Constraints manuscript No.
(will be inserted by the editor)

Not All FPRASs are Equal: Demystifying FPRASs for DNF-Counting

Kuldeep S. Meel · Aditya A. Shrotri · Moshe Y. Vardi

Received: date / Accepted: date

Abstract The problem of counting the number of solutions of a DNF formula, also called #DNF, is a fundamental problem in artificial intelligence with applications in diverse domains ranging from network reliability to probabilistic databases. Owing to the intractability of the exact variant, efforts have focused on the design of approximate techniques for #DNF. Consequently, several Fully Polynomial Randomized Approximation Schemes (FPRASs) based on Monte Carlo techniques have been proposed. Recently, it was discovered that hashing-based techniques too lend themselves to FPRASs for #DNF. Despite significant improvements, the complexity of the hashing-based FPRAS is still worse than that of the best Monte Carlo FPRAS by polylog factors. Two questions were left unanswered in previous works: Can the complexity of the hashing-based techniques be improved? How do the various approaches stack up against each other empirically?

In this paper, we first propose a new search procedure for the hashing-based FPRAS that removes the polylog factors from its time complexity. We then present the first empirical study of runtime behavior of different FPRASs for #DNF. The result of our study produces a nuanced picture. First of all, we observe that there is no single best algorithm that outperforms all others for all classes of formulas and input parameters. Second, we observe that the algorithm with one of the worst time complexities solves the largest number of benchmarks.

Author names are ordered alphabetically by last name and does not indicate contribution

K. S. Meel
National University of Singapore, Singapore
E-mail: meel@comp.nus.edu.sg

A. A. Shrotri
Rice University, Houston USA
E-mail: Aditya.Aniruddh.Shrotri@rice.edu

M. Y. Vardi
Rice University, Houston USA
E-mail: vardi@rice.edu

1 Introduction

Constrained counting is a fundamental problem in artificial intelligence with a wide variety of applications ranging from network reliability [1], probabilistic inference [2,3], probabilistic databases [4], quantified information flow [5], and the like. Given a set of constraints F, the problem of constrained counting seeks to compute the total number of solutions to F. In this work, we focus on the variant of constrained counting where F is expressed in Disjunctive Normal Form (DNF), henceforth denoted as DNF-Counting or #DNF. This problem is important in practice, as applications such as query evaluation in probabilistic databases [4] and failure-probability estimation of networks [6] reduce to it.

The problem of #DNF is known to be #P-complete [7], where #P is the class of counting problems for decision problems in NP. Due to the intractability of exact #DNF, the approximate variant of #DNF has been studied extensively by both theoreticians and practitioners. Of particular interest is to obtain (ε, δ) approximation, such that the count returned by the approximation scheme is within $(1+\varepsilon)$ factor of the exact count with confidence at least $1-\delta$, where ε and δ are supplied by the user.

In their seminal paper, Karp and Luby [8] proposed the first Fully Polynomial Randomized Approximation Scheme (FPRAS) for #DNF based on Monte Carlo sampling. We will henceforth use the term KL Counter to denote the FPRAS proposed by Karp et al. The time complexity of KL Counter is quadratic in the number of cubes (i.e., disjuncts) and linear in the number of the variables of the input formula F. Building on KL Counter, Karp et al. [9] proposed an improved FPRAS, henceforth denoted as KLM Counter, which has time complexity linear in the number of cubes. Vazirani [10] proposed a variant of KL Counter (denoted Vazirani Counter) with same time complexity as KL Counter, but combined with an enhancement proposed in [11], it requires fewer Monte Carlo samples than KL Counter.

Recently, Chakraborty et al. [12] showed that the hashing-based framework, which was originally proposed for approximate counting of CNF formulas, lends to an FPRAS scheme for #DNF as well. In particular, Chakraborty et al. proposed a hashing-based scheme called DNFApproxMC, whose time complexity was significantly worse than that of KLM Counter. Building on Chakraborty et al., Meel et al. [13] proposed an improvement to DNFApproxMC, which we refer to as SymbolicDNFApproxMC. The time complexity of SymbolicDNFApproxMC is $\tilde{O}(\mathsf{mn}\log(1/\delta)/\varepsilon^2)$, which is within polylog factors of that of KLM Counter.

Two key questions however, are still unanswered: 1) Is it possible to remove the polylog factors in the complexity of SymbolicDNFApproxMC? 2) How do the various approaches perform empirically? The desire to make an inquiry into the runtime performance of different FPRAS is not just intellectual; it stems from the fruitful results such a study has produced in the development of theory and tools for approximate CNF-Counting [14,15]. Despite the fact that some FPRAS have been around for over 30 years, a comprehensive experimental evaluation has not been performed for #DNF, to the best of our knowledge.

In this paper, we propose a new search technique for hashing-based algorithms that improves the complexity of SymbolicDNFApproxMC to $\mathcal{O}(\mathsf{mn}\log(1/\delta)/\varepsilon^2)$, which is the same as KLM Counter. Further, we present the first empirical study of runtime behavior of different FPRASs for #DNF. Similar to previous studies for SAT solvers, we conduct our study on classes of randomly generated DNF formulas covering a broad range of distribution parameters. The result of our study produces a nuanced picture. First of all, we observe that there is no single best algorithm that outperforms all other algorithms for all classes of formulas and input parameters. Second, we observe that the algorithm with one of the worst time complexities, $\mathsf{DNFApproxMC}$, solves the largest number of benchmarks. We believe that the above two results are significant as they demonstrate a gap between runtime performance and theoretical time complexity of approximate techniques for #DNF. Similar to studies of #CNF, this gap should serve as a guiding light for designing new #DNF algorithms, and for analyzing the structure of solution space of DNF formulas.

The rest of the paper is organized as follows: we introduce some notation in Section 2 and briefly review the various approaches to approximate DNF-Counting in Section 3. We present our new search procedure for hashing algorithms in Section 4. We describe experimental methodology in Section 5 and report on the results in Section 6. We offer our interpretation of these results in Section 7, and conclude in Section 8.

2 Preliminaries

A literal is a variable or the negation of a variable. A formula F over boolean variables is in Disjunctive Normal Form (DNF) if it is a disjunction over conjunctions of literals. Disjuncts in the formula are called *cubes* and we denote the i^{th}cube by F^i. Thus $F = F^1 \vee F^2 \vee \cdots \vee F^{\mathsf{m}}$. We will use n and m to denote the number of variables and number of cubes in the input DNF formula, respectively. The width of a cube F^i refers to the number of literals in cube F^i and is denoted by $\mathsf{width}(\mathsf{F}^\mathsf{i})$. We use w to denote the minimum of width over all the cubes of the formula, i.e. $\mathsf{w} = \min_i \mathsf{width}(\mathsf{F}^\mathsf{i})$.

We use $\Pr[A]$ to denote probability of an event A. For a given random variable Y, we use $\mathsf{E}[Y]$ and $\mathsf{V}[Y]$ to denote expectation and variance of Y.

We use capital boldface letters $\boldsymbol{A}$,$\boldsymbol{B}, \ldots$ to denote matrices, small boldface letters $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}, \ldots$ to denote vectors. We denote by $\boldsymbol{A}^{(\mathsf{p})}$ the sub-matrix of $\boldsymbol{A}$ consisting of the first p rows. Similarly, $\boldsymbol{b}^{(\mathsf{p})}$ denotes the sub-vector of $\boldsymbol{b}$ consisting of the first p elements of $\boldsymbol{b}$. We refer to $\boldsymbol{A}^{(\mathsf{p})}$ and $\boldsymbol{b}^{(\mathsf{p})}$ as "prefix-slices" of $\boldsymbol{A}$ and $\boldsymbol{b}$ respectively.

An assignment (vector) $\boldsymbol{x}$ of truth values to variables of F is called a satisfying assignment or witness if it makes F evaluate to true. Finding a satisfying assignment if one exists can be accomplished in polynomial time for DNF formulas. We denote the set of all satisfying assignments of F by $\mathcal{R}_F$. Given F, the constrained counting problem is to compute $|\mathcal{R}_F|$. A *fully polynomial randomized approximation scheme* (FPRAS) is a randomized algorithm that takes as input a formula F, a tolerance $\varepsilon \in (0, 1)$ and confidence parameter $\delta \in (0, 1)$ and outputs a random variable Y such that $\Pr[(1-\varepsilon)|\mathcal{R}_F| \leq Y \leq (1+\varepsilon)|\mathcal{R}_F|] \geq 1-\delta$ and the running time of the algorithm is polynomial in $|F|$, $1/\varepsilon$, $\log(1/\delta)$.

A hash function $h : \{0, 1\}^{\mathsf{q}} \to \{0, 1\}^{\mathsf{p}}$ partitions the elements of the domain $\{0, 1\}^{\mathsf{q}}$ into 2^{p} cells. $h(\boldsymbol{x}) = \boldsymbol{y}$ implies that h maps the assignment $\boldsymbol{x}$ to the cell $\boldsymbol{y}$. $h^{-1}(\boldsymbol{y}) = \{\boldsymbol{x}|h(\boldsymbol{x}) = \boldsymbol{y}\}$ is the set of assignments that map to the cell $\boldsymbol{y}$. We will be interested in calculating the cardinality of $\mathcal{R}_F \cap h^{-1}(\boldsymbol{y})$ for a randomly chosen h.

Hash functions of the form $h(\boldsymbol{x}) = \boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})}$ are commonly used in approximate counting. A base matrix $\boldsymbol{A}$ of dimension $\mathsf{q} \times \mathsf{q}$ is randomly sampled from a special set called a hash family. Similarly, base vectors $\boldsymbol{b}$ and $\boldsymbol{y}$ are chosen uniformly at random from $\{0,1\}^{\mathsf{q}}$. To obtain a hash function $h : \{0,1\}^{\mathsf{q}} \to \{0,1\}^{\mathsf{p}}$ and a cell in $\{0,1\}^{\mathsf{p}}$, the prefix-slices $\boldsymbol{A}^{(\mathsf{p})}, \boldsymbol{b}^{(\mathsf{p})}$ and $\boldsymbol{y}^{(\mathsf{p})}$ are constructed. Thus the hash function and the cell $h(\boldsymbol{x}) = \boldsymbol{y}$ is a system of linear equations modulo 2: $\boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})} = \boldsymbol{y}^{(\mathsf{p})}$. The solutions to this system of linear equations are the elements of the set $h^{-1}(\boldsymbol{y})$.

We will use the triple $\boldsymbol{A}^{(\mathsf{p})}, \boldsymbol{b}^{(\mathsf{p})}, \boldsymbol{y}^{(\mathsf{p})}$ to denote a hash function and a cell. We obtain different families of hash functions depending on the constraints imposed on the structure of the matrix $\boldsymbol{A}$. For example, if each element of $\boldsymbol{A}$ is chosen uniformly at random, we obtain a hash function from the random XOR family [16]. If $\boldsymbol{A}$ is sampled from the set of matrices in Reduced Row Echelon form, we obtain a hash function from the Row Echelon XOR family [13]. The technique for enumerating solutions in a cell also depends on the family of the hash function under consideration.

3 Approximation Algorithms for #DNF

Beginning with the seminal work of Karp and Luby [8], three Monte Carlo FPRASs for #DNF have been designed over the years [9,10]. Two more FPRASs were designed using the new hashing-based approach [12,13]. Besides developing FPRASs, considerable effort has also gone into developing *deterministic* approximation algorithms for #DNF [17,18,19] and the closely related problem of designing pseudorandom generators with short seeds [20,21,22]. The development of a fully polynomial time deterministic approximation algorithm for #DNF is still an open problem [19].

Motivated by applications of #DNF to probabilistic databases, several approaches to the design of approximate #DNF counters have been investigated from the perspective of query evaluation as well [23,24,25]. Such algorithms, however, either take exponential time in the worst case [23,25] or are designed to work on restricted classes of formulas such as monotone, read-once etc. [24]. An FPRAS similar to KL Counter was developed in the Multi-Instance Learning community for evaluating SVM kernels [26]. The FPRAS is designed to count the number of axis-parallel boxes that contain given points. However, the algorithm is identical to KL Counter when the problem instance is reduced to a DNF formula.

In summary, there is intense interest in practical applications of #DNF and a number of algorithmic schemes have been designed towards that end. The strongest guarantees on worst-case running time are provided by FPRASs, yet there does not exist a comprehensive experimental evaluation comparing them. In this work, we perform the first such empirical study of runtime behavior of different FPRASs. Before delving into experimental setup, we briefly review the five FPRASs from an algorithmic perspective. The purpose is two-fold:

1. to provide a unified overview of the state-of-the-art FPRASs for #DNF, and
2. to shed some light on the subtle differences within each variant algorithm of the Monte Carlo and Hashing frameworks. While the differences may seem inconsequential from a distance, our experiments show that they make a significant difference in practice.

Algorithm 1 Monte-Carlo-Count($\mathcal{A}$, $\mathcal{U}$)

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

3.1 Monte Carlo Framework

Algorithms built on Monte Carlo framework are randomized algorithms whose output can be wrong with a certain (usually small) probability [27]. Typically, these algorithms rely on drawing independent random samples to obtain numerical results. We refer the reader to [28] for further details. In the context of counting, the abstract Monte Carlo framework for finding cardinality of a set $\mathcal{A}$ in the universe $\mathcal{U}$ is shown in Algorithm 1.

In Algorithm 1, Y is an unbiased estimator for $\rho = |\mathcal{A}|/|\mathcal{U}|$. ρ is called the density of solutions. Also, Z is an unbiased estimator for $|\mathcal{A}|$. If $N = \mathcal{O}(\frac{\mathsf{V}[Z]}{\mathsf{E}[Z]^2}\log(1/\delta)/\varepsilon^2)$, we have $\Pr[(1-\varepsilon)|\mathcal{A}| \leq Z \leq (1+\varepsilon)|\mathcal{A}|] \geq 1-\delta$.

Algorithm 1 is an FPRAS if the number of samples N, and the time taken by line 3 and 4 are polynomial in the size of input[1].

In the context of this work, we have $\mathcal{A} = \mathcal{R}_F$. If F is a DNF formula with n variables and m cubes, we can employ Algorithm 1 by defining $\mathcal{U}$ to be the set of all assignments over n variables. A naive lower bound on $|\mathcal{R}_F|$ is $2^{\mathsf{n}-\mathsf{w}}$, where w is the minimum over width of all the cubes of F. If w is a small constant, then $\frac{1}{\rho} \geq \frac{1}{2^{\mathsf{w}}}$ which is polynomial in n and m and hence we require polynomially many samples. But if w is $O(\mathsf{n})$, then the lower bound does not polynomially bound the number of samples required which implies that this naive Monte Carlo counter is not an FPRAS.

The key insight by Karp et al. is to transform $\mathcal{R}_F$ and $\mathcal{U}$ into $\mathcal{R}'_F$ and $\mathcal{U}'$ such that $\frac{1}{\rho'} = |\mathcal{U}'|/|\mathcal{R}_F|$ is polynomially bounded, and it is also possible to recover $|\mathcal{R}_F|$ from $|\mathcal{R}'_F|$. We now discuss various transformations proposed over the years and the FPRASs these transformations yield.

KL Counter

Karp and Luby [8] developed the first FPRAS for #DNF, which we refer to as KL Counter. They defined a new universe $\mathcal{U}' = \{(\boldsymbol{x}, F^i) \mid \boldsymbol{x} \models F^i\}$, and the corresponding solution space $\mathcal{R}'_F$ as $\mathcal{R}'_F = \{(\boldsymbol{x}, F^i) \mid \boldsymbol{x} \models F^i \text{ and } \forall j < i, \boldsymbol{x} \not\models F^j\}$ for a fixed ordering of the cubes. They showed that $|\mathcal{R}_F| = |\mathcal{R}'_F|$ and that the ratio $|\mathcal{U}'|/|\mathcal{R}'_F| \leq \mathsf{m}$ and is therefore polynomially bounded. Consequently, the time complexity of the algorithm is $\mathcal{O}(\mathsf{m}^2\mathsf{n}\log(1/\delta)/\varepsilon^2)$. For our experiments, we employ an enhancement suggested in [11] which ensures optimal estimation of N. The enhancement is applicable, since the estimator used by KL Counter is a 0–1 estimator.

[1] Note that $\mathcal{A}$ is typically represented implicitly such as using constraints in DNF in the context of this paper

KLM Counter

Karp et al. [9] proposed an improvement of KL Counter by employing a non 0–1 estimator. To this end, the concept of 'coverage' of an assignment $\boldsymbol{x}$ in $\mathcal{U}'$ is introduced as $cover(\boldsymbol{x}) = \{j | \boldsymbol{x} \models F^j\}$. The first key insight is that $|\mathcal{R}'_F| = \sum_{(\boldsymbol{x},F^i)\in\mathcal{U}'} \frac{1}{|cover(\boldsymbol{x})|}$. The second insight was to define an estimator for $1/|cover(\boldsymbol{x})|$ using the geometric distribution. It is shown that the time complexity of KLM Counter is $\mathcal{O}(\mathsf{mn}\log(1/\delta)/\varepsilon^2)$, which is an improvement over KL Counter.

Vazirani Counter

A variant of KLM Counter was described in Vazirani [10], where $|cover(\boldsymbol{x})|$ is computed exactly by iterating over all cubes, avoiding the use of the geometric distribution. The advantage of Vazirani Counter, is that it is able to utilize the enhancement proposed in [11]. Consequently, Vazirani Counter requires fewer samples than KL Counter to achieve the same error bounds. The time for generating a sample, however, can be considerably more since the check for $\boldsymbol{x} \models F^j$ has to be performed for all cubes.

3.2 Hashing Framework

The key idea behind hashing-based counting is to partition the solution space of a given formula into *roughly equal small* cells of solutions, using randomly chosen 2-universal hash functions [16]. The crux of the framework is a search for the right number of hash constraints such that the number of solutions in a cell – $Y_{cell} = |\mathcal{R}_F \cap h^{-1}(\boldsymbol{y})|$ – is not too large, yet the tolerance and confidence obtained are as required. To calculate Y_{cell}, all the solutions in a randomly chosen cell are enumerated. If Y_{cell} is greater than a threshold $\mathsf{hiThresh} \in \mathcal{O}(1/\varepsilon^2)$, then the number of constraints are increased. The search ends when the number of hash constraints p is such that (1) $Y_{cell} < \mathsf{hiThresh}$ and (2) $Y_{cell} \geq \mathsf{hiThresh}$ when number of hash constraints is $\mathsf{p}-1$. The usage of 2-universal hash functions guarantees that the random variable Y_{cell} has low variance. Therefore, the final estimate $Y_{cell} \times 2^{\mathsf{p}}$, where 2^{p} is the total number of cells, is a good approximation of $|\mathcal{R}_F|$.

The abstract hashing-based counting framework is shown in Algorithm 2. The procedure ApproxMCCore (Algorithm 3) is invoked $t \in \mathcal{O}(\log(1/\delta))$ times in Algorithm 2, to get the required confidence $1 - \delta$ using majority vote. ApproxMCCore assumes access to a sub-procedure SampleHashFunction for sampling the base matrix and vectors $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$ as well as the number of variables in the hash function q. Note that q is not necessarily the same as the number of variables in the formula n. The procedure SampleHashFunction depends on the particular hash family used. A search sub-procedure is invoked in line 2 which returns the correct number of hash constraints p and the corresponding Y_{cell}. A binary search can be employed for this purpose, which is shown in Algorithm 4. The range of values of p to search, is provided by the functions GetLowerBound and GetUpperBound which depend on the input formula. The list FailRecord maintains the values of p for which $Y_{cell} < \mathsf{hiThresh}$ with $\mathsf{FailRecord}[p] = 0$ and those p for which $Y_{cell} \geq \mathsf{hiThresh}$ by $\mathsf{FailRecord}[p] = 1$. The search returns when p is found such that $\mathsf{FailRecord}[\mathsf{p}] = 0$ and $\mathsf{FailRecord}[\mathsf{p} - 1] = 1$. The procedure BSAT (Algorithm 5) is invoked for calculating

Algorithm 2 ApproxMC(F, ε, δ)

```
1: hiThresh ← $\mathcal{O}(\frac{1}{\varepsilon^2})$;
2: $t \leftarrow \mathcal{O}(\log(\frac{1}{\delta}))$;
3: EstimateList ← emptyList;
4: repeat $t$ times
5:     (numCells, $Y_{cell}$) ← ApproxMCCore($F$, hiThresh);
6:     AddToList(EstimateList, $Y_{cell}$ × numCells);
7: finalEstimate ← FindMedian(EstimateList);
8: return finalEstimate
```

Y_{cell}. Each time a solution is found using EnumerateNextSol, Y_{cell} is incremented by an amount calculated using the function ComputeIncrement, which can be instantiated to suit the particular counting problem. The procedure EnumerateNextSol depends on the type of formula F, as well as the family of the hash function $\boldsymbol{A}, \boldsymbol{b}$. The hash family also determines how a prefix slice is obtained from the call to Extract.

DNFApproxMC

Concrete counting algorithms for a class of formulas can be obtained from the above framework by choosing an appropriate family of hash functions along with the corresponding procedures SampleHashFunction, GetLowerBound, GetUpperBound, ComputeIncrement, Extract and EnumerateNextSol. For example, Chakraborty et al. [12] obtained an FPRAS for DNF formulas with complexity $\mathcal{O}((\mathsf{mn}^3 + \mathsf{mn}^2/\varepsilon^2)\log \mathsf{n} \log(1/\delta))$, using Random XOR hash functions with SampleHashFunction and Extract along with Gaussian Elimination for EnumerateNextSol.The upper bound, lower bound and increment were fixed to n,0 and 1 respectively. We denote the resulting algorithm as DNFApproxMC. In our experiments, we augmented DNFApproxMC with Row-Echelon Hash family (proposed in [13]), which improves the complexity from cubic to quadratic in n leading to better performance on all benchmarks.

SymbolicDNFApproxMC

The algorithm SymbolicDNFApproxMC proposed in [13] achieves better worst-case time complexity, made possible by three improvements over the original DNFApproxMC algorithm. First, the usage of Row Echelon hash functions eliminates the need for expensive Gaussian Elimination procedure. The concept of Symbolic Hashing enables hashing over a transformed solution space without modifying the input formula. Lastly, it was shown that a probabilistic estimate of Y_{cell} can be used in place of an exact count. The complexity of SymbolicDNFApproxMC is $\tilde{O}(\mathsf{mn}\log(1/\delta)/\varepsilon^2)$, which stems from the use of BinarySearch. We now present a new search technique called ReverseSearch (Algorithm 6), that removes the polylog factors (hidden in the $\tilde{O}$ notation) from the complexity of SymbolicDNFApproxMC to make it at par with the complexity achieved KLM Counter, and also improves its running time in practice.

Algorithm 3 ApproxMCCore(F, hiThresh)

```
1: $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}, \mathsf{q} \leftarrow$ SampleHashFunction();
2: $Y_{cell}, \mathsf{p} \leftarrow Search(F, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}, \mathsf{q}, \mathsf{hiThresh})$;
3: return $(2^{\mathsf{p}}, Y_{cell})$
```

Algorithm 4 BinarySearch($F, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$, q, hiThresh)

```
1: lo $\leftarrow$ GetLowerBound(); hi $\leftarrow$ GetUpperBound();
2: FailRecord[lo] $\leftarrow 1$; FailRecord[hi] $\leftarrow 0$;
3: FailRecord[$i$] $\leftarrow \perp$ for all $i$ other than lo and hi;
4: while $true$ do
5:     p $\leftarrow$ (hi + lo)/2;
6:     $\boldsymbol{A}^{\mathsf{p}}, \boldsymbol{b}^{\mathsf{p}}, \boldsymbol{y}^{\mathsf{p}} \leftarrow$ Extract($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$,p);
7:     $Y_{cell} \leftarrow$ BSAT($F, \boldsymbol{A}^{\mathsf{p}}, \boldsymbol{b}^{\mathsf{p}}, \boldsymbol{y}^{\mathsf{p}}$, q, hiThresh);
8:     if ($Y_{cell} \geq$ hiThresh) then
9:         if (FailRecord[p + 1] = 0) then
10:            $Y_{cell} \leftarrow$ BSAT($F, \boldsymbol{A}^{\mathsf{p}+1}, \boldsymbol{b}^{\mathsf{p}+1}, \boldsymbol{y}^{\mathsf{p}+1}$, q, hiThresh);
11:            return $Y_{cell}$, p + 1;
12:        FailRecord[$i$] $\leftarrow 1$ for all $i \in \{$lo, ... p$\}$;
13:        lo $\leftarrow$ p;
14:    else
15:        if (FailRecord[p − 1] = 1) then return $Y_{cell}$, p;
16:        FailRecord[$i$] $\leftarrow 0$ for all $i \in \{$p, ... hi$\}$;
17:        hi $\leftarrow$ p;
```

Algorithm 5 BSAT($F, \boldsymbol{A}^{\mathsf{p}}, \boldsymbol{b}^{\mathsf{p}}, \boldsymbol{y}^{\mathsf{p}}$, q, threshold)

```
1: $Y_{cell} \leftarrow 0$;
2: while $true$ do
3:     $s \leftarrow$ EnumerateNextSol($F, \boldsymbol{A}^{\mathsf{p}}, \boldsymbol{b}^{\mathsf{p}}, \boldsymbol{y}^{\mathsf{p}}$);
4:     if $s \neq \perp$ then
5:         $Y_{cell}$ = ComputeIncrement($s, Y_{cell}$, threshold);
6:     else
7:         return $Y_{cell}$;
8:     if $Y_{cell} \geq$ threshold then
9:         return threshold;
```

4 Reverse Search for Hashing-Based Algorithms

A close inspection of the SymbolicDNFApproxMC algorithm in [13] reveals that the polylog factors in the complexity analysis arise due to redundancy in enumerating solutions in successive calls to BSAT. In particular, the fact that the set $\{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})} = \boldsymbol{y}^{(\mathsf{p})}\}$ is a subset of $\{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p-1})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p-1})} = \boldsymbol{y}^{(\mathsf{p-1})}\}$ is not exploited. Each call to BSAT is agnostic of the previous ones, resulting in repeated enumeration of solutions. One work-around could be to buffer solutions from a call to BSAT in order to reuse them in the future. However, this involves additional space overhead and is not suitable when constraints are removed during binary search. Instead, we propose a different search technique which guarantees that every solution to the hash function is enumerated at most once, by eliminating redundancy during search space exploration. The technique makes use of the fact that the set $\{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p-1})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p-1})} = \boldsymbol{y}^{(\mathsf{p-1})}\}$ can be partitioned into $\{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})} = \boldsymbol{y}^{(\mathsf{p})}\}$ and $\{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})} = \boldsymbol{y}^{(*\mathsf{p})}\}$, where $\boldsymbol{y}^{(*\mathsf{p})}$ is the vector $\boldsymbol{y}^{(\mathsf{p})}$ with the pth (last) bit negated.

Algorithm 6 ReverseSearch($F, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$, q, hiThresh)

```
1: $Y_{total} = 0$;
2: hi ← GetUpperBound();
3: lo ← GetLowerBound();
4: p ← hi;
5: $\boldsymbol{A}^{(p)}, \boldsymbol{b}^{(p)}, \boldsymbol{y}^{(p)}$ ← ExtractSlice($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$,p, flip = false);
6: $Y_{cell}$ ← BSAT($F, \boldsymbol{A}^{(p)}, \boldsymbol{b}^{(p)}, \boldsymbol{y}^{(p)}$, q, hiThresh);
7: $Y_{total} = Y_{total} + Y_{cell}$;
8: if ($Y_{total} \geq$ hiThresh) then return hiThresh, p;
9: for p = hi; p ≥ lo; p = p − 1 do
10:     $\boldsymbol{A}^{(p)}, \boldsymbol{b}^{(p)}, \boldsymbol{y}^{(*p)}$ ← ExtractSlice($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$,p, flip = true);
11:     $Y_{cell}$ ← BSAT($F, \boldsymbol{A}^{(p)}, \boldsymbol{b}^{(p)}, \boldsymbol{y}^{(*p)}$, q, hiThresh − $Y_{total}$);
12:     $Y_{total} = Y_{total} + Y_{cell}$;
13:     if ($Y_{total} \geq$ hiThresh) then return ($Y_{total} - Y_{cell}$), p;
```

Algorithm 6 depicts procedure ReverseSearch. Y_{total} maintains the count of all the solutions enumerated so far. In lines 2-3, the bounds for the search for the right p are obtained. In line 5, a prefix slice with p $= hi$ constraints is extracted. We assume access to a procedure ExtractSlice which requires a slight modification of procedure Extract in [13]. ExtractSlice takes an additional Boolean argument 'flip' which determines if the last bit of $\boldsymbol{y}$ is to be flipped or not. The details of this procedure are provided in the Appendix. In line 8, if the cell-count obtained in line 6 is found to exceed hiThresh, then it implies that the true count is within $(1+\varepsilon)$ factor of 2^{q} with high probability, and the algorithm returns (hiThresh, p). Otherwise, the for-loop in line 9 is executed. In lines 10-11, $Y_{cell} = |\mathcal{R}_F \cap \{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})} = \boldsymbol{y}^{(*\mathsf{p})}\}|$ is evaluated by setting the 'flip' argument to true in ExtractSlice. After execution of line 12, we have that $Y_{total} = |\mathcal{R}_F \cap \{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p}-1)}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p}-1)} = \boldsymbol{y}^{(\mathsf{p}-1)}\}|$. Therefore, when Y_{total} exceeds hiThresh, the hash count along with the cell-count of the previous iteration are returned in line 13.

Theorem 1 *The complexity of* SymbolicDNFApproxMC*, when invoked with* ReverseSearch *is* $\mathcal{O}(\mathsf{mn}\log(1/\delta)/\varepsilon^2)$

Proof Sketch We defer the full proof to the appendix. The core sub-procedure of SymbolicDNFApproxMC is to obtain a probabilistic estimate of Y_{cell} in each invocation of BSAT. This is done as follows: 1) A solution $\boldsymbol{x}$ of the hash function is enumerated 2) Cubes of the input formula F are randomly sampled until a cube F^i is found such that $\boldsymbol{x} \models F^i$ 3) The number of steps required to find such a cube is used to calculate an estimator for Y_{cell}. The complexity of each such sample-and-check is $\mathcal{O}(\mathsf{n})$.

The effect of the use of binary search in [13] was two-fold. Firstly, BSAT was invoked $\mathcal{O}(\log\log \mathsf{m})$ times. Secondly, each call to BSAT possibly required the sampling of m × hiThresh cubes. The use of ReverseSearch, however, ensures that each call to BSAT is over a previously unexplored part of the solution space. This in turn ensures that exactly m × hiThresh cubes are sampled in total, instead of m × hiThresh × $\log\log \mathsf{m}$ as in [13]. Since sample-and-check is $\mathcal{O}(\mathsf{n})$, hiThresh $\in \mathcal{O}(1/\varepsilon^2)$ and SymbolicDNFApproxMCCore is invoked $\mathcal{O}(\log(1/\delta))$ times, the overall complexity is $\mathcal{O}(\mathsf{mn}\log(1/\delta)/\varepsilon^2)$. □

Naturally, one wonders whether employing ReverseSearch leads to gains in performance in practice. We compared the running times of SymbolicDNFApproxMC

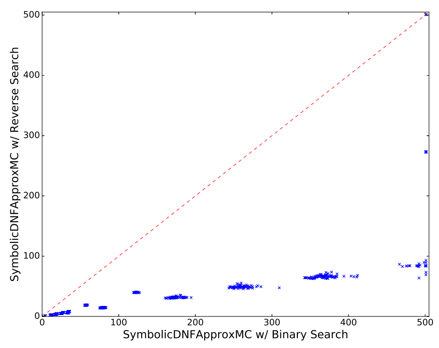

Fig. 1 Comparison of Running time of SymbolicDNFApproxMC with BinarySearch and ReverseSearch

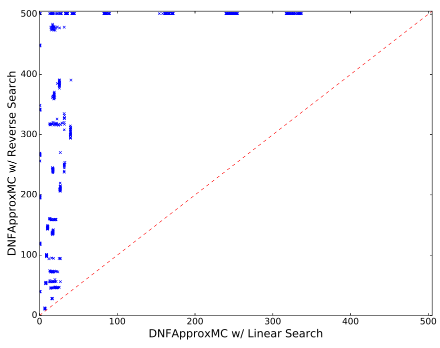

Fig. 2 Comparison of Running time of DNFApproxMC with LinearSearch and ReverseSearch

with BinarySearch and with ReverseSearch over wide classes of randomly generated DNF formulas with 100, 000 variables, number of cubes ranging from 10, 000 to 800, 000 and cube-widths ranging from 3 to 43. Figure 1 shows a scatter-plot of the results. A point (in blue) in the plot corresponds to one DNF formula in our test set. Its y-coordinate represents the time taken by SymbolicDNFApproxMC using ReverseSearch, while its x-coordinate represents time taken using BinarySearch. It can be seen that SymbolicDNFApproxMC with ReverseSearch is roughly four or five times faster than with BinarySearch. Therefore in the empirical study we describe next, we use ReverseSearch in all experiments involving SymbolicDNFApproxMC. Henceforth, we denote SymbolicDNFApproxMC with ReverseSearch as just SymbolicDNFApproxMC. Note, however, that DNFApproxMC does not benefit from ReverseSearch (Fig. 2). In fact, a simple linear search works best since our implementation uses efficient data structures for buffering solutions that obviate the need for reverse or binary searches.

5 Experimental Methodology

The objective of our experimental evaluation was to seek an answer for the following four key questions:

1. Runtime Variation: How does the running time of the algorithms vary across different benchmarks?
2. Benchmarks Solved: How many benchmarks can the algorithms solve overall?
3. Accuracy: How accurate are the counts returned by the algorithms?
4. $\varepsilon - \delta$ Scalability: How do the algorithms scale with the input tolerance and confidence?

For ease of exposition, we henceforth refer to the experiments corresponding to these questions as Runtime Variation, Benchmarks Solved, Accuracy and $\varepsilon - \delta$ Scalability respectively. A fair comparison requires careful consideration of several parameters, such as programming language of implementation, usage of libraries, configuration of the cluster, benchmark suite, measures of performance, and the

Table 1 Parameters used for generating random formulas and as input to algorithms

Experiment	**Formula Generation Parameters**			**Input Parameters**	
	#Vars n	#Cubes m	Width w	Tolerance ε	Confidence δ
Benchmarks Solved, Runtime Variation	100,000	$10^4 \leq \mathsf{m} < 9 \times 10^4$ steps of 2×10^4 & $10^5 \leq \mathsf{m} \leq 8 \times 10^5$ steps of 2×10^4	$3 \leq \mathsf{w} \leq 43$	0.8	0.36
Accuracy	$100 \leq \mathsf{n} < 1000$ & $1000 \leq \mathsf{n} \leq 7000$ variable step size	$30 \leq \mathsf{m} \leq 7000$ & $300 \leq \mathsf{m} \leq 35,000$ variable step size	$3 \leq \mathsf{w} \leq 2450$ variable step size	0.8	0.36
ε Scalability	100,000	50,000	12	$[0.04, 0.8]$	0.36
δ Scalability				0.8	$[0.03, 0.36]$

like. Given a long list of parameters, performing experimental evaluation of all possible combinations quickly becomes infeasible. Therefore, we had to arrive at choices for several parameters. We explain our rationale for all such choices and analyze the experimental results obtained.

5.1 Experimental Setup

We ran all experiments on a cluster. Each experiment had exclusive access to a node with Intel(R) Xeon(R) CPU E5-2650 v2 processors running at 2.60GHz. Only 1 core out of the 16 available on each node was used with a memory limit of 4GB. All algorithms were implemented in C++ and compiled with GCC version 5.4 with the O3 flag. To mitigate implementation bias, we used existing code and third-party libraries wherever possible. For instance, we used a library called M4RI [29] for implementing hash functions, GNU Bignum library for maintaining large counts. We adapted implementations of ApproxMC and Dagum et al.'s Monte Carlo enhancement from the ApproxMC and MayBMS [30] code-bases, respectively[2]. For a given algorithm and an input formula, we set the timeout to 500 seconds.

5.2 Benchmark Generation

To the best of our knowledge, there are no publicly-available standardized set of benchmarks for #DNF. We contacted the authors of works on probabilistic databases, but were unable to obtain non-synthetic benchmarks. This is because most works tend to rely on random data generators such as TPC-H [31] for testing prototype implementations of probabilistic databases [23,25].

Another approach could have been to use the complement of CNF formulas arising from works on CNF-Counting. Such CNF formulas, however, typically have counts that are exponentially smaller than 2^n. The DNF complements of those formulas thus have counts extremely close to 2^n. So naive Monte Carlo techniques would suffice.

There is a chicken-and-egg problem – lack of real-world benchmarks for testing prevents adoption of algorithms in practice, which in turn affects benchmark

[2] Code and results can be accessed at `https://gitlab.com/Shrotri/DNF_Counting`

availability. A salient goal of this work is to break this vicious cycle. A common trend in the CSP community is to use random benchmarks for empirical studies, when real-world problem instances are unavailable [32]. In the same vein, owing to a lack of publicly-available meaningful benchmarks, we conduct our study on random DNF formulas. Each formula with uniform cube-width was sampled as follows: To sample a cube, w variables were sampled uniformly at random, out of n possible choices and negated with probability 0.5. This process was repeated m times to get the final formula. For formulas with non-uniform cube-widths, the width of each cube was sampled uniformly at random between 3 and 43 in the previous procedure.

5.3 Parameters Used

The parameters used for generating random benchmarks for the various experiments is shown in Table 1. We used a set of 1080 benchmarks for experiments on Runtime Variation and Benchmarks Solved, covering a broad range of values of n, m, and w. We generated a different set of 600 much smaller formulas for the Accuracy experiment, as exact counts are needed to measure accuracy and the exact counter SharpSAT [33] timed out on most large formulas. For ε and δ Scalability, the idea was to find a setting of n, m, and w for which all FPRAS would take similar time with inputs $\varepsilon = 0.8$, $\delta = 0.36$, so as to provide a level playing field.

For all experiments besides Accuracy, the benchmark sets comprised of 20 random instances for each setting of n, m, and w. This was sufficient as we observed that the running time of all five algorithms tended to not vary much between instances. In particular, the median coefficient-of-variation for all algorithms was less than 18%; ergo the distribution of running times is sufficiently captured by the mean and adding more instances would provide no further insight.

Following previous studies of approximate counting techniques [34, 12], we used $\varepsilon = 0.8$ as base value for tolerance. Since the dependence of algorithms on δ is $\log(\frac{1}{\delta})$, we studied all the algorithms to find value of δ so that any value of δ smaller than that would simply require the algorithms more repetitions of the core algorithm. The value of δ computed from the above was 0.36, which we use in our experiments. For $\varepsilon - \delta$ Scalability, the respective value was varied while fixing the other to its base value.

6 Results

We ran experiments on Runtime Variation, Benchmarks Solved, Accuracy and $\varepsilon - \delta$ Scalability over a combined total of 1500+ benchmarks, requiring well over 3000 hours of computational effort on dedicated nodes.

6.1 Runtime Variation

We present a graph of the running time vs. the number of cubes for $\mathsf{w} = 3, 13, 23, 33, 43$ as well as for non-uniform cube-widths. This is shown in Figs. 3, 4, 5, 6, 7 and 9

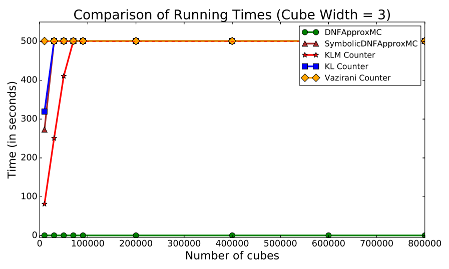

Fig. 3 Runtime Variation: DNFApproxMC is the best performer. Rest timeout.

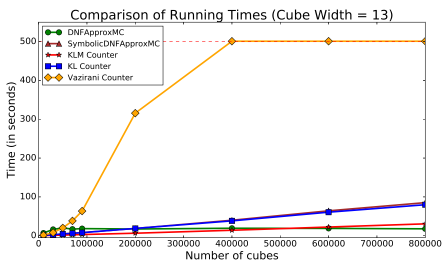

Fig. 4 Runtime Variation: DNFApproxMC and KLM Counter are the best performers

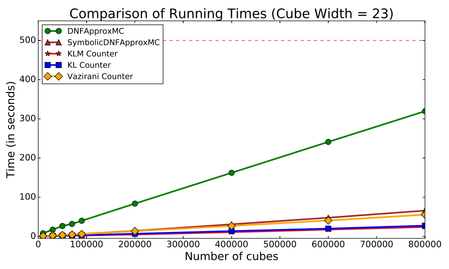

Fig. 5 Runtime Variation: KLM Counter and KL Counter are the best performers

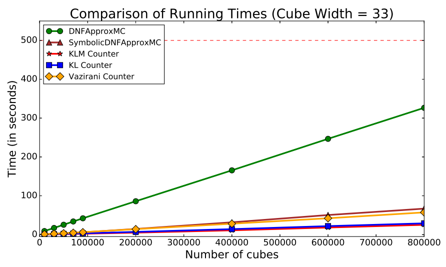

Fig. 6 Runtime Variation: KLM Counter and KL Counter are the best performers

respectively[3]. Each data point in the graphs represents the average running time of an algorithm over the 20 random formulas that were generated with the corresponding n, m and w. A note of caution should be exercised while interpreting results for small widths, as these formulas are easy for naive Monte Carlo strategies. For $w = 3$, we see that DNFApproxMC vastly outperforms other algorithms, taking under a second to solve all formulas (see: Fig. 3). Rest of the algorithms time out for formulas with number of cubes $m \geq 100,000$. For $w = 13$, it can be seen from Fig. 4 that DNFApproxMC and KLM Counter are the best performers. However, DNFApproxMC scales better with m. Vazirani Counter is the only algorithm to time out. For $w = 23$, we see that Monte Carlo algorithms, in particular KL Counter and KLM Counter, outperform the hashing-based algorithms. These algorithms also scale well with respect to m for $w = 23$. This trend continues for $w = 33$ and 43. We see that the behavior of the algorithms does not change above $w = 23$. For non-uniform widths, we see that the DNFApproxMC is again the best performer.

In summary, the performance of the Monte Carlo algorithms and SymbolicDNFApproxMC, improves significantly with the width of cubes, while DNFApproxMC dominates for low and non-uniform cube-widths and is more consistent overall.

6.2 Benchmarks Solved

Fig. 8 shows the cactus plot of all the different algorithms. We present the number

[3] Figures are best viewed online in color

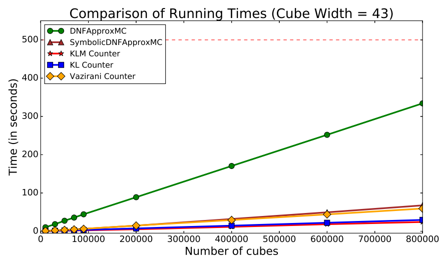

Fig. 7 Runtime Variation: KLM Counter and KL Counter are the best performers

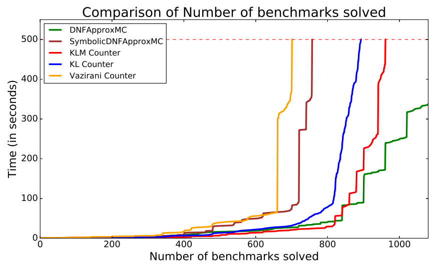

Fig. 8 Benchmarks Solved: DNFApproxMC solved all benchmarks

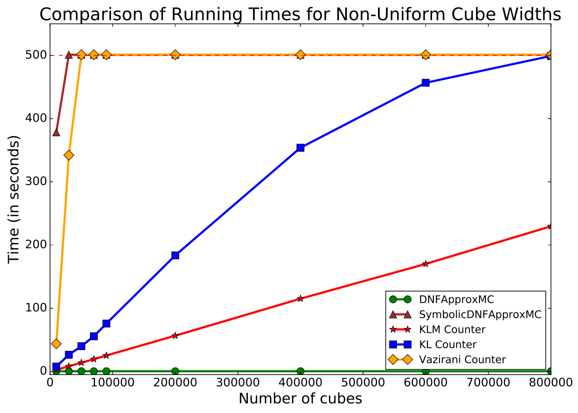

Fig. 9 Runtime Variation: DNFApproxMC dominates other algorithms

of benchmarks on x–axis and the total time taken on y–axis. A point (x, y) implies that x benchmarks took less than or equal to y seconds to solve. We see that DNFApproxMC completes all 1080 benchmarks in under 350 seconds which is well within the time limit of 500 seconds. All the other algorithms time out on at least 100 benchmarks.

6.3 Accuracy

Among the 600 formulas we generated for measuring accuracy, SharpSAT was able to return exact counts of 228 within a timeout of 8 hours for each. The observed mean and max errors of the counts returned by the five FPRAS for the 228 formulas, is shown in Table 2. If C is the exact count for a formula and Y is its estimate, then the error is calculated as $|C - Y|/C$. The errors for all algorithms are well within the tolerance $\varepsilon = 0.8$, that the algorithms were invoked with.

6.4 ε - δ Scalability

Fig. 10 shows the average time taken by the five algorithms over 20 instances when

Table 2 Accuracy of algorithms (invoked with $\varepsilon = 0.8$, $\delta = 0.36$)

Algorithm	Mean Error	Max Error
DNFApproxMC	0.09	0.36
SymbolicDNFApproxMC	0.21	0.42
KLM Counter	0.11	0.55
KL Counter	0.007	0.20
Vazirani Counter	0.001	0.04

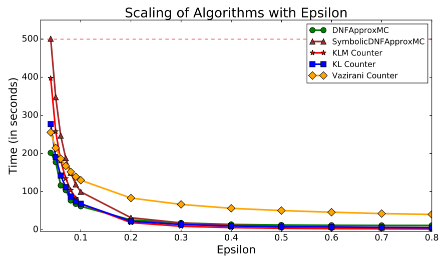

Fig. 10 ε Scalability: DNFApproxMC scales better than other algorithms

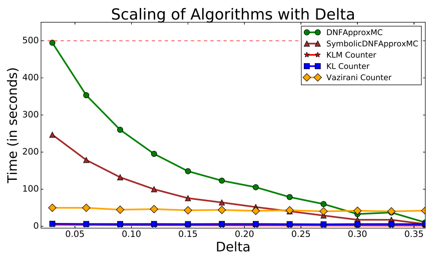

Fig. 11 δ Scalability: Monte Carlo FPRAS scale better

ε is varied between 0.04 and 0.8, keeping δ fixed at 0.36. The time complexity of all algorithms varies quadratically with $1/\varepsilon$, which also can be seen in the plotted curves. Nevertheless, DNFApproxMC scales better with $1/\varepsilon$ than all other algorithms.

Fig. 11 depicts the average time taken by the algorithms over the same 20 instances when δ is varied between 0.03 and 0.36, keeping ε fixed at 0.8. The time complexity of all five FPRAS has a $\mathcal{O}(\log(1/\delta))$ factor. However, the Monte Carlo algorithms scale extremely well for small δ, while SymbolicDNFApproxMC quickly times out, and DNFApproxMC also loses steam.

7 Discussion

The experiments on Runtime Variation and Benchmarks Solved make sense in the light of two key observations:

1. The counts of random DNF formulas tend to be extremely close to the upper-bound, i.e. $|\mathcal{R}_F| \approx min(2^n, \mathsf{m}*2^{\mathsf{n}-\mathsf{w}})$, a trend which was confirmed by the exact counts of SharpSAT
2. No. of samples required by the Monte Carlo FPRAS varies inversely with the solution density in the transformed space, i.e. $N \propto \frac{1}{\rho'}$ where $\rho' = \frac{|\mathcal{R}_F|}{\mathsf{m}*2^{\mathsf{n}-\mathsf{w}}}$

Together these imply that ρ' is close to 1 for all random formulas with large cube-widths. In such cases Monte Carlo FPRAS perform exceedingly well. Conversely, ρ' is low for small cube-widths and the Monte Carlo FPRAS time out. SymbolicDNFApproxMC too is affected adversely by small ρ' because of the symbolic space transform. In contrast, the running time of DNFApproxMC does not depend as heavily on either ρ or ρ', and therefore does not timeout on any formula

(Fig. 8).Thus DNFApproxMC is more robust across different formula types. This is also apparent in the experiment on formulas with non-uniform cube-widths (Fig. 9). The presence of a few short cubes in a formula is sufficient to make ρ' low, which enables DNFApproxMC to significantly dominate other algorithms.

The Monte Carlo algorithms perform substantially better than the hashing-based approaches in terms of δ Scalability. This can be attributed to the fact that the core sub-procedure of the hashing variants has to be repeated in order to boost confidence, which incurs a significant overhead. In contrast, for the Monte Carlo algorithms, only the number of samples required increases, which has low overhead. However, the marginal utility obtained by using small values for δ is debatable, as Table 2 shows that the counts returned by all five FPRAS are well within the input tolerance even for $\delta = 0.36$.

DNFApproxMC scales better with ε than the other FPRAS as seen in Fig. 10. We believe this is due to the use of efficient data structures for buffering solutions, in the implementation of DNFApproxMC . Algorithmic differences preclude the use of these data structures in the other FPRAS.

The best accuracy is obtained by Vazirani Counter (Table 2). However, this comes at a price. Vazirani Counter is markedly slower than KLM Counter and KL Counter despite requiring fewer samples. This is due to the additional time required by Vazirani Counter to generate a sample.

In summary, KLM Counter and KL Counter are the algorithms of choice when ρ' is known to be high. Naive Monte Carlo is sufficient when ρ is close to 1. However, when there is no information about the formula or when ρ and ρ' are known to be low, DNFApproxMC is a safe bet.

8 Concluding Remarks

Designing model counters for DNF formulas has been of practical as well as theoretical interest owing to applications in diverse domains in AI and beyond. Building on Chakraborty et al. [12], Meel et al. [13] proposed a hashing-based algorithm, SymbolicDNFApproxMC, whose time complexity was shown to be within polylog factors of the best known Monte Carlo schemes. Meel et al. left two key questions answered: (1) Are hashing-based techniques as powerful as Monte Carlo, i.e. is it possible to remove the polylog factors in the complexity of SymbolicDNFApproxMC?, and (2) How do the various approaches perform?

This paper provides positive answers to these questions. In particular, we first introduced a new reverse-search technique that makes the time complexity of a hashing-based FPRAS at par with the state-of-the art Monte Carlo techniques. Furthermore, our proposed scheme leads to up to $4-5\times$ gains over the previous scheme proposed by Meel et al. [13]. Moreover, the reverse-search is an enhancement of the general hashing-based counting framework, and is not limited to DNF-Counting, thereby opening future directions of research of its application to #CNF.

We also provided the first empirical study of the various FPRASs for #DNF. We compared three algorithms from the classical Monte Carlo framework, and two from the recently proposed hashing-based framework. Our experimental analysis leads to two important observations, which are not apparent from the theoretical analysis of these algorithms:

1. There is no panacea; different algorithms are well suited for different formula types and input parameters.
2. DNFApproxMC solves the most the number of benchmarks and is robust across different classes of formulas, despite poor complexity.

Owing to comprehensive testing on a wide array of formula classes and input parameters, we believe that these observations will carry over to real-world benchmarks as well. These observations illustrate a gap between theory and practice of #DNF which we hope will kick-start further empirical investigations and serve as a blueprint for future work on DNF-Counting.

Acknowledgements

Work supported in part by NSF grant IIS-1527668, NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", and by NUS ODPRT Grant, R-252-000-685-133. This is a post-peer-review, pre-copyedit version of an article published in Constraints. The final authenticated version is available online at: `https://link.springer.com/article/10.1007/s10601-018-9301-x`

References

1. L. Dueñas-Osorio, K.S. Meel, R. Paredes, M.Y. Vardi, SAT-based connectivity reliability estimation for power transmission grids. Tech. rep., Rice University (2017)
2. F. Bacchus, S. Dalmao, T. Pitassi, in *Proc. of FOCS* (2003), pp. 340–351. URL `http://dl.acm.org/citation.cfm?id=946243.946291`
3. T. Sang, P. Beame, H. Kautz, in *Prof. of AAAI* (2005), pp. 475–481
4. N. Dalvi, D. Suciu, The VLDB Journal **16**(4), 523 (2007)
5. F. Biondi, M. Enescu, A. Heuser, A. Legay, K.S. Meel, J. Quilbeuf, in *Proc. of VMCAI* (2018)
6. D.R. Karger, SIAM Review (2001)
7. L. Valiant, SIAM Journal on Computing **8**(3), 410 (1979)
8. R. Karp, M. Luby, Proc. of FOCS (1983)
9. R. Karp, M. Luby, N. Madras, Journal of Algorithms **10**(3), 429 (1989)
10. V.V. Vazirani, *Approximation algorithms* (Springer Science & Business Media, 2013)
11. P. Dagum, R. Karp, M. Luby, S. Ross, SIAM Journal on Computing **29**(5), 1484 (2000)
12. S. Chakraborty, K.S. Meel, M.Y. Vardi, in *Proc. of IJCAI* (2016)
13. K.S. Meel, A.A. Shrotri, M.Y. Vardi, in *Proc. of FSTTCS* (2017)
14. S. Ermon, C.P. Gomes, A. Sabharwal, B. Selman, in *Proc. of ICML* (2013), pp. 334–342
15. K.S. Meel, arXiv preprint arXiv:1806.02239 (2018)
16. J.L. Carter, M.N. Wegman, in *Proc. of STOC* (ACM, 1977), pp. 106–112
17. M. Luby, B. Veličković, Algorithmica **16**(4), 415 (1996)
18. L. Trevisan, in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques* (Springer, 2004), pp. 417–425
19. P. Gopalan, R. Meka, O. Reingold, Computational Complexity (2013)
20. M. Ajtai, A. Wigderson, in *Proc. of FOCS* (IEEE, 1985), pp. 11–19
21. N. Nisan, Combinatorica **11**(1), 63 (1991)
22. A. De, O. Etesami, L. Trevisan, M. Tulsiani, in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques* (Springer, 2010), pp. 504–517
23. D. Olteanu, J. Huang, C. Koch, in *ICDE* (IEEE, 2010), pp. 145–156
24. R. Fink, D. Olteanu, in *Proc. of ICDT* (ACM, 2011)
25. W. Gatterbauer, D. Suciu, ACM TODS **39**(1), 5 (2014)
26. Q. Tao, S. Scott, N. Vinodchandran, T.T. Osugi, in *Proceedings of the twenty-first international conference on Machine learning* (ACM, 2004), p. 101

27. L. Babai, Université tde Montréal Technical Report, DMS pp. 79–10 (1979)
28. R. Motwani, P. Raghavan, *Randomized algorithms* (2010)
29. M. Albrecht, G. Bard, *The M4RI Library – Version 20121224* (2012). URL `http://m4ri.sagemath.org`
30. J. Huang, L. Antova, C. Koch, D. Olteanu, in *Proc. of SIGMOD* (ACM, 2009)
31. TPC Benchmark H. `http://www.tpc.org/`
32. D. Mitchell, B. Selman, H. Levesque, in *Proc. of AAAI* (1992), pp. 459–465
33. M. Thurley, in *Proc. of SAT* (2006), pp. 424–429
34. S. Chakraborty, K.S. Meel, M.Y. Vardi, in *Proc. of CP* (2013), pp. 200–216

Appendix

For obtaining a concrete algorithm from the framework described in Algorithm 2, we need to instantiate the sub-procedures SampleHashFunction, GetLowerBound, GetUpperBound, EnumerateNextSol, ExtractSlice and ComputeIncrement for a particular counting problem. We now show how SymbolicDNFApproxMC [13], which uses Row Echelon XOR hash functions, and the concepts of Symbolic Hashing and Stochastic Cell-Counting, can be obtained through such instantiations. Then we prove that by substituting the BinarySearch procedure by ReverseSearch, the complexity of the resulting algorithm is improved by polylog factors.

SampleHashFunction

One can directly invoke the procedure SampleBase described in Algorithm 4 of [13] with minor modifications. This is shown in Algorithm 7. Note that the hash function $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$ so obtained belongs to the Row Echelon XOR family.

Algorithm 7 SampleHashFunction()

```
1: $\mathsf{q} \leftarrow \mathsf{n} - \mathsf{w} + \log \mathsf{m}$;
2: $sI \leftarrow \mathsf{n} - \mathsf{w} - \log \mathsf{hiThresh}$;
3: $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y} \leftarrow \mathsf{SampleBase}(\mathsf{q}, sI)$;
4: return $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}, \mathsf{q}$;
```

Lower and Upper Bounds

As shown in [13], it suffices to search between $\mathsf{n} - \mathsf{w} - \log \mathsf{hiThresh}$ and $\mathsf{n} - \mathsf{w} + \log \mathsf{m} - \log \mathsf{hiThresh}$ hash constraints. Therefore the functions GetLowerBound and GetUpperBound return these values respectively.

Extracting a prefix slice

Procedure ExtractSlice required for ReverseSearch is shown in Algorithm 8. If flip is false, ExtractSlice returns the result of the procedure Extract (described in [13]) directly. Otherwise, the p-th bit of $\boldsymbol{y}$ is negated before being passed to Extract.

Algorithm 8 ExtractSlice($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$, p, $flip$)

```
1: if $flip = true$ then
2:     $\boldsymbol{y}[\mathsf{p}] = \neg \boldsymbol{y}[\mathsf{p}]$;
3: $lo \leftarrow$ GetLowerBound;
4: return Extract($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$, p, q, $lo$);
```

EnumerateNextSol

SymbolicDNFApproxMC enumerates solutions in the cell, in the order of a Gray code sequence, for better complexity. This is achieved by invoking the procedure enumREX (Algorithm 1 in [13]).

ComputeIncrement

Procedure CheckSAT (Algorithm 10 adapted from [13]) can be used to compute the increments to Y_{cell} as shown in Algorithm 9. The assignment s is divided into a solution $\boldsymbol{x}$ and a cube F^i using the same Interpret function used in line 7 of Algorithm 6 in [13]. CheckSAT samples a cube at random in line 3 and checks if the assignment $\boldsymbol{x}$ satisfies it in line 5. The returned value follows the geometric distribution [9], and can be used to compute an accurate probabilistic estimate Y_{cell} of the true number of solutions in the cell [13].

Algorithm 9 ComputeIncrement(s, Y_{cell}, threshold)

```
1: $\boldsymbol{x}, F^i \leftarrow interpret(s)$;
2: return $Y_{cell}$ + CheckSAT($\boldsymbol{x}, F^i, Y_{cell}$, threshold);
```

Algorithm 10 CheckSAT($\boldsymbol{x}, F^i, Y_{cell}$, threshold)

```
1: $\mathsf{c}_{\boldsymbol{x}} \leftarrow 0$;
2: while $Y_{cell} + \mathsf{c}_{\boldsymbol{x}}/m <$ threshold do
3:     Uniformly sample $j$ from $\{1, 2, .., \mathsf{m}\}$;
4:     $\mathsf{c}_{\boldsymbol{x}} \leftarrow \mathsf{c}_{\boldsymbol{x}} + 1$;
5:     if $\boldsymbol{x} \models F^j$ then
6:         return $\mathsf{c}_{\boldsymbol{x}}/\mathsf{m}$;
7: return $\mathsf{c}_{\boldsymbol{x}}/\mathsf{m}$
```

Lemma 1 *The complexity of* BSAT *is* $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot \mathsf{threshold})$.

Proof: Y_{cell} is incremented by $\mathsf{c}_{\boldsymbol{x}}/\mathsf{m}$ in line 5 of BSAT after a call to ComputeIncrement and CheckSAT. Since BSAT returns after Y_{cell} reaches threshold, the sum of $\mathsf{c}_{\boldsymbol{x}}$ over all invocations of CheckSAT is $\mathsf{m} \cdot \mathsf{threshold}$. Every time $\mathsf{c}_{\boldsymbol{x}}$ is incremented, the check in line 5 of CheckSAT is performed which takes $\mathcal{O}(n)$ time. Moreover, EnumerateNextSol also takes $\mathcal{O}(n)$ time as enumREX in [13] takes $\mathcal{O}(n)$ time. As a result, the complexity of BSAT is $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot \mathsf{threshold})$. □

Lemma 2 *The complexity of* $\mathsf{ReverseSearch}$ *is* $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot \mathsf{hiThresh})$.

Proof: In $\mathsf{ReverseSearch}$, BSAT is invoked with different thresholds (say $T_1, T_2, T_3 \ldots$) in each iteration of the for loop in line 9 (Algorithm 6) depending on the value of Y_{total}. As a result of the check in line 13, it follows that $T_1 + T_2 + T_3 + \ldots = \mathsf{hiThresh}$. Therefore the complexity of all invocations of BSAT is $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot (T_1 + T_2 + T_3 + \ldots)) = \mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot \mathsf{hiThresh})$. The complexity of $\mathsf{ExtractSlice}$ in line 12 is $\mathcal{O}(\mathsf{n}(\log \mathsf{m} + \log(1/\varepsilon^2))^2)$ [13], and the loop in line 9 can be executed at most $\mathcal{O}(\log\log \mathsf{m})$ times. Therefore, the complexity of $\mathsf{ReverseSearch}$ is $\mathcal{O}(\log\log \mathsf{m} \cdot (\mathsf{n}(\log \mathsf{m} + \log(1/\varepsilon^2))^2) + \mathsf{m} \cdot \mathsf{n} \cdot \mathsf{hiThresh})$, which is $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot \mathsf{hiThresh})$. □

We are now ready to prove Theorem 1.

Theorem 1 *The complexity of* $\mathsf{SymbolicDNFApproxMC}$*, when invoked with* $\mathsf{ReverseSearch}$ *is* $\mathcal{O}(\mathsf{mn} \log(1/\delta)/\varepsilon^2)$

Proof: In Algorithm 2, $\mathsf{ApproxMCCore}$ is invoked $\mathcal{O}(\log(1/\delta))$ times, which in turn makes a call to $\mathsf{ReverseSearch}$. The complexity of $\mathsf{SampleHashFunction}$ is $\mathcal{O}(\mathsf{n}(\log \mathsf{m} + \log(1/\varepsilon^2)))$ [13]. Since $\mathsf{hiThresh} = \mathcal{O}(1/\varepsilon^2)$, the complexity of Algorithm 2 is $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot (1/\varepsilon^2) \cdot \log(1/\delta) + \mathsf{n}(\log \mathsf{m} + \log(1/\varepsilon^2))$, which is $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot (1/\varepsilon^2) \cdot \log(1/\delta))$. □