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 *probably approximately correct* (PAC) guarantees. 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 variables, n, in the input constraint. Since SAT solver calls are by far the computationally most expensive steps in these

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

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 $\mathsf{Pr}\Big[|R_{F\downarrow S}|/(1+\varepsilon) \leq c \leq (1+\varepsilon)|R_{F\downarrow S}|\Big] \geq 1-\delta$. The probabilistic guaranteee provided by a PAC counter is also called an (ε,δ) guarantee, for obvious reasons.

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 $\{h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n} a_{i,k} \cdot y[k]), a_{i,k} \in \{0,1\}, 1 \leq i \leq m, 0 \leq k \leq n\}$, 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 $H_{xor}(n,m)$. 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.

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

```
thresh ← 1 + 9.84 (1 + ε/(1+ε)) (1 + 1/ε)^2;
Y ← BSAT(F, thresh, S);
if (|Y| < thresh) then return |Y|;
t ← ⌈17 log_2(3/δ)⌉;
nCells ← 2; C ← emptyList; iter ← 0;
repeat
    iter ← iter + 1;
    (nCells, nSols) ← ApproxMC2Core(F, S, thresh, nCells);
    if (nCells ≠ ⊥) then AddToList(C, nSols × nCells);
until (iter < t);
finalEstimate ← FindMedian(C);
return finalEstimate
```

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 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 $(\perp, \perp)$. 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 × 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 $|R_{\langle F, h_m, \alpha_m\rangle\downarrow S}| < \mathsf{thresh}$ and $|R_{\langle F, h_{m-1}, \alpha_{m-1}\rangle\downarrow S}| \geq \mathsf{thresh}$. 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 $|R_{\langle F, h_m, \alpha_m\rangle\downarrow S}| \geq \mathsf{thresh}$. 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 $|R_{\langle F, h_m, \alpha_m\rangle\downarrow S}| \geq \mathsf{thresh}$. A straightforward implementation of BSAT makes up to thresh calls to a SAT solver to answer this question. Therefore, an invocation of ApproxMCCore makes $\mathcal{O}(\mathsf{thresh}.|S|)$ SAT solver calls. A key contribution of this paper is a new approach for choosing hash functions that allows ApproxMC2Core to make at most $\mathcal{O}(\mathsf{thresh}.\log_2 |S|)$ 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 m^{th} 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 m^{th} 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,

Algorithm 2 $\mathsf{ApproxMC2Core}(F, S, \mathsf{thresh}, \mathsf{prevNCells})$

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

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 shows the pseudo-code for ApproxMC2Core. After choosing h and α randomly, ApproxMC2Core checks if $|R_{\langle F, h, \alpha\rangle\downarrow S}| < \mathsf{thresh}$. 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 $|R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle\downarrow S}| < \mathsf{thresh}$ and $|R_{\langle F, h^{(m-1)}, \alpha^{(m-1)}\rangle\downarrow S}| \geq \mathsf{thresh}$. This ensures that nSols computed in line 7 is $|R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle\downarrow S}|$. Finally, ApproxMC2Core returns $(2^m, \mathsf{nSols})$, 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. $|R_{\langle F, h^{(m)}, \alpha^{(m)}\rangle\downarrow S}| \geq \mathsf{thresh}$. As boundary conditions, we set $\mathsf{BigCell}[0]$ to 1 and $\mathsf{BigCell}[|S| - 1]$ 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 $|R_{\langle F, h^{|S|-1}, \alpha^{|S|-1}\rangle\downarrow S}| < \mathsf{thresh}$. For every other i, $\mathsf{BigCell}[i]$ is initialized to $\bot$ (unknown value). Subsequently, we set $\mathsf{BigCell}[i]$ 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] 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 lin-

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

loIndex $\leftarrow 0$; hiIndex $\leftarrow |S| - 1$; $m \leftarrow$ mPrev;
BigCell$[0] \leftarrow 1$; BigCell$[|S| - 1] \leftarrow 0$;
BigCell$[i] \leftarrow \perp$ for all i other than 0 and $|S| - 1$;
while true **do**
  $Y \leftarrow \mathsf{BSAT}(F \wedge (h^{(m)}(S) = \alpha^{(m)}), \mathsf{thresh}, S)$;
  if ($|Y| \geq$ thresh) **then**
    if (BigCell$[m+1] = 0$) **then return** $m + 1$;
    BigCell$[i] \leftarrow 1$ for all $i \in \{1, \ldots m\}$;
    loIndex $\leftarrow m$;
    if ($|m - \mathsf{mPrev}| < 3$) **then** $m \leftarrow m + 1$;
    else if ($2.m < |S|$) **then** $m \leftarrow 2.m$;
    else $m \leftarrow (\mathsf{hiIndex} + m)/2$;
  else
    if (BigCell$[m-1] = 1$) **then return** m;
    BigCell$[i] \leftarrow 0$ for all $i \in \{m, \ldots |S|\}$;
    hiIndex $\leftarrow m$;
    if ($|m - \mathsf{mPrev}| < 3$) **then** $m \leftarrow m - 1$;
    else $m \leftarrow (m + \mathsf{loIndex})/2$;

early 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

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

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

Proof. Note that $\mathsf{Pr}[T_{m^*-3}] = \mathsf{Pr}\left[|R_{F, h^{m^*-3}, \alpha^{m^*-3}}| \leq \mathsf{thresh}\right]$. Noting that thresh $< \frac{3}{2}$pivot. Using Lemma 1 and putting $\beta = 3/8$ and $\mu_{m^*-3} \geq 4$pivot (ensuring $\beta\mu_{m^*-3} \geq$ thresh), we get $\mathsf{Pr}[T_{m^*-3}] \leq \frac{1}{1+25/64*8*4*4.92} \leq \frac{1}{62.5}$

To compute $\mathsf{Pr}[L_{m^*-2}]$, we employ Lemma 1 with $\mu_{m^*-2} \geq 2$pivot and $\beta = \frac{1}{1+\varepsilon}$ to obtain $\mathsf{Pr}[L_{m^*-2}] \leq \frac{1}{1+(\frac{\varepsilon}{1+\varepsilon})^2 2\mathrm{pivot}} \leq \frac{1}{20.68}$. Similarly, we obtain $\mathsf{Pr}[L_{m^*-1}] \leq \frac{1}{1+(\frac{\varepsilon}{1+\varepsilon})^2 \mathrm{pivot}} \leq \frac{1}{10.84}$.

Finally, since $\mathsf{Pr}[L_{m^*} \cup U_{m^*}] = \mathsf{Pr}\left[\left||R_{F, h^{m^*}, \alpha^{m^*}}| - \mu_{m^*}\right| \geq \frac{\varepsilon}{1+\varepsilon}\mu_{m^*}\right]$, we employ Lemma 1 with $\mu_{m^*} \geq$ pivot$/2$. Therefore, $\mathsf{Pr}[L_{m^*} \cup U_{m^*}] \leq 1/4.92$. □

Lemma 3. $\mathsf{Pr}[B] \leq 0.36$

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 $\mathsf{Pr}[B] \leq \mathsf{Pr}\left[\bigcup_{i \in \{1, \ldots |S|\}} \left(\overline{T_{i-1}} \cap T_i \cap (L_i \cup U_i)\right)\right]$.

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 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* $k \in \mathcal{O}(\frac{\log(|S|)\log(1/\delta)}{\varepsilon^2})$.

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 Appendix.

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.

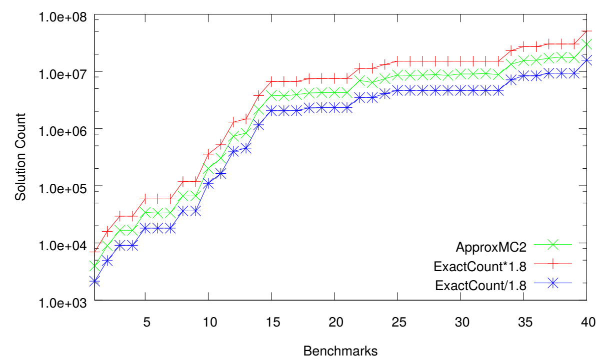

Figure 1: Quality of counts computed by ApproxMC2

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.

A Detailed Experimental Analysis

Table 2 presents an extended version of Table 1.

Benchmark	Vars	Clauses	ApproxMC2 Time	ApproxMC Time	ApproxMC2 SATCalls	ApproxMC SATCalls
case106	204	509	133.92	–	2377	–
case35	400	1414	215.35	–	1809	–
case146	219	558	4586.26	–	1986	–
tutorial3	486193	2598178	12373.99	–	1744	–
case202	200	544	149.56	–	1839	–
case203	214	580	165.17	–	1800	–
case205	214	580	300.11	–	1793	–
s953a_15_7	602	1657	161.41	–	1648	–
s953a_7_4	533	1373	16218.67	–	1832	–
case_1_b14_1	238	681	132.47	–	1814	–
case_2_b14_1	238	681	129.95	–	1805	–
case119	267	787	906.88	–	2044	–
case133	211	615	18502.44	–	2043	–
case_3_b14_1	238	681	125.69	–	1831	–
case204	214	580	166.2	–	1808	–
case136	211	615	9754.08	–	2026	–
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
20	15475	60994	1778.45	20557.24	2308	34815
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
81	10775	38006	158.93	10555.13	2220	33954
17	10090	27056	100.76	4874.39	1810	28407
29	8866	31557	87.78	3569.25	1712	28630
LoginService	8200	26689	21.77	101.15	1498	12520
19	6993	23867	126.23	11051.95	1827	31352
Pollard	7815	41258	12.8	16.55	1023	695
7	6683	24816	84.1	5332.76	2062	31195
doublyLinkedList	6890	26918	17.05	75.45	1615	10647
tree_delete	5758	22105	8.87	33.84	1455	7647
35	4915	10547	77.53	6074.75	2028	32096
80	4969	17060	76.88	5039.37	2389	30294
ProcessBean	4768	14458	213.78	15558.75	2296	33493
56	4842	17828	126.96	1024.36	2218	22988
70	4670	15864	68.18	1026.99	2307	23902
ProjectService3	3175	11019	190.98	19626.24	1715	36762
32	3834	13594	49.86	1102.68	1882	21835
55	3128	12145	90.33	7623.13	1810	28322
51	3708	14594	86.9	1538.87	2091	22115
109	3565	14012	77.69	917.19	1752	21104
NotificationServiceImpl2	3540	13425	22.2	74.76	2265	15186
aig_insertion2	2592	10156	13.18	120.56	2412	16729
53	2586	10747	32.29	248.26	1885	17680
ConcreteActivityService	2481	9011	6.01	33.56	1619	13072
111	2348	5479	42.49	567.25	1884	20383
aig_insertion1	2296	9326	24.91	127.94	2416	16779
case_3_b14_2	270	805	90.88	18114.84	2028	31194
ActivityService2	1952	6867	2.74	13.09	1542	9700
IterationService	1896	6732	3.39	16.74	1572	10570
squaring7	1628	5837	323.58	8774.17	1791	29298
ActivityService	1837	5968	2.39	11.62	1633	9606
10	1494	2215	135.04	4759.18	2020	30270
case_2_b14_2	270	805	90.17	13479.3	2002	31179
PhaseService	1686	5655	2.45	12.03	1617	9649
squaring9	1434	5028	308.34	6131.25	1718	29324
case_1_b12_2	827	2725	129.03	9964.91	1808	29328
UserServiceImpl	1509	5009	1.49	7.1	1480	7707
27	1509	2707	34.96	130.23	1885	17489
squaring8	1101	3642	250.2	9963.56	1784	29386
case_2_b12_2	827	2725	122.64	7967.12	1803	29342
case_1_b14_2	270	805	89.69	10777.71	2038	31187
case_0_b12_2	827	2725	134.65	8362.19	1808	29340
IssueServiceImpl	1393	4319	2.48	13.37	1589	10469
squaring10	1099	3632	290.64	6208.98	1773	29391
squaring11	966	3213	324.63	11111.49	1795	29280
s953a_3_2	515	1297	165.81	11968.07	1826	33920
squaring29	1141	4248	135.4	1290.88	2002	18662
squaring3	885	2809	281.29	8836.68	1802	27618
squaring28	1060	3839	129.46	1164.31	2091	18685
squaring6	885	2809	233.72	5799.3	1753	27580
s1196a_15_7	777	2165	73.26	2577.71	1938	23097
squaring30	1031	3693	117.53	1134.18	2006	18668

Continued on next page

Benchmark	Vars	Clauses	ApproxMC2 Time	ApproxMC Time	ApproxMC2 SATCalls	ApproxMC SATCalls
squaring1	891	2839	227.03	5145.1	1787	27557
squaring4	891	2839	274.71	6094.24	1774	27646
squaring2	885	2809	240.35	5112.72	1805	27577
squaring5	885	2809	352.17	6477.17	1819	27559
GuidanceService	988	3088	3.59	17.08	1632	13115
case_1_b14_3	304	941	109.46	7432.67	1829	28444
s1488_15_7	941	2783	1.57	5.02	1553	5867
squaring26	894	3187	102.08	787.16	1997	17569
case_3_b14_3	304	941	104.65	6821.33	1815	28424
case201	200	544	221.78	16171.04	1814	32970
squaring25	846	2947	110.25	791.63	2074	17437
tree_delete3	795	2734	46.39	562.39	1595	20763
s1488_7_4	872	2499	1.46	5.43	1523	6891
squaring27	837	2901	110.1	714.37	2028	17337
s1488_3_2	854	2423	1.8	6.51	1501	5527
case_2_b14_3	304	941	114.36	6643.4	1815	28443
s1238a_15_7	773	2210	66.87	713.17	1841	22792
case_0_b11_1	340	1026	123.65	6398.95	1777	29323
s1196a_7_4	708	1881	76.44	917.27	1800	22442
s1196a_3_2	690	1805	62.64	827.91	1711	22177
s1238a_7_4	704	1926	66.48	716.53	1813	22545
case_1_b11_1	340	1026	124.08	5754.05	1810	29352
s1238a_3_2	686	1850	77.88	895.66	1848	23171
GuidanceService2	715	2181	2.37	15.56	1605	13252
squaring23	710	2268	74.37	429.83	2358	15911
squaring22	695	2193	71.75	466.91	2357	15891
squaring20	696	2198	78.24	466.67	2357	15813
squaring21	697	2203	81.89	460.94	2451	15877
squaring24	695	2193	80.76	462.12	2363	15849
s832a_15_7	693	2017	6.01	29.68	1608	14808
s820a_15_7	685	1987	2.52	12.0	1483	12488
s832a_7_4	624	1733	2.47	11.66	1543	12713
s832a_3_2	606	1657	1.26	6.71	1717	11449
s820a_7_4	616	1703	2.41	9.83	1435	12328
s820a_3_2	598	1627	1.19	5.75	1646	10746
case34	409	1597	124.7	2665.47	1818	27561
s420_15_7	366	994	81.34	2011.14	2060	24871
case6	329	996	113.94	3233.94	2043	25750
s420_new_15_7	351	934	73.18	1897.5	2054	24885
case131	432	1830	76.96	1293.21	1852	24230
s420_7_4	312	770	82.7	2373.55	2049	24887
s420_new1_15_7	366	994	79.42	1732.28	2053	24868
case121	291	975	112.0	3046.07	1809	29418
case_0_b12_1	427	1385	67.81	914.84	1880	22212
squaring50	500	1965	31.92	190.39	2388	16703
squaring51	496	1947	37.45	230.85	2094	16804
case_1_b12_1	427	1385	66.94	866.66	1894	22152
case_2_b12_1	427	1385	63.55	797.71	1882	22206
s420_new1_7_4	312	770	85.19	2045.89	2061	24869
case125	393	1555	86.17	1324.85	2306	23975
case123	267	980	58.88	1625.83	2250	23066
case143	427	1592	71.83	696.46	2139	19449
s420_new_7_4	312	770	74.5	1485.23	2054	24887
case105	170	407	227.36	7361.33	2330	32045
case114	428	1851	24.83	151.71	1854	17679
case115	428	1851	29.09	173.42	1888	17659
case116	438	1881	31.59	156.56	1897	17636
s526a_15_7	453	1304	20.35	67.56	1887	15811
s526_15_7	452	1303	17.69	58.43	1898	15861
case126	302	1129	74.05	1312.09	2316	23068
s420_new_3_2	294	694	88.48	1577.85	2052	24925
s420_new1_3_2	294	694	93.87	1590.05	2053	24485
s420_3_2	294	694	97.18	1399.45	2052	24933
s526a_7_4	384	1020	13.39	46.53	1805	15711
case57	288	1158	57.97	703.78	1647	21193
s444_15_7	377	1072	8.43	26.74	1634	14897
case62	291	1165	71.35	833.88	1973	22174
s526_7_4	383	1019	20.55	44.41	1820	15200
s526_3_2	365	943	7.66	24.51	1964	14977
s526a_3_2	366	944	12.45	26.09	1772	15219
s382_15_7	350	995	22.29	67.46	1763	16207
registerlesSwap	372	1493	0.42	0.33	1018	685
s510_15_7	340	948	20.59	56.42	1840	16558
s510_7_4	316	844	18.06	73.38	1842	16622
case117	309	1367	0.75	3.44	1712	8665

Continued on next page

Benchmark	Vars	Clauses	ApproxMC2 Time	ApproxMC Time	ApproxMC2 SATCalls	ApproxMC SATCalls
case122	314	1258	17.59	67.53	1963	16806
case111	306	1358	0.62	2.86	1519	7686
case118	309	1367	0.84	3.44	1933	8650
case113	309	1367	0.93	3.77	1972	8624
s510_3_2	298	768	15.14	74.08	1871	16667
s349_15_7	285	829	13.18	76.65	1906	15850
s444_7_4	308	788	18.25	62.97	1766	16260
s298_15_7	292	870	0.86	4.08	1756	9569
case2	296	1116	10.08	39.7	1662	14956
s344_15_7	284	824	12.76	60.94	1887	15837
case3	294	1110	11.52	40.84	1648	14935
case110	287	1263	0.69	2.74	1776	7771
s444_3_2	290	712	6.48	18.76	1601	14909
s382_7_4	281	711	7.83	28.86	1538	14832
s382_3_2	263	635	5.33	21.47	1624	14915
case109	241	915	5.53	24.43	1711	13172
case132	236	708	22.7	94.67	1683	14076
s298_7_4	223	586	0.67	3.42	1690	9492
case135	236	708	19.74	68.81	1659	13858
case56	202	722	1.84	10.02	1676	13176
s298_3_2	205	510	0.59	2.92	1747	8670
case108	205	800	0.87	4.15	1731	9554
s344_7_4	215	540	14.53	47.24	1875	15887
case54	203	725	2.49	10.56	1679	13197
case5	176	518	72.42	474.93	2103	18572
case1	187	681	0.73	3.8	1726	10331
case46	176	660	0.64	3.53	1726	9572
case44	173	651	0.61	3.52	1754	9548
case124	133	386	66.62	653.36	1730	20333
s344_3_2	197	464	12.37	38.68	1896	15915
s349_7_4	216	545	41.0	39.31	1893	15854
case68	178	553	1.12	5.25	1744	10430
s349_3_2	198	469	18.26	40.07	1862	15841
s27_15_7	32	103	0.0	0.07	0	612
case8	160	525	9.68	37.17	1883	15874
case53	132	395	0.67	4.18	1741	11410
case55	149	442	2.18	8.88	1667	13128
case51	132	395	0.66	3.76	1740	11220
case38	143	568	0.34	1.31	1641	5956
case112	137	520	0.5	2.17	1975	8668
case52	132	395	0.85	3.83	1743	11357
case22	126	411	0.27	1.22	1516	6856
case21	126	411	0.28	1.2	1526	6808
case47	118	328	1.11	5.47	1756	11378
case45	116	421	0.29	1.49	1496	7662
case7	116	365	0.57	2.83	1739	10475
case43	116	421	0.31	1.54	1517	7726
case11	105	371	0.28	1.48	1458	7719
case4	103	316	0.37	1.71	1900	8515
case63	96	299	0.36	1.75	1630	8621
case64	93	285	0.4	1.85	1927	8748
case58	96	299	0.42	1.79	1884	8704
case59	93	285	0.39	1.75	1927	8723
s27_7_4	24	63	0.0	0.06	0	594
case59_1	93	285	0.39	1.69	1972	8642
case134	60	146	0.37	2.34	1710	11336
case101	72	178	2.12	10.02	1666	14100
case100	72	178	2.0	8.73	1675	14072
case23	77	235	0.22	0.7	1604	5034
case17	77	235	0.22	0.69	1608	5069
case137	60	146	0.52	2.43	1779	11219
case32	52	146	0.15	0.76	1372	4106
case127	36	104	0.01	0.06	0	509
case128	36	104	0.01	0.06	0	541
case25	68	195	0.18	0.44	1323	3266
case30	68	195	0.18	0.43	1341	3259
case26	53	148	0.16	0.55	1352	4120
case36	64	208	0.15	0.34	1338	2426
case27	52	146	0.15	0.51	1369	4156
case31	53	148	0.16	0.52	1374	4125
case29	65	190	0.15	0.28	1181	2360
case24	65	190	0.17	0.28	1227	2267
case33	51	143	0.18	0.52	1369	4199
case28	51	143	0.18	0.48	1316	4153
s27_3_2	20	43	0.02	0.08	0	588

Continued on next page

Benchmark	Vars	Clauses	ApproxMC2 Time	ApproxMC Time	ApproxMC2 SATCalls	ApproxMC SATCalls
case103	32	86	0.12	0.24	1233	2349
case102	34	92	0.15	0.25	1215	2357
TR_b14_2_linear	1570	4963	–	–	331	–
tableBasedAddition	1026	961	–	–	317	–
case144	765	2340	–	–	339	–
case_1_b14_even	1304	4057	–	–	289	–
63	7242	24379	–	–	421	–
TR_b14_1_linear	1287	3950	–	–	341	–
case18	579	1815	–	–	423	–
s35932_15_7	17918	44709	–	–	332	–
s35932_7_4	17849	44425	–	–	340	–
case14	247	649	–	–	423	–
case130	644	2056	–	–	423	–
s838_3_2	598	1414	–	–	422	–
case141	4155	11758	–	–	342	–
s5378a_7_4	3697	8448	–	–	287	–
case_1_ptb_2	2851	9506	–	–	327	–
reverse	75641	380869	–	–	322	–
s38417_3_2	25528	57586	–	–	319	–
s641_15_7	576	1399	–	–	1268258	–
case42	903	2735	–	–	304	–
case_1_b12_even3	4157	13049	–	–	323	–
tree_delete2	15573	59561	–	–	299	–
isolateRightmost	10057	35275	–	–	306	–
case12	737	2310	–	–	400	–
case1_b14_even3	1318	4093	–	–	331	–
case_2_b12_even1	2681	8492	–	–	330	–
s15850a_3_2	10908	24476	–	–	324	–
case_3_4_b14_even	1532	4761	–	–	319	–
s13207a_3_2	9368	20559	–	–	271	–
aig_traverse	82247	331100	–	–	347	–
30	29621	112297	–	–	324	–
lldelete1	198239	803606	–	–	345	–
71	5670	14616	–	–	418	–
case104	3666	11589	–	–	333	–
case9	279	753	–	–	423	–
TR_b14_even_linear	8809	28200	–	–	351	–
s13207a_7_4	9386	20635	–	–	316	–
tree_delete1	34998	135565	–	–	271	–
case_0_b12_even2	2669	8460	–	–	326	–
case_0_ptb_2	3391	11089	–	–	284	–
case_0_b14_1	812	3000	–	–	333	–
57	6917	23549	–	–	410901	–
case_1_b12_even2	2669	8460	–	–	311	–
s15850a_15_7	10995	24836	–	–	323	–
s9234a_7_4	6313	14555	–	–	290	–
case145	219	558	–	–	2498760	–
s9234a_15_7	6382	14839	–	–	287	–
TR_b12_even7_linear	8633	28088	–	–	347	–
case19	397	1126	–	–	424	–
logcount	19126	68146	–	–	352	–
jburnim_morton	101241	378557	–	–	358	–
s1423a_15_7	864	2248	–	–	329	–
case120	284	851	–	–	2509415	–
s5378a_3_2	3679	8372	–	–	298	–
case_2_b12_even2	2669	8460	–	–	329	–
case_2_ptb_2	2848	9498	–	–	317	–
case138	849	2253	–	–	310	–
case212	1189	3477	–	–	322	–
TR_b14_even2_linear	10329	33008	–	–	338	–
case_2_b12_even3	4157	13049	–	–	307	–
squaring41	4185	13599	–	–	345	–
case139	846	2163	–	–	313	–
squaring40	4173	13539	–	–	342	–
case_1_b12_even1	2681	8492	–	–	329	–
case39	245	650	–	–	420	–
case41	245	650	–	–	424	–
s838_7_4	616	1490	–	–	424	–
case37	1084	3159	–	–	306	–
case207	824	2128	–	–	287	–
s641_3_2	489	1039	–	–	396	–
84	15678	70956	–	–	339	–
s38584_3_2	23405	57394	–	–	321	–
tree_delete4	12389	47486	–	–	328	–
case20	397	1126	–	–	424	–

Continued on next page

Benchmark	Vars	Clauses	ApproxMC2 Time	ApproxMC Time	ApproxMC2 SATCalls	ApproxMC SATCalls
s713_3_2	509	1117	–	–	303	–
case140	488	1222	–	–	289	–
case208	824	2128	–	–	301	–
case49	1510	6505	–	–	423	–
TR_b14_even3_linear	8809	28200	–	–	347	–
ConcreteRoleAffectationService	395951	1520924	–	–	349	–
s38584_7_4	23423	57470	–	–	333	–
case61	282	753	–	–	423	–
s713_7_4	527	1193	–	–	883153	–
case50	843	3288	–	–	423	–
TR_ptb_1_linear	1969	6288	–	–	424	–
xpose	35519	159328	–	–	218	–
case_1_ptb_1	966	3035	–	–	424	–
110	18570	63471	–	–	311	–
54	21726	73499	–	–	338	–
llinsert2	116345	454527	–	–	334	–
s15850a_7_4	10926	24552	–	–	315	–
case107	618	1661	–	–	315	–
tree_insert_insert	5835	22436	–	–	423	–
s38417_15_7	25615	57946	–	–	314	–
compress	44901	166948	–	–	1302	–
TR_b12_even2_linear	8633	28088	–	–	332	–
signedAvg	30335	91854	–	–	327	–
s713_15_7	596	1477	–	–	449	–
s1423a_3_2	777	1888	–	–	334	–
case210	872	2937	–	–	287	–
s38417_7_4	25546	57662	–	–	329	–
case_0_b12_even1	2681	8492	–	–	325	–
s641_7_4	507	1115	–	–	1825384	–
case142	2457	7305	–	–	320	–
TR_b12_2_linear	2426	8373	–	–	400	–
squaring42	4173	13539	–	–	337	–
squaring60	5186	16134	–	–	343	–
s9234a_3_2	6295	14479	–	–	287	–
case3_b14_even3	1304	4057	–	–	322	–
case211	869	2929	–	–	290	–
case213	648	1891	–	–	268	–
TR_ptb_2_linear	3857	12774	–	–	424	–
case_2_ptb_1	963	3027	–	–	424	–
case10	328	878	–	–	329	–
case214	645	1883	–	–	424	–
s13207a_15_7	9455	20919	–	–	309	–
case_1_4_b14_even	1532	4761	–	–	327	–
107	8948	40147	–	–	324	–
tree_insert_search	82202	319077	–	–	346	–
case_0_b12_even3	4157	13049	–	–	324	–
case209	1189	3477	–	–	424	–
TR_device_1_even_linear	2447	7612	–	–	342	–
tree_search	81080	315697	–	–	332	–
case_0_ptb_1	1507	4621	–	–	424	–
squaring70	882	2663	–	–	339	–
case15	296	774	–	–	328	–
s35932_3_2	17831	44349	–	–	360	–
log2	185178	716257	–	–	360	–
s838_15_7	685	1774	–	–	424	–
s5378a_15_7	3766	8732	–	–	291	–
case40	245	650	–	–	422	–
s38584_15_7	23492	57754	–	–	325	–
partition	151795	689327	–	–	343	–
TR_b12_even3_linear	8633	28088	–	–	339	–
TR_b14_3_linear	1942	6228	–	–	348	–
TR_device_1_linear	1249	3927	–	–	347	–
TR_b12_1_linear	1914	6619	–	–	317	–
s1423a_7_4	795	1964	–	–	333	–
squaring12	1507	5210	–	8419.06	423	31880
squaring16	1627	5835	–	9926.56	423	31778
squaring14	1458	5009	–	13892.48	423	31842