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 \#\text{CNF} with strongly probably approximately correct (SPAC) guarantees, i.e., PAC guarantees 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 fully polynomial randomized approximation scheme (FPRAS) for \#\text{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 [Mee 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 \( \text{linear in } n \) to logarithmic in \( n \). Our algorithm provides SPAC, strongly probably approximately correct, guarantees; i.e., it computes a model count within a prescribed tolerance \( \varepsilon \) 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 \( \text{Vars}(F) \) be the set of variables appearing in \( F \). The set \( \text{Vars}(F) \) is also called the support of \( F \). An assignment \( \sigma \) of truth values to the variables in \( \text{Vars}(F) \) 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 \( \text{Wts}(F) \), and \( \text{Vars}(7) \) the projection of \( \text{Wts}(F) \) on \( S \). Furthermore, given a function \( h : \{0, 1\}^{\text{Vars}(F)} \rightarrow \{0, 1\}^m \) and an \( \alpha \in \{0, 1\}^m \), we use \( \text{Wts}(F, h, \alpha) \) to denote the projection of \( \text{Wts}(F) \) on \( S \) of the witnesses of \( F \) that are mapped to \( \alpha \) by \( h \), i.e., \( \text{Wts}(F, h, \alpha) = \{ y \in \{0, 1\}^m \mid h(y) = \alpha \} \). We write \( \Pr[X \in \mathcal{P}] \) 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 \( E[X] \) and its variance is denoted \( V[X] \).

The constrained counting problem is to compute \(|R_{F|S}|\) for a given CNF formula \( F \) and sampling set \( S \subseteq \text{Vars}(F) \). A probably approximately correct (or PAC) algorithm is a probabilistic algorithm \( \text{ApproxCount}((\cdot, \cdot, \cdot)) \) 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 \( \Pr[|R_{F|S}|/\left(1+\varepsilon\right) \leq c \leq (1+\varepsilon)|R_{F|S}|] \geq 1-\delta \). The probabilistic guarantee provided by a PAC counter is also called an \((\varepsilon, \delta)\) guarantee, for obvious reasons. A PAC counter that additionally ensures that the expected value of the returned count equals \(|R_{F|S}|\) 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 \(|R_{F|S}|\) 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 \( H_{\text{xor}}(n,m) \), plays a crucial role in our work. Let \( y[i] \) denote the \( i \)-th component of a vector \( y \). The family \( H_{\text{xor}}(n,m) \) can then be defined as \( \{ h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{m} a_{i,k} \cdot y[k]), a_{i,k} \in \{0, 1\}, 1 \leq i \leq m, 0 \leq k \leq n \} \), 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 \( H_{\text{xor}}(n,m) \). It was shown in [Gomes et al., 2007b] that \( H_{\text{xor}}(n,m) \) is 3-universal (and hence 2-universal too). We use \( h \overset{\mathcal{U}}{\leftarrow} H_{\text{xor}}(n,m) \) to denote the probability space obtained by choosing a function \( h \) uniformly at random from \( H_{\text{xor}}(n,m) \). The property of 2-universality guarantees that for all \( \alpha_1, \alpha_2 \in \{0, 1\}^m \) and for all distinct \( y_1, y_2 \in \{0, 1\}^n \), \( \Pr \left[ \bigwedge_{i=1}^{m} h(y_i) = \alpha_i \mid h \overset{\mathcal{U}}{\leftarrow} H_{\text{xor}}(n,m) \right] = 2^{-2m} \). 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., 2015]. 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
significantly hurdle 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. $O(n/\varepsilon)$, 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 $(\varepsilon, \delta)$ 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, $O(1/\varepsilon^2)$ times. Overall, ApproxMC makes a total of $O(n \log(1/\delta))$ 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 algorithm 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 $(\varepsilon, \delta)$ guarantees. In other work with $(\varepsilon, \delta)$ 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 bit-vectors [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.

### 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 $\varepsilon (> 0)$, and a confidence $1 - \delta \in (0, 1]$. It returns an estimate of $|R_{F,S}|$ within tolerance $\varepsilon$, 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 $|R_{F,S}|$). 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 $|R_{F,S}|$ 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 $R_{F,S}$, such that $|Y| = \min(\text{thresh}, |R_{F,S}|)$. 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 $|R_{F,S}|$. Otherwise, a core subroutine, called ApproxMC2Core in ApproxMC and

---

**Algorithm 1 ApproxMC2**

1: thresh $\leftarrow 1 + 9.84 \left(1 + \frac{1}{n \varepsilon^2}\right)(1 + \frac{1}{2})^2$
2: $Y \leftarrow \text{BSAT}(F, \text{thresh}, S)$
3: if $(|Y| < \text{thresh})$ then return $|Y|$
4: $t \leftarrow \lceil 17 \log_2(3/\delta) \rceil$
5: $\text{nCells} \leftarrow 2$; $C \leftarrow \text{emptyList}$; iter $\leftarrow 0$
6: repeat
7: iter $\leftarrow$ iter + 1
8: $(\text{nCells}, nSols) \leftarrow \text{ApproxMC2Core}(F, S, \text{thresh}, \text{nCells})$
9: if (nCells $\neq 1$) then AddToList($C$, nSols $\times$ nCells)
10: until (iter $< t$)
11: finalEstimate $\leftarrow \text{FindMedian}(C)$
12: return finalEstimate
ApproxMC2Core in ApproxMC2 is invoked. This subroutine tries to randomly partition \( R_{FS} \) into “small” cells using hash functions from \( H_{cor}(|S|, m) \), for suitable values of \( m \).

There is a small probability that this subroutine fails and returns \((\perp, \perp)\). Otherwise, it returns the number of cells, \( n\text{Cells} \), into which \( R_{FS} \) is partitioned, and the count of solutions, \( n\text{Sols} \), in a randomly chosen small cell. The value of \( |R_{FS}| \) is then estimated as \( n\text{Cells} \times n\text{Sols} \). 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 \(|R_{FS}|\).

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 \( R_{FS} \) 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 \((m, h_m, \alpha_m)\), where \( m \) is an integer in \( \{1, \ldots, |S| - 1\} \), \( h_m \) is a hash function chosen randomly from \( H_{cor}(|S|, m) \), and \( \alpha_m \) is a vector chosen randomly from \( \{0,1\}^m \), such that \(|R_{(F,h_m,\alpha_m)}| < \text{thresh} \) and \(|R_{(F,h_m-1,\alpha_m-1)}| \geq \text{thresh} \). In order to find such a triple, ApproxMCCore uses a linear search: it starts from \( m = 1 \), chooses \( h_m \) and \( \alpha_m \) randomly and independently from \( H_{cor}(|S|, m) \) and \( \{0,1\}^m \) respectively, and checks if \(|R_{(F,h_m,\alpha_m)}| \geq \text{thresh} \). If so, the partitioning is considered too coarse, \( h_m \) and \( \alpha_m \) are discarded, and the process repeated with the next value of \( m \); otherwise, the search stops. Let \( m^*, h^{*}_m \) and \( \alpha^{*}_m \) denote the values of \( m, h_m \) and \( \alpha_m \), respectively, when the search stops. Then ApproxMCCore returns \(|R_{(F,h^{*}_m,\alpha^{*}_m)}| \times 2^{m^*} \) as the estimate of \(|R_{FS}|\). If the search fails to find \( m, h_m \) and \( \alpha_m \) with the desired properties, we say that ApproxMCCore fails.

Every iteration of the linear search above invokes BSAT once to check if \(|R_{(F,h_m,\alpha_m)}| \geq \text{thresh} \). A straightforward implementation of BSAT makes up to \( \text{thresh} \) calls to a SAT solver to answer this question. Therefore, an invocation of ApproxMCCore makes \( O(\text{thresh} \cdot |S|) \) SAT solver calls. A key contribution of this paper is a new approach for choosing hash functions that allows ApproxMC2Core to make at most \( O(\text{thresh} \cdot \log_2 |S|) \) 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 \((h_m, \alpha_m)\) pairs for different values of \( m \), as it searches for the right partitioning of \( R_{FS} \). In contrast, in ApproxMC2Core, we randomly choose one function \( h \) from \( H_{cor}(|S|, |S| - 1) \), and one vector \( \alpha \) from \(|\{0,1\}^{s-1}| \). Thereafter, we use “prefix-slices” of \( h \) and \( \alpha \) to obtain \( h_m \) and \( \alpha_m \) for all other values of \( m \). Formally, for every \( m \in \{1, \ldots, |S| - 1\} \), the \( m \)-th prefix-slice of \( h \), denoted \( h(m) \), is a map from \(|\{0,1\}^i| \) to \(|\{0,1\}^i| \), such that \( h(m)(y)[i] = h(y)[i] \), for all \( y \in \{|0,1\}^i| \) and for all \( i \in \{1, \ldots, m\} \). Similarly, the \( m \)-th prefix-slice of \( \alpha \), denoted \( \alpha(m) \), is an element of \(|\{0,1\}^m| \) such that \( \alpha(m)(i)[i] = \alpha[i] \) for all \( i \in \{1, \ldots, m\} \). Once \( h \) and \( \alpha \) are chosen randomly, ApproxMC2Core uses \( h(m) \) and \( \alpha(m) \) as choices of \( h_m \) and \( \alpha_m \), respectively. The randomness in the choices of \( h \) and \( \alpha \) induces randomness in the choices of \( h_m \) and \( \alpha_m \). However, the \((h_m, \alpha_m)\) pairs chosen for different values of \( m \) are no longer independent. Specifically, \( h_j(y)[i] = h_{j+1}(y)[i] \) and \( \alpha_j[i] = \alpha_{j+1}[i] \) for \( 1 \leq j < k < |S| \) and for all \( i \in \{1, \ldots, m\} \). This lack of independence is a fundamental departure from ApproxMCCore.

Algorithm 2 shows the pseudo-code for ApproxMC2Core. After choosing \( h \) and \( \alpha \) randomly, ApproxMC2Core checks if \(|R_{(F,h,\alpha)}| < \text{thresh} \). If not, ApproxMC2Core fails and returns \((\perp, \perp)\). Otherwise, it invokes sub-routine LogSATSearch to find a value of \( m \) (and hence, of \( h(m) \) and \( \alpha(m) \)) such that \(|R_{(F,h(m),\alpha(m))}| < \text{thresh} \) and \(|R_{(F,h(m-1),\alpha(m-1))}| \geq \text{thresh} \). This ensures that \( n\text{Sols} \) computed in line 7 is \(|R_{(F,h(m),\alpha(m))}| \). Finally, ApproxMC2Core returns \((2^m, n\text{Sols})\), where \( 2^m \) gives the number of cells into which \( R_{FS} \) 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 \(|R_{(F,h_m,\alpha_m)}| \leq |R_{(F,h_{m-1},\alpha_{m-1})}| \). 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 \( R_{FS} \) with \( h^{(m)} \) is large, i.e. \(|R_{(F,h^{(m)},\alpha^{(m)})}| \geq \text{thresh} \). As boundary conditions, we set BigCell[0] to 1 and BigCell[|S| - 1] to 0. These are justified because (i) if \( R_{FS} \) is partitioned into \( 2^0 \) (i.e. 1) cell, line 3 of Algorithm 1 ensures that the size of the cell (i.e. \(|R_{FS}|\)) is at least thresh, and (ii) line 4 of Algorithm 2 ensures that \(|R_{(F,h^{(i)},\alpha^{(i)})}| < \text{thresh} \). For every other \( i \), BigCell[i] is initialized to \( \perp \) (unknown value). Subsequently, we set BigCell[i] to 1 (0) whenever we find that \(|R_{(F,h^{(i)},\alpha^{(i)})}| \) is at least as large (smaller than) \( \text{thresh} \).

In the context of probabilistic hashing-based counting algorithms like ApproxMC, it has been observed [Meel, 2014]
Algorithm 3 LogSATSearch($F, S, h, \alpha, \text{thresh}, mPrev$)

1: loIdx ← 0; hiIdx ← $|S| - 1$; $m$ ← $m$Prev;
2: BigCell[0] ← 1; BigCell[$|S| - 1$] ← 0;
3: BigCell[0] ← 0 for all $i$ other than 0 and $|S| - 1$;
4: while true do
5: \[ Y \leftarrow \text{BSAT}(G \land (h(m)(S) = \alpha(m))), \text{thresh}, S); \]
6: \[ \text{if } (Y) \geq \text{thresh} \text{ then} \]
7: \[ \text{if } (\text{BigCell}[:m + 1]) = 0 \text{ then return } m + 1; \]
8: \[ \text{BigCell}[i] ← 1 \text{ for all } i \in \{1, \ldots, m\}; \]
9: \[ \text{loIdx} ← m; \]
10: \[ \text{if } (m - mPrev < 3) \text{ then } m ← m + 1; \]
11: \[ \text{else if } (2m < |S|) \text{ then } m ← 2m; \]
12: \[ \text{else } m ← (\text{hiIdx} + \text{loIdx}) / 2; \]
13: \[ \text{else} \]
14: \[ \text{if } (\text{BigCell}[:m - 1]) = 1 \text{ then return } m; \]
15: \[ \text{BigCell}[i] ← 0 \text{ for all } i \in \{m, \ldots, |S|\}; \]
16: \[ \text{hiIdx} ← m; \]
17: \[ \text{if } (m - mPrev < 3) \text{ then } m ← m - 1; \]
18: \[ \text{else } m ← (m + \text{loIdx}) / 2; \]

that the “right” values of $m$, $h_m$ and $\alpha_m$ for partitioning $R_{F|S}$ 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 $m$Prev, 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 $m$Prev, viz. when $m - m$Prev < 3, before embarking on a galloping search. Specifically, if LogSATSearch finds that $|R_{(F, h(m), \alpha(m))|S}|$ ≥ $\text{thresh}$ after the linear search, it keeps doubling the value of $m$ until either $|R_{(F, h(m), \alpha(m))|S}|$ becomes less than $\text{thresh}$, or $m$ overshoots $|S|$. Subsequently, binary search is done by iteratively bisecting the interval between loIdx and hiIdx. This ensures that the search requires $O(\log_2 m^*)$ calls (instead of $O(\log_2 |S|)$ 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 $m$Prev 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 $(h_m, \alpha_m)$ pairs for different values of $m$, and the independence of successive invocations of ApproxMC2Core, 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 $1 \leq i \leq |S|$, let $\mu_i = R_{F|S}/2^i$. For every $\beta > 0$ and $0 < \varepsilon < 1$, we have the following:

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

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

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

Lemma 2. The following bounds hold:

1. \[ \Pr[T_{m^*}] \leq \frac{1}{0.25}; \]
2. \[ \Pr[L_{m^*}] \leq \frac{1}{2m^*}; \]
3. \[ \Pr[L_{m^*}] \leq \frac{1}{10.84}; \]
4. \[ \Pr[L_{m^*} \cup U_{m^*}] \leq \frac{1}{4.92}; \]

The proofs follow from the definitions of $m^*$, thresh, $\mu_i$, and from applications of Lemma 1 with appropriate values of $\beta$.

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

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

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 $\mu_i$, and from the linear ordering of $R_{(l,F,H(m^*),\alpha(m^*))}\subseteq S$.

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

**O2:** \[ \Pr\left( \bigcup_{j \in \{m^* - 1\}} T_j \cap (L_i \cup U_i) \right) \leq \Pr(\overline{T_{m^* - 1} \cap (L_i \cup U_i)}) \leq \Pr[L_{m^*} \cup U_{m^*}]. \]

**O3:** For $i \in \{m^* - 2, m^* - 1\}$, since $\text{thresh} \leq \mu_i (1 + \frac{1}{1+\varepsilon})$, 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 $\logSAT$Search proceeds. Our main theorem now follows from Lemma 3 and 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 $\Pr[|R_{F,S}|/(1 + \varepsilon) \leq c \leq (1 + \varepsilon)|R_{F,S}| \geq 1 - \delta$, and $k \leq O\left(\frac{\log(|S|) \log(1/\delta)}{\varepsilon^2}\right)$. Furthermore, $E[c] = |R_{F,S}|$. Hence ApproxMC2 is a SPAC counter.

Note that the number of SAT solver calls in ApproxMC [Chakraborty et al., 2013b] lies in $O\left(\frac{\log(|S|) \log(1/\delta)}{\varepsilon^2}\right)$, which is exponentially worse than the number of calls in ApproxMC2, for the same $\varepsilon$ and $\delta$. 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 $\#DNF$ 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 $O(n \log n)$ 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 $O(n \log n)$ to $O(\log n)$, 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 ApproxMC2Core that replaces $|R_{F,S}|$ with the total weight of assignments in $R_{F,S}$. 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 ApproxMC compare with that of ApproxMC 2? (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
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 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 ($\varepsilon_{obs}$) was calculated as $\max(\frac{\text{AprxCount}}{\text{ExactCount}} - 1, 1 - \frac{\text{ExactCount}}{\text{AprxCount}})$, where AprxCount is the estimate computed by ApproxMC2. We observe that the geometric mean of $\varepsilon_{obs}$ 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Vars</th>
<th>Clauses</th>
<th>ApproxMC2 Time</th>
<th>ApproxMC Time</th>
<th>ApproxMC2 SATCalls</th>
<th>ApproxMC SATCalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>tutorial3</td>
<td>486193</td>
<td>2598178</td>
<td>12373.99</td>
<td>–</td>
<td>1744</td>
<td>–</td>
</tr>
<tr>
<td>case204</td>
<td>214</td>
<td>580</td>
<td>166.2</td>
<td>–</td>
<td>1808</td>
<td>–</td>
</tr>
<tr>
<td>case205</td>
<td>214</td>
<td>580</td>
<td>300.11</td>
<td>–</td>
<td>1793</td>
<td>–</td>
</tr>
<tr>
<td>case133</td>
<td>211</td>
<td>615</td>
<td>18502.44</td>
<td>–</td>
<td>2043</td>
<td>–</td>
</tr>
<tr>
<td>s953a_15_7</td>
<td>602</td>
<td>1657</td>
<td>161.41</td>
<td>–</td>
<td>1648</td>
<td>–</td>
</tr>
<tr>
<td>llreverse</td>
<td>39912</td>
<td>167842</td>
<td>151.33</td>
<td>450.57</td>
<td>1516</td>
<td>4258</td>
</tr>
<tr>
<td>karatsuba</td>
<td>19594</td>
<td>82417</td>
<td>23553.73</td>
<td>28817.79</td>
<td>1378</td>
<td>13360</td>
</tr>
<tr>
<td>enqueueSeqSK</td>
<td>16466</td>
<td>58515</td>
<td>192.96</td>
<td>2036.09</td>
<td>2207</td>
<td>23321</td>
</tr>
<tr>
<td>progsyn_20</td>
<td>15475</td>
<td>60994</td>
<td>1778.45</td>
<td>20557.24</td>
<td>2308</td>
<td>34815</td>
</tr>
<tr>
<td>progsyn_77</td>
<td>14535</td>
<td>27573</td>
<td>88.36</td>
<td>1529.34</td>
<td>2054</td>
<td>24764</td>
</tr>
<tr>
<td>sort</td>
<td>12125</td>
<td>49611</td>
<td>209.0</td>
<td>3610.4</td>
<td>1605</td>
<td>27731</td>
</tr>
<tr>
<td>LoginService2</td>
<td>11311</td>
<td>41411</td>
<td>26.04</td>
<td>110.77</td>
<td>1533</td>
<td>10653</td>
</tr>
<tr>
<td>progsyn_17</td>
<td>10890</td>
<td>27056</td>
<td>100.76</td>
<td>4874.39</td>
<td>1810</td>
<td>28407</td>
</tr>
<tr>
<td>progsyn_29</td>
<td>8866</td>
<td>31557</td>
<td>87.78</td>
<td>3569.25</td>
<td>1712</td>
<td>28630</td>
</tr>
<tr>
<td>LoginService</td>
<td>8200</td>
<td>26689</td>
<td>21.77</td>
<td>101.15</td>
<td>1498</td>
<td>12520</td>
</tr>
<tr>
<td>doublyLinkedList</td>
<td>6890</td>
<td>26918</td>
<td>17.05</td>
<td>75.45</td>
<td>1615</td>
<td>10647</td>
</tr>
</tbody>
</table>

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.

**Figure 1:** Quality of counts computed by ApproxMC2

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**


