeps doubling the value of m until either $|R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle \downarrow S}|$ becomes less than thresh, or m overshoots $|S|$. Subsequently, binary search is done by iteratively bisecting the interval between loIndex and hiIndex. This ensures that the search requires $\mathcal{O}(\log_2 m^*)$ calls (instead of $\mathcal{O}(\log_2 |S|)$ calls) to BSAT, where m^* (usually $\ll |S|$) is the value of m when the search stops. Note also that a galloping search inspects much smaller values of m compared to a naive binary search, if $m^* \ll |S|$. Therefore, the formulas fed to the SAT solver have fewer xor clauses (or number of components of $h^{(m)}$) conjoined with F than if a naive binary search was used. This plays an important role in improving the performance of ApproxMC2.

In order to provide the right value of mPrev to LogSATSearch, ApproxMC2 passes the value of nCells returned by one invocation of ApproxMC2Core to the next invocation (line 8 of Algorithm 1), and ApproxMC2Core passes on the relevant information to LogSATSearch (lines 5–6 of Algorithm 2). Thus, successive invocations of ApproxMC2Core in ApproxMC2 are *no longer independent* of each other. Note that the independence of randomly chosen (h_m, α_m) pairs for different values of m, and the independence of successive invocations of ApproxMCCore, are features of ApproxMC that are exploited in its analysis [Chakraborty *et al.*, 2013b]. Since these independence no longer hold in ApproxMC2, we must analyze ApproxMC2 afresh.

4.2 Analysis

For lack of space, we state only some key results and provide sketches of proofs here. The complete proofs are deferred to a detailed technical report [Chakraborty *et al.*, 2016b].

Lemma 1. *For $1 \leq i < |S|$, let $\mu_i = R_{F\downarrow S}/2^i$. For every $\beta > 0$ and $0 < \varepsilon < 1$, we have the following:*

1. $\Pr\left[\left||R_{\langle F, h^{(i)}, \alpha^{(i)}\rangle \downarrow S}| - \mu_i\right| \geq \frac{\varepsilon}{1+\varepsilon}\mu_i\right] \leq \frac{(1+\varepsilon)^2}{\varepsilon^2 \mu_i}$
2. $\Pr\left[|R_{\langle F, h^{(i)}, \alpha^{(i)}\rangle \downarrow S}| \leq \beta\mu_i\right] \leq \frac{1}{1+(1-\beta)^2\mu_i}$

Proof sketch. For every $y \in \{0,1\}^{|S|}$ and for every $\alpha \in \{0,1\}^i$, define an indicator variable $\gamma_{y,\alpha,i}$ which is 1 iff $h^{(i)}(y) = \alpha$. Let $\Gamma_{\alpha,i} = \sum_{y \in R_{F\downarrow S}} (\gamma_{y,\alpha,i})$, $\mu_{\alpha,i} = \mathsf{E}[\Gamma_{\alpha,i}]$ and $\sigma^2_{\alpha,i} = \mathsf{V}[\Gamma_{\alpha,i}]$. Clearly, $\Gamma_{\alpha,i} = |R_{\langle F, h^{(i)}, \alpha\rangle \downarrow S}|$ and $\mu_{\alpha,i} = 2^{-i}|R_{F\downarrow S}|$. Note that $\mu_{\alpha,i}$ is independent of α and equals μ_i, as defined in the statement of the Lemma. From the pairwise independence of $h^{(i)}(y)$ (which, effectively, is a randomly chosen function from $H_{xor}(|S|, i)$), we also have $\sigma^2_{\alpha,i} \leq \mu_{\alpha,i} = \mu_i$. Statements 1 and 2 of the lemma then follow from Chebhyshev inequality and Paley-Zygmund inequality, respectively. □

Let B denote the event that ApproxMC2Core either returns $(\perp, \perp)$ or returns a pair $(2^m, \mathsf{nSols})$ such that $2^m \times$ nSols does not lie in the interval $\left[\frac{|R_{F\downarrow S}|}{1+\varepsilon}, |R_{F\downarrow S}|(1+\varepsilon)|\right]$. We wish to bound $\Pr[B]$ from above. Towards this end, let T_i denote the event $\left(|R_{\langle F, h^{(i)}, \alpha^{(i)}\rangle \downarrow S}| < \mathsf{thresh}\right)$, and let L_i and U_i denote the events $\left(|R_{\langle F, h^{(i)}, \alpha^{(i)}\rangle \downarrow S}| < \frac{|R_{F\downarrow S}|}{(1+\varepsilon)2^i}\right)$ and $\left(|R_{\langle F, h^{(i)}, \alpha^{(i)}\rangle \downarrow S}| > \frac{|R_{F\downarrow S}|}{2^i}(1 + \frac{\varepsilon}{1+\varepsilon})\right)$, respectively. Furthermore, let m^* denote the integer $\lfloor \log_2 |R_{F\downarrow S}| - \log_2\left(4.92\left(1 + \frac{1}{\varepsilon}\right)^2\right)\rfloor$.

Lemma 2. *The following bounds hold:*

1. $\Pr[T_{m^*-3}] \leq \frac{1}{62.5}$
2. $\Pr[L_{m^*-2}] \leq \frac{1}{20.68}$
3. $\Pr[L_{m^*-1}] \leq \frac{1}{10.84}$
4. $\Pr[L_{m^*} \cup U_{m^*}] \leq \frac{1}{4.92}$

The proofs follow from the definitions of m^*, thresh, μ_i, and from applications of Lemma 1 with appropriate values of β.

Lemma 3. $\Pr[B] \leq 0.36$

Proof sketch. For any event E, let $\overline{E}$ denote its complement. For notational convenience, we use T_0 and $U_{|S|}$ to denote the empty (or impossible) event, and $T_{|S|}$ and $L_{|S|}$ to denote the universal (or certain) event. It then follows from the definition of B that $\Pr[B] \leq \Pr\left[\bigcup_{i\in\{1,\dots|S|\}} \left(\overline{T_{i-1}} \cap T_i \cap (L_i \cup U_i)\right)\right]$.

We now wish to simplify the upper bound of $\Pr[B]$ obtained above. In order to do this, we use three observations, labeled O1, O2 and O3 below, which follow from the definitions of m^*, thresh and μ_i, and from the linear ordering of $R_{\langle F,h^{(m)},\alpha^{(m)}\rangle\downarrow S}$.

O1: $\forall i \leq m^* - 3$, $T_i \cap (L_i \cup U_i) = T_i$ and $T_i \subseteq T_{m^*-3}$,

O2: $\Pr[\bigcup_{i\in\{m^*,\ldots|S|\}} \overline{T_{i-1}} \cap T_i \cap (L_i \cup U_i)] \leq \Pr[\overline{T_{m^*-1}} \cap (L_{m^*} \cup U_{m^*})] \leq \Pr[L_{m^*} \cup U_{m^*}]$,

O3: For $i \in \{m^*-2, m^*-1\}$, since $\mathsf{thresh} \leq \mu_i(1+\frac{\varepsilon}{1+\varepsilon})$, we have $T_i \cap U_i = \emptyset$.

Using O1, O2 and O3, we get $\Pr[B] \leq \Pr[T_{m^*-3}] + \Pr[L_{m^*-2}] + \Pr[L_{m^*-1}] + \Pr[L_{m^*} \cup U_{m^*}]$. Using the bounds from Lemma 2, we finally obtain $\Pr[B] \leq 0.36$. □

Note that Lemma 3 holds regardless of the order in which the search in LogSATSearch proceeds. Our main theorem now follows from Lemma 3 and from the count t of invocations of ApproxMC2Core in ApproxMC2 (see lines 4-10 of Algorithm 1).

Theorem 4. *Suppose* ApproxMC2$(F, S, \varepsilon, \delta)$ *returns c after making k calls to a* SAT *solver. Then* $\Pr[|R_{F\downarrow S}|/(1+\varepsilon) \leq c \leq (1+\varepsilon)|R_{F\downarrow S}|] \geq 1-\delta$, *and* $k \in \mathcal{O}(\frac{\log(|S|)\log(1/\delta)}{\varepsilon^2})$. *Furthermore,* $\mathsf{E}[c] = |R_{F\downarrow S}|$. *Hence* ApproxMC2 *is a* SPAC *counter.*

Note that the number of SAT solver calls in ApproxMC [Chakraborty *et al.*, 2013b] lies in $\mathcal{O}(\frac{|S|\log(1/\delta)}{\varepsilon^2})$, which is exponentially worse than the number of calls in ApproxMC2, for the same ε and δ. Furthermore, if the formula F fed as input to ApproxMC2 is in DNF, the subroutine BSAT can be implemented in PTIME, since satisfiability checking of DNF + XOR is in PTIME. This gives us the following result.

Theorem 5. ApproxMC2 *is a fully polynomial randomized approximation scheme (FPRAS) for* #DNF.

Note that this is fundamentally different from FPRAS for $\#DNF$ described in earlier work, viz. [Karp *et al.*, 1989].

4.3 Generalizing beyond ApproxMC

So far, we have shown how ApproxMC2 significantly reduces the number of SAT solver calls vis-a-vis ApproxMC, without sacrificing theoretical guarantees, by relaxing independence requirements. Since ApproxMC serves as a paradigmatic representative of several hashing-based counting and probabilistic inference algorithms, the key ideas of ApproxMC2 can be used to improve these other algorithms too. We discuss two such cases below.

PAWS [Ermon *et al.*, 2013a] is a hashing-based sampling algorithm for high dimensional probability spaces. Similar to ApproxMC, the key idea of PAWS is to find the "right" number and set of constraints that divides the solution space into appropriately sized cells. To do this, PAWS iteratively adds independently chosen constraints, using a linear search. An analysis of the algorithm in [Ermon *et al.*, 2013a] shows that this requires $\mathcal{O}(n \log n)$ calls to an NP oracle, where n denotes the size of the support of the input constraint. Our approach based on dependent constraints can be used in PAWS to search out-of-order, and reduce the number of NP oracle calls from $\mathcal{O}(n \log n)$ to $\mathcal{O}(\log n)$, while retaining the same theoretical guarantees.

Building on ApproxMC, a weighted model counter called WeightMC was proposed in [Chakraborty *et al.*, 2014a]. WeightMC has also been used in other work, viz. [Belle *et al.*, 2015], for approximate probabilistic inference. The core procedure of WeightMC, called WeightMCCore, is a reworking of ApproxMCCore that replaces $|R_{F\downarrow S}|$ with the total weight of assignments in $R_{F\downarrow S}$. It is easy to see that the same replacement can also be used to extend ApproxMC2Core, so that it serves as the core procedure for WeightMC.

5 Evaluation

To evaluate the runtime performance and quality of approximations computed by ApproxMC2, we implemented a prototype in C++ and conducted experiments on a wide variety of publicly available benchmarks. Specifically, we sought answers to the following questions: (a) How does runtime performance and number of SAT invocations of ApproxMC2 compare with that of ApproxMC ? (b) How far are the counts computed by ApproxMC2 from the exact counts?

Our benchmark suite consisted of problems arising from probabilistic inference in grid networks, synthetic grid-structured random interaction Ising models, plan recognition, DQMR networks, bit-blasted versions of SMTLIB benchmarks, ISCAS89 combinational circuits, and program synthesis examples. For lack of space, we discuss results for only a subset of these benchmarks here. The complete set of experimental results and a detailed analysis can be found in [Chakraborty *et al.*, 2016b].

We used a high-performance cluster to conduct experiments in parallel. Each node of the cluster had a 12-core 2.83 GHz Intel Xeon processor, with 4GB of main memory, and each experiment was run on a single core. For all our experiments, we used $\varepsilon = 0.8$ and $\delta = 0.2$, unless stated otherwise. To further optimize the running time, we used improved estimates of the iteration count t required in ApproxMC2 by following an analysis similar to that in [Chakraborty *et al.*, 2013a].

5.1 Results

Performance comparison: Table 1 presents the performance of ApproxMC2 vis-a-vis ApproxMC over a subset of our benchmarks. Column 1 of this table gives the benchmark name, while columns 2 and 3 list the number of variables and clauses, respectively. Columns 4 and 5 list the runtime (in seconds) of ApproxMC2 and ApproxMC respectively, while columns 6 and 7 list the number of SAT invocations for ApproxMC2 and ApproxMC respectively. We use "–" to denote timeout after 8 hours. Table 1 clearly demonstrates that ApproxMC2 outperforms ApproxMC by 1-2 orders of magnitude. Furthermore, ApproxMC2 is able to compute counts for benchmarks that are beyond the scope of ApproxMC. The runtime improvement of ApproxMC2 can be largely attributed to the reduced (by almost an order of magnitude) number of SAT solver calls vis-a-vis ApproxMC.

There are some large benchmarks in our suite for which both ApproxMC and ApproxMC2 timed out; hence, we did

Benchmark	Vars	Clauses	ApproxMC2 Time	ApproxMC Time	ApproxMC2 SATCalls	ApproxMC SATCalls
tutorial3	486193	2598178	12373.99	–	1744	–
case204	214	580	166.2	–	1808	–
case205	214	580	300.11	–	1793	–
case133	211	615	18502.44	–	2043	–
s953a_15_7	602	1657	161.41	–	1648	–
llreverse	63797	257657	1938.1	4482.94	1219	2801
lltraversal	39912	167842	151.33	450.57	1516	4258
karatsuba	19594	82417	23553.73	28817.79	1378	13360
enqueueSeqSK	16466	58515	192.96	2036.09	2207	23321
progsyn_20	15475	60994	1778.45	20557.24	2308	34815
progsyn_77	14535	27573	88.36	1529.34	2054	24764
sort	12125	49611	209.0	3610.4	1605	27731
LoginService2	11511	41411	26.04	110.77	1533	10653
progsyn_17	10090	27056	100.76	4874.39	1810	28407
progsyn_29	8866	31557	87.78	3569.25	1712	28630
LoginService	8200	26689	21.77	101.15	1498	12520
doublyLinkedList	6890	26918	17.05	75.45	1615	10647

Table 1: Performance comparison of ApproxMC2 vis-a-vis ApproxMC. The runtime is reported in seconds and "–" in a column reports timeout after 8 hours.

not include these in Table 1. Importantly, for a significant number of our experiments, whenever ApproxMC or ApproxMC2 timed out, it was because the algorithm could execute *some, but not all* required iterations of ApproxMCCore or ApproxMC2Core, respectively, within the specified time limit. In all such cases, we obtain a model count within the specified tolerance, but with reduced confidence. This suggests that it is possible to extend ApproxMC2 to obtain an anytime algorithm. This is left for future work.

Approximation quality: To measure the quality of approximation, we compared the approximate counts returned by ApproxMC2 with the counts computed by an exact model counter, viz. sharpSAT [Thurley, 2006]. Figure 1 shows the model counts computed by ApproxMC2, and the bounds obtained by scaling the exact counts with the tolerance factor ($\varepsilon = 0.8$) for a small subset of benchmarks. The y-axis represents model counts on log-scale while the x-axis represents benchmarks ordered in ascending order of model counts. We observe that for *all* the benchmarks, ApproxMC2 computed counts within the tolerance. Furthermore, for each instance, the observed tolerance (ε_{obs}) was calculated as $\max(\frac{\text{AprxCount}}{|R_{F\downarrow S}|} - 1, 1 - \frac{|R_{F\downarrow S}|}{\text{AprxCount}})$, where AprxCount is the estimate computed by ApproxMC2. We observe that the geometric mean of ε_{obs} across all benchmarks is 0.021 – far better than the theoretical guarantee of 0.8. In comparison, the geometric mean of the observed tolerance obtained from ApproxMC running on the same set of benchmarks is 0.036.

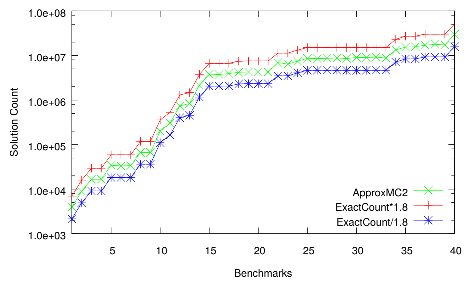

Figure 1: Quality of counts computed by ApproxMC2

6 Conclusion

The promise of scalability with rigorous guarantees has renewed interest in hashing-based counting techniques for probabilistic inference. In this paper, we presented a new approach to hashing-based counting and inferencing, that allows out-of-order-search with dependent hash functions, dramatically reducing the number of SAT solver calls from linear to logarithmic in the size of the support of interest. This is achieved while retaining strong theoretical guarantees and without increasing the complexity of each SAT solver call. Extensive experiments demonstrate the practical benefits of our approach vis-a-vis state-of-the art techniques. Combining our approach with more efficient hash functions promises to push the scalability horizon of approximate counting further.

Acknowledgements

The authors thank Jeffrey Dudek for several helpful discussions and comments on early draft of this paper. This work was supported in part by NSF grants IIS-1527668, CCF-1139011, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by BSF grant 9800096, by a grant from the Board of Research in Nuclear Sciences, India, Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097. Part of this work was done during a visit to the Israeli Institute for Advanced Studies.

References

[Bacchus *et al.*, 2003] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proc. of FOCS*, pages 340–351, 2003.

[Belle *et al.*, 2015] V. Belle, G. Van den Broeck, and A. Passerini. Hashing-based approximate probabilistic inference in hybrid domains. In *Proc. of UAI*, 2015.

[Brooks *et al.*, 2011] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng. *Handbook of Markov Chain Monte Carlo*. Chapman & Hall/CRC, 2011.

[Chakraborty *et al.*, 2013a] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable and nearly uniform generator of SAT witnesses. In *Proc. of CAV*, pages 608–623, 2013.

[Chakraborty *et al.*, 2013b] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Proc. of CP*, pages 200–216, 2013.

[Chakraborty *et al.*, 2014a] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In *Proc. of AAAI*, pages 1722–1730, 2014.

[Chakraborty *et al.*, 2014b] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, pages 1–6, 2014.

[Chakraborty *et al.*, 2015a] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi. On parallel scalable uniform sat witness generation. In *Proc. of TACAS*, pages 304–319, 2015.

[Chakraborty *et al.*, 2015b] S. Chakraborty, D. Fried, K. S. Meel, and M. Y. Vardi. From weighted to unweighted model counting. In *Proceedings of AAAI*, pages 689–695, 2015.

[Chakraborty *et al.*, 2016a] S. Chakraborty, K. S. Meel, R. Mistry, and M. Y. Vardi. Approximate probabilistic inference via word-level counting. In *Proc. of AAAI*, 2016.

[Chakraborty *et al.*, 2016b] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. Technical report, Department of Computer Science, Rice University, 2016.

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

[Cooper, 1990] G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990.

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

[Ermon *et al.*, 2013a] S. Ermon, C. Gomes, A. Sabharwal, and B. Selman. Embed and project: Discrete sampling with universal hashing. In *Proc. of NIPS*, pages 2085–2093, 2013.

[Ermon *et al.*, 2013b] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman. Optimization with parity constraints: From binary codes to discrete integration. In *Proc. of UAI*, 2013.

[Ermon *et al.*, 2014] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman. Low-density parity constraints for hashing-based discrete integration. In *Proc. of ICML*, pages 271–279, 2014.

[Gogate and Dechter, 2007] V. Gogate and R. Dechter. Approximate counting by sampling the backtrack-free search space. In *Proc. of the AAAI*, volume 22, page 198, 2007.

[Goldreich, 1999] O. Goldreich. The Counting Class #P. Lecture notes of course on ”Introduction to Complexity Theory”, Weizmann Institute of Science, 1999.

[Gomes *et al.*, 2007a] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. Short XORs for Model Counting; From Theory to Practice. In *SAT*, pages 100–106, 2007.

[Gomes *et al.*, 2007b] C. Gomes, A. Sabharwal, and B. Selman. Near-uniform sampling of combinatorial spaces using XOR constraints. In *Proc. of NIPS*, pages 670–676, 2007.

[Ivrii *et al.*, 2015] A. Ivrii, S. Malik, K. S. Meel, and M. Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, pages 1–18, 2015.

[Jerrum and Sinclair, 1996] M. Jerrum and A. Sinclair. The Markov Chain Monte Carlo method: an approach to approximate counting and integration. *Approximation algorithms for NP-hard problems*, pages 482–520, 1996.

[Karp *et al.*, 1989] R. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429–448, 1989.

[Kitchen and Kuehlmann, 2007] N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *Proc. of ICCAD*, pages 258–265, 2007.

[Koller and Friedman, 2009] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT press, 2009.

[Meel *et al.*, 2016] K. S. Meel, M. Vardi, S. Chakraborty, D. J. Fremont, S. A. Seshia, D. Fried, A. Ivrii, and S. Malik. Constrained sampling and counting: Universal hashing meets sat solving. In *Proc. of Beyond NP Workshop*, 2016.

[Meel, 2014] K. S. Meel. *Sampling Techniques for Boolean Satisfiability*. 2014. M.S. Thesis, Rice University.

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

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

[Sipser, 1983] M. Sipser. A complexity theoretic approach to randomness. In *Proc. of STOC*, pages 330–335, 1983.

[Soos *et al.*, 2009] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *Proc. of SAT*. Springer-Verlag, 2009.

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

[Tessem, 1992] B. Tessem. Interval probability propagation. *International Journal of Approximate Reasoning*, 7(3–5):95–120, 1992.

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

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

[Wainwright and Jordan, 2008] M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. *Found. Trends Machine Learning*, 1(1-2):1–305, 2008.

[Xue *et al.*, 2012] Y. Xue, A. Choi, and A. Darwiche. Basing decisions on sentences in decision diagrams. In *Proc. of AAAI*, 2012.

[Zhao *et al.*, 2016] S. Zhao, S. Chaturapruek, A. Sabharwal, and S. Ermon. Closing the gap between short and long xors for model counting. In *Proc. of AAAI (to appear)*, 2016.