Engineering an Efficient Approximate DNF-Counter [*]

Mate Soos[1] , **Divesh Aggarwal**[1,3] , **Sourav Chakraborty** [2] , **Kuldeep S. Meel**[1] , **Maciej Obremski**[3]

[1]National University of Singapore, Singapore
[2]Indian Statistical Institute, Kolkata
[3]The Centre for Quantum Technologies (CQT), Singapore

Abstract

Model counting is a fundamental problem in many practical applications, including query evaluation in probabilistic databases and failure-probability estimation of networks. In this work, we focus on a variant of this problem where the underlying formula is expressed in Disjunctive Normal Form (DNF), also known as #DNF. This problem has been shown to be #P-complete, making it often intractable to solve exactly. Much research has therefore focused on obtaining approximate solutions, particularly in the form of (ε, δ) approximations.

The primary contribution of this paper is a new approach, called pepin, an approximate #DNF counter that significantly outperforms prior state of the art approaches. Our work is based on the recent breakthrough in the context of union of sets in the streaming model. We demonstrate the effectiveness of our approach through extensive experiments and show that it provides an affirmative answer to the challenge of efficiently computing #DNF.

1 Introduction

The problem of model counting is fundamental in computer science, where one seeks to compute the total number of solutions to a given set of constraints. In this work, we focus on a variant of this problem where the underlying formula is expressed in Disjunctive Normal Form (DNF), also known as #DNF. This problem has many practical applications, including query evaluation in probabilistic databases [Dalvi and Suciu, 2007] and failure-probability estimation of networks [Karger, 2001]. The problem of #DNF is known to be #P-complete [Valiant, 1979], where #P is the class of counting problems for decision problems in NP. Due to the intractability of exact #DNF, much of the research has been focused on obtaining approximate solutions, particularly in the form of (ϵ, δ) approximations, where the count returned by the approximation scheme is within $(1 + \epsilon)$ of the exact count with confidence at least $1 - \delta$.

[*]The corresponding open-source tool is available at https://github.com/meelgroup/pepin

There has been a significant amount of research on the problem of approximate #DNF counting. Karp and Luby [Karp and Luby, 1983] proposed the first Fully Polynomial Randomized Approximation Scheme (FPRAS) for #DNF, known as the KL Counter. This was followed by the KLMCounter proposed by Karp, Luby, and Madras [1989] and the Vazirani Counter proposed by Vazirani [Vazirani, 2001]. More recently, Chakraborty, Meel, and Vardi [2016] showed that the hashing-based framework for approximate CNF counting can be applied to #DNF, leading to the DNFApproxMC algorithm. This was subsequently improved upon by Meel, Shrotri, and Vardi [2017; 2019] with the design of SymbolicDNFApproxMC algorithm.

Given the plethora of approaches with similar complexity, it is natural to wonder how they compare to each other. Meel, Shrotri, and Vardi [2017; 2019] conducted an extensive study to answer this question, producing a nuanced picture of the performance of these approaches. They observed that there is no single best algorithm that outperforms all others for all classes of formulas and input parameters. These results demonstrate a gap between runtime performance and theoretical bounds on the time complexity of techniques for approximate #DNF, thereby highlighting the room for improvement in the design of FPRAS for #DNF. In particular, they left open the challenge of designing an FPRAS that outperforms every other FPRAS.

The primary contribution of this paper is an affirmative answer to the above challenge. We present a new efficient approximate #DNF counter, called pepin, with (nearly) optimal time complexity that outperforms all the existing FPRAS algorithms when run on standard benchmark data. Our investigations are motivated by the recent breakthrough by Meel, Vinodchandran, and Chakraborty [2021] on approximating the volume of the union of sets in the streaming model. However, we found their algorithm to be highly impractical due to its reliance on sampling from the Binomial distribution and runtime overhead arising from requirement of a large amount of randomness.

To overcome these barriers, we first demonstrate that sampling from the Poisson distribution suffices to provide theoretical guarantees. We then propose algorithmic engineering innovations, such as a novel sampling scheme and the use of lazy data sampling to improve runtime performance. These innovations allow us to design pepin, a practically efficient approximate #DNF counter that outperforms every other FPRAS. In particular, over a benchmark suite of 900 instances, pepin attains a PAR-2 score[1] of 3.9 seconds while all prior techniques have PAR-2 score of over 150, thereby attaining a $40\times$ speedup.

The rest of the paper is organized as follows: we present notations and preliminaries in Section 2. We then present, in Section 3, a detailed overview of the prior approaches in the context of streaming that serve as the inspiration for our approach. We present the primary technical contribution, pepin, in Section 4 and present detailed empirical analysis in Section 5. Finally, we conclude in Section 6.

2 Notations and Preliminaries

In this paper, we consider Disjunctive Normal Form (DNF) formulas, which are disjunctions over conjunctions of literals. A literal is a variable or the negation of a variable. The disjuncts in a formula are referred to as *cubes*, and we use F^i to denote the i^{th} cube. A formula F with m cubes can be represented as $\mathsf{F} = \mathsf{F}^1 \vee \mathsf{F}^2 \vee ... \vee \mathsf{F}^\mathsf{m}$. We use n to denote the number of variables in the formula. The width of a cube F^i is the number of literals it contains and is denoted by $\mathsf{width}(\mathsf{F}^i)$.

Throughout this paper, $\log$ means logarithms to the base 2, and $\ln$ means logarithms to the base e. We use $\Pr[A]$ to denote the probability of an event A, and $\mu[Y]$ and $\sigma^2[Y]$ to denote the expectation and variance of a random variable Y, respectively. An assignment of truth values to the variables in a formula F is called a satisfying assignment or witness if it makes F evaluate to true. The set of all satisfying assignments of F is denoted by $\mathsf{Sol}(\mathsf{F})$. Computing a satisfying assignment, if one exists, can be done in polynomial time for DNF formulas. The constrained counting problem is to compute $|\mathsf{Sol}(\mathsf{F})|$.

We say that a randomized algorithm $\mathcal{A}$ is a FPRAS for a problem if, given a formula F, a tolerance parameter $\varepsilon \in (0,1)$, and a confidence parameter $\delta \in (0,1)$, $\mathcal{A}$ outputs a random variable Y such that the probability that $\Pr[\frac{1}{1+\varepsilon}|\mathsf{Sol}(\mathsf{F})| \leq Y \leq (1+\varepsilon)|] \geq 1-\delta$, and the running time of the algorithm is polynomial in $|\mathsf{F}|$, $1/\varepsilon$, and $\log(1/\delta)$.

2.1 Related Work

The problem of designing efficient techniques for #DNF has a long history. Starting with the work of Stockmeyer [Stockmeyer, 1983] and Sipser [Sipser, 1983], randomized polynomial time algorithms for approximately counting various #P problems were designed. In a breakthrough work, Karp and Luby [Karp and Luby, 1983] introduced the concept of Monte Carlo algorithms for #DNF. Since then, several FPRAS-s based on similar approaches have been developed [Karp *et al.*, 1989]. All these techniques use a mixture of sampling and carefully updating a counter. In recent years, hashing-based techniques have also been used to design FPRASs for #DNF [Chakraborty *et al.*, 2016; Meel *et al.*, 2017; 2019]. While the theoretical results are one part of the story, the practical usability of these FPRAS algorithms gives a different viewpoint. Various FPRAS algorithms for #DNF have been implemented, and their performances analyzed and compared in [Meel *et al.*, 2017; 2019]. There, the authors observed that no single FPRAS algorithm performed significantly better than the rest on all benchmarks.

Applications of #DNF to probabilistic databases also motivated a number of algorithms designed for approximate #DNF that try to optimize query evaluation [Olteanu *et al.*, 2010; Fink and Olteanu, 2011; Gatterbauer and Suciu, 2014; Tao *et al.*, 2004]. These algorithm are, however, either impractical (in terms of time complexity) or are designed to work on restricted classes of formulas such as read-once, monotone, etc. In addition to the randomized algorithms, a significant amount of effort has gone into designing deterministic approximation algorithms for #DNF [Luby and Velickovic, 1996; Trevisan, 2004; Gopalan *et al.*, 2013]. However, the challenge of developing a fully polynomial time deterministic approximation algorithm for #DNF remains open [Gopalan *et al.*, 2013].

Algorithm 1 MVC

```
1: Thresh ← $\left(\frac{\log(12/\delta)+\log \mathsf{m}}{\varepsilon^2}\right)$
2: $p \leftarrow 1$ ; $\mathcal{X} \leftarrow \emptyset$
3: for $i = 1$ to m do
4:     for all $s \in \mathcal{X}$ do
5:         if $s \models \mathsf{F}^i$ then remove $s$ from $\mathcal{X}$
6:     $N_i \leftarrow \mathsf{Binomial}(2^{n-\mathsf{width}(\mathsf{F}^i)}, p)$
7:     while $N_i + |\mathcal{X}|$ is more than Thresh do
8:         Remove each element of $\mathcal{X}$ with probability $1/2$
9:         $N_i = \mathsf{Binomial}(N_i, 1/2)$ and $p = p/2$
10:    $k = 0$
11:    for $j = 1$ to $N_i \log N_i$ do
12:        $s \leftarrow \mathsf{Sample}(\mathsf{F}^i)$
13:        if $s \notin \mathcal{X}$ then
14:            $\mathcal{X}$.Append($s$).
15:            $k = k + 1$
16:        if $k == N_i$ then break
17: Output $\frac{|\mathcal{X}|}{p}$
```

3 Background

As mentioned in Section 1, our algorithmic contributions are based on the recent advances in the streaming literature due to Meel, Vinodchandran, and Chakraborty [2021]. To put our contributions in context, we review their algorithm, henceforth referred to as MVC after the initials of the authors.

MVC is a sampling-based algorithm that makes a single pass over the given DNF formula. The high-level idea of the algorithm is to maintain a tuple $(\mathcal{X}, p)$ wherein $\mathcal{X}$ is a set of satisfying assignments while p indicates the probability with which every satisfying assignment of F is in $\mathcal{X}$. Since the number of solutions of F is not known a priori, the value of probability p is not a predetermined value but changes as the algorithm proceeds.

We now provide a description of MVC, whose pseudocode

[1]PAR-2 score is a penalized average runtime. It assigns a runtime of two times the time limit for each benchmark the tool timed out on.

is presented in Algorithm 1. MVC processes each cube sequentially and for every cube, it first removes all the solutions of F^i that belong to $\mathcal{X}$ (lines 4 – 5). In Line 6, we determine the number of solutions N_i that would be sampled from F^i if each solution of F^i was sampled (independently) with probability p. The distribution over the number N_i is simulated by the Binomial distribution. Since we do not want to store more than Thresh elements in $\mathcal{X}$, if $|\mathcal{X}| + N_i$ is larger than Thresh we decrease p and appropriately adjust N_i (by sampling from Binomial(N_i, p) and $\mathcal{X}$ (by removing each element of $\mathcal{X}$ with probability $1/2$). This is done in Lines 7 to 9. We now need to sample N_i distinct solutions of F^i uniformly at random: to accomplish this task, in Lines 10 – 16, we simply pick solutions of F^i uniformly at random with replacement until we have either generated N_i distinct solutions or the number of samples (with replacement) exceeds $N_i \log N_i$. Finally, in Line 17, we return our estimate as the ratio of $\frac{|\mathcal{X}|}{p}$. We refer the reader to [Meel *et al.*, 2021] for the theoretical analysis of MVC. It is worth remarking the worst-case time complexity of MVC is $\big(2\mathsf{n} \cdot (\log(12\mathsf{m}/\delta))^2 \log\log(12\mathsf{m}/\delta) \cdot \varepsilon^{-2} \log \varepsilon^{-1}\big)$.

Upon observing the existence of a new algorithm, our first step was to determine whether such an algorithm can translate to practical techniques for DNF counting. However, rather surprisingly, the resulting implementation could only handle a few hundred variables. The primary bottleneck to scalability is the reliance of MVC's algorithm on the subroutine Binomial(k, p) in line 6. State-of-the-art arbitrary precision libraries take prohibitively long time sampling from Binomial when the first argument is of the order 2^{100}, which is unfortunately necessary to handle formulas with more than a hundred variables. To further emphasize the overhead due to sampling from Binomial, a run of the algorithm would invoke Binomial roughly m times, and every such invocation when the first argument is of the order of 2^{100} is prohibitively slow to handle instances in practice. Since m for DNF instances is in the range of few ten to hundred thousand, such a scheme is impractical in contrast with state-of-the-art techniques that could handle such formulas in the order of a few seconds. At this point, it is worth remarking that the crucial underlying idea of the algorithm is to be able to sample every satisfying assignment of F^i with probability p, and the current analysis of MVC crucially relies on the usage of the Binomial distribution. Consequently, this raises the questions: *Is it possible to design an efficient algorithmic scheme based on the underlying ideas that can also lend itself to practical implementation?*

4 Technical Contributions

The primary contribution of our work is to resolve the aforementioned challenge. To this end, we present a new algorithmic scheme, pepin, that achieves significant runtime improvements over state-of-the-art techniques. As a first step, we seek to address the major bottleneck of MVC: avoiding dependence on Binomial by proposing a different sampling routine which no longer ensures that every solution of a given cube F^i is sampled independently with probability p. We present the new sampling scheme in Section 4.1; the resulting scheme allows our algorithm to be competitive with the state-of-the-art techniques. In order to achieve a significant runtime performance improvement, we profiled our implementation and discovered that sampling from every cube was the most expensive operation. As a remedy, we propose, inspired by lazy (vs eager) lemma proof generation in modern SMT solvers, *lazy sampling* to delay sampling as much as possible without losing correctness (Section 4.2). We then discuss several low-level but crucial enhancements in the implementation of pepin. Finally, we close the section with a theoretical analysis of the correctness of pepin.

Algorithm 2 pepin(F, ε, δ)

```
1:  Thresh ← $\max\left(12 \cdot \frac{\ln(24/\delta)}{\varepsilon^2}, 6(\ln\frac{6}{\delta} + \ln m)\right)$
2:  $p \leftarrow 1$ ; $\mathcal{X} \leftarrow \emptyset$
3:  for $i = 1$ to m do
4:      $t \leftarrow 2^{n-\mathsf{width}(\mathsf{F}^i)}$
5:      for $s \in \mathcal{X}$ do
6:          if $s \models \mathsf{F}^i$ then remove $s$ from $\mathcal{X}$
7:      while $p \geq \frac{\mathsf{Thresh}}{t}$ do
8:          Remove every element of $\mathcal{X}$ with prob. $1/2$
9:          $p = p/2$
10:     $N_i \leftarrow$ ComputeNumSamples$(t, p)$
11:     while $N_i + |\mathcal{X}| >$ Thresh do
12:         Remove every element of $\mathcal{X}$ with prob. $1/2$
13:         $N_i =$ Binom$(N_i, 1/2)$ and $p = p/2$
14:     $S \leftarrow$ GenerateSamples$(N_i, \mathsf{F}^i)$
15:     $\mathcal{X}$.Append(S)
16: Output $|\mathcal{X}|/p$
```

Algorithm 3 ComputeNumSamples(t, p)

```
1: Thresh$_1 \leftarrow \frac{12\mathsf{Thresh}^2 m}{\delta}$ ; Thresh$_2 \leftarrow \sqrt{\frac{\delta}{6m}}$; $N_i \leftarrow 0$
2: if $t \cdot p \geq$ Thresh$_2$ then
3:     if $t \leq$ Thresh$_1$ then
4:         $N_i \leftarrow$ Binom$(t, p)$
5:     else
6:         $N_i \leftarrow$ Pois$(t \cdot p)$
7: else
8:     $N_i \leftarrow$ Binom$(1, tp)$
9: return $N_i$
```

4.1 Subroutine ComputeNumSamples

In Algorithm 3, we are interested in sampling the number N_i of samples we get if, from a set of t elements, we sample each element independently with probability p, i.e., from the binomial distribution Binom(t, p). If t is large and p is not much larger than $1/t$, then the expected value of N_i is $t \cdot p$, which is small, and it is inefficient to toss t coins, each with probability p. We argue that if t is large, but $t \cdot p$ is small, then the statistical distance between Binom(t, p) and Pois$(t \cdot p)$ is small, and hence we only introduce a small additional error if we replace Binom(t, p) by the more efficiently samplable Pois(tp). Additionally, if $t \cdot p \lll 1$, then it is even more efficient (and we show that it still introduces only a small additional error) if we replace Binom(t, p) by Binom$(1, t \cdot p)$.

Algorithm 4 GenerateSamples(N_i, F^i)

```
1: $S \leftarrow \emptyset$
2: if $n - \mathsf{width}(\mathsf{F}^i) - 2 \cdot \log(1 + \mathsf{Thresh}) \leq \log \frac{6m}{\delta}$ then
3:     $k = 0$;
4:     for $j = 1$ to $N_i(\ln N_i + \ln \frac{6}{\delta} + \ln m)$ do
5:         $s \leftarrow \mathsf{Sample}(\mathsf{F}^i)$
6:         if $s \notin S$ then
7:             $S.\mathsf{Append}(s)$; $k \leftarrow k + 1$
8:         if $k == N_i$ then break;
9: else
10:     for $j = 1$ to $N_i$ do
11:         $s \leftarrow \mathsf{ConstructLazySample}(\mathsf{F}^i)$
12:         $S.\mathsf{Append}(s)$
13: return $S$
```

4.2 Subroutine GenerateSamples

As mentioned earlier, the above-proposed sampling scheme allows our algorithm to be on par with the prior state-of-the-art techniques. To achieve further speedup, we observed that the subroutine Sample(F^i) often takes over 99% of the runtime. Therefore, one wonders whether it is possible to *not sample*? At the outset, such a proposal seems counterintuitive as after all, pepin is a sampling-based technique. Upon further investigation, two observations stand out: (1) almost all samples generated by the Sample routine are removed in line 6 at some point in the future, and (2) to determine whether to remove s from $\mathcal{X}$, one needs to only determine whether $s \models \mathsf{F}^i$, which does not require one to know the assignment to all variables in s. Consequently, it is only required to generate assignment to variables in order to check whether $s \models \mathsf{F}^i$. We achieve such a design in the subroutine GenerateSamples, which we describe next.

The subroutine GenerateSamples is presented in Algorithm 4. The primary challenge in GenerateSamples is to handle the generation of N_i distinct solutions randomly from F^i as if we delay the generation of assignments to unassigned variables in F^i, then we would not know whether we have generated N_i unique solutions. To this end, we observe that when the number of unassigned variables (i.e. $n - \mathsf{width}(\mathsf{F}^i)$) is small, then the chances of repetitions among independent samples would be high and the cost of sample generation is low. Therefore, it is wise to generate the samples than to delay the sample generation (line 2– 8).

Now, we move to the case when the number of unassigned variables is large. In such a case, we seek to defer sampling and therefore, ConstructLazySample sets the value to only the variables that appear in F^i and for rest of the variables, it sets them to a special symbol MARK (i.e., s is a mapping from the set of variables to {TRUE, FALSE, and MARK}. Therefore, in contrast to relying on the expensive operation of pseudorandom generation, we can compute and store s at extremely high speed. Overall, we have deferred assignment to variables in s (except the ones corresponding to literals in F^i) at the time when we are required to check whether $s \models \mathsf{F}^j$ when a new cube F^j arrives. At such a time, for all the variables that are set to MARK in s but whose values are fixed in F^j, we use the pseudorandom generator to generate a random value for the corresponding variables. Note that once we have assigned all the variables corresponding to literals in F^j, we can perform the check whether $s \models \mathsf{F}^j$ by only checking whether s and F^j agree on assignment to variables corresponding to literals in F^j. If s and F^j do agree on all such variables, we can remove s, which showcases the strength of our approach as we could avoid all the work required to assign the variables in s that are still set to MARK.

4.3 Engineering Enhancements

Dense Matrix-based Sample Storage Sample storage for all samples is stored in a single contiguous pre-allocated memory array, similarly to a dense matrix representation. This helps with cache locality and ensures that when we check the samples to be emptied, we go forward, and only forward, in memory, with fixed jump sizes. This allows the memory subsystem to prefetch values the algorithm will read from memory, thereby masking memory latency, where memory latency can often be over 100x slower than instruction throughput in modern CPUs.

The current maximum number of samples always stay allocated, and we keep a stack where we have the next empty slot. When a sample is removed, we simply put their number on this stack. The size of the stack tells us the number of empty slots (i.e. unfilled sample slots).

Sample storage is further bit-packed. Each variable's value in the sample is represented as 2 bits, as we need to be able to represent not only TRUE and FALSE, but also MARK. The bit representation used is 00 = FALSE, 01 = TRUE, 11 = MARK, which allows us to quickly set all-MARK by filling the vector with 1's using highly optimized, SSE memset operations.

Sparse Matrix-based Sample Storage Since a large portion of the samples contain MARK values, one may ask whether it would be faster to use a sparse matrix representation where only 1's and 0's are stored, along with the number of consecutive MARKs following the 1 or 0. To check whether such a system would be faster, we have also developed an implementation that uses such a sparse matrix representation. Unfortunately, this implementation is very slow for anything but extremely sparse DNFs. We compare its performance to the dense matrix representation in Section 5.

Handling Arbitrary Precision We made extensive use of the GNU Bignum library [Granlund and the GMP development team, 2020] for all values that need high precision. We use MPQ for fractions such as sampling probabilities, and MPZ for large numbers such as the precision product. All bignum variables are pre-allocated and pre-initialized and, when appropriate, re-used to reduce dynamic memory allocation. Furthermore, observe that the sampling probability is always of the form 2^{-k} for integer k. Therefore, we only keep the exponent bits k and regenerate sample probability when needed. Since the GNU Bignum library has a special function to quickly generate values of the form 2^{-k}, such regeneration is fast.

4.4 Theoretical Analysis of the Algorithm

We will need the following bounds.

Lemma 1 (Chernoff Bound). *For any $n \in \mathbb{N}, p \in (0,1)$, $\varepsilon, \alpha > 0$,*

$$\Pr[\mathsf{Binom}(n,p) \geq np + \alpha] \leq e^{-\alpha^2/(2np+\alpha)} ,$$

and

$$\Pr[|\mathsf{Binom}(n,p) - np| \geq \varepsilon np] \leq 2e^{-\varepsilon^2 np/3} ,$$

The *statistical distance* between two random variables A, B is defined by

$$\begin{aligned}\Delta &= \frac{1}{2}\sum_v |\Pr[A=v] - \Pr[B=v]| \\ &= \sum_{v:\Pr[A=v]\geq\Pr[B=v]} (\Pr[A=v] - \Pr[B=v]) .\end{aligned}$$

We use $A \approx_\varepsilon B$ as shorthand for $\Delta(A,B) \leq \varepsilon$.

Lemma 2. *For any possibly randomized function α, if $\Delta(A\ ;\ B) \leq \varepsilon$, then $\Delta(\alpha(A)\ ;\ \alpha(B)) \leq \varepsilon$.*

We now bound the statistical distance between $\mathsf{Binom}(n,p)$ and $\mathsf{Binom}(1,np)$ and show that this is small when $np \ll 1$.

Lemma 3. *For any $n \in \mathbb{N}, p \in (0,1)$ such that $np < 1$, the statistical distance between $\mathsf{Binom}(n,p)$ and $\mathsf{Binom}(1,np)$ is at most*

$$\Delta(\mathsf{Binom}(n,p)\ ;\ \mathsf{Binom}(1,np)) < n^2p^2 .$$

Using Le Cam's theorem [Le Cam, 1960], we can bound the statistical distance between the Poisson's distribution and the Binomial distribution as follows.

Lemma 4. *For any $n \in \mathbb{N}, p \in (0,1)$, the statistical distance between $\mathsf{Pois}(np)$ and $\mathsf{Binom}(n,p)$ is at most*

$$\Delta(\mathsf{Binom}(n,p)\ ;\ \mathsf{Pois}(np)) < 2np^2 .$$

The following is a well known bound on the coupon collector problem that we will need.

Lemma 5. *Let there be a set $S = \{a_1, \ldots, a_n\}$. Let T be the number of times we need to draw independently and uniformly from the set S until all elements are drawn at least once. Then, we have that*

$$\Pr[T > n\ln n + cn] \leq e^{-c} .$$

The following is the well known birthday bound.

Lemma 6. *Let there be a set $S = \{a_1, \ldots, a_n\}$. Let T be the number of times we draw a uniformly random element from the set S. Then, the probability that the T elements are distinct is at least $1 - \frac{T^2}{n}$.*

For the purpose of the analysis, we will first analyze Algorithm 2 with one of the subroutines simplified. We will replace Algorithm 3 by Algorithm 5.

Algorithm 5 $\mathsf{ComputeNumSamplesBinom}(t,p)$

```
1: $N_i \leftarrow \mathsf{Binom}(t,p)$
2: **return** $N_i$
```

Lemma 7. *Consider the Algorithm 2 with the subroutine Algorithm 3 replaced by the subroutine Algorithm 5. Further, assume that in Step 14 of the algorithm, we obtain N_i uniformly distributed distinct samples from the solution set of F^i. Then, the algorithm outputs a number in $((1-\varepsilon)|\mathsf{Sol}(\mathsf{F})|, (1+\varepsilon)|\mathsf{Sol}(\mathsf{F})|)$ with probability at least $1 - \frac{\delta}{3}$.*

Now, we complete the proof of our main result.

Theorem 1. *Algorithm 2 outputs a number in $((1-\varepsilon)|\mathsf{Sol}(\mathsf{F})|, (1+\varepsilon)|\mathsf{Sol}(\mathsf{F})|)$ with probability at least $1-\delta$ and runs in expected time*

$$O\left(\frac{1}{\varepsilon^2} mn \log\frac{1}{\delta}\left(\log\frac{1}{\varepsilon} + \log\frac{1}{\delta} + \log m\right)\right) .$$

Proof. The algorithm from Lemma 7 is obtained from Algorithm 2 by:

1. Replacing the subroutine Algorithm 3 by the subroutine Algorithm 5.
2. Replacing the subroutine Algorithm 4 by an ideal procedure that obtains N_i uniformly distributed distinct samples from the solution set of F^i.

For item (1), we argue that the statistical distance between the distributions obtained by the two subroutines is at most $\delta/3$. Then by Lemma 2, this small distance is added to the probability that the algorithm does not output an estimate within the desired range. As for item (2), we argue that the probability that the subroutine Algorithm 4 does not output N_i uniformly distributed distinct samples from the solution set of F^i with probability at most $\delta/3$.

Combining the above steps, we get our desired probability of error at most δ.

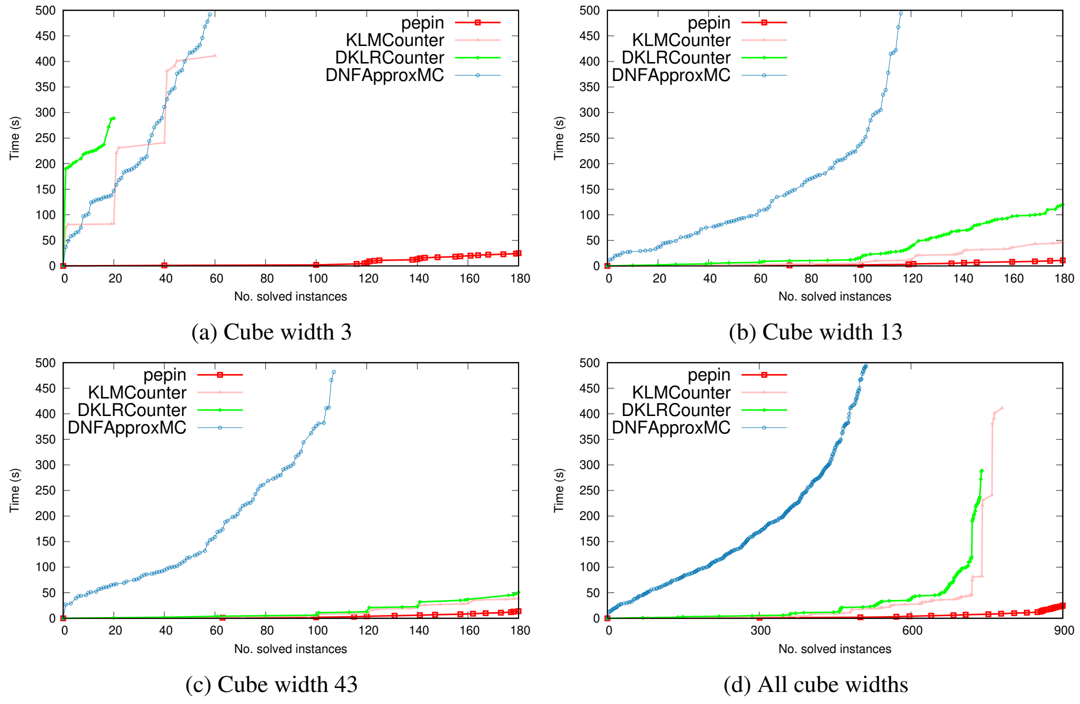
To argue for item (1), note the following. By Lemma 3, we increase statistical distance by at most t^2p^2 every time we sample from $\mathsf{Binom}(1,tp)$ instead of $\mathsf{Binom}(t,p)$. This happens only if $tp < \mathsf{Thresh}_2$ (See line 2 and 7 of the Algorithm 3). Moreover, Algorithm 3 is invoked at most m times, and so the statistical distance is increased by at most

$$m(tp)^2 \leq m\mathsf{Thresh}_2^2 \leq \frac{\delta}{6} .$$

Similarly, by Lemma 4, we increase statistical distance by at most $2tp^2$ every time we sample from $\mathsf{Pois}(tp)$ instead of $\mathsf{Binom}(t,p)$. This happens only if $tp < \mathsf{Thresh}$ (see lines 7-9 of Algorithm 2) and $t \geq \mathsf{Thresh}_1$ (see line 3 and 5 of Algorithm 3). Again, Algorithm 3 is invoked at most m times, and so the statistical distance is increased by at most

$$2mtp^2 = 2m\frac{(tp)^2}{t} \leq 2m\frac{\mathsf{Thresh}^2}{\mathsf{Thresh}_1} \leq \frac{\delta}{6} .$$

To argue for item (2), the subroutine Algorithm 4 might fail to output N_i uniformly distributed distinct samples from the solution set of F^i for one of two reasons: (i) $N_i(\ln N_i + \ln\frac{6m}{\delta} + \ln m)$ uniformly and independently generated samples from F^i do not contain N_i distinct samples (see lines 4-8 of Algorithm 4). (ii) If $\frac{(\mathsf{Thresh}+1)^2}{2^{n-\mathsf{width}(\mathsf{F}^i)}} < \frac{\delta}{6m}$ (See line 2, 9 of



(a) Cube width 3

(b) Cube width 13

(c) Cube width 43

(d) All cube widths

Figure 1: Performance comparison of pepin against the other counters, with different cube widths. As can be seen on the included plots, the cube width matters greatly for most counters other than pepin. This is due to the sparse sampling strategy employed by pepin.

Algorithm 4) and N_i uniformly random samples from F^i are not distinct.

By Lemma 5, the probability of each failure due to (i) is at most

$$e^{-\ln\frac{6m}{\delta}} \leq \frac{\delta}{6m}\,,$$

By Lemma 6 and using that $N_i \leq \mathsf{Thresh} < \mathsf{Thresh}+1$, the probability of failure due to (ii) is at most

$$\frac{N_i^2}{2^{n-\mathsf{width}(\mathsf{F}^i)}} \leq \frac{(\mathsf{Thresh}+1)^2}{2^{n-\mathsf{width}(\mathsf{F}^i)}} < \frac{\delta}{6m}\,.$$

Since the Algorithm 4 is invoked at most m times, the probability that there exists an i such that the corresponding invocation does not output N_i uniformly distributed distinct samples from the solution set of F^i is at most

$$m\cdot\frac{\delta}{6m} + m\cdot\frac{\delta}{6m} = \frac{\delta}{3}\,.$$

The desired bound on the error probability follows.

Now we bound the time complexity. The steps 5, 6 are executed at most $m\cdot|\mathcal{X}| < m\cdot\mathsf{Thresh}$ times. Since, the probability that p cannot drop below $\frac{1}{2^n}$ in steps 7-9, the steps 7-9 are executed at most $n+m$ times. Since, after step 10, $\mathbb{E}[N_i] < \mathsf{Thresh}$, and $|\mathcal{X}| \leq \mathsf{Thresh}$, steps 11-13 are executed $O(1)$ times for every i, and hence $O(m)$ times in total. So, the overall time complexity is dominated by Steps 10 and 14, and hence is bounded by

$$m\cdot\mathsf{Thresh}(\ln\mathsf{Thresh} + \ln\frac{6}{\delta} + \ln m)\cdot n$$
$$= O\left(\frac{1}{\varepsilon^2}mn\log\frac{1}{\delta}\left(\log\frac{1}{\varepsilon}+\log\frac{1}{\delta}+\log m\right)\right)\,.$$

□

5 Empirical Evaluation

We followed the methodology outlined in [Meel *et al.*, 2019]: we use the same benchmark generation tool with the parameters specified by the authors. In particular, the value of n varied from 100 to 100’000 while the value of m varied from 300 to 8×10^5 and the width of cubes varied from 3 to 43. Furthermore, in line with prior work, we set ε to 0.8 and δ to 0.36. We compare the runtime performance of pepin[2] with the prior state of the art techniques, KLMCounter [Karp *et al.*, 1989], DKLRCounter [Dagum *et al.*, 2000], and DNFApproxMC [Meel *et al.*, 2017]. These techniques were observed to be incomparable to each other while outperforming the rest of the alternatives in Meel et al.’s work. All our experiments were conducted on a high-performance computer cluster, each node consisting of 2xE5-2690v3 CPUs with 2x12 real cores and 96GB of RAM, i.e., 4GB limit per run. The timeout was set to be 500 seconds for all runs.

The primary objective of our empirical evaluation was to answer the following questions:

RQ 1 How does the runtime performance of pepin compare to the state of the art tools KLMCounter, DKLRCounter, and DNFApproxMC?

RQ 2 How accurate are the estimates computed by pepin?

[2]The pepin is available open-source at https://github.com/meelgroup/pepin

Counter	finished	PAR-2 score
pepin	900	3.9
KLMCounter	780	158
DKLRCounter	740	198
DNFApproxMC	511	527

Table 1: Comparing the PAR2 scores of the different counters over the performance problems.

In summary, we observe that pepin achieves significant runtime performance improvements over prior state of the art. In particular, pepin achieves a PAR-2 score of 3.9 seconds while the prior state of the art technique could only achieve a PAR-2 score of 158 seconds, thereby achieving a nearly 40× speedup. Furthermore, we observe that the observed ε is only 0.10 – significantly lower than $\varepsilon = 0.8$.

5.1 Performance Experiments

We follow the methodology of Meel et al. in the presentation and analysis of the results. Accordingly, we first present the cactus plots of performance comparisons in Fig. 1. The first three subfigures show the performance of the counters on DNFs with different cube widths, since the cube width has a significant effect on the performance of all solvers. The final subfigure shows the performance over all the cube widths.

The graph on Fig. 1a shows that at small cube widths the previous set of counters all exhibit comparable, and relatively poor, performance, with DKLRCounter performing the worst. In fact, even the best of the previous set of counters, KLMCounter, only managed to count 60 instances (for cube width 3) within the 500s timeout. In contrast, pepin shows remarkable performance here: it finished for all 180 instances for cube width 3, all under 25 seconds. This is primarily due to its lazy sample generation technique, which skips generating random values for variables not in the DNF clause, resulting in many saved computations for cube width 3.

As the cube width increases in Figs. 1b and 1c, the performance of pepin stays similarly good, while the other counters start exhibiting better performance, but never catching up to pepin. For larger cube widths, the lazy sampling starts to be less relevant while the careful design and implementation of the counter plays a more significant role.

Finally, Fig. 1d shows the performance of all the counter over all cube widths. Here, it is clear that pepin outperforms all the counters by a large margin. In fact, pepin is faster than any other counter on all files, except for 27 files, 24 of which are under 1s slower to count by pepin, and the remaining 3 are only under 3s slower to count.

We also present the PAR-2[3] scores for all the counters in Table 1. As shown in Table 1, pepin outperforms all other solvers by a wide margin, thereby, presenting an affirmative answer to the challenge posed by Meel et al. [2019].

Having established significant performance over prior state of the art, we now focus on analyzing pepin further. In particular, we evaluate the performance of three variants of pepin:

[3]PAR-2 score is a penalized average runtime. It assigns a runtime of two times the time limit for each benchmark the tool timed out on.

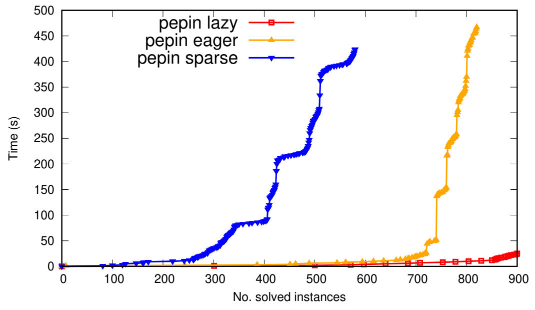


Figure 2: Comparing the performance of different variants of pepin

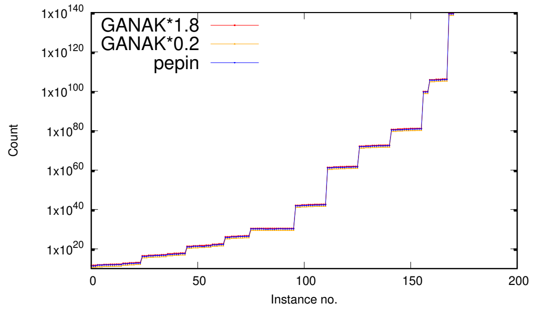


Figure 3: The count returned by pepin compared to the exact counter GANAK. All counts of pepin were well within the 80% permissible error rate as dictated by $\epsilon = 0.8$

default, eager, and sparse, where eager refers to the variant of pepin without the *lazy sampling* strategy while sparse refers to the variant of pepin with sparse matrix representation. The results are depicted in in Fig. 2. The results clearly demonstrate the significant performance improvements due to *lazy sampling* while also demonstrating that the usage of *sparse* matrix representation leads to an overall degradation of runtime performance.

5.2 Accuracy Experiments

To measure the accuracy of pepin, we compared the counts returned by pepin with that of the exact counter, GANAK, for all the instances for which GANAK could terminate successfully. Figure 3 shows the counts computed by pepin, and the bounds obtained by scaling the exact counts with the tolerance factor ($\varepsilon = 0.8$). The y-axis represents the counts on log-scale while the x-axis represents instances ordered in the increasing order of counts. We observed that for all the instances, pepin computed counts within the tolerance. Furthermore, the average mean of observed error for all benchmarks is 0.102– significantly better than the theoretical guarantee of $\varepsilon = 0.8$.

6 Conclusion

In this paper, we successfully tackled the challenge of designing an FPRAS for #DNF that outperforms other FPRAS in practice. An interesting direction of future work would be to further improve the data structures employed in pepin.

Acknowledgements

This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme[NRF-NRFFAI1-2019-0004] , Ministry of Education Singapore Tier 2 grant MOE-T2EP20121-0011, Ministry of Education Singapore Tier 1 Grant [R-252-000-B59-114], and Amazon Research Award. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg DA was supported by the bridging grant at Centre for Quantum Technologies titled "Quantum algorithms, complexity, and communication". MO was supported by the MOE-Tier 1 grant titled "Non-Malleability and Randomness- Foundations of Leakage and Tamper Resilient Cryptography" [A-8000614-00-00].

References

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

[Dagum *et al.*, 2000] Paul Dagum, Richard Karp, Michael Luby, and Sheldon Ross. An optimal algorithm for monte carlo estimation. *SIAM Journal on computing*, 29(5):1484–1496, 2000.

[Dalvi and Suciu, 2007] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.

[Fink and Olteanu, 2011] Robert Fink and Dan Olteanu. On the optimal approximation of queries using tractable propositional languages. In *ICDT*, pages 174–185, 2011.

[Gatterbauer and Suciu, 2014] Wolfgang Gatterbauer and Dan Suciu. Oblivious bounds on the probability of boolean functions. *ACM Trans. Database Syst.*, 39(1):5:1–5:34, 2014.

[Gopalan *et al.*, 2013] Parikshit Gopalan, Raghu Meka, and Omer Reingold. DNF sparsification and a faster deterministic counting algorithm. *Comput. Complex.*, 22(2):275–310, 2013.

[Granlund and the GMP development team, 2020] Torbjörn Granlund and the GMP development team. The GNU multiple precision arithmetic library. Website, Nov 2020. https://gmplib.org/gmp-man-6.2.1.pdf.

[Karger, 2001] David R Karger. A Randomized Fully Polynomial Time Approximation Scheme for the All-Terminal Network Reliability Problem. *SIAM Review*, 43(3):499–522, 2001.

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

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

[Le Cam, 1960] Lucien Le Cam. An approximation theorem for the poisson binomial distribution. *Pacific Journal of Mathematics*, 10(4):1181–1197, 1960.

[Luby and Velickovic, 1996] Michael Luby and Boban Velickovic. On deterministic approximation of DNF. *Algorithmica*, 16(4/5):415–433, 1996.

[Meel *et al.*, 2017] Kuldeep S. Meel, Aditya A. Shrotri, and Moshe Y. Vardi. On hashing-based approaches to approximate DNF-counting. In *Proc. of FSTTCS*, volume 93, pages 41:1–41:14, 2017.

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

[Meel *et al.*, 2021] Kuldeep S. Meel, N. V. Vinodchandran, and Sourav Chakraborty. Estimating the size of union of sets in streaming models. In *PODS*, pages 126–137, 2021.

[Olteanu *et al.*, 2010] Dan Olteanu, Jiewen Huang, and Christoph Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, pages 145–156, 2010.

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

[Stockmeyer, 1983] Larry J. Stockmeyer. The complexity of approximate counting (preliminary version). In *STOC*, pages 118–126, 1983.

[Tao *et al.*, 2004] Qingping Tao, Stephen Donald Scott, N. V. Vinodchandran, and Thomas Takeo Osugi. Svm-based generalized multiple-instance learning via approximate box counting. In *ICML*, volume 69 of *ACM International Conference Proceeding Series*. ACM, 2004.

[Trevisan, 2004] Luca Trevisan. A note on approximate counting for k-DNF. In *APPROX*, volume 3122, pages 417–426, 2004.

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

[Vazirani, 2001] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001.