RICE UNIVERSITY

Domain-Driven Approaches for Constrained Counting and Sampling

By

Aditya A. Shrotri

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

Moshe Vardi
Moshe Vardi (Dec 13, 2021 18:49 CST)

Moshe Vardi

(Chair) University Professor, Karen Ostrum George Distinguished Service Professor in Computational Engineering, Rice University

Supratik Chakraborty
Supratik Chakraborty (Dec 14, 2021 09:36 GMT+5.5)

Supratik Chakraborty

Bajaj Chair Professor in Department of Computer Science and Engineering, IIT Bombay

Devika Subramanian

Devika Subramanian

Professor of Computer Science and Electrical Engineering, Rice University

Illya Hicks (Dec 13, 2021 20:08 CST)

Illya Hicks

Professor of Computational and Applied Mathematics, Rice University

HOUSTON, TEXAS

May 2022

ABSTRACT

Domain-Driven Approaches for Constrained Counting and Sampling

by

Aditya A. Shrotri

Constrained Counting and Sampling are two fundamental problems in Computer Science, where the task is to count the number of solutions or satisfying assignments to a given set of constraints, or to sample a solution uniformly at random. Counting and sampling along with their approximate and weighted variants have been extensively studied in both theory and practice. However, this research effort has been disjointed, resulting in significant gaps in knowledge. On one hand, algorithms with worst-case polynomial running times are considered to be the gold standard by the theory community, but rarely scale well in practice. On the other hand, powerful general-purpose algorithms and tools developed by the AI and Formal Methods communities often fail to scale on 'easy' problems with polynomial upper bounds. The goal of this dissertation is to illuminate and address this disconnect. Specifically, we develop flexible techniques that natively exploit the structure inherent in domain-specific constraints. This often leads to significant performance gains over the popular approach which attempts to shoehorn all constraints to fit a rigid algorithm. Motivated by numerous practical applications and a lack of practically scalable tools with strong theoretical guarantees, we present new solutions for the concrete problems of DNF-Counting, conditional counting, computing the matrix permanent, sampling traces of a transition system and weighted sampling from low-treewidth CNF formulas. Our empirical

analyses reveal a nuanced picture wherein our approaches are seen to be a valuable addition to an algorithmic portfolio.

Acknowledgments

I am deeply grateful to my adviser, Moshe Vardi, for patiently guiding me on 'my terms'. He gave me immense freedom to pursue research directions of my interest, while making sure I didn't veer too much off course. His constant support and availability for his students, despite a huge number of commitments, is extraordinary. From him, I continue to learn how to see the big picture while being rigorous on the details as well as 'thinking fast and slow'. I am excited to continue exploring ideas and directions with him.

I also want to thank my other committee members, – Supratik Chakraborty, Illya Hicks and Devika Subramanian – for insightful feedback during my proposal and defense. Their suggestions shaped a collection of projects into a dissertation.

I want to thank Supratik Chakraborty both as a close collaborator and mentor as well as for hosting me as a visiting student for a semester at IIT Bombay. I am very lucky to have found support from someone with such technical expertise as well as the time (at the oddest hours) and patience to guide me through the trenches of research.

I am also grateful to Kuldeep Meel and Nina Narodytska as collaborators, friends and mentors during different stages of my Ph.D. I could not have asked for a better officemate than Kuldeep, during the first couple of years. I enjoyed and learnt a lot from our discussions, both technical and non-technical, and his work ethic continues to inspire. I am grateful to Nina for taking me on as an intern at VMware Research and for patiently guiding me through the field of explainable AI. Seasoned senior researchers who are willing and able to delve into the nitty-gritties including coding and experiments are rare, and I am truly lucky to have found such a mentor in

Nina. I also want to thank the folks at TRDDC, especially Kumar Madhukar and R. Venkatesh, for a great internship exploring the challenges of industrial research.

I want to thank Dror Fried and Suguman Bansal for all the fun times spent in Duncan Hall and beyond. I would not have been half as productive if not for the casual banter and the serious research discussions, as well as the camaraderie they inspired in the group as a whole. I am grateful to all past, present, extended and honorary LAPIS members, especially Abhinav, Afsaneh, Jeff, Kevin, Lucas, Shufang, Vu, Yong and Zhiwei.

I want to thank all the friends I made in the past six years, especially the 2015 Indian grad-students and the badi subset, LAPIS, the Brompton folks i.e. Ayush, Prathamesh, Vaideesh and Vivek; the poker group i.e. Eslaam, Gaurav, Kedar, Soumya and Zaid; the Houston game group including Rutvi, Swati and Zankhana and the large Counter Strike group. I am especially grateful to my roommate Yash Khemka who has been the common denominator, both in many groups and in many years of fun times. A large part of the work was done at Agora and Empire Cafes and with the support of AWS resources. Last but not least, I am humbled to have the support of old friends both in US and India including '12 che + chya', MAVAP-KJGANIS++ and Excelsior++.

Finally, I am eternally grateful to the extended Joshi and Shrotri families for their continued love and support. I am grateful to Kanchan and Subhash Dandage for being in my corner throughout these six years. I am indebted to Tanvi Modi, Trupti Gosaliya and Smita Modi for going out of their way and for warmly welcoming me in their lives at every possible step. I am deeply grateful to Jojo, Jiji, Mama, Mami, Shreyas, Shashank and families, Mama Mami, Sanket and family, Aaji Ajoba and Kaka, and in particular Atya, Kaka and family for making me feel at home in the US. Ultimately, I would not be at this juncture today without my parents Ulka and Aniruddha, whom I cannot begin to thank in any way.

Contents

Illustrations

Tables

Part I

Prologue

Chapter 1

Introduction

1.1 Background

Constrained Counting and Sampling are two fundamental problems in Computer Science. In Constrained Counting, also known as Discrete Integration, the task is to compute the total number of solutions to a given set of constraints over discrete variables. For Constrained Sampling, the task is to generate a solution uniformly at random from all the solutions to the given constraints. Concrete instantiations of these abstract problems can be defined by specifying a format for constraints.

1.1.1 Constraints

Informally, constraints are compact representations of Boolean-valued functions over a set of discrete variables *. For instance, constraints specified over Boolean variables using operators like AND ($\wedge$), OR ($\vee$), NOT ($\neg$) etc. are called Boolean formulas and the corresponding problems are known as propositional model counting and sampling respectively. Further restrictions on the structure of Boolean formulas give rise to constraint languages like the Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF). Such constraint types have been the subject of intense study both in theory and practice, owing to a trade-off between their expressive power and the hardness of queries like satisfiability, optimization, counting and sampling. For example, CNF formulas are highly expressive, in that all types of constraints over discrete spaces can be succinctly and mechanically encoded into CNF such that the

*See Chapter 2 for definitions and a deeper discussion

count is preserved [1, 2]. Ready availability of efficient off-the-shelf encodings to CNF make it an attractive constraint type. Instead of creating new algorithms and tools for each constraint type, one can simply focus on CNF formulas and reuse CNF counters and samplers to also count and sample from other constraint types. Unsurprisingly, CNF counting and sampling has myriad applications in Computer Science and beyond [3, 4]. However, this expressivity comes at a cost. Counting and sampling from CNF formulas (along with other queries) are complete for their respective complexity classes and are difficult to solve efficiently in practice. On the other end of the spectrum, DNF formulas are much less expressive. For example, only very select applications like querying probabilistic databases [5] and estimating network reliability [6] have native and efficient reductions to DNF-Counting. However, unlike CNF, queries like satisfiability, sampling and approximate counting can be answered very efficiently in worst-case polynomial time for DNF formulas. Thus the trade-off between expressivity of constraint types and hardness of queries is nuanced. The theory and AI communities have taken different approaches to exploring the divide.

1.1.2 Counting and Sampling in Theory

The theory community is primarily interested in establishing complexity lower bounds. Counting problems belong to the complexity class #P. We refer the reader to Arora and Barak [7] for a detailed discussion on complexity classes. The problem of counting the solutions of Boolean formulas is the canonical #P-Complete problem, analogous to Boolean satisfiability (SAT) being the canonical NP-Complete problem. Toda [8] showed that it possible to solve any problem in the polynomial hierarchy within polynomial time with just one call to a #P-oracle. This hints to the hardness of counting. Interestingly, Valiant [9] showed that counting the number of perfect matchings in a bipartite graph is also #P-Complete, despite the fact that the decision version, i.e. finding a perfect matching can be solved in polynomial time. The complexity landscape for counting is thus nuanced, and considerable effort has been devoted

to delineating the boundary between problems with polynomial time algorithms vs. those complete for some hard complexity class (c.f. [10]). Another line of work has explored complexity of approximate counting, wherein the goal is to compute the count up to a given multiplicative or additive factor, either deterministically or with high probability. Some problems, such as approximate DNF-Counting can be solved in time polynomial in both the size of the formula as well as the approximation factors [11], while others are provably hard under mild assumptions. Thus, similar to the exact case, the primary focus of the theory community for approximation algorithms has been to establish polynomial upper bounds for different constraint types, where possible. Uniform sampling is intimately related to counting in many ways [12, 13, 14, 15]. Jerrum, Valiant and Vazirani [14] demonstrated a reduction between uniform sampling and approximate counting, as well as the inter-reducibility between approximate counting and almost-uniform sampling. The latter is at the core of most polynomial time approximate counting algorithms. Bellare, Goldreich and Petrank [15] showed that uniform sampling can be done in probabilistic polynomial time with access to an NP oracle.

It is thus clear that the theory community is concerned with tractability only from the perspective of polynomial running time. Fine-grained analysis and algorithmic development that also takes into account the degree of the polynomial, log-factors and constants, has been largely absent in literature. Not surprisingly, it has been regularly observed that many algorithms that are worst-case polynomial do not yield tools that scale well in practice [16, 17]

1.1.3 Counting and Sampling in Practice

Interest in practical aspects of counting and sampling stems from the myriad applications in diverse domains across computer science such as probabilistic inference and partition function estimation [18, 3], quantifying information flow [4], testing and verification [19, 20], etc. Undeterred by asymptotic worst-case complexity, researchers

attempted to create algorithms and tools that worked well on real-world instances which often have exploitable structure. Early work was geared towards Boolean satisfiability (SAT). Beginning in the 90's, SAT solvers saw tremendous improvements in scalability [21, 22], and can now routinely solve formulas with millions of variables arising from real-world applications (c.f. [23]). Inspired by this success, in the 2000's, researchers began looking at problems believed to be even harder than SAT, including counting and sampling.

Today, the dominant paradigm for exact counting and sampling, is based on extending the classical SAT algorithm called CDCL [22] to exhaustively search the entire solution space with counting-specific enhancements like component caching [18] and special heuristics [24]. This approach requires the input constraints to be in CNF, which naturally allows it to serve as a one-stop-shop solution for all counting and sampling problems. We refer to this paradigm as the 'SAT-based approach' to counting and sampling. Barring a few very recent exceptions, most existing tools for counting and sampling such as `sharpSAT` [24], `Cachet` [25], `C2D` [26], `d4` [27], `Ganak` [28], `SPUR` [29], `KUS` [30], `WAPS` [31] etc are directly or indirectly based on this SAT-based paradigm.

On the approximate side, the state-of-the-art tools that provide strong PAC-style [32] guarantees with no hand-tuned parameters, such as ApproxMC [33], UniGen [16] and their successors [34, 35, 36], all leverage universal hash functions and special SAT solvers that can handle CNF as well as parity (XOR) constraints. Such SAT solvers, such as CryptoMiniSAT [37], extend CDCL with XOR-reasoning to achieve better scalability. Similar to the exact case, the SAT-based approach is thus the workhorse in approximation tools as well.

In summary, the SAT-based approach operating on CNF formulas is by far the most popular today for both counting and sampling. Given any application, the pipeline is to first encode the problem into CNF using standard count-preserving encodings [1] and the resulting formula is given as input to a SAT-based counter or

sampler. The advantages of this general-purpose approach are numerous. It enables modern counters and samplers to leverage years of research in SAT solving which had yielded spectacular improvements in scalability for the problem of satisfiability. A standardized approach across tools encourages modularity, code reuse, fast development and update cycles and community-building as was seen for SAT solvers. Additionally, a wealth of efficient encodings for various types of constraints to CNF are readily available [23]. This is especially beneficial for problems involving heterogeneous constraints such as those arising from planning [38], scheduling [39], model checking [40] etc. Thus the strength of the SAT-based approach is its generality – instead of developing new algorithms and tools from scratch for each new problem and constraint type, one can simply use standard parsimonious (count-preserving) encodings such as the Tsetin transformation [1], to translate the original problem instance into CNF and use an existing SAT-based tool on the encoded formula. Thus, this approach serves as a one-stop-shop solution for a variety of problems.

1.2 A Knowledge-Gap

Conventional wisdom in many disciplines favors general-purpose approaches over the specialized ones, for many of the same reasons for preferring SAT-based approaches just mentioned. For example, generic tools like CPLEX and Gurobi are the preferred solution for optimization problems across many disciplines and industries, general-purpose architectures won over special-purpose in early days of microprocessor development [41], and (CNF) SAT solvers became the go-to solution for the problem of Automatic Test Pattern Generation for hardware-verification over specialized circuit-SAT [42]. This wisdom has been captured in the principle "COTS (Commodity Off-the-Shelf) always wins".

The holy grail for computing has been proposed as "the user states the problem; the computer solves it" [43]. In the context of Constrained Counting and Sampling, it seems at first glance that the SAT-based approach is well-positioned to achieve

this goal. Domain experts can engineer good CNF-encodings, while SAT experts can work on improving heuristics and augmenting the underlying CDCL inference engine with more primitives.

A closer inspection, however, reveals significant obstacles in this simple story in the context of counting and sampling. Consider the problem of computing the permanent of a 0-1 matrix, which is equivalent to the problem of computing the number of perfect matchings in a bipartite graph. The matrix permanent is #P-Complete [9], but in a celebrated result, Jerrum, Sinclair and Vigoda [44] demonstrated a fully polynomial algorithm for multiplicative approximation. However, the degree of the polynomial is n^7, coupled with prohibitively large constants, which makes it all but useless in practice [17]. As an alternative practical approach, we experimented with state-of-the-art exact and approximate SAT-based counting tools such as `d4` and ApproxMC, with a variety of encodings from perfect-matching constraints to CNF. Naturally, given the benefits of generic tools backed by years of engineering effort, we expected SAT-based approaches to do well on this problem, especially since the permanent is in some sense easier than general CNF-Counting[†]. Yet, surprisingly, we consistently observed that SAT-based counters failed to scale beyond 15×15 size matrices, while a brute force approach with $\mathcal{O}(n \cdot 2^n)$ best and worst-case complexity, could compute the permanent of all matrices up to size 27×27. While domain-specific approaches can be expected to have an edge over generic SAT-based approaches, the fact that a naive brute force approach outperformed highly engineered state-of-the-art counters by a wide margin is a striking result.

[†]While both permanent and CNF-Counting are #P-Complete under counting reductions, there can be no parsimonious reduction from the CNF-Counting to the permanent unless P=NP. Further, there is a poly-time approximation algorithm for the permanent [44], while no such algorithm can exist for CNF-Counting unless NP=RP

1.3 Contributions

Unfortunately, the situation with the matrix permanent is not a one-off exception. We observed similar phenomena for other exact and approximate counting and sampling problems such as DNF-Counting, conditional counting, sampling traces of a transition system, and sampling for CNF formulas with low treewidth. Similar to the permanent, these problems are not as expressive or general (and therefore 'easier' in a sense) as compared to counting or sampling arbitrary CNF formulas. Nevertheless, the prevalent SAT-based approach performed significantly poorly on these problems, in our experiments. Further, owing to establishment of complexity bounds, these problems are, in a sense, considered 'solved' by the theory community. Despite many real-world applications, the upshot is that such problems are in academic no-man's-land.

The overarching contribution of this dissertation is to highlight this state-of-affairs and to offer balanced solutions through algorithmic innovation. We observe that in spite of all the advantages of the SAT-based approach, it fails to scale on domain-specific problems because the translation to CNF leads to the loss of structure that is inherent in native constraint-types [45]. Instead, in our works, we directly count and sample from native constraints without necessarily converting them to CNF first[‡]. In this endeavor, we are careful to avoid the other extreme of developing approaches for each problem completely from scratch, which is prone to low-level inefficiencies. Instead, we enhance and extend existing counting and sampling techniques to suit each problem. Leveraging proven techniques in a modular fashion allows us to reuse mature software libraries ensuring scalability. Tailoring techniques to the native constraint types allows us to easily exploit the problem structure. This is in stark contrast to the SAT-based approach which attempts to tailor the constraints (i.e. converts them to CNF) to suit the technique (exhaustive CDCL). In each case our empirical analy-

[‡]We operate on CNF representations only when necessary, for eg. when the underlying application has heterogeneous constraints

ses reveal a nuanced picture, wherein our counting and sampling techniques are seen to be a valuable addition to the algorithmic portfolio, often yielding state-of-the-art performance in a wide variety of scenarios. This belies the conventional belief "one approach (viz. SAT) to rule them all". Our results are analogous to those in the constraint satisfaction setting, where the strengths of CP and SAT solvers are often complementary to each other [46, 47].

Approximate Counting and Sampling In this regime, our techniques straddle the divide between the classical Monte Carlo and hashing-based paradigms, playing on the complementary strengths of each. We demonstrate how to tailor these techniques to the problems of DNF-Counting and certifying Machine Learning explanations.

We first consider the problem of DNF-Counting which has applications in areas like probabilistic databases [48] and network reliability [6]. Important questions regarding the fine-grained complexity of existing approaches as well as their performance in practice, were left open by prior work. Our key contributions here are two fold. First we propose an algorithmic enhancement to the hashing framework, which makes its complexity as good as that of the best Monte Carlo approach. Secondly, we present an extensive empirical study evaluating the performance of different algorithms. A nuanced picture emerges from our study, wherein worst-case complexity is seen not to be the last word for predicting practical scalability.

Next we introduce the problem of conditional counting in the context of explainable AI (XAI) and Machine Learning. Intense interest in making ML fair, safe and understandable has led to the emergence of a plethora of different tools for explaining opaque ML models like Deep Neural Networks. Yet, balancing scalability, generality and theoretical rigor in explanations remains an elusive goal. In our work, we seek to remedy this through a novel explanation framework, called CLIME, which is powered by constraints. We show that the task of certifying the correctness of explanations is actually an instance of the problem of conditional counting. We then present a new

approximate conditional counting algorithm that allows off-the-shelf hashing-based constrained samplers to be used as subroutines within a Monte Carlo framework. This novel combination yields significant improvements in scalability without sacrificing generality or rigor.

Exact Counting and Sampling Here, we present a framework based on representing pseudo-Boolean functions in compact factored forms [49] using datastructures called Algebraic Decision Diagrams (ADDs) [50]. The flexibility offered by factored representations allows this approach to be applied to diverse counting and sampling problems such as computing the matrix permanent, sampling traces of a transition system, and sampling solutions of low-treewidth CNF formulas. This is in stark contrast to the rigid SAT-based framework, which is unable to exploit the hidden problem structure in these domains.

First we consider the problem of computing the permanent of a 0-1 matrix. This problem has applications in a diverse areas like physics [51], organic chemistry [52], constraint programming [53, 54] etc. Despite intense theoretical interest in the problem, there is a prominent lack of practical algorithms that can scale without compromising correctness guarantees. Our key contribution is an algorithm that leverages ADDs to exploit a very general notion of 'structure' in matrices, which allows it to scale on a wide variety of benchmarks.

Next, we consider the problem of sampling traces of a transition system. This problem is of crucial interest to the hardware design community for verifying the behavior of large sequential circuits [55]. We present the first algorithm based on ADDs for sampling traces i.e. runs of a transition system (such as a sequential circuit) with *global* uniformity guarantees, as opposed to existing approaches which only offer local uniformity at each time-step [56, 57, 58]. Critical to the success of our approach is a novel ADD-pruning technique to keep memory requirements small.

Lastly, we consider the problem of sampling solutions of CNF constraints with low-

treewidth. Careful design of algorithms to exploit this property was shown to yield significant performance gains for counting [59], yet the same has not been replicated for sampling. In this work, we show that an existing approach based on ADDs for counting can also be extended to sampling resulting in significant performance gains over existing SAT-based tools like `WAPS` especially in the regime of low-treewidth formulas. A new top-down sampling framework is at the heart of practical gains in performance.

1.4 Benchmarks

A key aspect of evaluating the performance of algorithms in practice is the quality of the set of benchmarks used for the purpose. Ideally, the benchmark set should be representative of the real-world use cases of the algorithm being evaluated. However, there is often a significant gap between the time an approach is proposed and the time it is applied in the real world. Moreover, even after an approach is used in industry, relevant benchmarks and datasets may not be publicly available due to their proprietary nature or simply not having been published. Given that we attempt to tackle problems in 'academic no-man's-land' in this thesis, it is not surprising that we often face many of these issues. In fact, in some cases like DNF-Counting, there is a chicken-and-egg problem[§]. The lack of benchmarks representative of real-world applications hinders the development of practically scalable tools, which in turn makes it difficult to find use-cases of such tools in real applications, from which benchmarks can be derived.

In order to overcome these hurdles, we adopt the strategy of using standard benchmark sets where available, and augmenting them with randomly generated benchmarks as needed. For example, in the case of trace sampling and generating ML explanations, there are comprehensive suites of benchmarks used in prior work that

[§]See Sec. 4.5.

are directly useful for our purposes, which we use with minimal modifications. In the case of CNF sampling, we require weighted CNF formulas as input. However, there are only a limited number of CNF benchmarks in literature that naturally include weights – vast majority of formulas used in literature are unweighted. Therefore, along with one set of naturally weighted CNF formulas, we also use 'pseudo-weighted' formulas where we augment a set of unweighted CNF formulas with randomly generated weights, following [59]. In the case of the matrix permanent, we experiment on both randomly generated as well as real-world instances. Finally, in the case of DNF-Counting, we fully rely on randomly generated benchmarks due to a lack of real-world, publicly available instances.

Randomly generated benchmarks must be used carefully to avoid potential pitfalls. For example, in the case of SAT, it is well known that the state-of-the-art SAT solvers can perform poorly on random instances, and that the structure of solution spaces of real-world instances looks very different from that of random ones [23]. Results on random benchmarks, therefore, may not easily generalize to real-world instances. In our work, we take care to generate benchmarks in a principled way while avoiding over-inference from our results. For example, in case of DNF-Counting, we generate formulas across a broad range of parameters like cube-to-variable ratio, cube-width etc. In other words, we make no presumptions about the type of formulas that may be encountered in the real world, and experiment with a broad class of formulas so as to give a bird's-eye view of the performance landscape. Further, our results illuminate a nuanced picture, where different algorithms are better suited for different formula-types. We discuss these issues in more depth in the respective chapters.

1.5 Tools

The following tools have been developed as part of this dissertation:

1. SymbolicDNFApproxMC and DNFApproxMC: Implementations of hashing-based FPRASs for approximate DNF-Counting

2. CLIME: Constraint-driven Machine Learning explainer (based on a novel conditional-counting algorithm)

3. RysersADD: Counting tool for the 0-1 matrix permanent

4. TraceSampler: Uniform sampler of traces of a transition system

5. `DPSampler`: Weighted CNF-Sampler based on dynamic programming

1.6 Organization

This dissertation is divided into 4 parts: Part I: Prologue, Part II: Approximate Counting and Sampling, Part III: Exact Counting and Sampling, and Part IV: Epilogue.

In Chapter 2 (Part I) we introduce some preliminaries and notation related to counting and sampling. Then in Part II, we first give some background related to Approximate Counting and Sampling as well as the techniques of universal hashing and Monte Carlo sampling (Chapter 3). We present our work on the problem of DNF-counting in Chapter 4. Most of this work appeared in [60]. Chapter 5 presents our work on conditional counting in the context of explaining and certifying Machine Learning models using constraints. This work is to appear in AAAI 2022.

In Part III (Exact Counting and Sampling), we discuss background and techniques, viz. the SAT-based and ADD-based approaches in Chapter 6. We present our work on computing the permanent of a 0-1 matrix in Chapter 7. This chapter is based on [61]. Chapter 8 presents our work on uniform trace sampling which appeared in [62]. In Chapter 9 we discuss our work on the problem of weighted sampling of low treewidth CNF formulas.

We summarize and conclude in Chapter 10 (Part IV). For improving readability and ease of exposition, we defer some proofs, experimental results and extended discussion to the Appendix.

Chapter 2

Preliminaries

We take a general view of constraints in order to unify the different counting and sampling problems we discuss in subsequent chapters. In an abstract sense, a Boolean-valued constraint φ over a set of discrete variables $X = \{x_1, x_2, \ldots, x_n\}$ with domain $\mathcal{D} = d_1 \times d_2 \times \ldots \times d_n$ is a representation of the function $\mathcal{D} \to \{0, 1\}$ mapping each possible variable-assignment $\sigma \in \mathcal{D}$ to 1 or 0 (true or false). If the same function is represented in two different ways, say φ_1 and φ_2, then we write $\varphi_1 \equiv \varphi_2$. When clear from context, we will use φ to represent both the constraint and the underlying function, i.e. $\varphi : \mathcal{D} \to \{0, 1\}$. A satisfying assignment or solution of φ is an assignment $\sigma \in \mathcal{D}$ such that φ evaluates to 1 under σ, which is denoted as $\varphi[\sigma] = 1$. The set of solutions of φ is denoted $R_\varphi = \{\sigma \in \mathcal{D} \mid \varphi[\sigma] = 1\}$.

Definition 2.1. *Given a constraint φ over discrete variables $x_1, x_2, \ldots x_n$ and domain $\mathcal{D} = d_1 \times d_2 \times \ldots \times d_n$, the problem of Constrained Counting* is to compute $|R_\varphi| = \sum_{\sigma \in \mathcal{D}} \varphi[\sigma]$*

Definition 2.2. *Given a constraint φ over discrete variables $x_1, x_2, \ldots x_n$ and domain $\mathcal{D} = d_1 \times d_2 \times \ldots \times d_n$, the problem of Constrained Sampling is to return a random assignment S such that for all $\sigma \in \mathcal{D}$, we have*

$$\Pr[S = \sigma] = \begin{cases} \frac{1}{|R_\varphi|} & if\ \sigma \models \varphi \\ 0 & otherwise \end{cases}$$

*Note that the definition of a constraint and Constrained Counting given here is more general than the ones used in #CSP or Holant problem [10].

It is easy to see that both counting and sampling are at least as hard as satisfiability which simply asks if a solution exists. The problem of deciding satisfiability for arbitrary constraints is NP-Complete while sampling can be done in probabilistic polynomial time with access to an NP oracle [15], and counting is #P-Complete (a class which contains the entire polynomial hierarchy).

Constraint types restrict structure and/or the domain of constraints. For example, if each variable of φ is restricted to the Boolean domain and only Boolean operators such as AND, OR, NOT etc. are allowed, then φ is simply a Boolean formula. If we further restrict the structure of φ to be a conjunction of disjunctions of literals (i.e. variables or their negations), then φ is said to be in Conjunctive Normal Form (CNF). If φ is a disjunction of conjunctions of literals, then it is said to be in Disjunctive Normal Form (DNF). Note that constraints may be implicitly defined, such as those on graphs. For example, the set of all paths of length 4 in a graph can be viewed as constraint that evaluates to true for all ordered subsets of edges that correspond to a path of length 4. The corresponding counting problem is then to compute the number of paths of length 4 in a given graph. Similarly, the set of perfect matchings in a bipartite graph can also be viewed as a constraint from this lens.

The complexity of counting, sampling and satisfaction varies widely for different constraint types. For instance, for CNF formulas, satisfiability is NP-Complete, counting is #P-Complete and sampling can be done in probabilistic polynomial time with access to an NP oracle. On the other hand, for DNF formulas, satisfiability and sampling are both polynomial time, while counting is #P-Complete. For Boolean formulas consisting of conjunctions of XOR constraints, all three problems can be done in polynomial time. Charting out the complexity map for counting and sampling has been the subject of intense research (c.f. [10]).

Apart from constraint types, it is also possible to define tractability in terms of parameters like treewidth, cliquewidth etc. [63]. Additionally, it is also possible to define weighted variants of counting and sampling, wherein a weight function mapping

each variable assignment to positive weight is provided along with the constraints, and the goal is to obtain the cumulative weight of all assignments in R_φ or sample an assignment from R_φ with probability proportionate to its weight. We consider these issues in more detail in Part III.

Researchers have also considered approximate variants of counting and sampling. Different notions of 'good' approximations have been defined in literature; in this dissertation we focus on a strong notion of probabilistic approximation with guarantees similar to those of Probabilistically Approximately Correct (PAC) learning [32].

Definition 2.3. *Given a constraint φ over discrete variables $x_1, x_2, \ldots x_n$ and domain $\mathcal{D} = d_1 \times d_2 \times \ldots \times d_n$, along with two real numbers ε, δ such that $0 \leq \varepsilon, \delta \leq 1$ the problem of approximate counting is to compute an estimate C of $|R_\varphi|$ such that* $\Pr[(1-\varepsilon) \cdot |R_\varphi| \leq C \leq (1+\varepsilon) \cdot |R_\varphi|] \geq 1 - \delta$

Definition 2.4. *Given a constraint φ over discrete variables $x_1, x_2, \ldots x_n$ and domain $\mathcal{D} = d_1 \times d_2 \times \ldots \times d_n$, along with a real numbers ε such that $0 \leq \varepsilon \leq 1$ the problem of almost-uniform sampling is to return a random assignment S such that for all $\sigma \in \mathcal{D}$, we have*

$$\frac{(1-\varepsilon)}{|R_\varphi|} \leq \Pr[S = \sigma] \leq \frac{(1+\varepsilon)}{|R_\varphi|} \quad if\ \sigma \models \varphi$$

$$\Pr[S = \sigma] = 0 \quad otherwise$$

The parameter ε is known as the tolerance and δ is the confidence. Note that in the preceding two definitions, one can instead find an approximation between the multiplicative factors of $1/(1+\varepsilon)$ and $(1+\varepsilon)$ (instead of $(1-\varepsilon)$ and $(1+\varepsilon)$) and allow ε to be greater than 1 as well. The corresponding definitions are equivalent and used interchangeably in literature. Of particular interest, are approximation algorithms that run in polynomial time both in terms of the size of φ as well as $1/\varepsilon$ and $1/\delta$. Such algorithms are called Fully Polynomial Randomized Approximation Schemes (FPRAS) for counting, and Fully Polynomial Approximate Uniform Samplers (FPAUS) for sampling. For instance, the algorithm by Karp and Luby [11] is

an FPRAS for DNF counting, while an FPAUS and an FPRAS for bipartite perfect matchings was given in [44]. In contrast, no FPRAS for CNF counting can exist, unless NP=RP.

Part II

Approximate Counting and Sampling

Chapter 3

Background

3.1 History

Approximate counting and sampling have a rich and varied history since the problems along with the first FPRAS for DNF-Counting were introduced by Karp and Luby [11]. Their algorithm was based on the paradigm of Monte Carlo sampling, which eventually led to the further development of the more advanced Markov Chain Monte Carlo (MCMC) method [64], which has been used to design various FPRASs (such as that for the matrix permanent [44]), as well as for powering many Machine Learning algorithms by allowing sampling from complex distributions (c.f. [65]). Nevertheless, its performance in practice leaves a lot to be desired. For instance, the FPRAS for the matrix permanent has large degree and constants in its asymptotic running time, and has been found to be all but useless in practice [17]. Further, for constraints such as those imposed by Bayesian Networks, it is known that an FPRAS is not possible unless NP=RP [66]. For such problems, MCMC requires exponentially many samples to achieve mixing and yield PAC-style guarantees. Practitioners therefore terminate the sampling before the number of steps mandated by theory have been performed. This leads to loss of guarantees which can be unacceptable for safety-critical applications like those in health-care. Fortunately, this gap between theory and practice was recently addressed by a series of algorithms, viz. ApproxMC [33] for counting and UniGen [16] for sampling, which were based on universal hashing [67] and special SAT solvers. Building on earlier works [12, 13, 14, 15], ApproxMC and UniGen achieved scalability on real-world CNF benchmarks arising from various problem domains without compromising on PAC-style guarantees. This was made possible

by use of low-independence hash functions based on random XOR formulas, as well as a specialized SAT solver called CryptoMiniSAT [37], which allowed fast solving of CNF and XOR constraints. While the worst-case complexity is exponential, it was observed that ApproxMC, UniGen and their successors [68, 34, 35] were successful on real-world problem instances, thanks to the ability of the SAT solver to exploit the structure of such problems. The latest versions of ApproxMC and UniGen are able to scale to formulas with hundreds of thousands of variables.

This progress by ApproxMC and UniGen suggested that the SAT-based paradigm may be powerful enough for many practical purposes. In particular, given its ability to count complex CNF formulas, we expected ApproxMC to also do well on DNF formulas encoded into CNF, given that DNF satisfiability is trivial. Surprisingly, this was not the case. When we tested ApproxMC on (CNF-encodings of) DNF benchmarks obtained from the application of probabilistic databases, we found that it failed to count formulas in a 1000 second timeout. In contrast, a simplistic naive Monte Carlo algorithm could solve many of these benchmarks in under a second, but its performance was not robust. In a similar vein, in our work on certifying ML explanations, we encountered CNF formulas with high solution density, which proved to be very difficult for ApproxMC (and even for exact SAT-based tools). This wide chasm in expected vs. obtained results, suggested that the SAT-based approach may not be the be-all, end-all for approximate counting in practice, as previously thought. It called for techniques that could combine the best of Monte Carlo and hashing-based paradigms, without necessarily relying on SAT solving.

In our works on DNF-Counting and conditional counting, we designed counting and sampling techniques leveraging the complementary strengths of hashing and Monte Carlo paradigms. Hashing offers generality while the simplicity of Monte Carlo can make it extremely fast when applicable. Our key contribution in this regard was to marry these two hitherto unrelated paradigms to obtain algorithms that were robust across different problem regimes. In the rest of this chapter, we give a bird's

eye view of these paradigms. In the following two chapters, we discuss our works on DNF-Counting and conditional counting respectively, that combine the strengths of these paradigms in novel ways to obtain fast and robust performance, both in theory and practice.

3.2 Monte Carlo Framework

Algorithms built on Monte Carlo framework are randomized algorithms whose output can be wrong with a certain (usually small) probability [69]. Typically, these algorithms rely on drawing independent random samples to obtain numerical results. We refer the reader to [70] 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.

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

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. Note that $\mathcal{A}$ is typically represented implicitly using constraints. In the context of counting, we have $\mathcal{A} = \mathcal{R}_\varphi$, given an

input constraint φ.

In general, it is hard to get a good estimate of N without some prior knowledge about the problem instance. An overestimate of N can lead to poor sample complexity, while an underestimate leads to loss of theoretical guarantees. Dagum et al. [71] proposed a Monte Carlo algorithm that could compute a PAC estimate for the cardinality of $\mathcal{A}$ while estimating N on-the-fly. Their approach was general-purpose in that it made no assumptions on the type or size of $\mathcal{A}$. They also proved that their approach has sample complexity that is only a constant factor away from the theoretical optimal, which also makes it very fast in practice.

The simplicity of the Monte Carlo framework coupled with the theoretical and practical benefits of the approach of [71], make it a formidable tool for counting and sampling. Nevertheless, the Achilles heel of this paradigm is the fact that the sample complexity is proportional to the ratio of te size of $\mathcal{A}$ to the size of the universe $\mathcal{U}$. This makes it impractical in cases when this ratio is exponentially small, as is the case in many real-world applications.

3.3 Hashing Framework

The USP of the hashing framework, as developed in ApproxMC and UniGen for CNF formulas, is its ability to exploit the problem structure in a given problem instance using SAT solvers. This ability allows for counting and sampling from exponentially small solution spaces, and sets it apart from the previous naive Monte Carlo paradigm.

The abstract hashing framework is presented in Alg. 2. $\mathsf{hiThresh}$ and t are constants computed based on the input values of ε and δ respectively. p and q represent the number of constraints and variables in the hash function $h \in \mathcal{H}$ respectively. 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 [67]. The crux of the framework is a search for the right number of hash constraints such that the number of solutions in a cell – $Y_{cell} = |\mathcal{R}_\varphi \cap h^{-1}|$ – is

Algorithm 2 ApproxMC$(\varphi, \varepsilon, \delta)$

hiThresh $\leftarrow \mathcal{O}(\frac{1}{\varepsilon^2})$;
$t \leftarrow \mathcal{O}(\log(\frac{1}{\delta}))$;
EstimateList $\leftarrow$ emptyList;
$\mathcal{H} \leftarrow ChooseHashFamily()$;
repeat t times
    $h \leftarrow$ SampleHashFunction$(\mathcal{H}, \mathsf{q})$;
    $Y_{cell}, \mathsf{p} \leftarrow Search(\varphi, h, \mathsf{q}, \text{hiThresh})$;
    AddToList(EstimateList, $Y_{cell} \times 2^{\mathsf{p}}$);
finalEstimate $\leftarrow$ FindMedian(EstimateList);
return finalEstimate

Algorithm 3 LinearSearch$(\varphi, h, \mathsf{q}, \text{hiThresh})$

for $\mathsf{p} \in \{0, \ldots, \mathsf{q}\}$ **do**
    $Y_{cell} \leftarrow$ SaturatingCounter$(\varphi, h, \mathsf{p}, \mathsf{q}, \text{hiThresh})$
    if $Y_{cell} <$ hiThresh **then**
        return Y_{cell}, p

not too large, yet the tolerance and confidence obtained are as required. This search is encapsulated in a call on line 7 of Alg. 2. For ease of understanding, in Alg. 3 we present a simple linear search starting from 0 hash constraints, as was done in [16]. Note that in [68], binary search was used in its place which reduced the number of SAT calls to logarithmic in the number of variables, while in Chapter 4 we present yet another 'reverse' search procedure that improves the complexity of the hashing-based DNF-Counting algorithm.

In Alg. 3, to calculate Y_{cell}, all the solutions in a randomly chosen cell are counted up to the threshold $\mathsf{hiThresh}$, using a procedure called $\mathsf{SaturatingCounter}$ called on line 2. $\mathsf{SaturatingCounter}$ can be implemented through calls to a SAT solver that can handle CNF and XOR constraints comprising of the input formula φ and the hash constraints h respectively. In particular, a SAT solver can be used to enumerate solutions of $\varphi \wedge h$, one by one, using the concept of blocking clauses [16]. This procedure is scalable in practice, as we only need to enumerate $\mathsf{hiThresh}$ many solutions, which is much smaller compared to $|R_\varphi|$. While enumerating solutions in this way, if Y_{cell} exceeds the 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. In Alg. 2, therefore, the estimate $Y_{cell} \times 2^{\mathsf{p}}$ (line 8), where 2^{p} is the total number of cells, is a good approximation of $|\mathcal{R}_\varphi|$. The final answer is the median (line 9) of t independent invocations of the for loop (lines 5-8), which ensures that the confidence is amplified to the required value of $1 - \delta$.

Chapter 4

DNF-Counting

4.1 Introduction

DNF-Counting or #DNF is an important problem in practice, as applications such as query evaluation in probabilistic databases [5] and failure-probability estimation of networks [6] reduce to it. In their seminal paper, Karp and Luby [11] 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 φ. Building on KL Counter, Karp et al. [72] proposed an improved FPRAS, henceforth denoted as KLM Counter, which has time complexity linear in the number of cubes. Vazirani [73] proposed a variant of KL Counter (denoted Vazirani Counter) with same time complexity as KL Counter, but combined with an enhancement proposed in [71], it requires fewer Monte Carlo samples than KL Counter.

Besides the Monte Carlo paradigm, one can also use the hashing paradigm for approximate DNF-Counting. A naive way to do so is to encode DNF formulas into CNF, using Tseitin encoding*. One can directly invoke ApproxMC on the encoded CNF formulas, to get the count of the original DNF formula, because of the fact that Tseitin encoding is parsimonious. In this chapter, we refer to this method simply as ApproxMC. It is natural to expect ApproxMC to do well here, as DNF satisfiability

*We highlight that multiplicative approximation is not closed under complement, so it is not possible avoid using Tseitin encoding by considering the complement CNF formula of a given DNF formula.

is trivial as compared to general CNF satisfiability. In preliminary experiments, however, we observed that ApproxMC severely underperformed on formulas stemming from probabilistic database benchmarks. In particular, ApproxMC timed out on all benchmarks after 1000 seconds, while a naive Monte Carlo algorithm took under a second for each. In fact, for all formulas considered in this chapter, ApproxMC fared very poorly. Clearly, the SAT-based approach is not suited for the problem of DNF-Counting, and we do not discuss it further.

Despite the failure of ApproxMC, it turns out that the hashing paradigm can still be used for DNF-Counting, by operating on DNF formulas directly, instead of going via CNF and SAT. Chakraborty et al. [68] 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. The algorithm relies on Gaussian elimination for solving DNF+XOR formulas directly, instead of converting DNF to CNF. However, the time complexity of this hashing-based scheme of Chakraborty et al., called DNFApproxMC, was cubic in the number of variables which is significantly worse than that of KLM Counter. Building on Chakraborty et al., in our earlier work (not part of this dissertation) [74], we 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. This was made possible through the techniques of Symbolic Hashing, Stochastic Cell-Counting and Row-Echelon hash functions. Despite being tailored for obtaining improvements in time complexity for DNF-Counting, these techniques do not sacrifice generality. In particular, they are enhancements of the underlying hashing framework and are not tied to any particular constraint type. For instance, they can also be used with vanilla ApproxMC for CNF counting. A detailed study of these techniques in alternative contexts is an interesting direction for future work.

Nevertheless, two key questions were left unanswered in previous work: 1) Is it possible to remove the polylog factors in the complexity of SymbolicDNFApproxMC?

2) How do the other 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 [75, 76]. 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 chapter, we first present a new 'reverse 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. Similar to other techniques used in SymbolicDNFApproxMC, reverse search also balances generality and specificity to yield theoretical and practical improvements. In particular, in the context of DNF counting, it leads to removal of poly-log factors in complexity. At the same time, it can also be used with vanilla ApproxMC for CNF counting, where it can result in fewer SAT solvers through solution reuse. 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, 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 chapter is organized as follows: we introduce some notation in Section 4.2 and briefly review the various approaches to approximate DNF-Counting in Section 4.3. We present our new search procedure for hashing algorithms in Section

4.4. We describe experimental methodology in Section 4.5 and report on the results in Section 4.6. We offer our interpretation of these results in Section 4.7, and conclude in Section 4.8.

4.2 Preliminaries

A literal is a variable or the negation of a variable. A formula φ 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 φ^i. Thus $\varphi = \varphi^1 \vee \varphi^2 \vee \cdots \vee \varphi^{\mathsf{m}}$. 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 φ^i refers to the number of literals in cube φ^i and is denoted by $\mathsf{width}(\varphi^{\mathsf{i}})$. We use w to denote the minimum of width over all the cubes of the formula, i.e. $\mathsf{w} = \min_i \mathsf{width}(\varphi^{\mathsf{i}})$.

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 $\boldsymbol{A}$,$\boldsymbol{B}$,... to denote matrices, small boldface letters $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$,... 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 φ is called a satisfying assignment or witness if it makes φ 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 φ by $\mathcal{R}_\varphi$. Given φ, the Constrained Counting problem is to compute $|\mathcal{R}_\varphi|$. A *fully polynomial randomized approximation scheme* (FPRAS) is a randomized algorithm that takes as input a formula φ, a tolerance $\varepsilon \in (0,1)$ and confidence parameter $\delta \in (0,1)$ and outputs a random variable Y such that $\Pr[(1-\varepsilon)|\mathcal{R}_\varphi| \leq Y \leq (1+\varepsilon)|\mathcal{R}_\varphi|] \geq 1-\delta$ and the running time of the algorithm is polynomial in $|\varphi|$, $1/\varepsilon$, $\log(1/\delta)$.

A hash function $h : \{0,1\}^{\mathsf{q}} \rightarrow \{0,1\}^{\mathsf{p}}$ 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 $\mathcal{R}_\varphi \cap h^{-1}(\boldsymbol{y})$ 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 [67]. 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 [74]. The technique for enumerating solutions in a cell also depends on the family of the hash function under consideration.

4.3 Approximation Algorithms for #DNF

Beginning with the seminal work of Karp and Luby [11], three Monte Carlo FPRASs for #DNF have been designed over the years [72, 73]. Two more FPRASs were designed using the new hashing-based approach [68, 74]. Besides developing FPRASs, considerable effort has also gone into developing *deterministic* approximation algorithms for #DNF [77, 78, 79] and the closely related problem of designing pseudorandom generators with short seeds [80, 81, 82]. The development of a fully polynomial time deterministic approximation algorithm for #DNF is still an open problem [79].

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 [83, 84, 85]. Such algorithms, however, either take exponential time in the worst case [83, 85] or are designed to work on restricted classes of formulas such as monotone, read-once etc. [84]. An FPRAS similar to KL Counter was developed in the Multi-Instance Learning community for evaluating SVM kernels [86]. 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.

4.3.1 Monte Carlo Framework

We present the different Monte Carlo-based FPRASs using the framework of Algorithm 1. If φ 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 $|\mathcal{R}_\varphi|$ is $2^{\mathsf{n}-\mathsf{w}}$, where w is the minimum over width of all the cubes of φ. 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 $\mathcal{R}_\varphi$ and $\mathcal{U}$ into $\mathcal{R}'_\varphi$ and $\mathcal{U}'$ such that $\frac{1}{\rho'} = |\mathcal{U}'|/|\mathcal{R}_\varphi|$ is polynomially bounded, and it is also possible to recover $|\mathcal{R}_\varphi|$ from $|\mathcal{R}'_\varphi|$. We now discuss various transformations proposed over the years and the FPRASs these transformations yield.

KL Counter

Karp and Luby [11] developed the first FPRAS for #DNF, which we refer to as KL Counter. They defined a new universe $\mathcal{U}' = \{(\boldsymbol{x}, \varphi^i) \mid \boldsymbol{x} \models \varphi^i\}$, and the corresponding solution space $\mathcal{R}'_\varphi$ as $\mathcal{R}'_\varphi = \{(\boldsymbol{x}, \varphi^i) \mid \boldsymbol{x} \models \varphi^i \text{ and } \forall j < i, \boldsymbol{x} \not\models \varphi^j\}$ for a fixed ordering of the cubes. They showed that $|\mathcal{R}_\varphi| = |\mathcal{R}'_\varphi|$ and that the ratio $|\mathcal{U}'|/|\mathcal{R}'_\varphi| \leq \mathsf{m}$ 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 [71] which ensures optimal estimation of N. The enhancement is applicable, since the estimator used by KL Counter is a 0–1 estimator.

KLM Counter

Karp et al. [72] 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 $cover(\boldsymbol{x}) = \{j | \boldsymbol{x} \models \varphi^j\}$. The first key insight is that $|\mathcal{R}'_\varphi| = \sum_{(\boldsymbol{x},\varphi^i)\in\mathcal{U}'} \frac{1}{|cover(\boldsymbol{x})|}$. 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 [73], 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 [71]. 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 $\boldsymbol{x} \models \varphi^j$ has to be performed for all cubes.

4.3.2 Hashing Framework

We first flesh out the abstract hashing framework presented in Chapter 3, in more detail in Algs. 4,5,6 and 7. The procedure ApproxMCCore (Algorithm 5) is invoked $t \in \mathcal{O}(\log(1/\delta))$ times in Algorithm 4, 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 6. 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 $Y_{cell} <$ hiThresh with FailRecord$[p] = 0$ and those p for which $Y_{cell} \geq$ hiThresh by FailRecord$[p] = 1$. The search returns when p is found such that FailRecord[p] $= 0$ and FailRecord[p $- 1] = 1$. The procedure SaturatingCounter (Algorithm 7) is invoked for calculating 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 φ, 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. [68] 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 [74]), which improves the complexity from cubic to quadratic in n leading to better performance on all benchmarks.

SymbolicDNFApproxMC

The algorithm SymbolicDNFApproxMC proposed in [74] 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 8), 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 4 ApproxMC($\varphi, \varepsilon, \delta$)

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

Algorithm 5 ApproxMCCore(φ, hiThresh)

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

4.4 Reverse Search for Hashing-Based Algorithms

A close inspection of the SymbolicDNFApproxMC algorithm in [74] reveals that the polylog factors in the complexity analysis arise due to redundancy in enumerating solutions in successive calls to SaturatingCounter. In particular, the fact that the set $\{\boldsymbol{x} \,|\, \boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})} = \boldsymbol{y}^{(\mathsf{p})}\}$ is a subset of $\{\boldsymbol{x} \,|\, \boldsymbol{A}^{(\mathsf{p}-1)}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p}-1)} = \boldsymbol{y}^{(\mathsf{p}-1)}\}$ is not exploited. Each call to SaturatingCounter is agnostic of the previous ones, resulting in repeated enumeration of solutions. One work-around could be to buffer solutions from a call to SaturatingCounter 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 $\{\boldsymbol{x} \,|\, \boldsymbol{A}^{(\mathsf{p}-1)}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p}-1)} = \boldsymbol{y}^{(\mathsf{p}-1)}\}$ can be partitioned into $\{\boldsymbol{x} \,|\, \boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})} =$

Algorithm 6 BinarySearch($\varphi, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}, \mathsf{q}, \mathsf{hiThresh}$)

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

Algorithm 7 SaturatingCounter($\varphi, \boldsymbol{A}^{\mathsf{p}}, \boldsymbol{b}^{\mathsf{p}}, \boldsymbol{y}^{\mathsf{p}}, \mathsf{q}, \mathsf{threshold}$)

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

$\boldsymbol{y}^{(\mathsf{p})}\}$ 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 8 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 $\mathsf{p} = hi$ constraints is extracted. We assume access to a procedure ExtractSlice which requires a slight modification of procedure Extract in [74]. 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 $(\mathsf{hiThresh}, \mathsf{p})$. Otherwise, the for-loop in line 9 is executed. In lines 10-11, $Y_{cell} = |\mathcal{R}_\varphi \cap \{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p})}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p})} = \boldsymbol{y}^{(*\mathsf{p})}\}|$ is evaluated by setting the 'flip' argument to true in ExtractSlice. After execution of line 12, we have that $Y_{total} = |\mathcal{R}_\varphi \cap \{\boldsymbol{x} \mid \boldsymbol{A}^{(\mathsf{p}-1)}\boldsymbol{x} \oplus \boldsymbol{b}^{(\mathsf{p}-1)} = \boldsymbol{y}^{(\mathsf{p}-1)}\}|$. Therefore, when Y_{total} exceeds hiThresh, the hash count along with the cell-count of the previous iteration are returned in line 13.

Theorem 4.1. *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 SaturatingCounter. This is done as follows: 1) A solution $\boldsymbol{x}$ of the hash function is enumerated 2) Cubes of the input formula φ are randomly sampled until a cube φ^i is found such that $\boldsymbol{x} \models \varphi^i$ 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 [74] was two-fold. Firstly, SaturatingCounter was invoked $\mathcal{O}(\log\log \mathsf{m})$ times. Secondly, each call to SaturatingCounter possibly required the sampling of $\mathsf{m} \times \mathsf{hiThresh}$ cubes. The use of ReverseSearch, however, ensures

that each call to SaturatingCounter is over a previously unexplored part of the solution space. This in turn ensures that exactly $\mathsf{m} \times \mathsf{hiThresh}$ cubes are sampled in total, instead of $\mathsf{m} \times \mathsf{hiThresh} \times \log\log \mathsf{m}$ as in [74]. Since sample-and-check is $\mathcal{O}(\mathsf{n})$, $\mathsf{hiThresh} \in \mathcal{O}(1/\varepsilon^2)$ and SymbolicDNFApproxMCCore is invoked $\mathcal{O}(\log(1/\delta))$ times, the overall complexity is $\mathcal{O}(\mathsf{mn}\log(1/\delta)/\varepsilon^2)$. □

Algorithm 8 ReverseSearch$(\varphi, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}, \mathsf{q}, \mathsf{hiThresh})$

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

Naturally, one wonders whether employing ReverseSearch leads to gains in performance in practice. We compared the running times of SymbolicDNFApproxMC 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 4.1 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

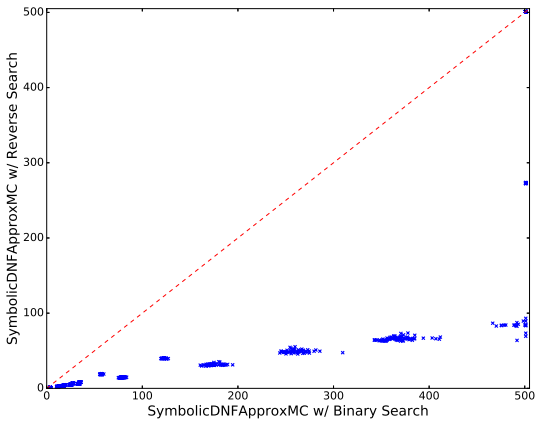


Figure 4.1 : Comparison of Running time of SymbolicDNFApproxMC with BinarySearch and ReverseSearch

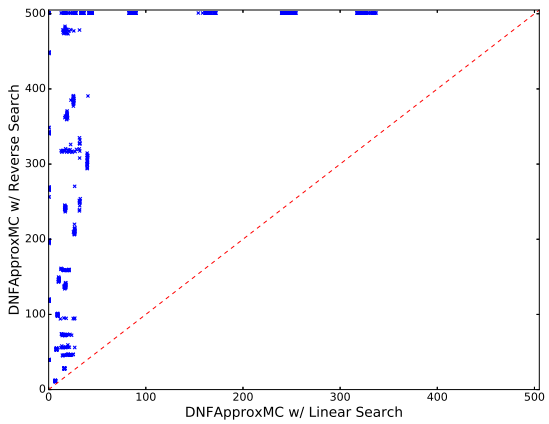


Figure 4.2 : Comparison of Running time of DNFApproxMC with LinearSearch and ReverseSearch

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. 4.2). 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.

4.5 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 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.

4.5.1 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 [87] 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 [88] code-bases, respectively[†]. For a given algorithm and an input formula, we set the timeout to 500 seconds.

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

Table 4.1 : 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	$[0.03, 0.36]$

4.5.2 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 [89] for testing prototype implementations of probabilistic databases [83, 85].

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 avail-

ability. 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 [90]. 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.

4.5.3 Parameters Used

The parameters used for generating random benchmarks for the various experiments is shown in Table 4.1. 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 [24] 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 [33, 68], 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.

4.6 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.

4.6.1 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. 4.3, 4.4, 4.5, 4.6, 4.7 and 4.9 respectively[‡]. 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 $\mathsf{w} = 3$, we see that DNFApproxMC vastly outperforms other algorithms, taking under a second to solve all formulas (see: Fig. 4.3). Rest of the algorithms time out for formulas with number of cubes $\mathsf{m} \geq 100,000$. For $\mathsf{w} = 13$, it can be seen from Fig. 4.4 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 $\mathsf{w} = 23$, 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 $\mathsf{w} = 23$. This trend continues for $\mathsf{w} = 33$ and 43. We see that the behavior of the algorithms does not change above $\mathsf{w} = 23$. For non-uniform

[‡]Figures are best viewed online in color

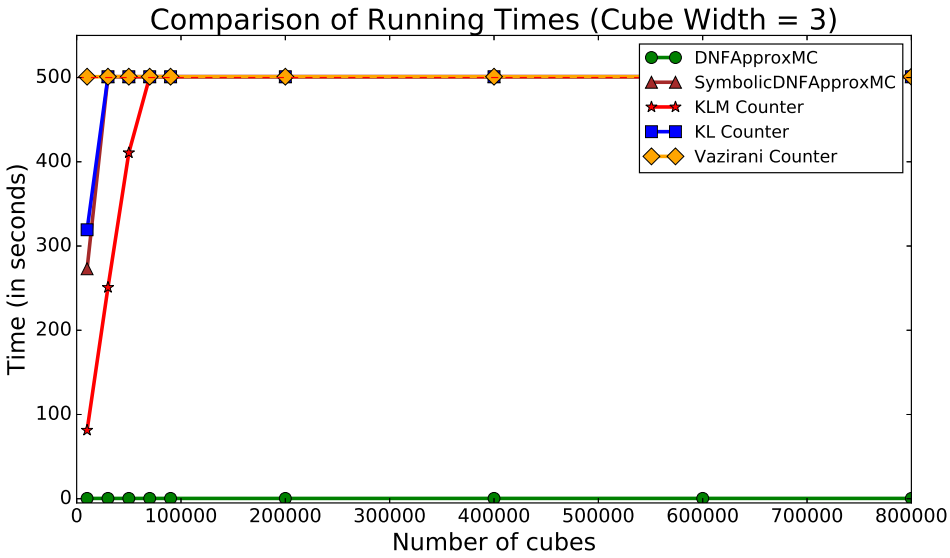


Figure 4.3 : Runtime Variation: DNFApproxMC is the best performer. Rest timeout.

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.

4.6.2 Benchmarks Solved

Fig. 4.8 shows the cactus plot of all the different algorithms. We present the number 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.

4.6.3 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

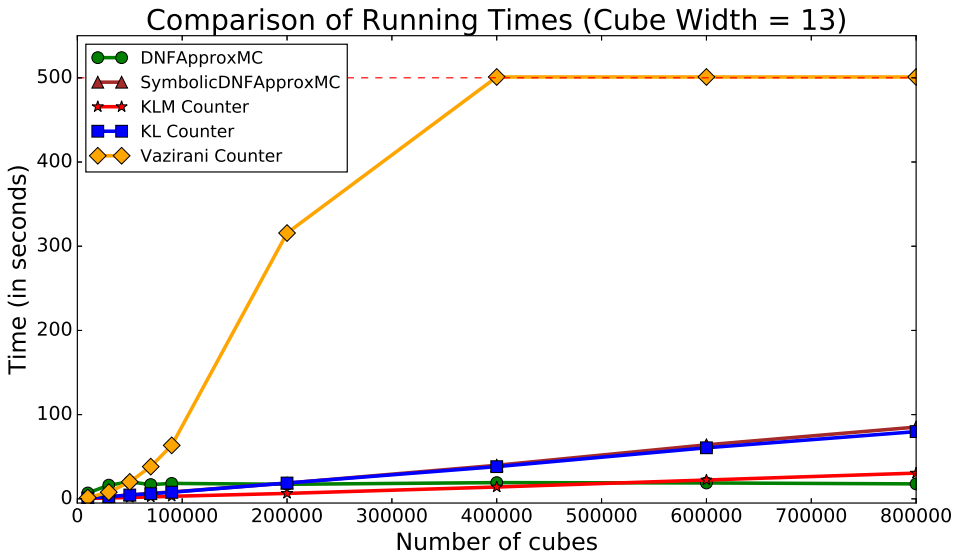


Figure 4.4 : Runtime Variation: DNFApproxMC and KLM Counter are the best performers

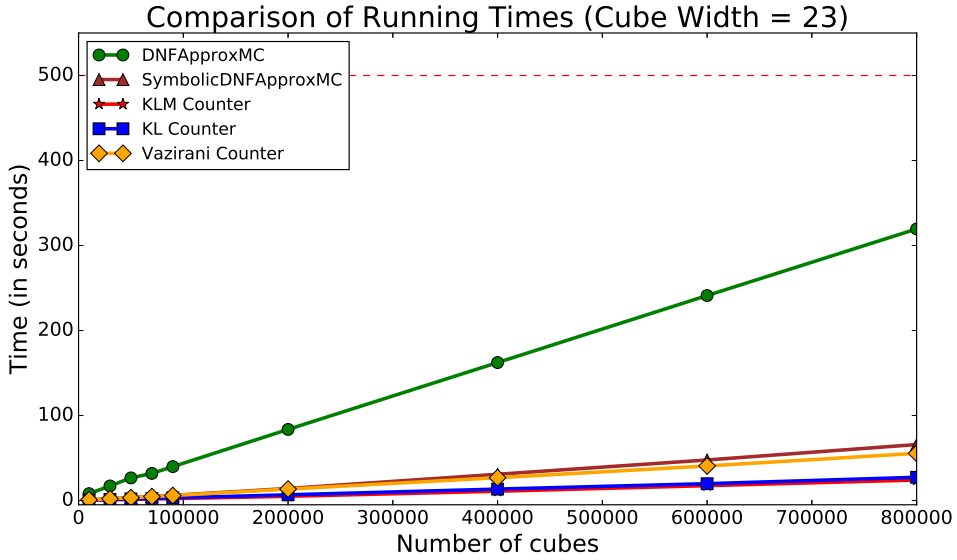


Figure 4.5 : Runtime Variation: KLM Counter and KL Counter are the best performers

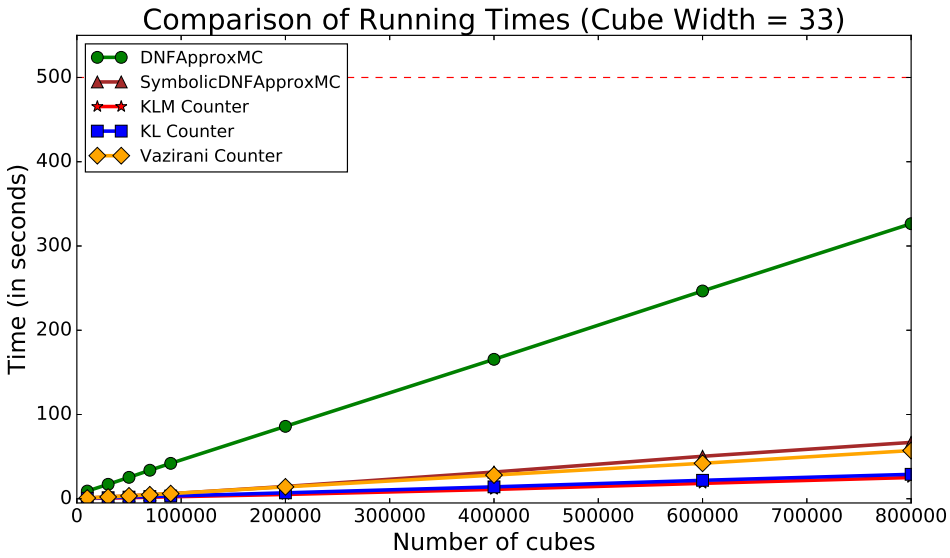


Figure 4.6 : Runtime Variation: KLM Counter and KL Counter are the best performers

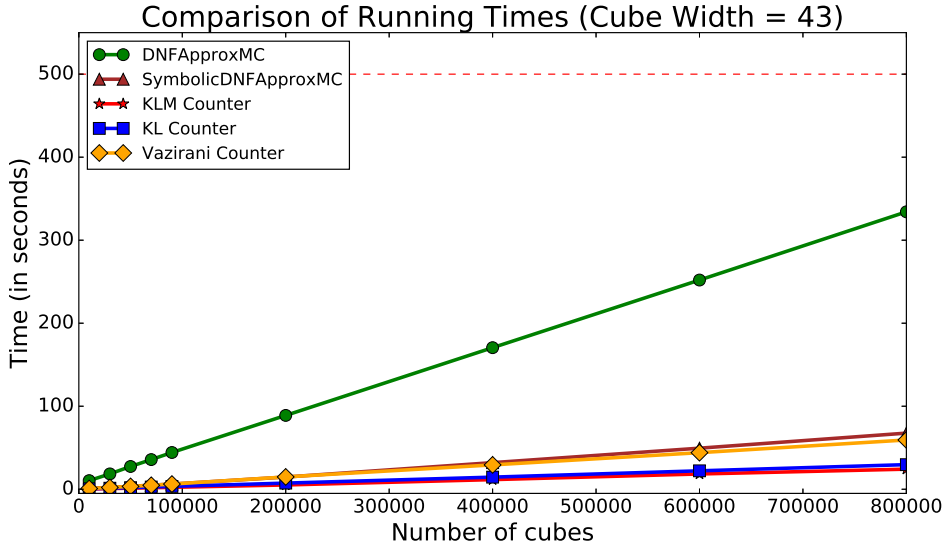


Figure 4.7 : Runtime Variation: KLM Counter and KL Counter are the best performers

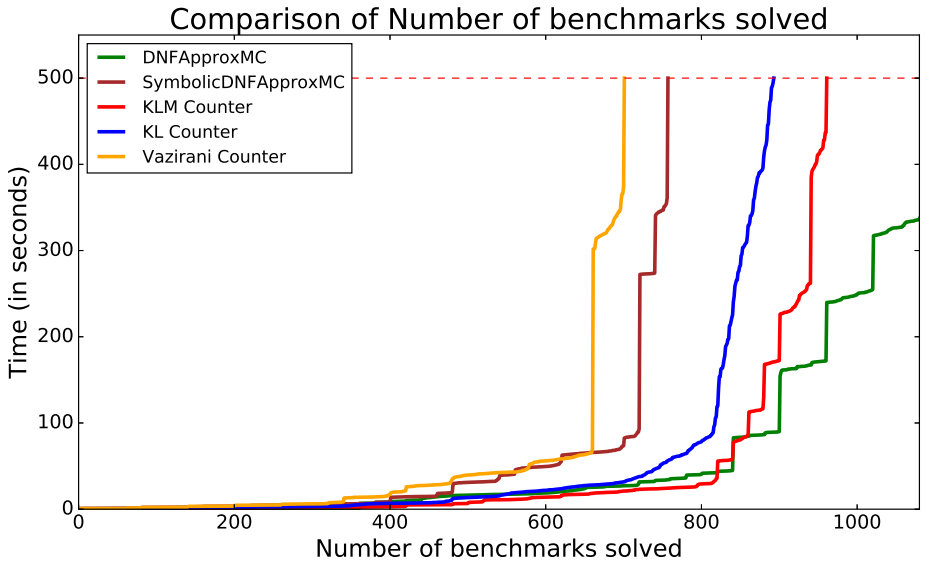


Figure 4.8 : Benchmarks Solved: DNFApproxMC solved all benchmarks

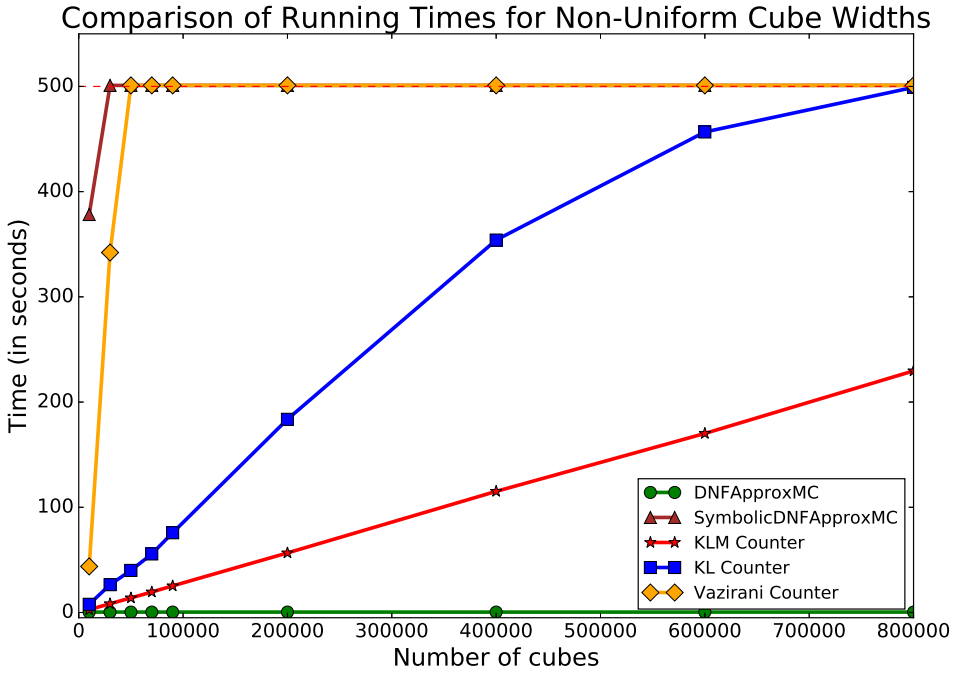


Figure 4.9 : Runtime Variation: DNFApproxMC dominates other algorithms

Table 4.2 : 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

and max errors of the counts returned by the five FPRAS for the 228 formulas, is shown in Table 4.2. 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.

4.6.4 ε - δ Scalability

Fig. 4.10 shows the average time taken by the five algorithms over 20 instances when 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. 4.11 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.

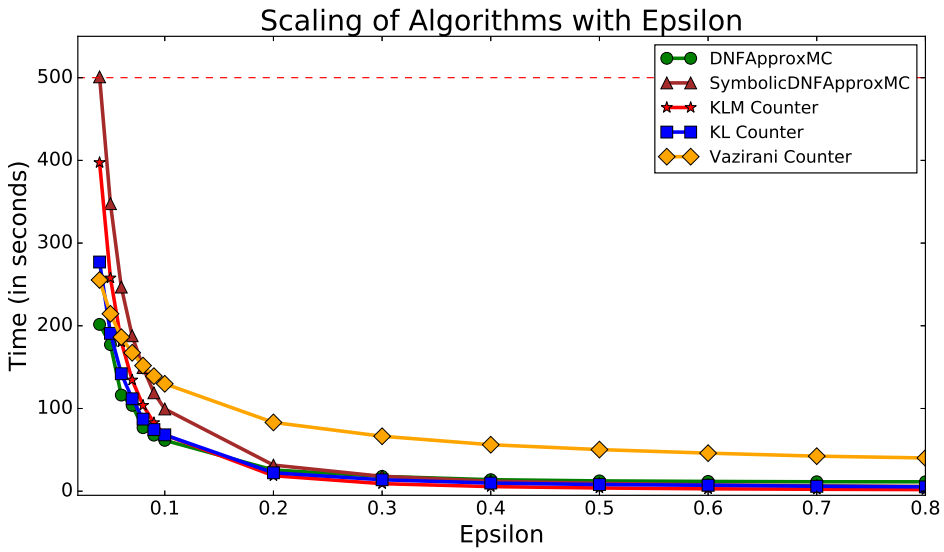


Figure 4.10 : ε Scalability: DNFApproxMC scales better than other algorithms

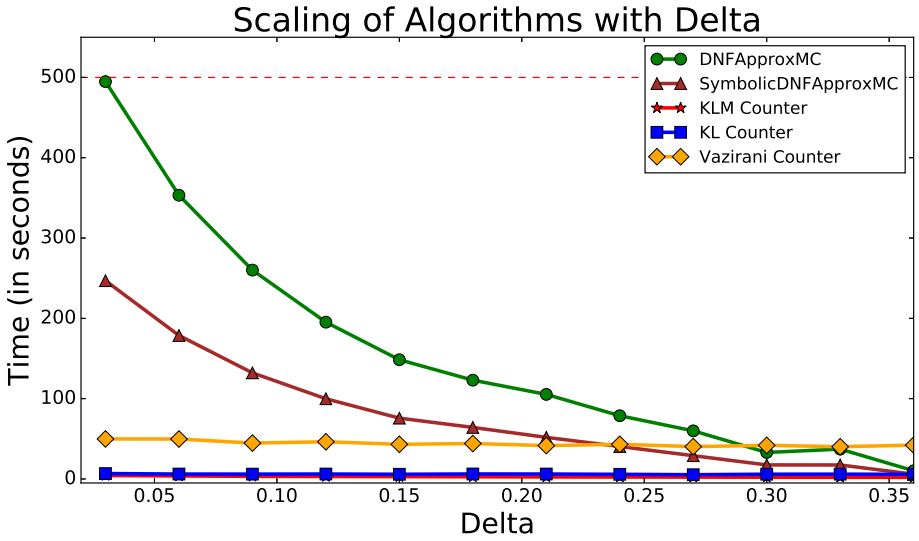


Figure 4.11 : δ Scalability: Monte Carlo FPRAS scale better

4.7 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. $|\mathcal{R}_\varphi| \approx min(2^n, \mathsf{m} * 2^{\mathsf{n}-\mathsf{w}})$, 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 $\rho' = \frac{|\mathcal{R}_\varphi|}{\mathsf{m} * 2^{\mathsf{n}-\mathsf{w}}}$

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. 4.8).Thus DNFApproxMC is more robust across different formula types. This is also apparent in the experiment on formulas with non-uniform cube-widths (Fig. 4.9). 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 4.2 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. 4.10.

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 4.2). 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.

4.8 Chapter Summary

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. It was clear from our preliminary experiments that SAT-based approaches like ApproxMC were not suited for this problem, owing to their inability to exploit DNF-specific optimizations. Taking a non-SAT-based approach, building on Chakraborty et al. [68], we had proposed a hashing-based algorithm, SymbolicDNFApproxMC, in our previous work [74], whose time complexity was shown to be within polylog factors of the best known Monte Carlo schemes. However, two key questions were left unanswered: (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?

The present work 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. [74]. 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. In this context, Row-Echelon hash functions are crucial for achieving scalability by obviating the need for expensive Gaussian Elimination

Chapter 5

Conditional Counting for Explainable AI

5.1 Introduction

In this chapter, we introduce the problem of conditional counting and discuss our solution from the perspective of its application in Machine Learning and eXplainable AI (XAI). In particular, we present a novel constraint-driven framework, called 'CLIME', for explaining opaque ML models like deep neural networks and random forests. An important feature missing in previous ML explainers, is the ability to rigorously measure the quality of explanations they generate, in a theoretically grounded way. Towards this end, we first identified that the problem of conditional counting is at the heart of the task of certifying explanation quality. We proposed a novel approximation algorithm for conditional counting that combines the strengths of Monte Carlo and hashing frameworks to achieve scalability without the loss of theoretical guarantees of accuracy. Equipped with this new algorithm, CLIME not only generates meaningful explanations but also provides rigorous guarantees of their quality. We emphasize that although we present our approximate conditional counting algorithm in the context of ML explanations, it is a general-purpose estimator and can potentially be used for other probabilistic verification tasks, and conditional probability estimation as well.

The rest of this chapter is organized as follows. We first introduce conditional counting in Sec 5.1.1. Then in Sec 5.1.2 we give some background on the field of XAI and introduce our framework called CLIME. We introduce some notation and preliminaries in Sec 5.2 and discuss the explanation generation part of CLIME in more detail in Sec. 5.3. Sec. 5.4 is devoted to explanation certification for which we

present our algorithm for conditional counting. We present our empirical study on CLIME in Sec. 5.5 discuss related work in Sec. 5.6 and summarize in Sec. 5.7.

5.1.1 Conditional Counting

Definition 5.1. *Let φ_A and φ_B be two constraints over the same set of discrete variables. The problem of conditional counting is to compute the ratio*

$$\rho = \frac{\#(\varphi_A \wedge \varphi_B)}{\#\varphi_B} \tag{5.1}$$

To the best of our knowledge, this problem has not been explicitly defined previously in this way, although it is frequently encountered implicitly in applications areas like probabilistic inference and verification. In the case of probabilistic inference, the ratio ρ can be seen to be conditional probability of two events A and B in some sample space, i.e. $\Pr[A|B] = \rho$. In the context of verification, φ_B can encode some 'model' that is to be verified against a desirable property encoded by φ_A. Then Eqn. 5.1 gives the degree to which the model satisfies the property. In our work, we encounter conditional counting in this latter sense, where the 'model' is a user-defined sub-space of inputs, the property encodes a quality metric, and ρ gives the quality of the explanation in the sub-space.

5.1.2 XAI and Constraint-Driven Explanations

The field of eXplainable AI (XAI) has emerged out of the need for humans to understand the complex and opaque decision processes governing modern Machine Learning models. Researchers seek to develop both naturally interpretable models [91, 92, 93, 94] as well as post-hoc explanations for opaque models like Deep Neural Networks and ensembles [95, 96]. State-of-the-art learning approaches in most domains, however, are uninterpretable and necessitate the latter approach.

A number of different approaches have been proposed in literature for generating post-hoc explanations (c.f. [97]). A broad class of techniques explain individual

predictions by capturing the behavior of the opaque model in a small neighborhood of the input instance in two phases [95, 98, 99]. In the first phase, the input instance is perturbed by according to some criteria and in the second phase the behavior of the model on the perturbed instances is captured using various interpretable artifacts such as linear classifiers [95, 96], gradients [100, 101], counterfactuals [102], subgraphs of GNNs [103] etc.

Different choices for each phase yield different tradeoffs between flexibility, computational cost and theoretical rigor. For example, one of the earliest and most popular post-hoc explainers called LIME [95] employs a fixed heuristic perturbation procedure, and uses a simple linear classifier trained on the perturbed instances as the interpretable artifact. The advantages are that LIME is model-agnostic in that it can explain predictions of any black-box model, and is reasonably fast in practice. However, it suffers from drawbacks like lack of a strong theoretical foundation and susceptibility to adversarial attacks [104] among others. The explainer SHAP [96] rectified some of these issues by using Shapley values from game theory for axiomatically deriving the coefficients of the linear model. While Shapley values are provably 'ideal' under some mild assumptions, they are expensive to compute [105, 106], and in practice we have to resort to approximations or accept the loss of model-agnosticity. Recently, [107] proposed to employ user-defined constraints to generate explanations. In particular, they allow the user to supply domain knowledge through constraints in the form of linear inequalities over the input space. These constraints guide the perturbation procedure for generating counterfactual explanations. While constraints provide flexibility in tailoring explanations, the modeling language (linear inequalities) is restrictive and the proposed algorithm is not model-agnostic. Thus, striking a balance between flexibility, rigor, and computational efficiency remains a major challenge.

In this chapter, we present our work on addressing this challenge via design of an efficient constraint-driven perturbation framework that provides robust theoreti-

cal guarantees. In particular, we allow constraints to be expressed as Boolean formulas, which are known to be expressive enough to succinctly encode *all* types of constraints on discrete spaces [108]. This gives the user the flexibility to drill down into the structure of input space. By leveraging advances in Formal Methods, we can generate perturbed samples with strong theoretical guarantees on adherence to the desired distribution, while scaling to large formulas in practice [35, 31]. This allows us to rigorously measure the fidelity of the generated interpretable artifact to the input model i.e., how closely the artifact actually explains the model. Previous works on certifying explanation fidelity solely relied on approximate CNF counting, and required the model to be encoded as a Boolean formula, which severely limited their applicability and scalability. In contrast, our algorithm for conditional counting marries the strengths of Monte Carlo and hashing frameworks to achieve scalability while being truly model-agnostic. Our perturbation framework is decoupled from the artifact generation phase, and the generated samples can be used to train any surrogate classifier that is appropriate for the task. Following LIME, we build a linear model over the generated samples as the interpretable artifact. We experimentally demonstrate how the resulting tool, called CLIME, can be used for generating high quality explanations.

In summary, our contributions are as follows:

1. Framework for precisely crafting explanations for specific subspaces of the input domain through logical constraints

2. A theoretical framework and an efficient algorithm for conditional counting that unifies Monte Carlo and hashing frameworks, for estimating the 'true' explanation fidelity up to any desired accuracy

3. Empirical study showing the efficacy of constraints in

    - Efficient fidelity computation with strong guarantees
    - Zooming in and refining explanations guided by fidelity

- Detecting and foiling adversarial attacks

5.2 Preliminaries

We follow notations from [95]. Let $D = (X, y) = \{(x^1, y^1), (x^2, y^2), \ldots, (x^n, y^n)\}$ denote the input dataset from some distribution $\mathcal{D}$ where $x^i \in \mathbb{R}^d$ is a vector that captures the feature values of the ith sample, and $y^i \in \{\mathcal{C}_0, \mathcal{C}_1\}$ is the corresponding class label*. We use subscripts, i.e. x_j, to denote the j^{th} feature of the vector x. We denote by $f : \mathcal{R}^d \rightarrow [0, 1]$ the opaque classifier that takes a data point x^i as input and returns the probability of x^i belonging to $\mathcal{C}_1$. We assume that an instance x is assigned label $l_f(x) = \mathcal{C}_1$ if $f(x) \geq 0.5$ and $l_f(x) = \mathcal{C}_0$ otherwise.

Surrogate Linear Models. The exact problem formulation varies depending on the choice of interpretable artifact to be generated. Following LIME, SHAP and a host of other popular methods, we choose a simple linear model, as our explanation artifact. Specifically, given a classifier f, the task is to learn a linear model g such that g mimics the behavior of f in the neighborhood of some given point x. The function g is built on an 'interpretable domain' of inputs rather than the original domain. To do so, the original features (that can be continuous or categorical) are mapped to Boolean features. While $x \in \mathbb{R}^d$ represents an instance in the original domain, we use prime-notation, i.e. $x' \in \{0, 1\}^{d'}$ to represent an instance in the interpretable domain. Using Boolean features is a natural choice for 'interpretable domain', as we can understand explanations in terms of a presence/absence of a feature's value. Thus g operates in the interpretable domain $\{0, 1\}^{d'}$. Existing explainers differ in the way the x is perturbed and g is trained. For instance, LIME perturbs the interpretable instance x' by randomly changing 1s to 0s in the binary representation. The generated samples $z'^1, z'^2, \ldots \in \mathcal{Z}'$ are mapped back to the original space as $z^1, z^2, \ldots \in \mathcal{Z}$, where $\mathcal{Z}$ and

*We focus on binary classification; extension to multi-class classification follows by 1-vs-rest approach.

$\mathcal{Z}'$ are called the neighborhoods of x and x' in the respective spaces. The instances $z'^1, z'^2, \ldots$ with corresponding labels $f(z^1), f(z^2) \ldots$ are used as the training set along with a heuristic loss function for building a linear model g using regression. In Sec. 5.3, we discuss the limitations of this approach and show how constraint-sampling can mitigate some of these issues.

Boolean (logical) constraints and uniform sampling. We use notation standard in the area of Boolean Satisfiability (SAT). A Boolean formula over n variables $\varphi : \{0,1\}^n \rightarrow \{0,1\}$ assigns a truth value 0/1 or false/true to each of the 2^n assignments to it's variables and is constructed using logical operators like AND ($\wedge$), OR ($\vee$), NOT ($\neg$), XOR ($\oplus$) etc. An assignment of truth values to variables denoted $s \in \{0,1\}^n$ is said to satisfy φ (denoted $s \models \varphi$) iff $\varphi(s) = 1$. The total number of assignments that satisfy φ is denoted as $\#\varphi = \sum_{s \in \{0,1\}^n} \varphi(s)$. An algorithm is said to be a (perfectly) uniform sampler if it takes as input an arbitrary formula φ and returns an assignment s^* such that $\forall s \models \varphi$, we have $Pr[s^* = s] = \frac{1}{\#\varphi}$. An almost-uniform sampler is an algorithm that takes, along with φ, a parameter $\varepsilon > 0$ as input and returns s^* such that $\forall s \models \varphi$, we have $\frac{1}{(1+\varepsilon)\#\varphi} \leq Pr[s^* = s] \leq \frac{1+\varepsilon}{\#\varphi}$. The tools WAPS [31] and UniGen3 [35] are state-of-the-art perfectly uniform and almost-uniform samplers respectively.

Fidelity. The notion of fidelity aims to capture how closely the explainer model 'reflects' the behavior of the opaque model, and can be seen as a measure of the quality of the explanation [95, 109]. The fidelity $\widehat{\rho}$ of the explainer model g to the opaque model f is calculated as the precision of g [109], i.e. the fraction of the sampled neighbors where the output class of f and g agree. Let $\mathcal{Z}$ and $\mathcal{Z}'$ be the neighborhoods of x and x' as defined above. Then,

$$\widehat{\rho} = \frac{\sum_{z' \in \mathcal{Z}'} \mathcal{I}[l_f(z) = l_g(z')]}{|\mathcal{Z}'|} \tag{5.2}$$

where $\mathcal{I}$ is the indicator function, and z is the preimage of z'.

5.3 Constraint-Driven Explanations

We present our framework CLIME which belongs to a large class of post-hoc explainers that operate in two-phases: in the first phase, it perturbs an input instance x and in the second phase, it generates an interpretable model g to explain the prediction of the opaque model f on x. The new distinctive capability of CLIME is that it lets the user specify constraints on the input space to define the allowed perturbations of x in a model-agnostic way. Next, we discuss our choice for the constraint modeling language and give high-level overview of the two-phase algorithm.

Constraint modeling language. We assume that the constraints are specified in propositional logic, i.e. as a Boolean formula φ . Boolean constraints are powerful enough to represent log-linear family of distributions [110], yet allow fast sampling of solutions either uniformly or with a user-provided bias [31], thanks to the advances in SAT technology. Boolean constraints are also easy to use, and many toolkits for formal analysis such as model-checkers [111] require their input to be specified using Boolean logic.

As an example, assume that φ represents the constraint that at least k features must be fixed for some user-defined k. For image data this constraint enforces the requirement that at least k superpixels must be 'on', while for text it forces at least k words from x to be present in each sample. This blocks out very sparse data ensuring that only informative instances are used for training the explainer model. Example 5.3.1 describes more scenarios where constraints are useful.

The first phase: sampling data points. The CLIME framework generates explanations on instances sampled (almost-) uniformly from user-defined subspaces which are defined through constraints. In this work, we employ techniques for (almost) uniformly sampling solutions of constraints for generating explanations, but we note that the extension to biased (weighted) sampling is straightforward [112].

The pseudo-code of the constrained explanation framework is presented in Alg. 9. Along with the input instance x CLIME also takes as input a Boolean formula φ. The variables of φ are exactly the Boolean features of the interpretable domain $\mathcal{Z}'$, and the solutions φ is the user-defined subspace $\mathcal{U}^{\mathcal{Z}'}$ i.e. $\mathcal{U}^{\mathcal{Z}'} = \{s \in \{0,1\}^n \mid s \models \varphi\}$.

The samples generated from φ determine the neighborhood $\mathcal{Z}'$ (line 1 of Alg. 9). Note that $\mathcal{U}^{\mathcal{Z}'}$ is the universe of all possible assignments from which $\mathcal{Z}'$ is sampled. We assume access to a procedure *getSamples* that returns N independent samples satisfying φ. The algorithm takes as input a parameter ε, which represents the tolerance to deviation from perfectly-uniform sampling. If $\varepsilon = 0$, then the call to *getSamples* in line 1 must be to a perfectly uniform sampler like WAPS [31], otherwise an almost-uniform sampler like Unigen3 [35] suffices. We highlight that CLIME is the first framework with a capability to incorporate constraints to generate explanations in model-agnostic settings, to the best of our knowledge.

The second phase: learning an explainer. We adopt LIME's method for training a linear explainer model for the second (artifact generation) phase. The samples z'^i are mapped back to the original domain, and the output of f on each z^i and the distance of each z^i to x are used for training g in line 6, where at most K coefficients are allowed to be non-zero to ensure interpretability. More formally, let the complexity of an explanation g be denoted as $\Omega(g)$ (complexity of a linear model can be the number of non-zero weights), and let $\pi_x(z)$ denote the proximity measure between inputs x and $z \in \mathcal{Z}$ ($\pi_x(z)$ can be defined using cosine or L_2 distance). The objective function for training g is crafted to ensure that g: (1) approximates the behavior of f accurately in the vicinity of x where the proximity measure is high, and (2) achieves low complexity and is thereby interpretable. The explanation is obtained as $g^* = argmin_{g \in G}\ L(\pi_x, g, f) + \Omega(g)$ where G is the set of all linear classifiers and the loss function L is defined as: $L(f, g, \pi_x) = \sum_{z \in \mathcal{Z}, z' \in \mathcal{Z}'} [f(z) - g(z')]^2 \pi_x(z)$. Intuitively, the loss function captures how unfaithful g is to f in the neighborhood of x.

Example 5.3.1. *We consider the bank dataset [113] that was also used in [107]. The bank dataset contains bank client data that describes client characteristics as well as their communications with a bank. The model predicts whether a client will subscribe for the term deposit. Four* integrity *constraints were proposed for this dataset by [107]. Integrity constraints enforce data consistency and accuracy.*

- ***Previous contacts with a client.*** *Two constraints were proposed. A client has not been contacted before iff the time since previous contact is undefined. A client has not been not contacted before iff the previous outcome is unknown. In terms of the features, these constraints are expressed as ('previous' = 0) ⇔ ('pdays' = undefined) ⇔ ('poutcome' = unknown).*

- ***Features interdependencies.*** *Two constraints were proposed. If a client is a student then they are not married and younger than 35 years old. If a client has an 'admin' job that their education is secondary.*

We can find explanations for a random sample in two scenarios. First, we do not supply constraints to the explainer. In this case, we get an explanation: I = ('duration', 'housing', 'previous', 'poutcome','pdays'), which consists of the 5 most important features (by weight) that contributed to the prediction of the opaque classifier on the input instance. If we add integrity constraints then we get a different explanation (i.e. different set of top 5 features): J = ('duration', 'pdays', 'housing', 'campaign', 'loan').

At this point, it is hard for the user to judge the quality of these explanations I and J. In the next section, we present a technique that enables users to make this judgement.

□

5.4 Certifying Explanation Quality

For increasing user trust, it is necessary to provide a measure of the quality of expla-

Algorithm 9 ExplainWithCLIME$(f, \varphi, \varepsilon, N, x, x', \pi_x, K)$

Input: f: Opaque classifier φ: Boolean constraints
ε: Tolerance N: Number of samples
π_x: Similarity kernel K: Length of explanation

Output: g: Interpretable linear classifier

$\mathcal{Z}' \leftarrow$ getSamples$(\varphi, \varepsilon, N)$;
$Z \leftarrow \{\}$
for $z' \in \mathcal{Z}'$ **do**
    $Z \leftarrow Z \cup \{z', f(z), \pi_x(z)\}$
$g \leftarrow$ K-LASSO(Z, K)

Algorithm 10 computeFidelity$(f, g, \varepsilon, \delta, \gamma)$

Input: f: Opaque Model g: Explainer Model
ε: Tolerance δ: Confidence γ: Threshold

Output: $\widehat{\rho}$: Estimate of ρ (see Thm. 5.2)

if checkThreshold$(f, g, \varepsilon, \delta, \gamma)$ == True **then**
    return $\perp$ ▷ $\widehat{\rho} \leq \gamma - \varepsilon$; report failure
/*Threshold check passed; compute 2-sided bound*/
$\widehat{\rho} \leftarrow AA'(0.4 * \varepsilon, 0.4 * \varepsilon, \delta)$ ▷ See Appendix B.2.1
return $\widehat{\rho}$

Algorithm 11 checkThreshold$(f, g, \varepsilon, \delta, \gamma)$

Input: f: Opaque Model g: Explainer Model

ε: Tolerance δ: Confidence γ: Threshold

Output: True with high probability if $\rho \leq \gamma - \varepsilon$

$\nu \leftarrow \min(\varepsilon + \varepsilon^2/2 - \gamma\varepsilon/2, (\varepsilon - \gamma\varepsilon/2)/(1 + \varepsilon/2))$

/* compute the number of samples N based on $\varepsilon, \delta, \gamma$*/

$N \leftarrow \frac{1}{2\nu^2} \log(\frac{1}{\delta})$

$\mathcal{Z}' \leftarrow$ getSamples$(\varphi, \varepsilon/2, N)$

$C \leftarrow 0$

/* compute sample fidelity */

for $z' \in \mathcal{Z}'$ **do**

/*z is the preimage of z'*/

$c \leftarrow \mathcal{I}[l_f(z) = l_g(z')]$

$C \leftarrow C + c/N$

if $C \leq \gamma$ **then**

/*Value below threshold; terminate early*/

return True

else

return False

nations generated. A fundamental requirement from a high-quality explainer model is that it should closely mimic the behavior of the opaque model in the specified neighborhood. This is especially important for explanations of user defined sub-spaces, as it may be possible that no simple explanation exists for a large subspace, and further refinements to the constraints may be required to get a high-quality explanation.

The fidelity metric, as defined in Eqn. 5.2, aims to quantify this property, in terms of the fraction of samples in the neighborhood $\mathcal{Z}'$, on which the prediction made by the explainer model matches the prediction of the opaque model. Two parameters influence the accuracy of the fidelity score: the number of samples in $\mathcal{Z}'$ and the quality of these samples, i.e. their uniformity in the universe $\mathcal{U}^{\mathcal{Z}'}$ of all such possible samples. Both of these parameters were chosen heuristically in prior works [95, 109], which raises the question, *is the fidelity score, as measured by Eqn. 5.2 trustworthy?* Intuitively, a score measured on 10 samples will not be as accurate as one measured on 10000 due to randomness inherent in any sampling procedure. Such uncertainties can be unacceptable in, for instance, safety-critical applications of XAI such as healthcare [114].

We address this gap by first rigorously defining fidelity, and then presenting an efficient algorithm for computing it. We observe that the true fidelity score is the one that is calculated on *all* possible instances belonging to a user-defined subspace of inputs $\mathcal{U}^{\mathcal{Z}'}$, i.e.

$$\rho = \frac{\sum_{z' \in \mathcal{U}^{\mathcal{Z}'}} \mathcal{I}[l_f(z) = l_g(z')]}{|\mathcal{U}^{\mathcal{Z}'}|} \tag{5.3}$$

Relation to Conditional Counting We highlight that the task of computing ρ as given by Eqn. 5.3 is an instance of conditional counting. To see this, notice that R.H.S. of Eqn. 5.3 can be see to be the count of solutions for the constraint $\mathcal{I}[l_f(z) = l_g(z')]$ conditioned on the count of solutions to the user-defined sub-space φ, i.e. $|\mathcal{U}^{\mathcal{Z}'}|$. In other words, we can take φ_A in Eqn. 5.1 to be the constraint $\mathcal{I}[l_f(z) = l_g(z')]$ and φ_B to be the user-defined sub-space encoded by φ.

In practice, φ can have hundreds of variables and exponentially many solutions which makes enumerating all elements of $\mathcal{U}^{\mathcal{Z}'}$ in the numerator of Eqn. 5.3 infeasible. Thus, computing ρ exactly is usually intractable. Approximating ρ can be faster, but requires formal guarantees to be meaningful. We observe that the score $\widehat{\rho}$, as measured by Eqn. 5.2, is the 'sample mean' of the true 'population mean' ρ, as defined by Eqn. 5.3. This observation allows us to compute the estimate $\widehat{\rho}$ in theoretically grounded way, so as to statistically guarantee its closeness to ρ.

We use a PAC-style notion of approximation [32], which provides strong probabilistic guarantees on the accuracy of the output. The goal is to find an approximation ρ that is within user-defined tolerance of the true value with high confidence. Specifically, we wish to compute $\widehat{\rho}$ such that

$$\Pr[(1-\varepsilon)\rho \leq \widehat{\rho} \leq (1+\varepsilon)\rho] \geq (1-\delta) \tag{5.4}$$

where $\varepsilon > 0$, $\delta > 0$ are user-defined tolerance and confidence.

To the best of our knowledge, no existing approach is directly applicable for finding a good approximation of ρ, in a model-agnostic way. The technique presented by [115], requires the opaque model to be encoded as a Boolean formula, severely limiting both its scalability as well as the types of models that can be explained. On the other hand, algorithms based on Monte Carlo sampling such as the AA algorithm by [71], are known to be fast when ρ is high, but require far too many samples when ρ is low [60]. They also require perfectly uniform samples, while it may only be feasible to generate almost-uniform samples from the universe $\mathcal{U}^{\mathcal{Z}'}$.

In this section, we propose an efficient and model-agnostic estimation algorithm based on [71], that is able to work with almost-uniform samples and also terminates quickly if the quantity being approximated is small. Two key insights inform the design of our approach: we first observe that ε-almost uniform sampling can change the value of ρ at most by a factor of $(1+\varepsilon)$. Secondly, in typical scenarios, users are interested in two-sided bounds on fidelity (as given by Eqn. 5.4) only if it is high enough. If the fidelity is lower than some threshold, say 0.1, then it doesn't matter if

it is 0.05 or 0.01, since the explanation will be unacceptable in either case. In other words, below a certain threshold, one-sided bounds suffice.

Procedure computeFidelity (Alg. 10) is used for computing $\widehat{\rho}$, given an opaque model f, an explainer model g and three parameters ε, δ and γ that control the precision of $\widehat{\rho}$. computeFidelity invokes checkThreshold (Alg. 11) on line 1. checkThreshold first computes the number of samples N required for the probabilistic guarantees, and then invokes a sampler through *getSamples* as in Alg. 9. If $\widehat{\rho} \leq \gamma - \varepsilon$, then checkThreshold returns True with probability at least $1-\delta$ and computeFidelity reports failure on line 2. This check ensures that the sample complexity remains low even if the fidelity is very small, which is a common pitfall for Monte Carlo algorithms. If checkThreshold returns False, then computeFidelity makes a call to procedure AA' (line 4), which is an adaptation of the algorithm by [71] that provides the guarantees of Eqn. 5.4 with almost-uniform samples (see Appendix B.2.1). Theorem 5.2 captures the guarantees and the behavior of the framework.

Theorem 5.2. *If $\rho \leq \gamma - \varepsilon$, then* computeFidelity *returns $\perp$ with high probability (i.e. at least $1-\delta$). If $\rho \geq \gamma + \varepsilon$, w.h.p., it returns an estimate $\widehat{\rho}$ such that* $\Pr[(1-\varepsilon)\rho \leq \widehat{\rho} \leq (1+\varepsilon)\rho] \geq (1-\delta)$.

We highlight that our certification framework is more general than just fidelity computation, and is applicable broadly to the more general problem of conditional counting. In Appendix B.2.1, we show how Algs. 10 and 11 can be used for accurately estimating the true mean of any 0/1 random variable with almost-uniform samples and early termination.

Example 5.4.1. *We continue with Example 5.3.1. Now, the user can use the fidelity metric to compare quality of explanations. We compute the fidelity score for both explanations I and J that we obtained without and with constraints, respectively. We get that fidelity(I)= 0.91 and fidelity(J)= 0.90. First, the user notices that the fidelity scores are the same for these explanations, so I and J can be seen as explanations of*

the same quality. Second, these fidelity scores can be considered low, hinting the user to refine the input space. In the next section we show, through an extensive evaluation, how the user can perform such a refinement to obtain high quality explanations. □

5.5 Experiments

We seek to answer the following research questions through our empirical study:

1. How scalable is the certification framework presented in Sec. 5.4?
2. What benefits do constraints provide for analysing ML models?
3. How susceptible are constrained explanations to adversarial attacks?

5.5.1 Efficiency of certification

A salient benefit of leveraging the AA-algorithm of [71] for the fidelity computation approach of Sec. 5.4, is that its sample complexity is guaranteed to be close-to-optimal. Nevertheless, the practical performance of our approach is unknown apriori, given the added cost of generating (almost-) uniform samples from constraints. Therefore, in this experiment, we evaluate the scalability of our framework vs. that of the technique of [115].

We implemented and ran Algs. 10, 11 on 150 benchmarks used in [115]. The benchmarks are CNF formulas that encode the Anchor [109] explanations of Binarized Neural Networks trained on Adult, Recidivism and Lending datasets. The true fidelity of an explanation can be computed from the count of the number of solutions of the corresponding formula. We compared the running-time of our tool to the time taken by the approach of [115], which utilizes the state-of-the-art approximate model-counting tool called ApproxMC [35]. Note that both ApproxMC and our tool provide the same probabilistic guarantees on the returned estimate [34]. The results are shown as a scatter-plot in Fig. 5.1. The x-coordinate of a point in blue represents the time taken by ApproxMC on a benchmark, while the y-coordinate represents

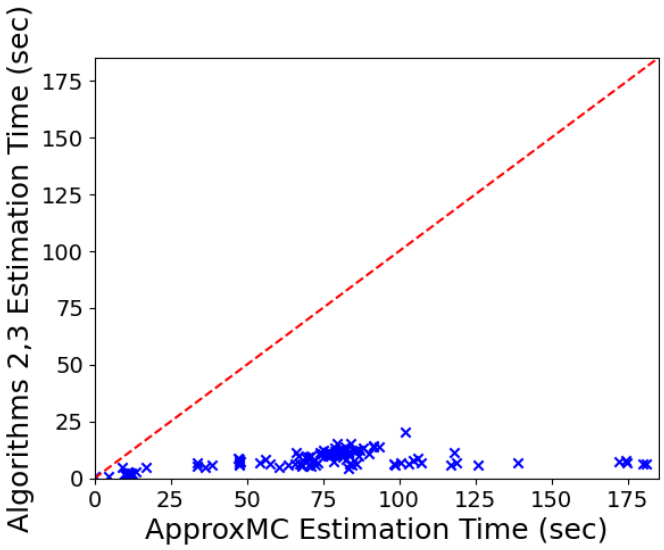


Figure 5.1 : Scalability of Algs. 2,3 vs. ApproxMC

the time taken by our approach. As all the points are far below the diagonal dotted red line, we can infer that ApproxMC takes significantly longer than our tool to compute the same estimate. In fact, on average (geometric mean), our tool is 7.5× faster than ApproxMC. It is clear from Fig. 5.1 that our algorithm scales far better than the alternative, despite being more general and *model-agnostic*. We also experimentally compared the scalability of our tool to ApproxMC for different values of input tolerance ε and confidence δ. We found that our tool scales significantly better than ApproxMC for tighter tolerance and confidence values. Thus, our experiments demonstrate that our tool significantly outperforms the state-of-the-art. We provide more details and results in Appendix B.2.4.

5.5.2 Model analysis

First, we consider the bank dataset that was proposed in [107] that we describe in Example 5.3.1. We train 10 random forest models with different random seeds and the same hyper-parameters as in [107]. The average accuracy of these models is 90%.

In all experiments, we compute the average fidelity scores over 100 input explanations and 10 RF models.

Let us consider a scenario where the user needs to explain why a client who has not been contacted in the past made a decision to not subscribe for a term deposit. We find explanations for 100 samples per model in two scenarios: (a) without constraints and (b) with integrity constraints, as in Example 5.4.1, and average the result. We get fidelity scores 0.90 and 0.89 for scenarios (a) and (b), respectively. These result confirm scores that we obtain for a single instance in Example 5.4.1. However, the obtained fidelity scores might not be acceptable for the user. A low fidelity score can indicate either that the constrained space needs to be refined or that the interpretable artifact needs to be changed (ex: using decision trees instead of linear classifiers to explain non-linear decision boundaries). We highlight that unlike [107], under our modular framework, it is easy to learn a different artifact. In this work, we focus on constraint refinement and assume that the user intends to continue drilling down by specifying *user-defined* constraints to better communicate their focus space to an explainer.

To achieve their goal, the user can specify an additional constraint: 'consider only clients that have not been previously contacted'. So, the user adds this constraint on top of integrity constraints, creating a new setup: (c) CLIME is supplied with integrity and the user constraint. We again compute the average fidelity score that is 0.98 for the scenario (c). Clearly, adding the user-defined constraint allowed to refine the input space to obtain high quality explanations.

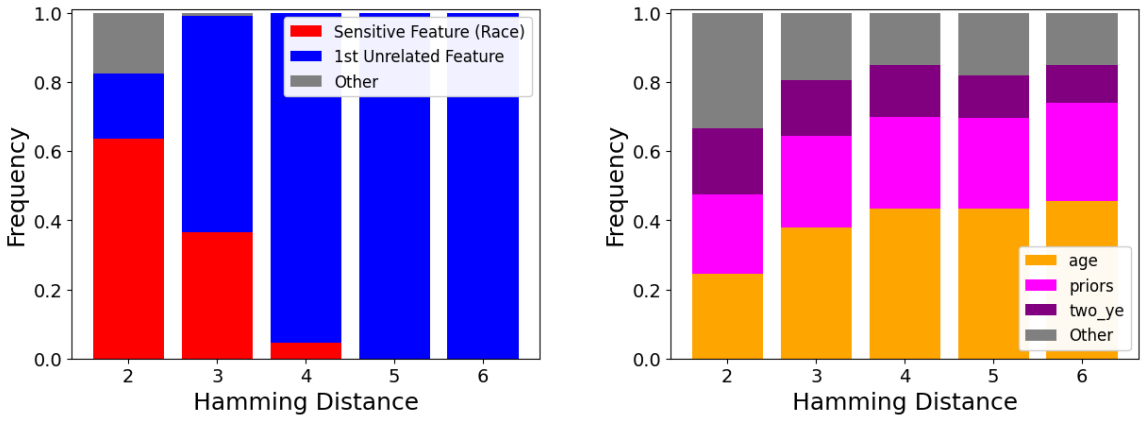
Note that the user-defined constraint 'a client has not been contacted before' triggers integrity constraints (see Example 5.3.1 for the definition of integrity constraints) forcing that 'the time since previous contact should be undefined' and 'the previous outcome should be unknown'. Hence, these features, i.e. 'previous', 'pdays', 'poutcome' in the dataset, are *fixed* by the user's constraints. Therefore, these fixed features should not appear in the explanations. The next table shows the top 5 fea-

Scenario	Top five features (left to right) in explanations				
(a)	'previous'	'pdays'	'duration'	'poutcome'	'housing'
(c)	'age'	'contact'	'duration'	'personal loan'	'month'

tures that were used in explanations for the scenarios (a) and (c) in 100 instances.

Note that three fixed features are often chosen by CLIME *without* constraints (scenario (a)), making these explanations less useful for the user. In contrast, CLIME with constraints (scenario (c)) never chooses these features in its explanations demonstrating the correct behaviour.

Second, we consider the adult dataset [116], originally taken from the Census bureau. It is used for predicting whether or not a given adult person earns more than $50K a year depending on various attributes, e.g. race, sex, education, hours of work, etc. We pre-processed columns with continuous features, e.g. the pre-processor discretizes the capital gain and capital loss features into categorical features, e.g. 'Unknown', 'Low' and 'High'[109]. We train 10 Random Forest models with different random seeds and 20 trees and max depth is 7. The accuracy is 83% on average. We compute the average fidelity score of 100 inputs explanations and RF models. Consider a scenario when the user wants to find an explanation for male individuals without a college degree. So, the user adds a constraint (Education ∈ [dropout, high-school, some-college]) and (Sex = Male). Next, they find that the average fidelity score is 0.68 for explanations in this constrained space. This indicates a need for refining the input space. For example, they can add a constraint that there is no information available about individuals' Capital, i.e.: (Capital Gain = 'Unknown') and (Capital Loss = 'Unknown'). In this more constrained space, the average fidelity score of explanations increases to 0.98. Therefore the user can be confident that explanations are reliable.



(a) Adversarial Classifier

(b) Non-Adversarial Classifier

Figure 5.2 : Recidivism: Top CLIME explanation distribution vs. Hamming Distance

5.5.3 Detecting Adversarial Attacks

[104] presented a way to craft an adversarial attack that seeks to hide the biased predictions made by a biased classifier (eg: one that decides credit card applications solely on a sensitive feature like race) from detection by post-hoc explainers like LIME, by exploiting the ad-hoc perturbation procedure that can generate Out-of-Distribution (OOD) samples. In this experiment we evaluate the suitability of constraints in foiling such attacks.

Adversarial classifier setup. Given a biased classifier, [104] construct an adversarial classifier that 'wraps' around the biased classifier aiming to hide its biases. It relies on the ability to successfully train a classifier to predict whether an input sample is in- or out-of-distribution (OOD). This classifier, called the 'OOD detector', is used as a submodule in the adversarial classifier to predict whether an input instance is OOD or not. If the input instance is OOD, then the adversarial classifier outputs a *prediction based on an innocuous feature*. If the instance is not OOD, then it outputs the biased classifier's prediction. When LIME is used for explaining a prediction made by the adversarial classifier, almost all of LIME's perturbed samples are detected to be OOD, resulting in the innocuous feature appearing as the top explanation. In this

way, the sensitive feature (eg. race), is successfully hidden, when in reality it was the ground-truth explanation. The attack crucially relies on the ability to train an OOD detector to tell apart LIME's perturbed samples and instances from the original distribution with high accuracy. Note that accurately measuring the fidelity of the explainer model alone is not sufficient to detect such an attack as the problem lies in the way the opaque classifier is constructed and not in the resulting explanation.

A blind spot of the OOD detector We hypothesized that for discrete data, the OOD detector should have trouble differentiating between instances from the original dataset and instances with low Hamming distances to points in the original dataset. We emphasize here, that LIME almost always generates samples at a high Hamming distance from the input instance, as it perturbs each feature independently. Consequently, the penalization of high Hamming distance, either through the loss function or through explicit filtering by rejection sampling, is ultimately ineffective, as the number of samples is ad-hoc and fixed apriori. This may partly explain LIME's susceptibility to such attacks.

Our idea is to generate constrained samples based on their Hamming distance h from the original input. For adversarial classifiers, for small values of h, we hit the blind spot of the OOD detector and our sampled instances will be marked as in-distribution. So, sensitive features will be revealed by CLIME. As we move away from the instance, the innocuous features should dominate. Towards this end, we constructed a formula φ^h for a given Hamming distance h and a set of instances S, such that the solutions of φ^h are exactly those instances that are at most at a distance h from each instance in S. We took S to be the set of 3 nearest neighbors of the instance being explained. If our hypothesis is correct, the adversarial classifier should have trouble hiding the sensitive features in the subspace $\mathcal{Z}_{\varphi^h}$ for low values of h. Thus, sensitive features should appear in the top CLIME explanations when h is small, and innocuous features when h is large.

To test our hypothesis, we generated 200 CLIME explanations using discretized

versions of the three datasets used by [104]. We computed the relative frequency of a feature appearing as the top explanation for different values of h. The results for the Recidivism dataset with 1 innocuous feature are depicted in Figure 5.2a (we defer results on other datasets and an extended discussion to Appendix B.3 for lack of space). We plot the hamming distance h on the X-axis and the frequency of seeing a feature as the top explanation on the Y-axis. We see the sensitive feature 'race' appears as the top explanation for 60% of the instances explained when $h = 2$. In contrast, the innocuous feature (1st Unrelated Feature), appears almost exclusively as the top explanation when $h \geq 5$. For comparison, we depict results on CLIME explanations for a non-adversarial classifier in Fig. 5.2b. We see that the relative frequency with which a feature appears as the top explanation changes very little for different hamming distances. This confirms our hypothesis. In contrast, for LIME explanations (not shown), the innocuous feature appears as the top explanation for *all* 200 instances. We thus conclude that CLIME can not only avoid being fooled, but also potentially detect the adversarial natural of an opaque classifier by observing the change in the top features with and without Hamming distance constraints. While it may be possible to craft even more sophisticated attacks, these results clearly demonstrate CLIME's ability to detect adversarial attacks that exploit OOD sampling.

5.6 Related Work

We focus on constrained explanations in this section, and defer a fuller discussion of other related work to Appendix B.1 for lack of space. To the best of our knowledge, the work of [107] is the first and only approach to incorporate user-defined constraints into the explanation process explicitly. Approaches like [117] allow some restrictions on the types of allowed perturbations (such as rotations and deletions for images), but are much more limited compared to the expressive power of a full-fledged formally defined constraint language.

The approach of Deutch and Frost [107] is the closest to CLIME, yet differs in

several ways. Firstly, they generate counterfactual explanations as the interpretable artifact, which are defined as the smallest changes to the input instance that make the opaque predictor label it differently. In contrast, CLIME is a feature attribution method that directly explains the relative importance of features contributing to a prediction. Further, [107] employ a two-step 'perturb and project' method for incorporating constraints into the counterfactual generation phase, wherein the input instance is first perturbed in the direction of the target label without constraints, and is then projected on to the constrained space. This method is not guaranteed to converge and also imposes restrictions on the type of models it can explain. In contrast, CLIME directly samples perturbations satisfying the constraints, and is completely agnostic to the model being explained.

5.7 Chapter Summary

We presented a modular model-agnostic explanation framework CLIME that is able to operate on constrained subspaces of inputs. We introduced a new estimation algorithm that enables computation of an explanation's quality up to any desired accuracy. XAI is inherently human-centric, and in this light, our framework empowers the user to iteratively refine the explanation according to their needs. We demonstrated concrete scenarios where the user can zoom in to the input space guided by the fidelity metric.

The key contribution in the realm of Constrained Counting and Sampling is a new approximation algorithm for the problem of conditional counting (see Eqn. 5.1). Our approach allows the samples from φ_B to be drawn almost-uniformly instead of exactly uniformly, as required by classical Monte Carlo. This allows approximate samplers like UniGen to be used for generating the candidate samples to check against φ_A. In many cases, it is only possible to generate approximately-uniform samples using tools like UniGen due to scalability issues with exact tools. Further, our algorithm uses the Monte Carlo based AA algorithm by Dagum et al. [71] as a building block

which yields close-to-optimal sample complexity. Coupled with our insight for early termination, we get a fast practical algorithm that comprehensively outperforms prior work [115].

Part III

Exact Counting and Sampling

Chapter 6

Background

6.1 History

Algorithms for Boolean satisfiability, such as DPLL [118] had been proposed as far back as the 1960s. Parallely, algorithms for exact counting such as the polynomial time FKT algorithm [119, 120] for counting the number of perfect matchings in planar graphs were proposed around the same time. Nevertheless, despite theoretical interest surrounding these problems, practical implementations were scarce. This scenario changed in the 90s, with the introduction of GRASP [22] and Chaff [21] SAT solvers, which first introduced the now-ubiquitous SAT techniques like CDCL, lazy unit propagation, VSIDS heuristics and the like. The 'SAT-revolution' started by these solvers prompted researchers to look at problems considered to be even harder than SAT, among them propositional model counting.

Early exact model counters such as `CDP` [121], `Cachet` [25] and `sharpSAT` [24] were based on extending the existing CDCL framework to exhaustively search the solution space for all possible models, along with counting-specific enhancements like component-caching, implicit Boolean Constraint Propagation etc. Simultaneously, in the related subfield of Knowledge Compilation (KC), compact representations of Boolean functions such as d-DNNF [122] were proposed, which turned out to be equivalent to the trace of modern model counters. This led to synergistic development in KC and model-counting communities, owing to the fact that compilers like `C2D` [26] and `d4` [27] easily doubled as model-counters. The first exact uniform sampling tool, `SPUR` [29], was proposed soon after, and it extended `sharpSAT` to allow generating samples on the fly using the technique of reservoir sampling. The tools `KUS` [30]

and `WAPS` [31] took the approach of explicitly compiling and annotating the d-DNNF representation in-memory, which allowed for very fast sampling through amortization of the compilation cost.

Similar to the approximate case, this progress in exact counting suggested it was a one-stop-shop for all real-world benchmarks. However, in our experiments on the problem of computing the permanent of a 0-1 matrix, we found that all exact counters performed rather poorly even compared to a brute-force approach. Ryser's algorithm [123], which is a brute force, $n \times 2^n$-step approach, was able to compute the permanent for all matrices up to size 27×27, while tools like `d4` timed out on matrices of size 14×14, except when they were very sparse. Interestingly, even modern SAT solvers like Glucose were unable to determine the existence of a solution within the same time limit, for certain matrices above size 24×24, regardless of the encoding used to for translating Exact-One constraints into CNF. Similar results were also seen for problems like sampling traces of a transition system. These results indicated that SAT-based exact approaches were incapable of fully exploiting the structure inherent in such problems and that alternative techniques may be needed.

In this part of the thesis, we discuss our work in bridging this gap, by leveraging datastructures called Algebraic Decision Diagrams (ADDs) [50] for compactly representing the solution space using factored representations [49]. We show how this general approach can be applied to diverse domains like computing the matrix permanent (Chapter 7), sampling traces of a transition system (Chapter 8), and sampling solutions of low treewidth CNF formulas (Chapter 9). In the rest of this chapter, we introduce the SAT-based and ADD-based paradigms for counting and sampling.

6.2 SAT-based approach

The SAT-based approach for counting and sampling at its core, is based on the CDCL (Conflict Driven Clause Learning) framework [22] for satisfiability. For purposes of counting, CDCL is run exhaustively until the entire solution space is explored, instead

of stopping at the first satisfying assignment. For sampling, the fastest SAT-based samplers like `KUS` and `d4` first compile a datastructure called the Decomposable Deterministic Negation Normal Form (d-DNNF). This datastructure allows fast generation of samples by performing a top-down random walk guided by the model-counts of partial assignments. In this section, we give an overview of the common techniques used in algorithms based on this paradigm. For a detailed exposition on the topic, we refer the reader to [23].

6.2.1 CDCL

The CDCL algorithm is an enhancement of the DPLL algorithm [118] with clause learning. The original DPLL algorithm is a simple backtracking search coupled with Unit Propagation. A clause with a single literal is known as a unit clause, and the existence of a such a clause in a formula, can be used to infer that the corresponding variable must be assigned the polarity of the literal in order to satisfy the formula. Thus given a CNF formula, the DPLL algorithm first carries out all possible unit propagations and then picks a variable to assign, and recurses on the two branches corresponding to the the positive and negative assignments to the chosen variable.

The CDCL algorithm, enhances this simple procedure, with conflict analysis and non-chronological backtracking. A conflict is said to arise when any partial assignment to the variables leads to the setting of all variables in a clause to false. In such a case it is possible to both learn a clause such that adding the clause to the original formula does not change its satisfiability, and it also prevents the algorithm from exploring the same partial assignment that led to the conflict. Non-chronological backtracking allows jumping back multiple levels in the search tree upon detection of a conflict, as opposed to one level at a time.

Early SAT solvers like zChaff [21] incorporated additional techniques like lazy unit propagation, VSIDS heuristic, learnt clause deletion strategies etc. which allowed to it scale much better than previous tools and heralded the SAT revolution.

6.2.2 Exhaustive CDCL with Component Caching

The same algorithm, can be easily extended to exhaustively search the entire solutions space for satisfying assignments. However, it is necessary to tailor the algorithm for counting in order to allow it to scale. One such critical enhancement is component analysis and caching.

A CNF formula can be partitioned into sets of clauses such that no two sets share any variables. Each such set is called a component, which can also be thought of as a (smaller) CNF formula. Given such a partition of a formula, the count of the formula is simply the product of the counts of each component. This fact can be used to greatly speed up counting by avoiding expressly counting every combination of partial satisfying assignments to each component. Further, it is often seen that the same components reappear during the exhaustive search. Therefore, one can also store and reuse the count of a component after encountering it for the first time. Over the years, many SAT-based counters such as `Cachet`, `sharpSAT`, `Ganak` etc. have refined and enhanced these basic principles to obtain extremely efficient component analysis and caching in practice.

Besides component analysis, techniques like implicit BCP [24], VSADS heuristic (as opposed to VSIDS for SAT), etc. have also been crucial to the success of the SAT-based approach. A detailed discussion can be found in [23].

6.2.3 d-DNNF Representation

Darwiche and Marquis [122] pioneered the area of Knowledge Compilation (KC), where the overarching idea is to compile a given Boolean function (generally represented as a Boolean formula), into a representation that allows for fast querying of the function. KC is the study of different representations and queries, with a goal to analyze time and space complexities as well as practical performance. For example, the well-known Ordered Binary Decision Diagrams (OBDDs) [124] are one such representation, and they allow many queries like model-counting, function equivalence,

Prime Implicants etc. to be performed in polynomial time in the size of the OBDD. However, there is a tradeoff between supported queries and size of the representation. OBDDs are known to blow-up in size and are not directly useful for model counting, as they typically do not scale beyond a roughly a hundred variables.

Darwiche [125] introduced another such representation, called the d-DNNF, which is more compact in lieu of supporting fewer polynomial-time queries. A d-DNNF is essentially a DAG where (1) each internal node labeled with AND or an OR, (2) each leaf is labeled with a literal or 0/1 (false/true), (3) no two children of an AND node share any variables (4) children of OR nodes are mutually inconsistent. Crucially, however, it supports model counting and sampling in polynomial time in the size of the representation. Further, Huang and Darwiche [126] showed that the trace of modern model counters like `Cachet` are actually equivalent to a subset of d-DNNF, called decision-DNNF, where each OR node has exactly two children and is associated with a decision variable that appears in opposite polarities in each branch. This deep connection between model-counters and d-DNNF has proven to be mutually beneficial, and modern tools often do both counting and compilation (eg. `d4` [27], `DSharp` [127]). Further, tools like `KUS` and `WAPS` explicitly compile a given CNF formula into d-DNNF to allow for fast sampling, while the tool `SPUR` does it on-the-fly using the model counter `sharpSAT` as a backend.

6.3 ADD-based approach

6.3.1 Algebraic Decision Diagrams

Let X be a set of Boolean-valued variables. An Algebraic Decision Diagram (ADD) is a data structure used to compactly represent a function of the form $f : 2^X \to \mathbb{R}$ as a Directed Acyclic Graph (DAG). For functions with logical structure, an ADD representation can be exponentially smaller than the explicit representation. Originally designed for matrix multiplication and shortest path algorithms, ADDs have been used for a variety of applications including Bayesian inference [128, 129], stochastic

planning [130], and model counting [131, 49]. ADDs were originally proposed as a generalization of Binary Decision Diagrams (BDDs), which can only represent functions of the form $g : 2^X \to \{0, 1\}$. Formally, an ADD is a 4-tuple (X, T, ρ, G) where X is a set of Boolean variables, the finite set $T \subset \mathbb{R}$ is called the carrier set, $\rho : X \to \mathbb{N}$ is the diagram variable order, and G is a rooted directed acyclic graph satisfying the following three properties:

1. Every terminal node of G is labeled with an element of T.

2. Every non-terminal node of G is labeled with an element of X and has two outgoing edges labeled 0 and 1. The node at the other of the 1-edge is called the 'then-child' of the parent node v (denoted $v.then$) and the node at the other end of the 0-edge is called the 'else-child' (denoted $v.else$).

3. For every path in G, the labels of visited non-terminal nodes must occur in increasing order under ρ.

ADDs and BDDs differ in the carrier set T; for ADDs $T \subset \mathbb{R}$ while for BDDs, $T = \{0, 1\}$. We use lower case letters $f, g, \ldots$ to denote both functions from Booleans to reals as well as the ADDs representing them. Many operations on such functions can be performed in time polynomial in the size of their ADDs. We list some such operations that will be used in our discussion.

- *Product*: The product of two ADDs representing functions $f : 2^X \to \mathbb{R}$ and $g : 2^Y \to \mathbb{R}$ is an ADD representing the function $f \cdot g : 2^{X \cup Y} \to \mathbb{R}$, where $f \cdot g(\tau)$ is defined as $f(\tau \cap X) \cdot g(\tau \cap Y)$ for every $\tau \in 2^{X \cup Y}$,

- *Sum*: Defined in a way similar to the product.

- *If-Then-Else (ITE):* This is a ternary operation that takes as inputs a BDD f and two ADDs g and h. $ITE(f, g, h)$ represents the function $f \cdot g + \neg f \cdot h$, and the corresponding ADD is obtained by substituting g for the leaf '1' of f and h for the leaf '0', and simplifying the resulting structure.

- *Additive Quantification*: The existential quantification operation for Boolean-valued functions can be extended to real-valued functions by replacing disjunction with addition as follows. The additive quantification of $f : 2^X \rightarrow \mathbb{R}$ is denoted as $\exists x.f : 2^{X \setminus \{x\}} \rightarrow \mathbb{R}$ and for $\tau \in 2^{X \setminus \{x\}}$, we have $\exists x.f(\tau) = f(\tau) + f(\tau \cup \{x\})$.

ADDs share many properties with BDDs. For example, there is a unique minimal ADD for a given variable order ρ, called the *canonical ADD*, and minimization can be performed in polynomial time. Similar to BDDs, the variable order can significantly affect the size of the ADD. Hence heuristics for finding good variable orders for BDDs carry over to ADDs as well. ADDs typically have lower *recombination efficiency*, i.e. number of shared nodes, vis-a-vis BDDs. Nevertheless, sharing or recombination of isomorphic sub-graphs in an ADD is known to provide significant practical advantages in representing matrices, vis-a-vis other competing data structures. The reader is referred to [124, 132] for more details on these decision diagrams.

6.3.2 Factored Representations and Applications to Counting

While ADDs can be used for representing pseudo-Boolean functions as-is, it has been observed that for various functions arising from practical applications, the corresponding ADD representation quickly exceeds memory requirements. For example, typical CNF benchmarks arising from domains like probabilistic inference, logic circuits etc. consist of thousands of variables and clauses, and representing such functions using monolithic ADDs is nigh impossible [49].

However, it has been observed that it is possible to get around this issue by effectively factoring such large functions in smart ways, and representing each factor separately using an ADD. While the benefits of factored representations had been identified in the context of MDPs in the late 90s [133], it was recently shown that the same principle can also yield significant improvements in scalability in the context of model counting [49]. Dudek et al. [49, 59] proposed a 2-phase algorithm for model

counting based on factoring the given formula smartly and using ADDs to represent and operate on intermediate functions. In the first phase, a plan for the model counting query is generated, either using tree decomposition solvers or special heuristics. In the second phase, the plan so constructed is used for performing a series of project and join (i.e. additive quantification and product) operations, which finally yields the model count. Dudek et al. showed that this approach can be competitive with the SAT-based one, especially in cases when the treewidth is low.

It is known that (singular) OBDDs are less compact than d-DNNF representations [122]. However, a Tree-of-BDDs [134, 135] representation was shown to be incomparable in terms of succinctness to d-DNNF [136]. In other words, there exist functions for which Tree-of-BDDs is more compact than d-DNNF and vice versa. In this context, the Tree-of-ADDs representation, as compiled implicitly by model counters like `ADDMC` and `DPMC` [49, 59] is seen as a viable alternative to the SAT-based approach. In our works, we illuminated the fact that the Tree-of-ADDs representation underlies the approach of Dudek et al. We leveraged this idea for constructing the first weighted sampler to expressly target low-treewidth CNF formulas. Further, we come up with domain-specific factorization methods (as opposed to the generic ones employed previously) for effectively solving problems like computing the matrix permanent, and sampling traces of a transition system. In fact, the contrast between the SAT-based and ADD-based approaches in these problems is more vast, as compared to the results of Dudek et al.; we often see that our ADD-based algorithms far outperform competing SAT-based tools.

Chapter 7

Matrix Permanent

7.1 Introduction

Many Constrained Counting problems reduce to counting problems on graphs. For instance, learning probabilistic models from data reduces to counting the number of topological sorts of directed acyclic graphs [137], while computing the partition function of a monomer-dimer system reduces to computing the number of perfect matchings of an appropriately defined bipartite graph [138]. In this chapter, we focus on the last class of problems – that of counting perfect matchings in bipartite graphs. It is well known that this problem is equivalent to computing the *permanent* of the 0-1 bi-adjacency matrix of the bipartite graph. We refer to these two problems interchangeably in the remainder of this chapter.

Given an $n \times n$ matrix $\boldsymbol{A}$ with real-valued entries, the permanent of $\boldsymbol{A}$ is given by $perm(\boldsymbol{A}) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$, where S_n denotes the symmetric group of all permutations of $1, \ldots n$. This expression is almost identical to that for the determinant of $\boldsymbol{A}$; the only difference is that the determinant includes the sign of the permutation in the inner product. Despite the striking resemblance of the two expressions, the complexities of computing the permanent and determinant are vastly different. While the determinant can be computed in time $\mathcal{O}(n^{2.4})$, Valiant [139] showed that computing the permanent of a 0-1 matrix is #P-Complete, making a polynomial-time algorithm unlikely [8]. Further evidence of the hardness of computing the permanent was provided by Cai, Pavan and Sivakumar [140], who showed that the permanent is also hard to compute on average. Dell et al. [141] showed that there can be no algorithm with sub-exponential time complexity, assuming a weak version of the Exponential

Time Hypothesis [7] holds.

The determinant has a nice geometric interpretation: it is the oriented volume of the parallelepiped spanned by the rows of the matrix. The permanent, however, has no simple geometric interpretation. Yet, it finds applications in a wide range of areas. In chemistry, the permanent and the permanental polynomial of the adjacency matrices of fullerenes [142] have attracted much attention over the years [143, 52, 144]. In constraint programming, solutions to All-Different constraints can be expressed as perfect matchings in a bipartite graph [53]. An estimate of the number of such solutions can be used as a branching heuristic to guide search [54, 145]. In physics, permanents can be used to measure quantum entanglement [51] and to compute the partition functions of monomer-dimer systems [138].

Since computing the permanent is hard in general, researchers have attempted to find efficient solutions for either approximate versions of the problem, or for restricted classes of inputs. In this chapter, we restrict our attention to exact algorithms for computing the permanent. The asymptotically fastest known exact algorithm for general $n \times n$ matrices is Nijenhuis and Wilf's version of Ryser's algorithm [123, 146], which runs in time $\Theta(n \cdot 2^n)$ for all matrices of size n. For matrices with bounded treewidth or clique-width [147, 148], Courcelle, Makowsky and Rotics [149] showed that the permanent can be computed in time linear in the size of the matrix, i.e., computing the permanent is Fixed Parameter Tractable (FPT). A large body of work is devoted to developing fast algorithms for sparse matrices, i.e. matrices with only a few entries set to non-zero values [150, 52, 151, 152] in each row. Note that the problem remains #P-Complete even when the input is restricted to matrices with exactly three 1's per row and column [153].

An interesting question to ask is whether we can go beyond sparse matrices in our quest for practically efficient algorithms for the permanent. For example, can we hope for practically efficient algorithms for computing the permanent of *dense* matrices, i.e., matrices with almost all entries non-zero? Can we expect efficiency when the

rows of the matrix are "similar", i.e. each row has only a few elements different from any other row (sparse and dense matrices being special cases)? Existing results do not seem to throw much light on these questions. For instance, while certain non-sparse matrices indeed have bounded clique-width, the aforementioned result of Courcelle et al [63, 149] does not yield practically efficient algorithms as the constants involved are enormous [154]. The hardness of non-sparse instances is underscored by the fact that SAT-based model counters do not scale well on these, despite the fact that years of research and careful engineering have enabled these tools to scale extremely well on a diverse array of problems. We experimented with a variety of CNF-encodings of the permanent on state-of-the-art counters like D4 [27]. Strikingly, no combination of tool and encoding was able to scale to matrices even half the size of those solved by Ryser's approach in the same time, despite the fact that Ryser's approach has exponential complexity even in the best case.

We now present our work where we demonstrate that practically efficient algorithms for the permanent can indeed be designed for large non-sparse matrices if the matrix is represented compactly and manipulated efficiently using a special class of data structures. Specifically, we propose using *Algebraic Decision Diagrams* [50] (ADDs) to represent matrices, and design a version of Ryser's algorithm to work on this symbolic representation of matrices. This effectively gives us a symbolic version of Ryser's algorithm, as opposed to existing implementations that use an explicit representation of the matrix. ADDs have been studied extensively in the context of formal verification, and sophisticated libraries are available for compact representation of ADDs and efficient implementation of ADD operations [155, 156]. The literature also contains compelling evidence that reasoning based on ADDs and variants scales to large instances of a diverse range of problems in practice, cf. [50, 157]. Our use of ADDs in Ryser's algorithm leverages this progress for computing the permanent. Significantly, there are several sub-classes of matrices that admit compact representations using ADDs, and our algorithm works well for all these classes. Our empirical study

provides evidence for the first time that the frontier of practically efficient permanent computation can be pushed well beyond the class of sparse matrices, to the classes of dense matrices and, more generally, to matrices with "similar" rows. Coupled with a technique known as early abstraction, ADDs are able to handle sparse instances as well. In summary, the symbolic approach to permanent computation shows promise for both sparse and dense classes of matrices, which are special cases of a notion of row-similarity.

The rest of this chapter is organized as follows: in Section 7.2 we introduce Ryser's original algorithm and other concepts that we will use in this chapter. We discuss related work in Section 7.3 and present our algorithm and analyze it in Section 7.4. Our empirical study is presented in Sections 7.5 and 7.6 and we conclude in Section 7.7.

7.2 Preliminaries

We denote by $\boldsymbol{A} = (a_{ij})$ an $n \times n$ 0-1 matrix, which can also be interpreted as the bi-adjacency matrix of a bipartite graph $G_{\boldsymbol{A}} = (U \cup V, E)$ with an edge between vertex $i \in U$ and $j \in V$ iff $a_{ij} = 1$. We will denote the ith row of $\boldsymbol{A}$ by r_i. A perfect matching in $G_{\boldsymbol{A}}$ is a subset $\mathcal{M} \subseteq E$, such that for all $v \in (U \cup V)$, exactly one edge $e \in \mathcal{M}$ is incident on v. We denote by $perm(\boldsymbol{A})$ the permanent of $\boldsymbol{A}$, and by $\#PM(G_{\boldsymbol{A}})$, the number of perfect matchings in G. A well known fact is that $perm(\boldsymbol{A}) = \#PM(G_{\boldsymbol{A}})$, and we will use these concepts interchangeably when clear from context.

7.2.1 Ryser's Formula

The permanent of $\boldsymbol{A}$ can be calculated by the principle of inclusion-exclusion using Ryser's formula: $perm(\boldsymbol{A}) = (-1)^n \sum_{S \subseteq [n]} (-1)^{|S|} \prod_{i=1}^{n} \sum_{j \in S} a_{ij}$. Algorithms implementing Ryser's formula on an explicit representation of an arbitrary matrix $\boldsymbol{A}$ (not necessarily sparse) must consider all 2^n subsets of $[n]$. As a consequence, such algo-

rithms have at least exponential complexity. Our experiments show that even the best known existing algorithm implementing Ryser's formula for arbitrary matrices [146], which iterates over the subsets of $[n]$ in Gray-code sequence, consistently times out after 1800 seconds on a state-of-the-art computing platform when computing the permanent of $n \times n$ matrices, with $n \geq 35$.

7.3 Related Work

Valiant showed that computing the permanent is $\#P$-complete [139]. Subsequently, researchers have considered restricted sub-classes of inputs in the quest for efficient algorithms for computing the permanent, both from theoretical and practical points of view. We highlight some of the important milestones achieved in this direction.

A seminal result is the Fisher-Temperly-Kastelyn algorithm [119, 120], which computes the number of perfect matchings in planar graphs in PTIME. This result was subsequently extended to many other graph classes (c.f. [158]). Following the work of Courcelle et al., a number of different width parameters have been proposed, culminating in the definition of ps-width [159], which is considered to be the most general notion of width [160]. Nevertheless, as with clique-width, it is not clear whether it lends itself to practically efficient algorithms. Bax and Franklin [161] gave a Las Vegas algorithm with better expected time complexity than Ryser's approach, but requiring $\mathcal{O}(2^{n/2})$ space.

For matrices with at most $C \cdot n$ zeros, Servedio and Wan [150] presented a $(2-\varepsilon)^n$-time and $\mathcal{O}(n)$ space algorithm where ε depends on C. Izumi and Wadayama [151] gave an algorithm that runs in time $\mathcal{O}^*(2^{(1-1/(\Delta \log \Delta))n})$, where Δ is the average degree of a vertex. On the practical side, in a series of papers, Liang, Bai and their co-authors [52, 162, 152] developed algorithms optimized for computing the permanent of the adjacency matrices of fullerenes, which are 3-regular graphs.

In recent years, practical techniques for propositional model counting (#SAT) have come of age. State-of-the-art exact model counters like `DSharp` [127] and `D4` [27]

also incorporate techniques from knowledge compilation. A straightforward reduction of the permanent to #SAT uses a Boolean variable x_{ij} for each 1 in row i and column j of the input matrix $\boldsymbol{A}$, and imposes Exact-One constraints on the variables in each row and column. This gives the formula $F_{perm(\boldsymbol{A})} = \bigwedge_{i\in[n]} ExactOne(\{x_{ij} : a_{ij} = 1\}) \wedge \bigwedge_{j\in[n]} ExactOne(\{x_{ij} : a_{ij} = 1\})$. Each solution to $F_{perm(\boldsymbol{A})}$ is a perfect matching in the underlying graph, and so the number of solutions is exactly the permanent of the matrix. A number of different encodings can be used for translating Exact-One constraints to Conjunctive Normal Form (see Section 7.5.1). We perform extensive comparisons of our tool with `D4` and `DSharp` with six such encodings.

7.4 Representing Ryser's Formula Symbolically

As noted in Sec. 7.2, an explicit implementation of Ryser's formula iterates over all 2^n subsets of columns and its complexity is in $\Theta(n \cdot 2^n)$. Therefore, any such implementation takes exponential time even in the best case. A natural question to ask is whether we can do better through a careful selection of subsets over which to iterate. This principle was used for the case of sparse matrices by Servedio and Wan [150]. Their idea was to avoid those subsets for which the row-sum represented by the innermost summation in Ryser's formula, is zero for at least one row, since those terms do not contribute to the outer sum in Ryser's formula. Unfortunately, this approach does not help for non-sparse matrices, as very few subsets of columns (if any) will yield a zero row-sum.

It is interesting to ask if we can exploit similarity of rows (instead of sparsity) to our advantage. Consider the ideal case of an $n \times n$ matrix with *identical rows*, where each row has k ($\leq n$) 1s. For any given subset of columns, the row-sum is clearly the same for all rows, and hence the product of all row-sums is simply the n^{th} power of the row-sum of one row. Furthermore, there are only $k + 1$ distinct values (0 through k) of the row-sum, depending on which subset of columns is selected. The number of r-sized column subsets that yield row-sum j is clearly $\binom{k}{j} \cdot \binom{n-k}{r-j}$, for

$0 \leq j \leq k$ and $j \leq r \leq n-k+j$. Thus, we can directly compute the permanent of the matrix via Ryser's formula as $perm(\boldsymbol{A}) = (-1)^n \sum_{j=0}^{k} \sum_{r=j}^{n-k+j} (-1)^r \binom{k}{j} \cdot \binom{n-k}{r-j} \cdot j^n$. This equation has a more compact representation than the explicit implementation of Ryser's formula, since the outer summation is over $(k+1).(n-k+1)$ terms instead of 2^n terms.

Drawing motivation from the above example, we propose using memoization to simplify the permanent computation of matrices with similar rows. Specifically, if we compute and store the row-sums for a subset $S_1 \subset [n]$ of columns, then we can potentially reuse this information when computing the row-sums for subsets $S_2 \supset S_1$. We expect storage requirements to be low when the rows are similar, as the partial sums over identical parts of the rows will have a compact representation, as shown above.

While we can attempt to hand-craft a concrete algorithm using this idea, it turns out that ADDs fit the bill perfectly. We introduce Boolean variables x_j for each column $1 \leq j \leq n$ in the matrix. We can represent the summand $(-1)^{|S|} \prod_{i=1}^{n} \sum_{j \in S} a_{ij}$ in Ryser's formula as a function $f_{Ryser} : 2^X \rightarrow \mathbb{R}$ where for a subset of columns $\tau \in 2^X$, we have $f_{Ryser}(\tau) = (-1)^{|\tau|} \prod_{i=1}^{n} \sum_{j \in \tau} a_{ij}$. The outer sum in Ryser's formula is then simply the Additive Quantification of f_{Ryser} over all variables in X. The permanent can thus be denoted by the following equation:

$$perm(\boldsymbol{A}) = (-1)^n \ . \ \exists x_1, x_2, \ldots x_n . (f_{Ryser}) \tag{7.1}$$

We can construct an ADD for f_{Ryser} incrementally as follows:

- **Step 1**: For each row r_i in the matrix, construct the Row-Sum ADD $f_{RS}^{r_i}$ such that $f_{RS}^{r_i}(\tau) = \sum_{j:a_{ij}=1} \mathbb{1}_\tau(x_j)$, where $\mathbb{1}_\tau(x_j)$ is the indicator function taking the value 1 if $x_j \in \tau$, and zero otherwise. This ADD can be constructed by using the sum operation on the variables x_j corresponding to the 1 entries in row r_i.

- **Step 2**: Construct the Row-Sum-Product ADD $f_{RSP} = \prod_{i=1}^{n} f_{RS}^{r_i}$ by applying the product operation on all the Row-Sum ADDs

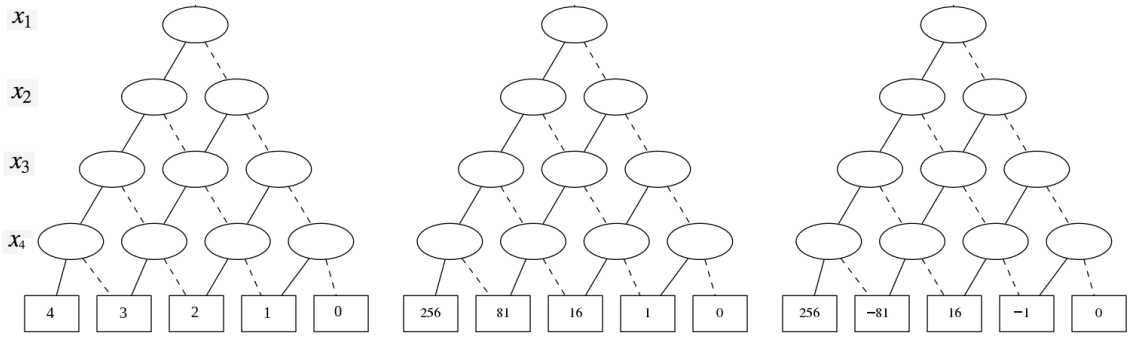


Figure 7.1 : (a) f_{RS}, (b) f_{RSP} and (c) f_{Ryser} for a 4×4 matrix of all 1s

- **Step 3**: Construct the Parity ADD $f_{PAR} = ITE(\bigoplus_{j=1}^{n} x_j, -1, +1)$, where $\bigoplus$ represents exclusive-or. This ADD represents the $(-1)^{|S|}$ term in Ryser's formula.

- **Step 4**: Construct $f_{Ryser} = f_{RSP}.f_{PAR}$ using the product operation.

Finally, we can additively quantify out all variables in f_{Ryser} and multiply the result by $(-1)^n$ to get the permanent, as given by Equation 7.1.

The size of the ADD f_{RSP} will be the smallest when the ADDs $f_{RS}^{r_i}$ are exactly the same for all rows r_i, i.e. when all rows of the matrix are identical. In this case, the ADDs $f_{RS}^{r_i}$ and f_{RSP} will be isomorphic; the values at the leaves of f_{RSP} will simply be the n^{th} power of the values at the corresponding leaves of $f_{RS}^{r_i}$. An example illustrating this for a 4×4 matrix of all 1s is shown in Fig. 7.1. Each level of the ADDs in this figure corresponds to a variable (shown on the left) for a column of the matrix. A solid edge represents the 'true' branch while a dotted edge represents the 'false' branch. Observe that sharing of isomorphic subgraphs allows each of these ADDs to have 10 internal nodes and 5 leaves, as opposed to 15 internal nodes and 16 leaves that would be needed for a complete binary tree based representation.

The ADD representation is thus expected to be compact when the rows are "similar". Dense matrices can be thought of as a special case: starting with a matrix of

all 1s (which clearly has all rows identical), we change a few 1s to 0s. The same idea can be applied to sparse matrices as well: starting with a matrix of all 0s (once again, identical rows), we change a few 0s to 1s. The case of very sparse matrices is not interesting, however, as the permanent (or equivalently, count of perfect matchings in the corresponding bipartite graph) is small and can be computed by naive enumeration. Interestingly, our experiments show that as we reduce the sparsity of the input matrix, constructing f_{RSP} and f_{Ryser} in a monolithic fashion as discussed above fails to scale, since the sizes of ADDs increase very sharply. Therefore we need additional machinery.

First, we rewrite Equation 7.1 in terms of the intermediate ADDs as:

$$perm(\boldsymbol{A}) = (-1)^n \; . \; \exists x_1, x_2, \ldots x_n . \left(f_{PAR} \cdot \prod_{i=1}^{n} f_{RS}^{r_i} \right) \tag{7.2}$$

We then employ the principle of early abstraction to compute f_{Ryser} incrementally. Note that early abstraction has been used successfully in the past in the context of SAT solving [163], and recently for weighted model counting using ADDs in a technique called ADDMC [49]. The formal statement of the principle of early abstraction is given in the following theorem.

Theorem 7.1. [49] *Let X and Y be sets of variables and $f : 2^X \rightarrow \mathbb{R}$, $g : 2^Y \rightarrow \mathbb{R}$. For all $x \in X \setminus Y$, we have $\exists_x (f \cdot g) = (\exists_x (f)) \cdot g$*

Since the product operator is associative and additive quantification is commutative, we can rearrange the terms of Equation 7.2 in order to apply early abstraction. This idea is implemented in Algorithm RysersADD, which is motivated by the weighted model counting algorithm in [49].

Algorithm RysersADD takes as input a 0-1 matrix $\boldsymbol{A}$, a diagram variable order ρ and a cluster rank-order η. η is an ordering of variables which is used to heuristically partition rows of $\boldsymbol{A}$ into clusters using a function clusterRank, where all rows in a cluster get the same rank. Intuitively, rows that are almost identical are placed in the same cluster, while those that differ significantly are placed in different clusters.

Algorithm 12 RysersADD($\boldsymbol{A}, \rho, \eta$)

1: $m \leftarrow max_{x \in X}\ \eta(x)$;
2: **for** $i = m, m-1, \ldots, 1$ **do**
3: $\quad \kappa_i \leftarrow \{f^r_{RS} : r$ is a row in $\boldsymbol{A}$ and clusterRank$(r, \eta) = i\}$;
4: $f_{Ryser} \leftarrow f_{PAR}$; ▷ f_{PAR} and each f^r_{RS} are constructed using the diagram variable order ρ
5: **for** $i = 1, 2, \ldots, m$ **do**
6: $\quad$ **if** $\kappa_i \neq \varnothing$ **then**
7: $\quad\quad$ **for** $g \in \kappa_i$ **do**
8: $\quad\quad\quad f_{Ryser} \leftarrow f_{Ryser} \cdot g$;
9: $\quad\quad$ **for** $x \in Vars(f_{Ryser})$ **do**
10: $\quad\quad\quad$ **if** $x \notin (Vars(\kappa_{i+1}) \cup \ldots \cup Vars(\kappa_m))$ **then**
11: $\quad\quad\quad\quad f_{Ryser} \leftarrow \exists_x(f_{Ryser})$
12: **return** $(-1)^n \times f_{Ryser}(\varnothing)$

Furthermore, the clusters are ordered such that there are non-zero columns in cluster i that are absent in the set of non-zero columns in clusters with rank $> i$. As we will soon see, this facilitates keeping the sizes of ADDs under control by applying early abstraction.

Algorithm RysersADD proceeds by first partitioning the Row-Sum ADDs of the rows $\boldsymbol{A}$ into clusters according to their cluster rank in line 3. Each Row-Sum ADD is constructed according to the diagram variable order ρ. The ADD f_{Ryser} is constructed incrementally, starting with the Parity ADD in line 4, and multiplying the Row-Sum ADDs in each cluster κ_i in the loop at line 7. However, unlike the monolithic approach, early abstraction is carried out within the loop at line 9. Finally, when the execution reaches line 12, all variables representing columns of the input matrix have been abstracted out. Therefore, f_{Ryser} is an ADD with a single leaf node that contains the (possibly negative) value of the permanent. Following Equation 7.2, the algorithm returns the product of $(-1)^n$ and $f_{Rsyer}(\varnothing)$.

The choice of the function clusterRank and the cluster rank-order η significantly affect the performance of the algorithm. A number of heuristics for determining clusterRank and η have been proposed in literature, such as Bucket Elimination [164], and Bouquet's Method [165] for cluster ranking, and MCS [166], LexP [167] and LexM [167] for variable ordering. Further details and a rigorous comparison of these heuristics are presented in [49]. Note that if we assign the same cluster rank to all rows of the input matrix, Algorithm RysersADD reduces to one that constructs all ADDs monolithically, and does not benefit from early abstraction.

7.4.1 Implementation Details

We implemented Algorithm 12 using the library Sylvan [156] since unlike CUDD [155], Sylvan supports arbitrary precision arithmetic – an essential feature to avoid overflows when the permanent has a large value. Sylvan supports parallelization of ADD operations in a multi-core environment. In order to leverage this capability, we cre-

ated a parallel version of RysersADD that differs from the sequential version only in that it uses the parallel implementation of ADD operations natively provided by Sylvan. Note that this doesn't require any change to Algorithm RysersADD, except in the call to Sylvan functions. While other non-ADD-based approaches to computing the permanent can be parallelized as well, we emphasize that it is a non-trivial task in general, unlike using Sylvan. We refer to our sequential and parallel implementations for permanent computation as RysersADD and RysersADD-P respectively, in the remainder of the discussion. We implemented our algorithm in C++, compiled under GCC v6.4 with the O3 flag. We measured the wall-times for both algorithms. Sylvan also supports arbitrary precision floating point computation, which makes it easy to extend RysersADD for computing permanent of real-valued matrices. However, we leave a detailed investigation of this for future work.

7.5 Experimental Methodology

The objective of our empirical study was to evaluate RysersADD and RysersADD-P on randomly generated instances (as done in [162]) and publicly available structured instances (as done in [52, 152]) of 0-1 matrices.

7.5.1 Algorithm Suite

As noted in Section 9.3, a number of different algorithms have been reported in the literature for computing the permanent of sparse matrices. Given resource constraints, it is infeasible to include all of these in our experimental comparisons. This is further complicated by the fact that many of these algorithms appear not to have been implemented (eg: [150, 151]), or the code has not been made publicly accessible (eg: [52, 152]). A fair comparison would require careful consideration of several parameters like usage of libraries, language of implementation, suitability of hardware etc. We had to arrive at an informed choice of algorithms, which we list below along with our rationale:

- RysersADD and RysersADD-P: For the dense and similar rows cases, we use the monolithic approach as it is sufficient to demonstrate the scalability of our ADD-based approach. For sparse instances, we employ Bouquet's Method (List) [165] clustering heuristic along with MCS cluster rank-order [166] and we keep the diagram variable order the same as the indices of columns in the input matrix (see [49] for details about the heuristics). We arrived at these choices through preliminary experiments. We leave a detailed comparison of all combinations for future work.

- *Explicit Ryser's Algorithm*: We implemented Nijenhuis and Wilf's version [146] of Ryser's formula using Algorithm H from [168] for generating the Gray code sequence. Our implementation, running on a state-of-the-art computing platform (see Section 7.5.2), is able to compute the permanent of all matrices with $n \leq 25$ in under 5 seconds. For $n = 30$, the time shoots up to approximately 460 seconds and for $n \geq 34$, the time taken exceeds 1800 seconds (time out for our experiments). Since the performance of explicit Ryser's algorithm depends only on the size of the matrix, and is unaffected by its structure, sparsity or row-similarity, this represents a complete characterization of the performance of the explicit Ryser's algorithm. Hence, we do not include it in our plots.

- *Propositional Model Counters*: Model counters that employ techniques from SAT-solving as well as knowledge compilation, have been shown to scale extremely well on large CNF formulas from diverse domains. Years of careful engineering have resulted in counters that can often outperform domain-specific approaches. We used two state-of-the-art exact model counters, viz. D4 [27] and `DSharp` [127], for our experiments. We experimented with 6 different encodings for At-Most-One constraints: (1) Pairwise [23], (2) Bitwise [23], (3) Sequential Counter [169], (4) Ladder [170, 171], (5) Modulo Totalizer [172] and (6) Iterative Totalizer [173]. We also experimented with ADDMC, an ADD-based model

Table 7.1 : Parameters used for generating random matrices

Experiment	**Matrix Size n**	C_f**, where** $C_f \cdot n$ **matrix entries flipped**	**Starting Matrix Row Density** ρ	**#Instances**	**Total Benchmarks**
Dense	30, 40, 50, 60, 70	1, 1.1, 1.2, 1.3, 1.4	1	20	500
Sparse	30, 40, 50, 60, 70	3.9, 4.3, 4.7, 5.1, 5.5	0	20	500
Similar	40, 50, 60, 70, 80	1, 1.1, 1.2, 1.3, 1.4	0.7, 0.8, 0.9	15	1125

counter [49]. However, it failed to scale beyond matrices of size 25; ergo we do not include it in our study.

We were unable to include the parallel #SAT counter countAtom [174] in our experiments, owing to difficulties in setting it up on our compute set-up. However, we could run countAtom on a slightly different set-up with 8 cores instead of 12, and 16GB memory instead of 48 on a few sampled dense and similar-row matrix instances. Our experiments showed that countAtom timed out on all these cases. We leave a more thorough and scientific comparison with countAtom for future work.

7.5.2 Experimental Setup

Each experiment (sequential or parallel) had exclusive access to a Westemere node with 12 processor cores running at 2.83 GHz with 48 GB of RAM. We capped memory usage at 42 GB for all tools. We implemented explicit Ryser's algorithm in C++, compiled with GCC v6.4 with O3 flag. The RysersADD and RysersADD-P algorithms were implemented as in Section 7.4.1. RysersADD-P had access to all 12 cores for parallel computation. We used the python library PySAT [175] for encoding matrices into CNF. We set the timeout to 1800 seconds for all our experiments. For purposes of reporting, we treat a memory out as equivalent to a time out.

7.5.3 Benchmarks

The parameters used for generating random instances are summarized in Table 7.1. We do not include matrices with $n < 30$ since the explicit Ryser's algorithm suffices (and often performs the best) for such matrices. The upper bound for n was chosen

such that the algorithms in our suite either timed out or came close to timing out. For each combination of parameters, random matrix instances were sampled as follows:

1. We started with an $n \times n$ matrix, where the first row had $\rho \cdot n$ 1s at randomly chosen column positions, and all other rows were copies of the first row.

2. $C_f \cdot n$ randomly chosen entries in the starting matrix are flipped i.e. 0 flipped to 1 and vice versa.

For the dense case, we start with a matrix of all 1s while for the sparse case, we start with a matrix of all 0s, and used intermediate row density values for the similar-rows case. We chose higher values for C_f in the sparse case because for low values, the bipartite graph corresponding to the generated matrix had very few perfect matchings (if any), and these could be simply counted by enumeration. We generated a total of 2125 benchmarks covering a broad range of parameters. For all generated instances, we ensured that there was at least one perfect matching, since the case with zero perfect matchings can be easily solved in polynomial time by algorithms like Hopcroft-Karp [176]. In order to avoid spending inordinately large time on failed experiments, if an algorithm timed out on all generated random instances of a particular size, we also report a time out for that algorithm on all larger instances of that class of matrices. We also double-check this by conducting experiments with the same algorithm on a few randomly chosen larger instances.

The SuiteSparse Matrix Collection [177] is a well known repository of structured sparse matrices that arise from practical applications. We found 26 graphs in this suite with vertex count between 30 and 100, of which 18 had at least one perfect matching. Note that these graphs are not necessarily bipartite; however, their adjacency matrices can be used as benchmarks for computing the permanent. A similar approach was employed in [178] as well.

Fullerenes are carbon molecules whose adjacency matrices have been used extensively by Liang et al. [52, 178, 152] for comparing tools for the permanent. We

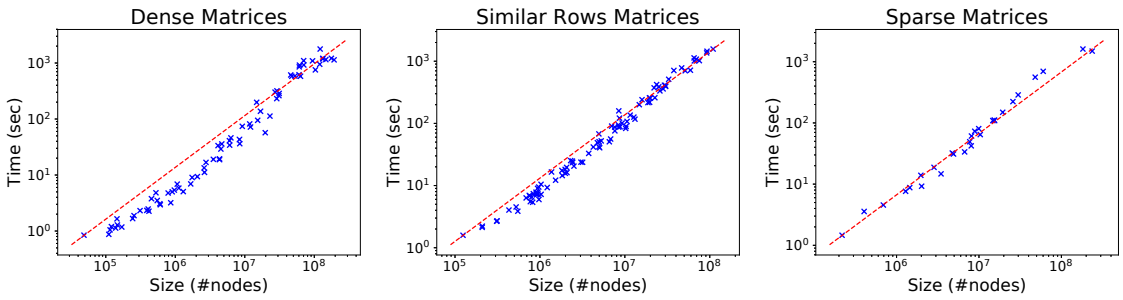


Figure 7.2 : Comparison of ADD Size vs. Time taken for a subset of random benchmarks

were able to find the adjacency matrices of C_{60} and C_{100}, and have used these in our experiments.

7.6 Results

We first study the variation of running time of RysersADD with the size of ADDs involved. Then we compare the running times of various algorithms on sparse, dense and similar-row matrices, as well as on instances from SuiteSparse Matrix Collection and on adjacency matrices of fullerenes C_{60} and C_{100}. The total computational effort of our experiments exceeds 2500 hours of wall clock time on dedicated compute nodes.

7.6.1 ADD size vs time taken by RysersADD

In order to validate the hypothesis that the size of the ADD representation is a crucial determining factor of the performance of RysersADD, we present 3 scatter-plots (Fig. 7.2) for a subset of 100 instances, of each of the dense, sparse and similar-rows cases. In each case, the 100 instances cover the entire range of C_f and n used in Table 7.1, and we plot times only for instances that didn't time out. The plots show that there is very strong correlation between the number of nodes in the ADDs and the time taken for computing the permanent, supporting our hypothesis.

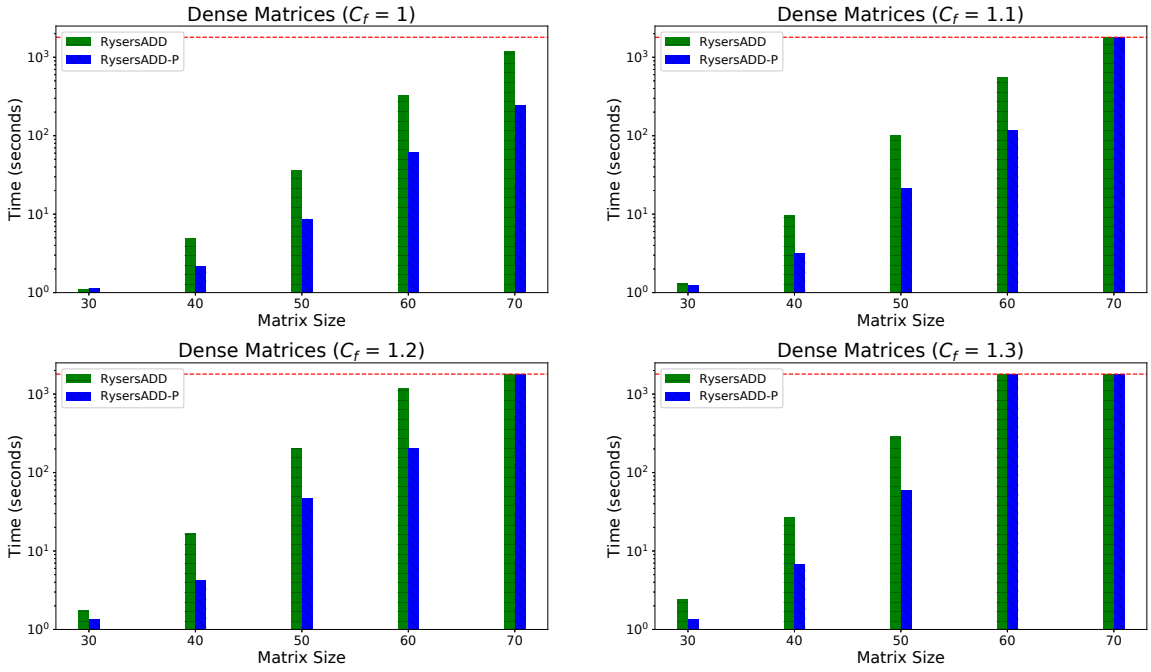


Figure 7.3 : Performance on Dense Matrices. D4, DSharp (not shown) timeout on all instances

7.6.2 Performance on dense matrices

We plot the median running time of RysersADD and RysersADD-P against the matrix size n for dense matrices with $C_f \in \{1, 1.1, 1.2, 1.3\}$ in Fig. 7.3. We only show the running times of RysersADD and RysersADD-P, since D4 and DSharp were unable to solve any instance of size 30 for all 6 encodings. We observe that the running time of both the ADD-based algorithms increases with C_f. This trend continues for $C_f = 1.4$, which we omit for lack of space. RysersADD-P is noticeably faster than RysersADD, indicating that the native parallelism provided by Sylvan is indeed effective.

7.6.3 Performance on sparse matrices

Fig. 7.4 depicts the median running times of the algorithms for sparse matrices with $C_f \in \{3.9, 4.3, 4.7, 5.1\}$. We plot the running time of the ADD-based approaches with early abstraction (see Sec. 7.5.1). Monolithic variants (not shown) time out on all

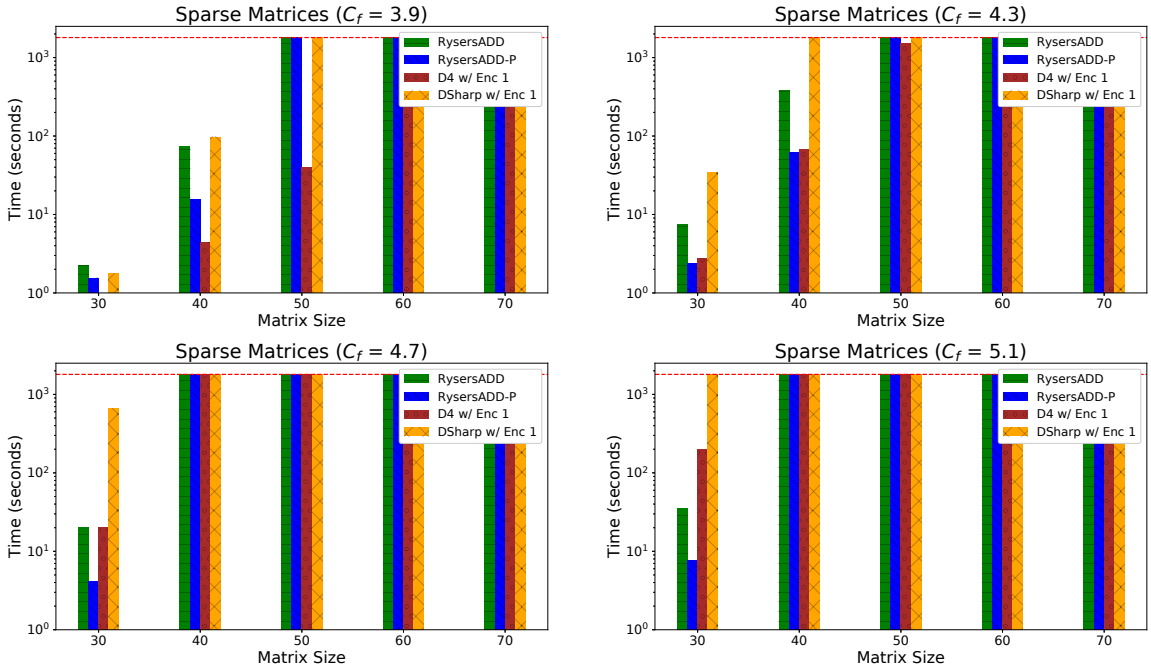


Figure 7.4 : Performance on Sparse Matrices

instances with $n \geq 40$. For D4 and `DSharp`, we plot the running times only for Pairwise encoding of At-Most-One constraints, since our preliminary experiments showed that it substantially outperformed other encodings. We see that D4 is the fastest when sparsity is high i.e. for $C_f \leq 4.3$, but for $C_f \geq 4.7$ the ADD-based methods are the best performers. `DSharp` is outperformed by the remaining 3 algorithms in general.

7.6.4 Performance on similar-row matrices

Fig. 7.5 shows plots of the median running time on similar-row matrices with $C_f = \{1, 1.1, 1.2, 1.3\}$. We only present the case when $\rho = 0.8$, since the plots are similar when $\rho \in \{0.7, 0.9\}$. As in the case of dense matrices, D4 and `DSharp` were unable to solve any instance of size 40, and hence we only show plots for RysersADD and RysersADD-P. The performance of both tools is markedly better than in the case of dense matrices, and they scale to matrices of size 80 within the 1800 second timeout.

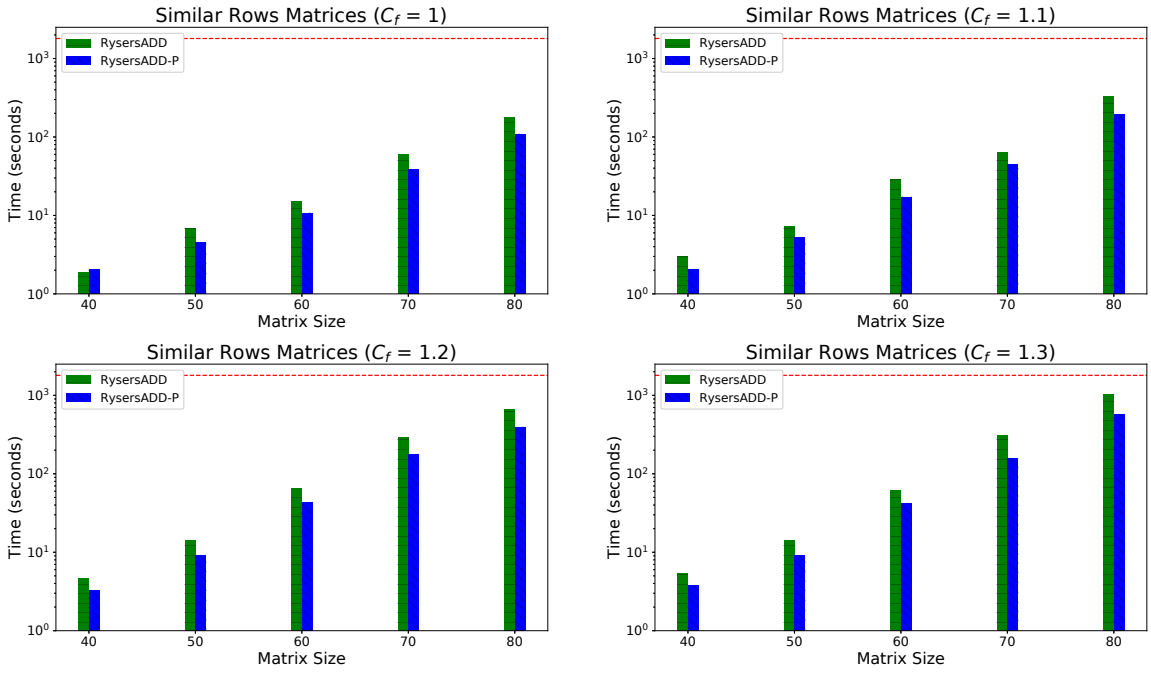


Figure 7.5 : Performance on similar-rows matrices. D4, `DSharp` (not shown) timeout on all instances.

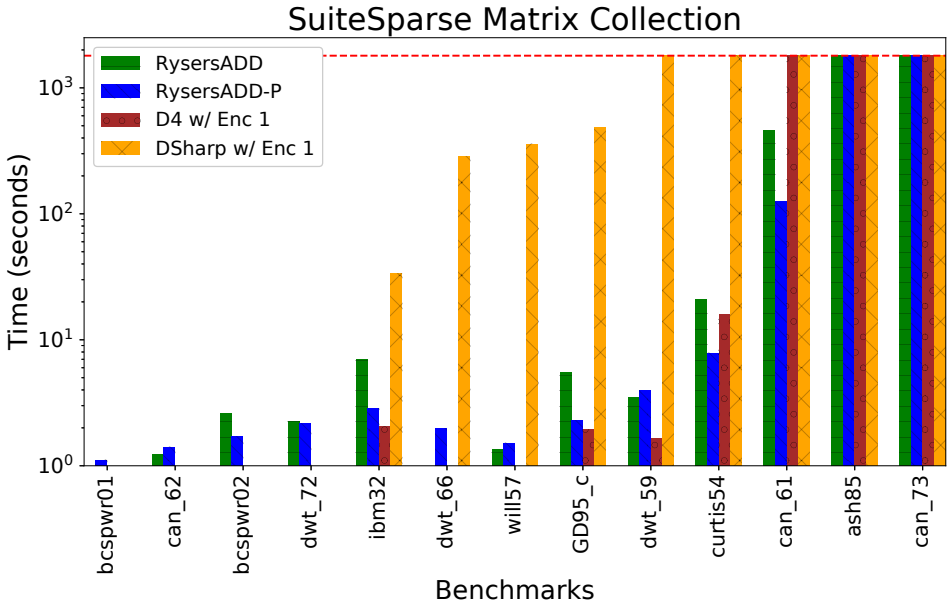


Figure 7.6

Table 7.2 : Running Times on the fullerene C_{60}. EA: Early Abstraction Mono: Monolithic

Tool	D4						DSharp						RysersADD		RysersADD-P	
Encoding / Mode	1	2	3	4	5	6	1	2	3	4	5	6	EA	Mono	EA	Mono
Time (sec)	94.8	150.5	150.6	136	158	156	TimeOut						96.4	TimeOut	57.1	TimeOut

7.6.5 Performance on SuiteSparse Matrix Collection

We report the performance of algorithms RysersADD, RysersADD-P, D4 and DSharp on 13 representative graphs from the SuiteSparse Matrix Collection in Fig. 7.6. Except for the first 4 instances, which can be solved in under 5 seconds by all algorithms, we find that D4 is the fastest in general, while the ADD-based algorithms outperform DSharp. Notably, on the instance "can_61", both D4 and DSharp time out while RysersADD and RysersADD-P solve it comfortably within the alloted time. We note that the instance "can_61" has roughly $9n$ 1s, while D4 is the best performer on instances where the count of 1s in the matrix lies between $4n$ and $6n$.

7.6.6 Performance on fullerene adjacency matrices

We compared the performance of the algorithms on the adjacency matrices of the fullerenes C_{60} and C_{100}. All the algorithms timed out on C_{100}. The results for C_{60} are shown in Table 7.2. The columns under D4 and DSharp correspond to 6 different encodings of At-Most-One constraints (see Sec. 7.5.1). It can be seen that RysersADD-P performs the best on this class of matrices, followed by D4. The utility of early abstraction is clearly evident, as the monolithic approach times out in both cases.

Discussion: Our experiments show the effectiveness of the symbolic approach on dense and similar-rows matrices, where neither D4 nor DSharp are able to solve even a single instance. Even for sparse matrices, we see that decreasing sparsity has lesser

effect on the performance of ADD-based approaches as compared to D4. This trend is confirmed by "can_61" in the SuiteSparse Matrix Collection as well, where despite the density of 1s being $9n$, RysersADD and RysersADD-P finish well within timeout, unlike D4. In the case of fullerenes, we note that the algorithm in [52] solved C_{60} in 355 seconds while the one in [152] took 5 seconds, which are in the vicinity of the times reported in Table 7.2. While this is not an apples-to-apples comparison owing to differences in the computing platform, it indicates that the performance of general-purpose algorithms like RysersADD and D4 can be comparable to that of application-specific algorithms.

7.7 Chapter Summary

In this work we introduced a symbolic algorithm called RysersADD for permanent computation based on augmenting Ryser's formula with Algebraic Decision Diagrams. We demonstrated, through rigorous experimental evaluation, the scalability of RysersADD on both dense and similar-rows matrices, where existing approaches fail. Coupled with the technique of early abstraction [49], RysersADD performs reasonably well even on sparse matrices as compared to dedicated approaches. Our key contribution in the context of Constrained Counting is to comprehensively demonstrate the versatility of ADDs and factored representations for solving combinatorial problems, vis-a-vis the rigidity of the SAT-based approach.

Chapter 8

Sampling Traces of a Transition System

8.1 Introduction

Simulation-based functional verification is a crucial yet time-consuming step in modern electronic design automation flows [55]. In this step, a design is simulated with a large number of input stimuli, and signals are monitored to determine if coverage goals and/or functional requirements are met. For complex designs, each input stimulus typically spans a large number of clock cycles. Since exhaustive simulation is impractical for real designs, using "good quality" stimuli that result in adequate coverage of the system's runs in targeted corners is extremely important [179]. *Constrained random verification, or CRV,* [180, 181, 182, 20] offers a practical solution to this problem. In CRV, the user provides constraints to ensure that the generated stimuli are valid and also to steer the system towards bug-prone corners. To ensure *diversity*, CRV allows randomization in the choice of stimuli satisfying a set of constraints. This can be very useful when the exact inputs needed to meet coverage goals or to test functional requirements are not known [182, 183]. In such cases, it is best to generate stimuli such that the resulting runs are uniformly distributed in the targeted corners of its behavior space. Unfortunately, state-of-the-art CRV tools [56, 57, 58, 184, 185] do not permit such uniform random sampling of input stimuli. Instead, they allow inputs to be assigned random values from a constrained set at specific simulation steps. This of course lends diversity to the generated stimuli. However, it gives no guarantees on the distribution of the resulting system runs. In this chapter, we present our work on remedying this problem. Specifically, we present a technique for generating input stimuli that *guarantees* uniform (or user-specified bias in) distribution of the

resulting system runs. Note that this is significantly harder than generating any one run satisfying a set of constraints.

We represent a run of the system by the sequence of states through which it transitions in response to a (multi-cycle) input stimulus. Important coverage metrics (viz. transition coverage, state sequence coverage, etc. [186]) are usually boosted by choosing stimuli that run the system through diverse state sequences. Similarly, functional requirements (viz. assertions in SystemVerilog [185], PSL [187], Specman E [57], UVM [56] and other formalisms [188]) are often stated in terms of temporal relations between states in a run of the system. Enhancing the diversity of state sequences in runs therefore improves the chances of detecting violations, if any, of functional requirements. Consequently, generating input stimuli such that the resulting sequences of states, or *traces*, are uniformly distributed among all traces consistent with the given constraints is an important problem. Significantly, given a sequence of states and the next-state transition function, the input stimuli needed to induce the required state transitions at each clock cycle can be easily obtained by independent SAT/SMT calls for each cycle. Hence, our focus in the remainder of the chapter is the core problem of sampling a system's traces uniformly at random from the set of all traces (of a given length) that satisfy user-specified constraints.

To see why state-of-the-art CRV techniques [56, 57, 58, 184, 185] often fail to generate stimuli that produce a uniform distribution of traces, consider the sequential circuit with two latches (x_0 and x_1) and one primary input, shown in Fig. 8.1a. The state transition diagram of the circuit is shown in Fig. 8.1b. Suppose we wish to uniformly sample traces that start from the initial state $s_0 = (x_1 = 0, x_0 = 0)$ and have 4 consecutive state transitions. From Fig. 8.1b, there are 7 such traces: $\omega_1 = s_0s_1s_1s_1s_1$, $\omega_2 = s_0s_1s_1s_1s_2$, $\omega_3 = s_0s_1s_1s_2s_2$, $\omega_4 = s_0s_1s_2s_2s_2$, $\omega_5 = s_0s_3s_1s_1s_1$, $\omega_6 = s_0s_3s_1s_1s_2$ and $\omega_7 = s_0s_3s_1s_2s_2$. Hence, each of these traces must be sampled with probability 1/7. Unfortunately, the state transition diagram of a sequential circuit can be exponentially large (in number of latches), and is often infeasible to

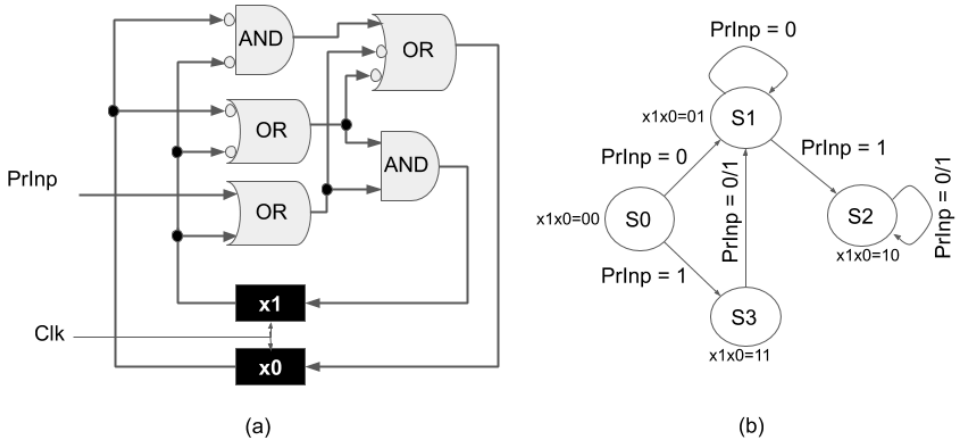


Figure 8.1 : (a) Sequential circuit, (b) State transition diagram

construct explicitly. Hence we must sample traces without generating the state transition diagram explicitly. The primary facility in existing CRV techniques to attempt such sampling is to choose values of designated inputs randomly at specific steps of the simulation. In our example, without any information about the state transition diagram, the primary input of the circuit in Fig. 8.1a must be assigned a value 0 (or 1) with probability 1/2 independently in each of the 4 steps of simulation. This produces the traces ω_1 and ω_2 with probability 1/16 each, ω_3, ω_5 and ω_6 with probability 1/8 each, and ω_4 and ω_7 with probability 1/4 each. Notice that this is far from the desired uniform distribution. In fact, it can be shown that for every choice of bias for sampling 0/1 values of the primary input at each state, we get a non-uniform distribution of ω_1 through ω_7.

The trace-sampling problem can be shown to be at least as hard as uniformly sampling satisfying assignments of Boolean formulas. The complexity of the latter problem has been extensively studied [12, 14, 15], and no efficient algorithms are known. Therefore, efficient algorithms for sampling traces are unlikely to exist. Nevertheless, a trace sampling technique that works efficiently in practice for many problem instances is likely to be useful even beyond CRV, viz. in test generation

using Bounded Model Checking [189].

The primary contributions of the work presented in this chapter are as follows:

1. A novel algorithm for sampling fixed-length traces of a transition system using Algebraic Decision Diagrams (ADDs) [132], with provable guarantees of uniformity (or user-provided bias). The following are distinctive features of our algorithm.

   (a) It uses iterative squaring, thereby requiring only $\log_2 N$ ADDs to be pre-computed when sampling traces of N consecutive state transitions. This allows our algorithm to scale to traces of a few hundred transitions in our experiments.

   (b) It is easily adapted when the trace length is not a power of 2, and also when implementing weighted sampling of traces with multiplicative weights.

   (c) It pre-compiles the i-step transition relation for $\log_2 N$ different values of i to ADDs. This allows it to quickly generate multiple trace samples once the ADDs are constructed. Thus the cost of ADD construction gets amortized over the number of samples, which is beneficial in CRV settings.

2. A comparative study of an implementation of our algorithm (called TraceSampler) with alternative approaches based on (almost)-uniform sampling of propositional models, that provide similar uniformity guarantees. Our experiments demonstrate that our approach offers significant speedup and is the fastest over 90% of the benchmarks.

8.2 Preliminaries

8.2.1 Transition Systems and Traces

A synchronous sequential circuit with n latches implicitly represents a transition system with 2^n states. Hence, synchronous sequential circuits serve as succinct represen-

tations of finite-state transition systems. We use "sequential circuits" and "transition systems" interchangeably in the remainder of the paper to refer to such systems.

Formally, a transition system with k Boolean state variables $X = \{x_0, \ldots, x_{k-1}\}$ and m primary inputs is a 5-tuple (S, Σ, t, I, F), where $S = \{0,1\}^k$ is the set of states, $\Sigma = \{0,1\}^m$ is the input alphabet, $I \subseteq S$ is the set of initial states, $F \subseteq S$ is the set of target (or final) states, and $t : S \times \Sigma \to S$ is the state transition function such that $t(s,a) = s'$ iff there is a transition from state $s \in S$ on input $a \in \Sigma$ to state $s' \in S$. We view each state in $S = \{0,1\}^k$ as a valuation of $(x_{k-1} \ldots x_0)$. For notational convenience, we use the decimal representation of the valuation of $(x_{k-1} \ldots x_0)$ as a subscript to refer to individual states. For instance, s_0 and s_{2^k-1} are the states with all-zero and all-one assignments to $x_{k-1} \ldots x_0$ respectively. We refer to multiple versions of the state variables X as $X^0, X^1, \ldots$

Given a transition system, a *trace* ω of length N (> 0) is a sequence of $N+1$ states such that $\omega[0] \in I, \omega[N] \in F$ and $\forall i \in \{0, \ldots, N-1\}\ \exists a \in \Sigma\ s.t.\ t(\omega[i], a) = \omega[i+1]$, where $\omega[i]$ represents the i^{th} state in the trace. We denote the set of all traces of length N by Ω_N. Given a trace $\omega \in \Omega_N$, finding an input sequence $\alpha \in \Sigma^N$ such that the i^{th} element, viz. $\alpha[i]$, satisfies $\omega[i+1] = t(\omega[i], \alpha[i])$ for all $i \in \{0, \ldots N-1\}$, requires N independent SAT solver calls. With state-of-the-art SAT solvers [37], this is unlikely to be a concern with the number of primary inputs m ranging upto tens of thousands. Therefore, finding a sequence of inputs that induces a trace is relatively straightforward, and we will not dwell on this any further. Our goal, instead, will be to sample a trace $\omega \in \Omega_N$ uniformly at random. Formally, if the random variable Y corresponds to a random choice of traces, we'd like to have $\forall \omega \in \Omega_N \quad \Pr[Y = \omega] = \frac{1}{|\Omega_N|}$. Given a weight function $w : \Omega_N \to \mathbb{R}^+$, the related problem of weighted trace sampling requires us to sample such that $\forall \omega \in \Omega_N \quad \Pr[Y = \omega] = \frac{w(\omega)}{\sum_{\omega \in \Omega_N} w(\omega)}$.

Since we are concerned only with sequences of states, we will henceforth assume that transitions of the system are represented by a transition relation $\widehat{t} : S \times S \to \{0,1\}$, where $\widehat{t}(s,s') \Leftrightarrow \exists a \in \Sigma\ s.t.\ t(s,a) = s'$. For notational convenience, we

Symbol	Meaning
X	Set of Boolean variables $x_1, x_2, \ldots, x_k$
S	Set of states $s_1, s_2, \ldots, s_{2^k-1}$
Ω_N	Set of all traces 'ω', of length N
t	Transition function
w	Weight function
Π_v	Set of all paths 'π' in a DD starting at node v

Table 8.1 : Summary of notation

abuse notation and use $t(s, s')$ for $\widehat{t}(s, s')$, when there is no confusion.

A multiplicative weight function assigns a weight to each state transition, and defines the weight of a trace as the product of weights of the transitions in the trace. Formally, let $\widehat{w} : S \times S \to \mathbb{R}^{\geq 0}$ be a weight function for state transitions, where $\widehat{w}(s_i, s_j) > 0$ if $t(s_i, s_j)$ holds, and $\widehat{w}(s_i, s_j) = 0$ otherwise. Then, the multiplicative weight of a trace $\omega \in \Omega_N$ is defined as $w(\omega) = \prod_{i=0}^{N-1} \widehat{w}(\omega[i], \omega[i+1])$. The unweighted uniform sampling problem can be seen to be the special case where $\widehat{w}(s_i, s_j) = 1$ whenever $t(s_i, s_j)$ holds.

We denote the set of leaves of a decision diagram (DD) t by $\mathit{leaves}(t)$, and the root of the DD by $\mathit{root}(t)$. We denote the vertices of the DAG by v, set of parents of v in the DAG by $P(v)$, and value of a leaf v by $\mathit{val}(v)$. A path from a node v to $\mathit{root}(t)$ in a DD t, denoted as $\pi = v_0 v_1 \ldots v_h$, is defined to be a sequence of nodes such that $v_0 = v$, $v_h = \mathit{root}(t)$ and $\forall i \ \ v_{i+1} \in P(v_i)$. We use Π_v denote the set of all paths to the root starting from some node v in the DD. For a set V of nodes, we define $\Pi_V = \cup_{v \in V} \Pi_v$. The special set Π represents all paths from all leaves to the root of a DD. Our notational setup is briefly summarized in Tab. 8.1.

8.3 Related Work

We did not find any earlier work on sampling traces of sequential circuits with provable uniformity guarantees. As mentioned earlier, constrained random verification tools [184, 185, 57, 56, 58] permit values of selected inputs to be chosen uniformly (or with specified bias) from a constrained set at some steps of simulation. Nevertheless, as shown in Section 8.1, this does not necessarily yield uniform traces.

Arenas et al. [190] gave a fully-polynomial randomized approximation scheme for approximately counting words of a given length accepted by a Non-deterministic Finite Automaton (NFA). Using Jerrum et al's reduction from approximate counting to sampling [14], this yields an algorithm for sampling words of an NFA. Apart from the obvious difference of sampling words vs. sampling traces, Arenas et al's technique requires the state-transition diagram of the NFA to be represented explicitly, while our focus is on transition systems that implicitly encode large state-transition diagrams.

Given a transition system, sampling traces of length N can be achieved by sampling satisfying assignments of the propositional formula obtained by "unrolling" the transition relation N times. Technique for sampling models of propositional formulas, viz. [29, 30, 31] for uniform sampling and [16, 191, 192] for almost uniform sampling, can therefore be used to sample traces. The primary bottleneck in this approach is the linear growth of propositional variables with the trace length and count of Boolean state variables. We compare our tool with state-of-the-art samplers WAPS [31] and UniGen2 [192], and show that our approach performs significantly better.

8.4 Algorithms

For clarity, we assume that the length of traces, i.e. N, is a power of 2; the case when N is not a power of 2 is discussed later. A naive approach would be to use a single BDD to represent all traces of length N, by appropriately unrolling the transition system, and then sample traces from the BDD. Such monolithic representations, however, are known to blow up [49]. Therefore, we use $\log_2 N$ ADDs, where the i^{th}

ADD ($1 \leq i \leq \log_2 N$) represents the count of 2^i-length paths between different states of the transition system. The i^{th} ADD is constructed from the $(i-1)^{th}$ ADD by a technique similar to iterative squaring [193, 194]. A trace is sampled by recursively sampling states from each ADD according to the weights on the leaves.

The detailed algorithm for constructing ADDs is presented in Algorithm 13. We assume that the transition relation is defined over 2 copies, viz. X^0 and X^1, of the state variables, and that an additional $\log_2 N$ copies, viz. $X^2 \ldots X^{(\log_2 N)+1}$, are also available. In each step of the for loop on line 2, the $(i-1)^{th}$ ADD is squared to obtain the i^{th} ADD after additively abstracting out X^i in line 4. Each ADD $t_i(X^0, X^i, X^{i+1})$ represents the count of 2^i-length traces from X^0 to X^{i+1} that pass through X^i at the half-way point. Note that $g(X^{i-1}, X^i)$ and $g(X^i, X^{i+1})$ in line 3 are the same ADD, but with variables renamed. Finally, in line 5, we take the product of the $\log_2 N^{th}$ ADD with the characteristic functions for the initial and final states, represented as ADDs. Although Algorithm 13 correctly computes all ADDs, in practice, we found that it often scaled poorly for values of N beyond a few 10s. On closer scrutiny, we found that this was because the ADD t_0 (and other ADDs derived from it) encoded information about transitions from states unreachable in N steps (and hence of no interest to us). Therefore, we had to aggressively optimize the ADD computations by *restricting* (see [195]) each ADD t_i with an over-approximation of the set of reachable states relevant to that t_i. We discuss this optimization in detail in Sec. 8.5.

Once the ADDs are constructed, the sampling of the $N+1$ states of the trace is done by Algorithm 22. States $\omega[0], \omega[N/2]$ and $\omega[N]$ are sampled from the $\log_2 N^{th}$ ADD in a call to Algorithm 19 in line 2. Then Algorithm 15 is recursively called to sample the first and second halves of the trace in lines 3 and 4. In each recursive call, Algorithm 15 invokes the procedure in Algorithm 19, to sample the state at the mid-point of the current segment of the trace under consideration, and recurses on each of the two halves thus generated.

In sampleFromADD (Algorithm 19), the $\log_2 N^{th}$ ADD is used as-is for sampling

(lines 1,2), while other ADDs are first simplified by substituting the values of state variables in $\omega[lo]$ and $\omega[hi]$, that have been sampled previously and provided as inputs to sampleFromADD (lines 3,4). The role of the rest of the algorithm is to sample a path from a leaf to the root in a bottom-up fashion, with probability proportional to the value of the leaf. Towards this end, a leaf is first sampled in lines 5-8. We assume access to a procedure *weighted_sample* that takes as input a list of elements and their corresponding weights, and returns a random element from the list with probability proportional to its weight. Once a leaf is chosen, we traverse up the DAG in the loop on line 9. This is done by iteratively sampling a parent with probability proportional to the number of paths reaching the parent from the root (lines 10-12). The quantity $|\Pi_v|$ denotes the number of paths from a node v to the root, and can be easily computed by dynamic programming. If some levels are skipped between the current node v and its parent p, then the number of paths reaching the current node from the parent are scaled up by a factor of $2^{level(p)-level(v)-1}$ (line 12). This is because each skipped level contributes a factor of 2 to the number of paths reaching the root. Once a parent is sampled, the value of the corresponding state variable is updated in the trace in lines 13-17, where the procedure $getTracePosition$ is assumed to return the index of the state (in the trace ω) and the index of the state variable (in the set X of state variables) corresponding to the parent node. $getTracePosition$ can be implemented by maintaining a map between the state variables and the variable order in the DD. The random values for variables in the skipped levels between the parent and the current node are sampled in lines 18 and 19.

Non Power-of-2 trace lengths When the trace length N is not a power of two, we modify the given sequential circuit so that the distribution of traces of length N' ($> N$) of the modified circuit is identical to the distribution of length-N prefixes of these traces. Conceptually, the modification is depicted in Fig. 8.2. Here, the "Saturate-at-N" counter counts up from 0 to N and then stays locked at N. Once

the count reaches N, the next state and current state of the original circuit are forced to be identical, thanks to the multiplexer. Therefore, the modified circuit's trace, when projected on the latches of the original circuit, behaves exactly like a trace of the original circuit up to N steps. Subsequently, the projection remains stuck at the state reached after N steps. Hence, by using the modified circuit and by choosing $N' = 2^{\lceil (\log_2 N) \rceil}$, we can assume w.l.o.g. that the length of a trace to be sampled is always a power of 2.

Weighted Sampling A salient feature of Algorithms 1-4 is that the same framework can be used for weighted sampling (instead of uniform) as defined in Section 8.2, with one small modification: if the input t_0 to Algorithm 13 is an ADD instead of a BDD, where the values of leaves are the weights of each transition, then it can be shown that drawSample will sample a trace with probability proportional to its weight, where the weight of a trace is define multiplicatively as in Section 8.2.

Algorithm 13 *makeADDs*(t_0, N, f, I)

Input: t_0: 1-step transition relation

N: trace length

f: characteristic function of target states

I: characteristic function of initial states

Output: ADDs $t_1 \dots t_{\log_2 N}$: 2^i-step transition relations t_i

1: $g \leftarrow t_0$;

2: **for** $i = 1, 2, \dots, \log_2 N$ **do**

3: $\quad t_i(X^0, X^i, X^{i+1}) \leftarrow g(X^0, X^i) \times g(X^i, X^{i+1})$;

▷ $\times$ is ADD multiplication

4: $\quad g \leftarrow \exists X^i \ t_i$; ▷ Additively abstract vars in X^i

5: $t_{\log_2 N} \leftarrow t_{\log_2 N} \wedge f(X^{(\log_2 N)+1}) \wedge I(X^0)$

6: **return** $t_1 \dots t_{\log_2 N}$

Algorithm 14 drawSample$(t_1, \ldots, t_{\log_2 N})$

1: $\omega \leftarrow []$ ▷ Initialize empty trace

/* Sample 0, $N/2$ and N^{th} states from $\log_2 N^{th}$ ADD */

2: $\omega[0], \omega[N/2], \omega[N] \leftarrow sampleFromADD(\log_2 N, t_{\log_2 N}, 0, N, \omega)$;

/* Sample states $0 \ldots N/2$ */

3: $\omega[0 \ldots N/2] \leftarrow$ drawSample_rec$((\log_2 N) - 1, 0, N/2, \omega)$;

/* Sample states $N/2 \ldots N$ */

4: $\omega[N/2 \ldots N] \leftarrow$ drawSample_rec$((\log_2 N) - 1, N/2, N, \omega)$;

5: **return** ω

Algorithm 15 drawSample_rec$(i, lo, hi, \omega, t_1, .., t_{\log_2 N})$

1: $mid \leftarrow (lo + hi)/2$;

2: $\cdot, \omega[mid], \cdot \leftarrow sampleFromADD(i, t_i, lo, hi, \omega)$;

▷ Sample $\omega[mid]$. ($\omega[lo], \omega[hi]$ unchanged)

3: $\omega[lo \ldots mid] \leftarrow$ drawSample_rec$(i - 1, lo, mid, \omega)$;

4: $\omega[mid \ldots hi] \leftarrow$ drawSample_rec$(i - 1, mid, hi, \omega)$;

5: **return** ω

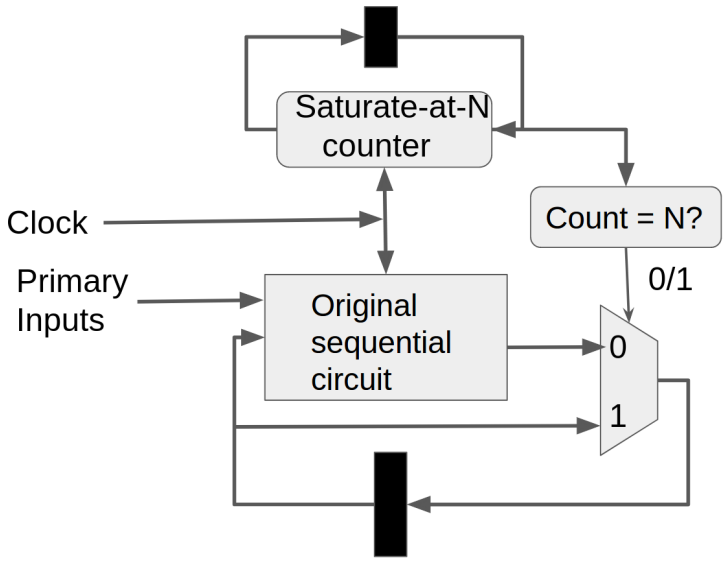


Figure 8.2 : Modified Circuit for Non-Power-of-2 Trace Lengths

Algorithm 16 $sampleFromADD(i, t_i, lo, hi, \omega)$

$mid \leftarrow (lo + hi)/2$;
if $i == \log_2 N$ **then** ▷ Use whole ADD for sampling
    $\hat{t} \leftarrow t_i$;
else ▷ Reduce ADD with states previously sampled
    $\hat{t} \leftarrow Substitute(t_i, \omega[lo], \omega[hi])$
$wtList \leftarrow []$; ▷ Array for weights
    /*Sample a leaf*/
for $v_l \in leaves(\hat{t})$ **do**
    $wtList[l] \leftarrow val(v_l) * |\Pi_{v_l}|$
$v \leftarrow weighted\_sample(wtList, leaves(\hat{t}))$
    /*Sample parents up to root*/
while $v \neq root(\hat{t})$ **do**
    for $p \in P(v)$ **do** ▷ Find weights of all parents
        $wtList[p] \leftarrow 2^{level(p)-level(v)-1} * |\Pi_p|$;
        ▷ Weight adjusted for skipped levels
    $p^* \leftarrow weighted\_sample(wtList, P(v))$;
    $j_s, j_b \leftarrow getTracePosition(p^*, i, lo, hi)$;
        ▷ j_s is state index, j_b is variable bit index
    if $then\_child(p^*) == v$ **then**
        $\omega[j_s][j_b] \leftarrow True$
    else
        $\omega[j_s][j_b] \leftarrow False$
    for each $v_{skipped}$ between p^* and v **do**
        $j_s, j_b \leftarrow getTracePosition(v_{skipped}, i, lo, hi)$;
        $\omega[j_s][j_b] \leftarrow random\_bit()$ ▷ For skipped vars
    $v \leftarrow p^*$
return $\omega[lo], \omega[mid], \omega[hi]$

8.5 Improved Iterative Squaring

In this section, we present a more efficient version of Alg. 29. To see where gains in efficiency can be made, note that the t_is generated using Alg. 29, encode transitions that are never used during sampling. For instance, the ADD $t_{(\log_2 N)-1}$ as constructed by Alg. 29, is only used by the procedure drawSample for sampling states $\omega[N/4]$ (given $\omega[0]$ and $\omega[N/2]$) and $\omega[3N/4]$ (given $\omega[N/2]$ and $\omega[N]$). Thus, $t_{(\log_2 N)-1}$ should only be concerned with states reachable in exactly 0,$N/4$, $N/2$, $3N/4$ or N steps from the initial set. However, the $t_{(\log_2 N)-1}$ constructed by Alg. 29 also contains information about other $2^{(\log_2 N)-1}$-step transitions from states not reachable in those many step from the initial set. This information is clearly superfluous and only serves to increase the size of the ADD. Such information is present in all t_is and exists because the iterative squaring framework of Alg. 29 squares all transitions in the loop on lines 2-4 regardless of the initial state, final state and reachability conditions. We give an improved squaring framework, presented in Algs. 17 and 18. The idea is to first compute (over-approximations of) sets of states reachable in exactly i steps from the initial set, for $1 \leq i \leq N$ (Alg. 18). We then restrict each ADD t_i by the over-approximations of only those reachable state sets it depends on (Alg. 17).

The set $\alpha[i-1]$ for $1 \leq i \leq \log_2 N$ in Alg. 18 represents the set that will be used for restricting the X^0 variable set of t_i, while the set $\beta[i-1]$ will be used for restricting the X^i variable set of t_i. The (over-approximate) set of states reachable after exactly j steps from the initial state, denoted r_j, is computed in line 5 starting from the initial set by taking the (over-approximate) image under t_0 of the reachable set after $j-1$ steps. Computing an exact image is often difficult for large benchmarks, hence an over-approximation of the image can be used. The literature contains a wide spectrum of heuristic techniques that can be used to trade-off space for time of computation. Once r_j is computed, we disjoin the appropriate elements of α and β with r in lines 7-11. The special case of $(N/2)^{th}$ reachable set is handled separately in lines 12-13.

After α and β sets are computed, we use them to restrict g and $\hat{g}$ in lines 3-4 of

Alg. 17. The restrict operation is the one proposed in [195]. If $f = Restrict(g, h)$ then $f = g$ wherever h is true, and f is undefined otherwise. This operation can be more efficient than conjunction, and is sufficient for our purposes as we explicitly enforce initial state condition in Line 7 of Alg. 17.

Algorithm 17 $makeADDs(t_0, N, \alpha, \beta, f, I)$

Input: t_0: 1-step transition function

N: trace length

f: final-state function

I: initial-state function

α, β: reachable state-sets

Output: ADDs $t_1 \dots t_{\log_2 N}$: 2^i-step transition relations t_i

1: $g \leftarrow t_0$;

2: **for** $i = 1, 2, \dots, \log_2 N$ **do**

3: $\quad \hat{g}(X^i, X^{i+1}) \leftarrow Restrict(g(X^i, X^{i+1}), \beta[i-1](X^i))$

4: $\quad g(X^0, X^i) \leftarrow Restrict(g(X^0, X^i), \alpha[i-1](X^0))$;

5: $\quad t_i(X^0, X^i, X^{i+1}) \leftarrow g(X^0, X^i) \times \hat{g}(X^i, X^{i+1})$;

6: $\quad g \leftarrow \exists X^i \ t_i$; ▷ Additively abstract vars in X^i

7: $t_{\log_2 N} \leftarrow t_{\log_2 N} \wedge f(X^{(\log_2 N)+1}) \wedge I(X^0)$

8: **return** $t_1 \dots t_{\log_2 N}$

8.6 Analysis

8.6.1 Hardness of Counting/Sampling Traces

Counting and sampling satisfying assignments of an arbitrary Boolean formula, say φ, can be easily reduced to counting and sampling, respectively, of traces of a transition system. From classical results on counting and sampling in [9, 13, 14, 15], it follows

Algorithm 18 *computeReachableSets*(t_0, I)

1: $r_0 \leftarrow I$; ▷ Initialize r to be the initial state set
2: $\alpha \leftarrow [I, \ldots, I]$;
▷ Initialize array of $(\log_2 N)$ initial state functions.
3: $\beta \leftarrow [0, \ldots, 0]$;
▷ Initialize array of $(\log_2 N)$ Boolean 0 functions.
4: **for** $j \in \{1, \ldots, N\}$ **do**
5: $r_j \leftarrow \boldsymbol{Im}(r_{j-1}, t_0)$ ▷ Find (over approx.) image of r under t_0
6: **for** each $i \in \{0, 1, 2, \ldots (\log_2 N) - 2\}$ **do**
7: **if** $j\%(2^{i+1}) == 0$ **then**
8: $\alpha[i] \leftarrow \alpha[i] \vee r_j$
9: **else**
10: $\beta[i] \leftarrow \beta[i] \vee r_j$
11: break;
12: **if** $j == N/2$ **then**
13: $\beta[(\log_2 N) - 1] = \beta[(\log_2 N) - 1] \vee r_j$
14: **return** α, β

that counting traces is #P-hard and uniformly sampling traces can be solved in probabilistic polynomial time with access to an NP-oracle.

To see how the reduction works, suppose the support of φ has n variables, say $x_1, \ldots x_n$. We construct a transition system (S, Σ, t, I, F), where $S = \{0,1\}^n$ and the set of state variables is $X = \{x_1, \ldots x_n\}$. We let $\Sigma = \{0,1\}$ and define the transition function $t : \{0,1\}^n \times \{0,1\} \to \{0,1\}^n$as follows: $t(x_1, \ldots, x_n, a)[0] = \varphi(x_1, \ldots, x_n)$ and $t(x_1, \ldots, x_n, a)[1] = t(x_1, \ldots, x_n, a)[2] = \ldots = t(x_1, \ldots, x_n, a)[n] = 0$, for $a \in \{0,1\}$. In other words, the 0^{th} next-state bit is determined by φ regardless of the input a, while the rest of the next-state bits are always 0. We define $I = \{0,1\}^n$ and $F = \{1000 \cdots 0\}$. It is easy to see that counting/sampling traces of length 1 of this transition system effectively counts/samples satisfying assignments of φ.

8.6.2 Random Walks and Uniform Traces

It is natural to ask if uniform trace-sampling can be achieved by a Markovian random walk, wherein the outgoing transition from a state is chosen according to a probability distribution specific to the state. Unfortunately, we show below that this cannot always be done. Since uniform sampling is a special case of weighted sampling, the impossibility result holds for weighted trace sampling too.

Consider the transition system in Fig. 8.1. We've seen in Section 8.1 that there are 7 traces of length 4. Hence a uniform sampling would generate each of these traces with probability 1/7. Suppose, the probability of transitioning to state s_j from state s_i is given by $\Pr[(s_i, s_j)]$. For uniform sampling, we require $\Pr[(s_i, s_j)] > 0$ if $\exists a \in \Sigma.\, s_j = t(s_i, a)$, and also $\sum_{s_j\,:\,\exists a,\, s_j = t(s_i,a)} \Pr[(s_i, s_j)] = 1$. Now, consider the traces $\omega_1 = s_0 s_1 s_1 s_1 s_1$ and $\omega_2 = s_0 s_1 s_1 s_1 s_2$. Let $\Pr[(s_0, s_1)] = c\ (> 0)$ and $\Pr[(s_1, s_1)] = d\ (> 0)$. This implies that $\Pr[(s_1, s_2)] = 1 - d\ (> 0)$. Thus, the probability of sampling ω_1 is $c.d^3$. For uniformity, $c.d^3 = 1/7$. Similarly, from ω_2, we get $c.d^2.(1-d) = 1/7$. From these two equations, we obtain $c.d^2 = 2/7$. Therefore, $d = \frac{cd^3}{cd^2} = 1/2$. It follows from the equation $cd^3 = 1/7$ that $c = 8/7$. However, this

	X^0	X^i
t_1	$j \in \{0, 2, 4, 6, ...\}$	$j \in \{1, 3, 5, 7, ...\}$
t_2	$j \in \{0, 4, 8, 12, ...\}$	$j \in \{2, 6, 10, 14, ...\}$
t_3	$j \in \{0, 8, 16, ...\}$	$j \in \{4, 12, 20, ...\}$
...	...	...

Table 8.2 : Reachable sets r_j that X^0, X^i variables of t_i depend on

is not a valid probability measure. Therefore, it is impossible to uniformly sample traces of this transition system by performing a Markovian random walk.

8.6.3 Correctness of Algorithms

We now turn to proving the correctness of algorithms presented in the previous section. We first prove the correctness of the improved iterative squaring framework (Sec. 8.5). Alg. 18 (lines 8-10) ensures that $\alpha[i-1]$ is computed as a disjunction of r_j's for values of j given in the first column and row i of Tab. 8.2, while $\beta[i-1]$ is computed from r_j's for values of j given on the i^{th} row and second column. Therefore, to show the correctness of Algs. 17 and 18, we show in Lemma 8.1 that the X^0 and X^i variable sets of t_i will only be instantiated with (over-approximations of) sets of states reachable in the number of steps given in the appropriate column of Tab. 8.2.

Lemma 8.1. *Let S_q^p denote the set of states that the variable set X^p of t_q will be instantiated with by Alg. 22, when used in conjunction with Algs. 18 and 17. Then $\forall s \in S_i^0$, we have $s \in r_j$ for some j given in column 1 and row i of Tab. 8.2, and $\forall s \in S_i^i$, we have $s \in r_j$ for some j given in column 2 and row i of Tab. 8.2.*

Proof. We show by induction on i from $\log_2 N$ down to 1. The base case is shown by the fact that $t_{\log_2 N}$ is used exactly once by drawSample and the X^0 variables are used only for sampling the initial state while X^i is used for sampling $\omega[N/2]$. Thus $S^0_{\log_2 N} \subseteq r_0$ and $S^{\log_2 N}_{\log_2 N} \subseteq r_{N/2}$. The former condition is satisfied by the limits of the

for-loop in Line 6 of Alg. 18, while the latter condition is satisfied by lines 12-13 of Alg. 18. This completes the base case.

Now assume that the lemma holds for some i. We will prove that the lemma holds for $i-1$ as well. First note that t_i is used by sampleFromADD for sampling some state $\omega[m]$ given states $\omega[m-2^{i-1}]$ and $\omega[m+2^{i-1}]$. Thereafter, t_{i-1} is used in 2 cases: (1) for sampling $\omega[m+2^{i-2}]$ given $\omega[m]$ and $\omega[m+2^{i-1}]$; and (2) for sampling $\omega[m-2^{i-2}]$ given $\omega[m]$ and $\omega[m-2^{i-1}]$. Thus the X^0 variables of t_{i-1} will be instantiated with the same states as for X^0 variables of t_i in case (1). In case (2), X^0 vars of t_{i-1} will be instantiated with the same states as for X^i vars of t_i. Thus the states instantiating X^0 vars of t_{i-1} are the union of the states instantiating X_0 and X^i variables of t_i, i.e., $S^0_{i-1} = S^0_i \cup S^i_i$. The values in Tab. 8.2 reflect this fact, and by our inductive assumption S^0_i and S^i_i were computed correctly. This proves that $\forall s \in S^0_i$, $s \in r_j$ for some j given in column 1 and row i of Tab. 8.2. To complete the inductive argument we still need to show that $\forall s \in S^i_i$, $s \in r_j$ for some j given in column 2 and row i of Tab. 8.2. To see this, first note that the X^i variables of t_i will only be instantiated with states reachable in 2^{i-1} steps from the states instantiating the X^0 variables of t_i. This is reflected in Tab. 8.2. For instance, in row 3 ($i = 3$), $r_4, r_{12}, r_{20} \ldots$ in column 2 are exactly the set of states reachable in $2^{i-1} = 4$ steps from $r_0, r_8, r_{16} \ldots$ respectively, in column 1. Since we showed that S^0_i has been computed correctly, this completes the proof. □

Let $c(l, s_i, s_j)$ denote the number of traces of length 2^l starting in state s_i and ending in state s_j. Note that $c(l, s_i, s_j) = \sum_{s_k \in S} c(l-1, s_i, s_k) \times c(l-1, s_k, s_j)$. We use the fact that sampleFromADD ensures that the parent of a node v is sampled independently of the path from an ADD leaf to v chosen so far. Conditional independence also holds for whole traces; given the states at two indices in a trace, the states within the trace segment delineated by the indices are sampled independently of the states outside the trace segment. The following lemmas characterize the behavior of the sampling framework (Algs. 22–19).

Lemma 8.2. *For $1 \leq i \leq \log_2 N$, the ADD t_i computed by makeADDs is such that $\forall s_{j_1}, s_{j_2}, s_{j_3} \in S$, we have $t_i(s_{j_1}, s_{j_2}, s_{j_3}) = c(i-1, s_{j_1}, s_{j_2}) \times c(i-1, s_{j_2}, s_{j_3})$*

Proof. We will prove by induction on i.

Base case: We have $\forall s_{j_1}, s_{j_2} \in S\ t_0(s_{j_1}, s_{j_2}) = c(0, s_{j_1}, s_{j_2})$ by definition. From line 3 of Alg. 29, we then have $\forall s_{j_1}, s_{j_2}, s_{j_3} \in S, t_1(s_{j_1}, s_{j_2}, s_{j_3}) = c(0, s_{j_1}, s_{j_2}) \times c(0, s_{j_2}, s_{j_3}))$

Induction step: Assume the lemma holds up to some i, i.e. $\forall s_{j_1}, s_{j_2}, s_{j_3} \in S\ t_i(s_{j_1}, s_{j_2}, s_{j_3}) = c(i-1, s_{j_1}, s_{j_2}) \times c(i-1, s_{j_2}, s_{j_3})$. After execution of line 4 of Alg. 29, we will have $\forall s_{j_1}, s_{j_3} \in S\ g(s_{j_1}, s_{j_3}) = \sum_{s_{j_2}} c(i-1, s_{j_1}, s_{j_2}) \times c(i-1, s_{j_2}, s_{j_3}) = c(i, s_{j_1}, s_{j_3})$. Then in the next iteration of the loop after line 3, we will have $\forall s_{j_1}, s_{j_2}, s_{j_3} \in S\ t_{i+1}(s_{j_1}, s_{j_2}, s_{j_3}) = c(i, s_{j_1}, s_{j_2}) \times c(i, s_{j_2}, s_{j_3})$. □

Lemma 8.3. *Let Z denote the random path from a leaf to the root of ADD $\hat{t}$ (see Alg. 4) chosen by* sampleFromADD*. Then*

$$\forall \pi \in \Pi \ \ \Pr[Z = \pi] = \frac{val(\pi[0])}{\sum_{v \in leaves(\hat{t})} val(v) \cdot |\Pi_v|} \tag{8.1}$$

Proof. The leaf v_l is sampled with probability $\Pr[\pi[0] = v_l] = \frac{val(v_l) \cdot |\Pi_{v_l}|}{\sum_{v \in leaves(\hat{t})} val(v) \cdot |\Pi_v|}$. Thereafter, each parent p^* is sampled with probability $\Pr[\pi[i] = p^* | \pi[i-1] = v] = \frac{|\Pi_{p^*}| \cdot 2^\gamma}{\sum_{p \in P(v)} |\Pi_p| \cdot 2^\gamma}$, where $\gamma = level(p) - level(v) - 1$. But note that $\Pi_v = \sum_{p \in P(v)} (|\Pi_p| \cdot 2^\gamma)$. Then, substituting in the identity $\Pr[Z = \pi] = \left(\Pr\left[\pi[0]\right] \cdot \prod_i \Pr\left[\pi[i] \middle| \pi[i-1]\right] \right)$, gives the lemma. □

In the next two lemmas, '*lo*' and '*hi*' refer to trace indices passed as arguments to sampleFromADD, and $mid = (lo + hi)/2$.

Lemma 8.4. *Suppose* sampleFromADD *is invoked with $i < \log_2 N$, $\omega[lo] = s_{j_1}$ and $\omega[hi] = s_{j_3}$. Let M denote the random state returned by* sampleFromADD *for $\omega[mid]$. Then for all $s_{j_2} \in S$, we have $Pr\left[M = s_{j_2} \ \middle|\ \omega[lo] = s_{j_1}, \omega[hi] = s_{j_3}\right] = \frac{c(i-1, s_{j_1}, s_{j_2}) \times c(i-1, s_{j_2}, s_{j_3})}{c(i, s_{j_1}, s_{j_3})}$*

Proof. We note that for any ADD t_i s.t. $i < \log_2 N$, we reduce the ADD by substituting $\omega[lo], \omega[hi]$ in line 4 of Alg. 19. In the resultant ADD $\hat{t}$, each paths from root to leaf yields a valuation for $\omega[mid]$. Therefore, if π is the path traversed in $\hat{t}$ corresponding to some state s_{j_2}, then $Pr\big[\omega[mid] = s_{j_2} \big| \omega[lo] = s_{j_1}, \omega[hi] = s_{j_3}\big] = \Pr[Z = \pi]$. We now need to prove that the R.H.S. of Eqn. 8.1 is the same as the desired conditional probability expression. In Eqn. 8.1, the numerator $val(\pi[0]) = t_i(s_{j_1}, s_{j_2}, s_{j_3}) = c(i-1, s_{j_1}, s_{j_2}) \times c(i-1, s_{j_2}, s_{j_3})$, by Lemma 8.2. The denominator of Eqn. 8.1 is $\sum_{v \in leaves(\hat{t})}(val(v) * |\Pi_v|) = \sum_{s_{j_2}} c(i-1, s_{j_1}, s_{j_2}) \times c(i-1, s_{j_2}, s_{j_3})$ which is $c(i, s_{j_1}, s_{j_3})$. □

Lemma 8.5. *Let* sampleFromADD *be invoked with $i = \log_2 N$, and let L, M and H denote the random states returned for $\omega[lo]$, $\omega[mid]$ and $\omega[hi]$ respectively. Then for all $s_{j_1}, s_{j_2}, s_{j_3} \in S$ s.t. $I(s_{j_1})$ and $f(s_{j_3})$ hold, we have* $\Pr\big[(L = s_{j_1}) \wedge (M = s_{j_2}) \wedge (H = s_{j_3})\big] = \frac{c(i-1, s_{j_1}, s_{j_2}) \times c(i-1, s_{j_2}, s_{j_3})}{|\Omega_N|}$

Proof. By definition, $|\Omega_N| = \sum_{s_{j_1}, s_{j_3}} c(N, s_{j_1}, s_{j_3})$, when $s_{j_1} \models I$ and $s_{j_3} \models f$. The rest of the proof is similar to that of Lem. 8.4. □

Theorem 8.6. *Let Y be a random trace returned by an invocation of* drawSample. *For all $\omega \in \Omega_N$, we have* $\Pr[Y = \omega] = \frac{1}{|\Omega_N|}$.

Proof. Recursively halving ω, we get $\Pr[Y = \omega] = \Pr[(\omega[0] = s_{j_1}) \wedge (\omega[N/2] = s_{j_2}) \wedge (\omega[N] = s_{j_3})] \cdot \Pr[(\omega[N/4] = s_{j_4}) | (\omega[0] = s_{j_1}) \wedge (\omega[N/2] = s_{j_2})] \cdot \Pr[(\omega[3N/4] = s_{j_5}) | (\omega[N/2] = s_{j_2}) \wedge (\omega[N] = s_{j_3})] \ldots$ Substituting values in the RHS from Lemmas 8.4 and 8.5, we get the result by noting that $\forall s_{j_1}, s_{j_2} \in S\ c(0, s_{j_1}, s_{j_2}) \in \{0, 1\}$ since the transition system is deterministic. □

8.7 Empirical Evaluation

We have implemented our algorithms in a tool called TraceSampler. The objective

of our empirical study was to compare TraceSampler* with other state-of-the-art approaches in terms of number of benchmarks solved as well as speed of solving.

Experimental Setup As noted in Section 8.3, UniWit [16], UniGen [191] and UniGen2 [192] are state-of-the-art tools for almost uniform sampling and SPUR [29], KUS [30] and WAPS [31] are similar tools for uniform sampling of SAT witnesses. We compare TraceSampler with UniGen2 and WAPS in our experiments, since these are currently among the best almost-uniform and uniform samplers respectively, of SAT witnesses. We invoke both WAPS and UniGen2 with default settings. Although UniGen2 is capable of operating in parallel, we invoke it in serial mode to ensure fairness of comparison.

We ran all our experiments on a high performance cluster. Each experiment had access to one core on an Intel Xeon E5-2650 v2 processor running at 2.6 GHz, with 4GB RAM. We used GCC 6.4.0 for compiling TraceSampler with O3 flag enabled, along with CUDD library version 3.0 with dynamic variable ordering enabled. We set a timeout of 7200 seconds for each experiment. For experiments that involved converting benchmarks in Aiger format to BDD (explained below), we allotted 3600 seconds out of 7200 exclusively for this conversion. We attempted to generate 5000 samples in each instance. We called an experiment successful or completed, if the sampler successfully sampled 5000 traces within the given timeout.

Benchmarks We used sequential circuit benchmarks from the Hardware Model Checking Competition [40] and ISCAS89 [196] suites. Each benchmark represents a sequential circuit in the And-Inverter Graph (AIG) format. In general, primary outputs of such a circuit can indicate if target states have been reached, and can be used to filter the set of traces from which we must sample. In our experiments, however, we ignored the primary outputs, and sampled from all traces of a given

*Code available at `https://gitlab.com/Shrotri/tracesampler`

length N starting from the all-zero starting state. We attempt uniform sampling of traces in our experiments, as the benchmarks do not provide weights for transitions.

As mentioned in Section 8.2, we need to existentially quantify the primary inputs from the transition functions to get the transition relations. This is done either explicitly or implicitly depending on the sampler to be used. TraceSampler requires the transition relation t in the form of a BDD while WAPS and UniGen2 require a CNF formula. We used a straightforward recursive procedure for converting t (as AIG) to a BDD. We then quantified out the primary inputs using library functions in CUDD. For converting t to CNF there were two choices: (1) by obtaining the prime cover using a built-in operation in CUDD, or (2) using Tseitin encoding to convert the AIG to CNF by introducing auxiliary variables. The CNF obtained from the first method has no auxiliary variables that need to be existentially quantified; hence it can be used with WAPS and the D4 compiler, which does not support existential quantification.[†] In contrast, the second method can be used in conjunction with UniGen2, since the auxiliary variables need to be projected out.

Within the available time and memory, we obtained a total of 310 pre-processed AIG files, out of which 102 could be converted to BDDs with primary inputs existentially quantified out. The number of latches for these 102 benchmarks ranged between 5 and 175, and the median number of latches was 32. The distribution of number of of latches is depicted in Fig. 8.3. We restricted the start state to be all-zeros since this is implicit in the AIG format. For each benchmark, we attempted to sample traces with lengths 2, 4, 8, 16, 32, 64, 128 and 256. We chose this range of trace lengths since a vast majority of benchmarks in HWMCC-17 (particularly, benchmarks in the DEEP category) required bounds within 256 [197]. Further, we observed that none of the tools were able to consistently scale beyond traces of length 256. We refer to a benchmark and a given trace length as an 'instance'. We thus generated

[†]WAPS also can work with the DSharp compiler, which supports existential quantification. However, in our experiments we found that DSharp provided incorrect answers.

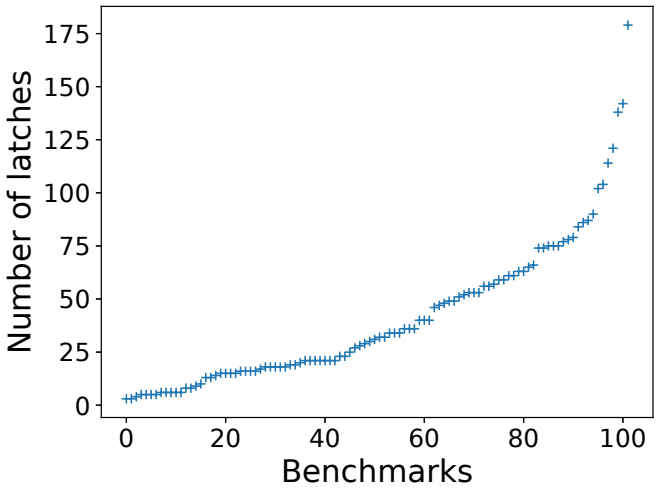


Figure 8.3 : Distribution of benchmark sizes (number of latches)

$102 \times 8 = 816$ instances for BDD based approaches. For CNF based approaches, the unrolling was done by appropriately unrolling the transition relation. Note that the first CNF-based approach was applicable to only 102 benchmarks that could be converted to BDDs, while the direct conversion from AIG to CNF was technically possible for all 310 benchmarks. However, we primarily report on the 816 instances even for AIG-encoded CNFs. We discarded formulas with more than 10^6 clauses, as the files became too large.

Results In our experiments we found that UniGen2 fared better with formulas encoded from BDDs, vis-a-vis formulas encoded directly from AIGs. All reported results are, therefore, on BDD-encoded formulas. We only report on instances with at least 100 distinct traces of the given length, since trace sampling can be trivially implemented by enumerating traces, if the trace-count is small.

We consistently found that TraceSampler outperformed both WAPS and UniGen2 by a substantial margin. We present a comparison of the performance of the 3 tools on the 816 instances where BDD construction succeeded. Figure 8.5 shows a cactus plot of the number of experiments completed in the given time, with the number of

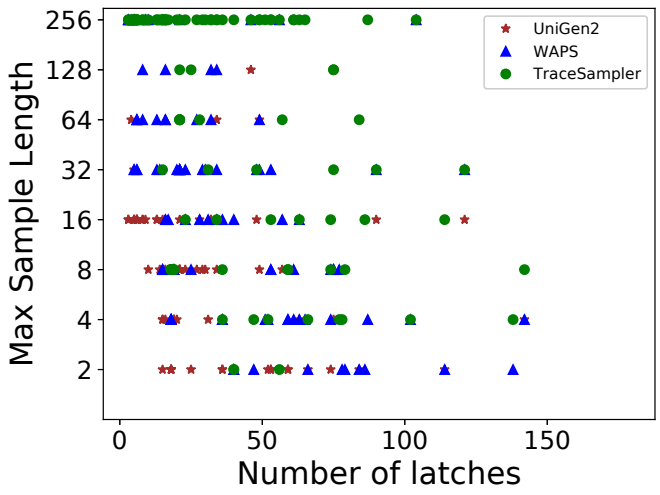


Figure 8.4 : Length of longest trace sampled vs. number of latches for each benchmark

instances on x–axis and the total time taken on y–axis. A point (x, y) implies that x instances took less than or equal to y seconds to solve. TraceSampler is able to complete 502 experiments out of 816 – almost 200 more than WAPS and 350 more than UniGen2.

TraceSampler was also fastest on the majority of instances. Among a total of 503 instances on which at least one sampler succeeded, TraceSampler was fastest on 446 (88.7%) while WAPS and UniGen2 were fastest on 33 (6.5%) and 24 (4.8%) respectively. For instances on which both tools were successful, we found that on average (geometric mean) TraceSampler offered a speedup of 25× compared to UniGen2 and 3 relative to WAPS. Overall, TraceSampler was able to sample traces 3.5 times longer on average (geometric mean) as compared to WAPS and 10 times longer as compared to UniGen2. Further, TraceSampler is able to sample traces of length 256 from 52 benchmarks, while WAPS and UniGen2 are able to sample 256-length traces from 12 and 3 benchmarks respectively. Fig. 8.4 depicts the distribution of the maximum length of traces each algorithm is able to sample from, relative to the size (number of latches) of the corresponding benchmarks. It can be seen that TraceSampler is

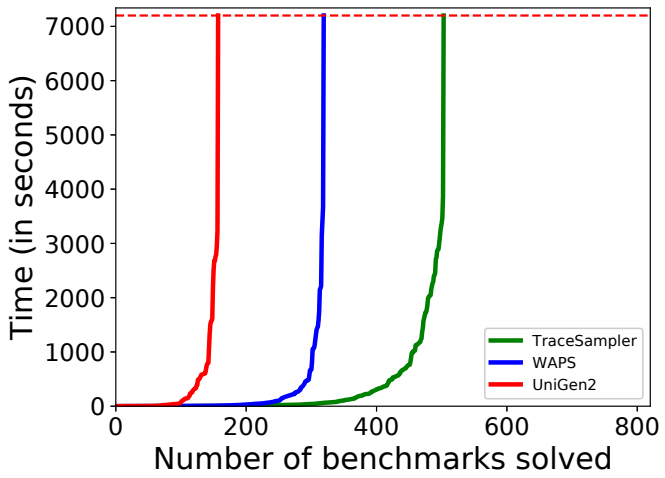


Figure 8.5 : Performance Comparison of TraceSampler with WAPS and UniGen2.

generally able to sample longer traces from larger benchmarks.

WAPS and TraceSampler proceed in two phases — the compilation phase where a d-DNNF or ADDs are constructed, and the sampling phase where the constructed structures are traversed. When only considering the time required for compilation, TraceSampler offered speedup factor of 16 compared to WAPS.

Output Distribution While Theorem 8.6 guarantees uniformity of distribution of traces generated by TraceSampler, we performed a simple experiment to compare the actual distribution of traces generated by TraceSampler with that generated by WAPS – a perfectly uniform sampler. The instance we selected had 8192 distinct traces. We generated 10^6 traces samples using both TraceSampler and WAPS, computed the frequency of occurrence of each trace, and grouped traces occurring with the same frequency. This is shown in Fig. 8.6 where a point (x, y) indicates that x distinct traces were generated y times. It can be seen that the distributions generated by TraceSampler and WAPS are practically indistinguishable, with Jensen-Shannon distance 0.003. Similar trends were observed for other benchmarks as well.

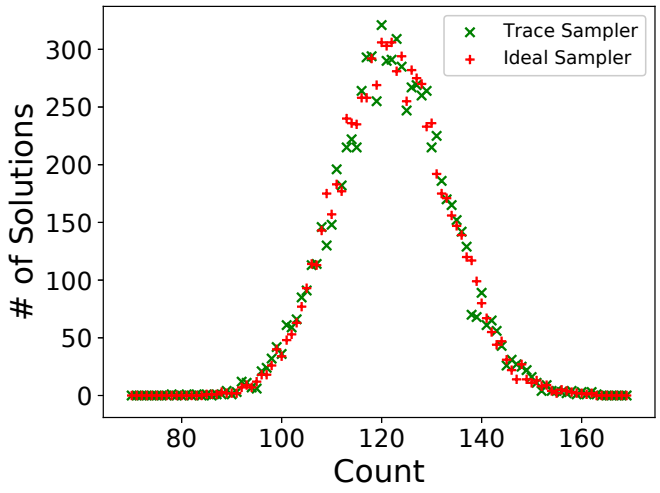


Figure 8.6 : Distributions of generated samples.

Comparison with ApproxMC3 The UniGen series of samplers are based on the approximate counting tool ApproxMC. At the time of writing this paper, the latest version of ApproxMC, called ApproxMC3, had not been incorporated into UniGen. In order to obtain an idea of the kind of performance gains one can expect from UniGen with updated counting sub-modules, we ran experiments to count the number of traces of a given length with ApproxMC3. TraceSampler was able to sample traces 5× longer than what ApproxMC3 could count, while providing 17× speedup, on average. Thus, on benchmarks where BDD construction was successful, TraceSampler was clearly the best choice. However, ApproxMC3 was able to count the number of 8-long traces in 62 cases out of 208 in which BDD construction failed. Yet, the time required for sampling using UniGen2 usually far exceeds that required to count, as the counting subroutine is invoked multiple times for obtaining the desired number of samples. The times reported for ApproxMC3 therefore, are a generous lower-bound on the times that would be required for sampling.

Discussion Our experiments show that TraceSampler is the algorithm of choice for uniformly sampling traces and is even able to outperform the state-of-the-art model

counter. We consistently observed that most of the time used for ADD construction is spent in dynamic variable reordering; given a good variable order, ADD construction is usually very fast. In industrial settings, a good variable order may be available for the circuits of interest. In addition, compilation can often be done 'off-line' resulting in its cost getting amortized over the generated samples. In this light, the compilation time speedup of TraceSampler relative to WAPS is encouraging.

A drawback of using BDDs and ADDs is that they often blow-up in size. Indeed, we found that conversion of AIG to BDD failed on 208 benchmarks. Nevertheless, we found that UniGen2 was also unable to finish sampling a single instance (with trace length 8) of these 208 benchmarks, as well. This indicates that the problem may lie deeper in the transition structure, rather than in the variable order.

It is worth noting that CRV runs typically span hundreds of thousands of clock cycles, while we (and other approaches that provide uniformity guarantees) can sample traces of a few hundred transitions at present. This is because trace sampling requires solving global constraints over the entire length of the trace, while in CRV, local constraints over short segments of an otherwise long trace need to be solved. We believe these are complementary strengths that can be used synergistically. Specifically, a CRV tool can be used to drive a system into a targeted (possibly bug-prone) corner over a large number of clock cycles. Subsequently, one can ensure provably good coverage of the system's runs in this corner by uniformly sampling traces for the next few hundred cycles. We believe this synergy can be very effective in simulation-based functional verification.

8.8 Chapter Summary

In this paper, we introduced a symbolic algorithm based on ADDs for sampling traces of a transition system (sequential circuit) with provable uniformity (or bias) guarantees. We demonstrated its scalability vis-a-vis competing SAT-based approaches that provide similar guarantees, through an extensive empirical study. Our main contribu-

tion related to Constrained Sampling is to demonstrate domain-specific factorization, coupled with ADD-compilation to yield fast amortized solution sampling. Specifically, our idea of enhanced iterative squaring implicitly encodes a factored representation for the problem of trace sampling, thereby underscoring the benefits domain-driven approaches.

Chapter 9

Sampling Solutions of Low-Treewidth CNF Formulas

9.1 Introduction

Given a Boolean formula φ over n variables and a user-defined weight function w assigning a non-negative real weight to all 2^n assignments, the problem of weighted sampling is to randomly generate an assignment that satisfies φ with probability proportional to the weight of the assignment. If the weight function is uniform over all assignments, then the problem is called uniform sampling. The problems of uniform and weighted sampling have diverse applications in various domains like probabilistic inference [18], testing and verification [19, 20, 62].

As noted in Chapter 6, there is a deep connection between sampling and the problem of model counting: given an algorithm for counting the number of solutions of a given formula, it is possible to generate uniform samples with polynomially many queries to the counter [14]. Consequently, a number of sampling tools have been developed in recent years leveraging advances in model counting [29, 30, 31]. For example, the state-of-the-art sampling tool, `WAPS` [31], is based on *knowledge compilation (KC)* approaches to model counting. The key idea of KC approaches is to compile the input formula into a succinct data structure (e.g., a d-DNNF for `WAPS`) which allows for fast sampling (e.g., through a Markovian random walk on a d-DNNF).

Recently, a line of model counting work that leverages *dynamic programming* has evolved in parallel to KC approaches [198, 49, 59, 199]. It was shown that by leveraging tree decompositions and datastructures known as Algebraic Decision Diagrams

(ADDs) [50], it is possible to perform model counting extremely efficiently on formulas with low treewidth [200, 59]. This property of having low treewidth has been noted to be common in many real-world benchmarks [201]. For low-treewidth instances, the tool `DPMC` was shown [59] to outperform state-of-the-art d-DNNF-based tools like `d4` [27]. `d4` is also the d-DNNF compiler used in the sampler `WAPS`.

While this synergistic interplay between counting, sampling and KC has been profitable for d-DNNF-based tools, this interplay has not been leveraged for ADD-based dynamic programming algorithms like `DPMC`. The question left unanswered is: can we perform sampling by exploiting treewidth using dynamic programming and ADDs?

In this chapter, we present our work where we answered this question in the positive. Our algorithm, `DPSampler`, operates in three phases. In the first phase an execution plan in the form of a project-join tree is computed, based on the tree decomposition of the input formula. In second phase, `DPSampler` compiles the input CNF into a succinct Tree-of-ADDs representation based on the plan generated in first phase. In third phase, the tree is traversed to generate a random sample. This decoupling of planning, compilation and sampling phases enables usage of various libraries for each purpose in a black-box fashion and makes it trivial to parallelize. Further, our novel ADD-sampling algorithm avoids the need for expensive dynamic memory allocation required in previous work [62]. Extensive experiments over diverse sets of benchmarks arising from applications in AI show `DPSampler` is more scalable and versatile than existing approaches. The contributions of our work can be summarized as follows:

1. `DPSampler`, the first weighted and uniform sampler based on dynamic programming that is able to exploit tree decompositions;

2. An "in-place" ADD-sampling algorithm that avoids dynamic memory allocation and deallocation; and

3. An empirical study that demonstrates that `DPSampler` has strong performance on diverse benchmarks.

9.2 Preliminaries

Boolean formulas and Pseudo-Boolean Functions. A *pseudo-Boolean function* over a set X of Boolean variables is a function $f : \{0,1\}^X \to \mathbb{R}$, where $\{0,1\}^X$ denotes the set of all possible assignments to the variables in X. For notational convenience, we sometimes denote an assignment $\omega \in \{0,1\}^X$ to be a set of literals i.e., $\omega = \bigcup_{x \in X} \{lit_x\}$ where lit_x is either x (x assigned to true) or $\neg x$ (x assigned to false). If $Y \subset X$, and $\omega \in \{0,1\}^X$ then we denote the restriction of ω to the variables in Y as ω_Y.

A Boolean formula φ over variables X represents a pseudo-Boolean function over X, denoted $[\varphi] : \{0,1\}^X \to \mathbb{R}$, where for all $\omega \in \{0,1\}^X$, if ω satisfies φ i.e. $\omega \models \varphi$, then $[\varphi](\omega) \equiv 1$ else $[\varphi](\omega) \equiv 0$. In a Boolean formula, a *clause* is a non-empty disjunction of literals. A *CNF formula* is a Boolean formula consisting of a non-empty set (conjunction) of clauses.

Operations on pseudo-Boolean functions include *product* and *projections*. We define product as follows.

Definition 9.1 (Product)**.** *Let X and Y be sets of Boolean variables. The* product *of functions $f : \{0,1\}^X \to \mathbb{R}$ and $g : \{0,1\}^Y \to \mathbb{R}$ is the function $f \cdot g : \{0,1\}^{X \cup Y} \to \mathbb{R}$ defined for all $\omega \in \{0,1\}^{X \cup Y}$ by $(f \cdot g)(\omega) \equiv f(\omega_X) \cdot g(\omega_Y)$.*

Product generalizes conjunction: if φ and ψ are propositional formulas, then $[\varphi] \cdot [\psi] = [\varphi \wedge \psi]$.

Next, we define (additive) projection, which marginalizes a single variable.

Definition 9.2 (Projection)**.** *Let X be a set of Boolean variables and $x \in X$. The* projection *of a function $f : \{0,1\}^X \to \mathbb{R}$ w.r.t. x is the function $\sum_x f : \{0,1\}^{X \setminus \{x\}} \to \mathbb{R}$ defined for all $\omega \in \{0,1\}^{X \setminus \{x\}}$ by $\left(\sum_x f\right)(\omega) \equiv f(\omega \cup \{\neg x\}) + f(\omega \cup \{x\})$.*

Note that projection is commutative, i.e., $\sum_x \sum_y f = \sum_y \sum_x f$ for all variables $x, y \in X$ and functions $f : \{0,1\}^X \to \mathbb{R}$. Given a set $X = \{x_1, x_2, \ldots, x_n\}$, define $\sum_X f \equiv \sum_{x_1} \sum_{x_2} \ldots \sum_{x_n} f$. Our convention is that $\sum_\varnothing f \equiv f$.

Weighted Sampling and Counting This paper is concerned with the problem of weighted sampling:

Definition 9.3 (Weighted Sample)**.** *Let X be a set of Boolean variables, φ be a Boolean formula over X, and $w : \{0,1\}^X \to \mathbb{R}^{\geq 0}$ be a pseudo-Boolean function (called the* weight function*).*

A random variable S with sample space $\{0,1\}^X$ is a w-weighted sample of φ *if, for all $\omega \in \{0,1\}^X$,*

$$\Pr[S = \omega] = \begin{cases} w(\omega)/w(\varphi) & \text{if } \omega \models \varphi \\ 0 & \text{if } \omega \not\models \varphi \end{cases}$$

where $w(\varphi) \equiv \sum_{\omega \models \varphi} w(\sigma)$ is a normalization factor.

If w is a constant function, then a w-weighted sample of φ is also called a *uniform sample* of φ. The normalization factor $w(\varphi) = \sum_{\omega \models \varphi} w(\sigma)$ is well-studied independently and is known as the *weighted model count* of φ w.r.t. w.

We focus on sampling with respect to literal-weight functions, which are weight functions that can be expressed as products of weights associated with each literal:

Definition 9.4 (Literal-Weight Function)**.** *A pseudo-Boolean function $w : \{0,1\}^X \to \mathbb{R}^{\geq 0}$ is a* literal-weight function (over X) *if there exist $w(x), w(\neg x) \in \mathbb{R}^{\geq 0}$ for each $x \in X$ such that, for all $\omega \in \{0,1\}^X$,*

$$w(\omega) = \prod_{\substack{x \in X \\ \omega(x)=1}} w(x) \cdot \prod_{\substack{x \in X \\ \omega(x)=0}} w(\neg x).$$

For ease of exposition, we assume that all literal weights are normalized so that $w(x) + w(\neg x) = 1$ for all x, which does not affect the sampling probabilities.

If $w : \{0,1\}^X \to \mathbb{R}^{\geq 0}$ is a literal-weight function and $X' \subseteq X$, we use $w(X')$ as shorthand for the pseudo-Boolean function $w(X') : \{0,1\}^{X'} \to \mathbb{R}^{\geq 0}$ defined for all $\omega \in \{0,1\}^{X'}$ by $w(X')(\omega) = \prod_{\substack{x \in X' \\ \omega(x)=1}} w(x) \cdot \prod_{\substack{x \in X' \\ \omega(x)=0}} w(\neg x)$.

Graphs For a graph G, we denote the set of vertices/nodes by $\mathcal{V}(G)$ and set of edges by $\mathcal{E}(G)$. We denote graphs which are trees by T and the leaves of T as $\mathcal{L}(T)$. A *rooted tree* (T, r) is a tree T together with a distinguished root node $r \in \mathcal{V}(T)$. The children of a node $n \in \mathcal{V}(T)$ in a rooted tree are denoted $\mathcal{C}(n)$, and $\mathcal{C}(n) = \varnothing$ if $n \in \mathcal{L}(T)$. The set of ancestors of n are denoted as $\mathcal{A}(n)$. Note that the nodes in $\mathcal{C}(n)$ are necessarily adjacent to n in T, while a node in $\mathcal{A}(n)$ is adjacent to n only if it is the (unique) parent of n.

9.3 Related Work

[14] showed a deep connection between counting and sampling; in particular they showed that uniform sampling can be polynomially reduced to exact and approximate counting. [15] demonstrated a technique to generate uniform samples related to an NP-Oracle in probabilistic polynomial time. These approaches, however, are known to scale poorly on real world instances [76]. Similarly, BDD-based sampling techniques [202] have also been shown to suffer from scalability issues [203].

The first exact uniform sampling tool to scale on non-trivial benchmarks was `SPUR` [29], which generated samples on-the-fly without explicit compilation. Subsequently, the sampler `KUS` [30], which relied on d-DNNF compilation was shown to significantly outperform `SPUR`. The tool `WAPS` [31] extended KUS to support weighted and projected sampling, and was shown to convincingly outperform even the approximate weighted sampling tool WeightGen [192]. To the best of our knowledge, `WAPS` is currently the state-of-the-art exact weighted sampler. In Sec. 9.6 we perform an extensive empirical comparison between `DPSampler` and `WAPS`.

There is also an extensive line of work on *approximately-uniform* sampling, in

which the sampling probability approximates the uniform one. The UniGen line of algorithms [191, 192, 35] provides strong guarantees on the "almost-uniformity" of generated samples, while tools such as QuickSampler [204] and XOR-Sample [205] provide weak or no guarantees on the output distribution. In this work, we focus exclusively on perfect (that is, no approximation) sampling.

Starting with the seminal work of [122], a wide variety of tractable representations of Boolean functions such as d-DNNFs and SDDs [206], along with variants of OBDDs [124], have been explored in literature under the umbrella of *Knowledge Compilation*. Additionally, [131] analyzed pseudo-Boolean representations including Algebraic Decision Diagrams [50]. A number of compilers have also been developed such as `d4` [27], C2D [26], dSharp [127] etc. The compiled form generated by `DPSampler` is closely related to the Tree-of-BDDs (ToB) language developed in [134, 135] and further analyzed in [136]. We note, however, that `DPSampler` actually compiles a Tree-of-ADDs with some stark differences to the variants of ToB anaylzed in [136]: (1) ToBs are compiled by a two-pass algorithm, while `DPSampler` requires only one pass; and (2) Model counting query can be performed in polynomial time on Tree-of-ADDs as generated by `DPSampler`, while it is unknown whether model counting can be performed in polynomial time for ToBs. A complete analysis of Tree-of-ADDs á la [136] is an interesting direction for fuutre work.

9.4 Sampling from an ADD

We first consider the problem of sampling an assignment to the variables of a single ADD, given a partial assignment to some of its variables, with probability proportionate to the weight of the assignment. We use this as a subprocedure in the sampling phase of `DPSampler`.

Such an ADD-sampling algorithm was previously presented in [62], in the context of a different problem of trace-sampling, as discussed in the previous chapter. While the algorithm of [62] could be used as-is for our purpose, it suffers from serious

Algorithm 19 sampleFromADD(f, w, ω)

Input: f: An ADD (X, S, ρ, G)

Input: w: A literal-weight function over X

Input: ω: An assignment to $Z \subseteq X$

Output: ω': An assignment to $Y = X \setminus Z$

$v \leftarrow root(f)$

computeWeights$(f, w, v, \omega, \varnothing)$

$\omega' \leftarrow \varnothing$

while $v \notin leaves(f)$ **do**

  if $x_v \in \omega$ **then** ▷ $x_v \in Z$ and assigned True

    $v_{next} \leftarrow v.then$

  else if $\neg x_v \in \omega$ **then** ▷ $x_v \in Z$ and assigned False

    $v_{next} \leftarrow v.else$

  else ▷ $x_v \in Y$ i.e. unassigned

    $t\_wt \leftarrow v.then.wt \times w(x_v)$

    $e\_wt \leftarrow v.else.wt \times w(\neg x_v)$

    $rand\_bit \leftarrow weighted\_sample(t\_wt, e\_wt)$

    if $rand\_bit == True$ **then**

      $\omega' \leftarrow \omega' \cup \{x_v\}$ ▷ Assign x_v to True

      $v_{next} \leftarrow v.then$

    else

      $\omega' \leftarrow \omega' \cup \{\neg x_v\}$ ▷ Assign x_v to False

      $v_{next} \leftarrow v.else$

  for $x \in X \setminus Z$ s.t. $\rho(x_v) < \rho(x) < \rho(x_{v_{next}})$ **do**

    $rand\_bit \leftarrow weighted\_sample(w(x), w(\neg x))$

    if $rand\_bit == True$ **then**

      $\omega' \leftarrow \omega' \cup \{x\}$ ▷ Assign x to True

    else

      $\omega' \leftarrow \omega' \cup \{\neg x\}$ ▷ Assign x to False

  $v \leftarrow v_{next}$

return ω'

Algorithm 20 computeWeights$(f, w, v, \omega, visited)$

Input: f: An ADD (X, S, ρ, G)

Input: w: A literal-weight function over X

Input: v: A node in f

Input: ω: An assignment to $Z \subseteq X$

Input: *visited*: Set of ADD nodes previously visited by this function; *visited* is modified in each recursive call.

Output: $v.wt$: The weight of v (see Lemma 9.5)

1: **if** $v \in visited$ **then**
2: **return** $v.wt$
3: $visited \leftarrow visited \cup \{v\}$
4: **if** $v \in leaves(f)$ **then**
5: **return** $v.val$

/* x_v is the variable labeling node v */

6: **if** $x_v \in \omega$ **then** ▷ $x_v \in Z$ and assigned True
7: $v.wt \leftarrow$ computeWeights$(f, w, v.then, \omega, visited)$
8: **else if** $\neg x_v \in \omega$ **then** ▷ $x_v \in Z$ and assigned False
9: $v.wt \leftarrow$ computeWeights$(f, w, v.else, \omega, visited)$
10: **else** ▷ x_v is unassigned
11: $t\_wt \leftarrow$ computeWeights$(f, w, v.then, \omega, visited)$
12: $e\_wt \leftarrow$ computeWeights$(f, w, v.else, \omega, visited)$
13: $v.wt \leftarrow t\_wt \times w(x_v) + e\_wt \times w(x_v)$
14: **return** $v.wt$

drawbacks in practice, in the context of CNF-sampling. In particular, that algorithm traversed the ADD from leaves to root in order to compute the sampling probabilities for each variable. For this, it was necessary to first eliminate all the variables from the input ADD that were already assigned, through an operation called *cofactoring* [207]. Although cofactoring is linear in the size of the ADD in theory, it entails the construction of a separate ADD, which may incur significant overhead in practice. In the present context, this operation would have to be performed hundreds to thousands of times *per sample*, depending on the size of the project-join tree, making sampling expensive and negating the benefits of cost amortization through compilation. We highlight that in the context of trace-sampling, this slow down was vastly overshadowed by the time taken to compile the ADDs. As a result, the performance of TraceSampler was not affected adversely.

We present here a faster top-down algorithm for ADD-sampling. Procedure sampleFromADD (Alg. 19), takes as input an ADD f along with a partial assignment ω to some variables in the support of f, and randomly samples values for the unassigned variables in f's support. In the next section, we show how the same algorithm can be used to sample an assignment from a Tree-of-ADDs recursively.

sampleFromADD first calls procedure computeWeights (line 2 of Alg. 19) for computing the sampling weights for each variable, and then performs a root-to-leaf random walk using the computed weights, sampling values for unassigned variables in the process. We assume that each node v of an ADD has an additional variable $v.wt$, for storing weights.

Procedure computeWeights (Alg. 20) computes, for each node v in an ADD f, the cumulative weight of all the partial assignments in the sub-ADD rooted at v. This cumulative weight is computed recursively using the values of v's children (see Lemma 9.5). This weight is stored in the variable $v.wt$ for retrieval later. Lines 1-3 ensure that each node in the ADD is processed only once, thereby ensuring running time linear in the size of the ADD. If a variable x_v at a node v is already assigned,

then the checks on lines 6 and 8 ensure that only the branch corresponding to the assigned value is explored. If x_v has not been previously assigned then both branches are recursively explored (lines 10-13). In this case, the weight of branch is computed as the weight of the child node scaled by the corresponding literal-weight of x_v.

Lemma 9.5. *Let wt be the return value of* computeWeights *invoked on an ADD* $f = (X, S, \rho, G)$ *with weight function* w*, an unvisited node* v *and an assignment* ω *to the variables* $Z \subseteq X$*. Let* $Y = X \setminus Z$ *be the set of unassigned variables,* $Y_{\geq v} = \{x \in Y \mid \rho(x) \geq \rho(x_v)\}$*, and* $Z_{\geq v} = \{x \in Z \mid \rho(x) \geq \rho(x_v)\}$*. Then we have*

$$wt = \sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\omega_{Z_{\geq v}}] \tag{9.1}$$

The weights computed by computeWeights are used for performing a top-down random walk on the ADD in procedure sampleFromADD in lines 3-27. If the variable x_v at a node v has already been assigned, then in lines 5-8, the appropriate branch is taken. Otherwise, in lines 9-18, a value for x_v is sampled. We assume access to a procedure *weighted_sample* that takes two positive real numbers, say a and b as parameters, and returns a random bit c such that $\Pr[c = true] = \frac{a}{a+b}$. In lines 15 and 18, the appropriate branch is chosen, depending on the value just sampled for x_v. Lines 19-26 sample values for skipped variables between v and the chosen child v_c, using the corresponding literal weights.

Lemma 9.6. *Let* sampleFromADD *be invoked on an ADD* $f = (X, S, \rho, G)$*, weight function* w*, and an assignment* ω *to the variables* $Z \subseteq X$*. Let* $Y = X \setminus Z$ *be the set of unassigned variables. Then* sampleFromADD *returns an assignment* ω' *to the variables in* Y *with probability*

$$\Pr[Y = \omega' | Z = \omega] = \frac{w(\omega') \cdot f[\omega', \omega]}{\sum_Y w(Y) \cdot f[\omega]} \tag{9.2}$$

Algorithm 21 `DPSampler`(X, φ, w, n)

Input: X: A set of variables

Input: φ: A CNF formula over X

Input: w: A literal-weight function over X

Input: n: The number of weighted samples to generate

Output: $\omega_1, \cdots, \omega_n$: for each $1 \leq i \leq n$, ω_i is an independent w-weighted sample of φ.

1: $\mathcal{T} \leftarrow \mathsf{Plan}(\varphi)$ ▷ See Sec. 9.5.1

2: $S \leftarrow \mathsf{Compile}(\mathcal{T})$ ▷ See Sec. 9.5.2

3: **for** each i in $1 \leq i \leq n$ **do**

4: $\quad \omega_i \leftarrow \mathsf{drawSample}(\mathcal{T}, root(\mathcal{T}), S, w, \varnothing)$ ▷ Alg. 22; see Sec. 9.5.3

5: **return** $\omega_1, \cdots, \omega_n$

9.5 Sampling from a Boolean formula

We now present our algorithm `DPSampler`, a three-phase algorithm for exact weighted sampling, in Alg. 21. While we use existing techniques [59] for the first phase (planning), the other phases of `DPSampler` (compilation and sampling) are novel.

First, in the planning phase, a data-structure known as a project-join tree [59] is computed which serves as a blueprint for subsequent computations. Next, in the compilation phase, a Tree-of-ADDs is computed through a sequence of product and additive quantification, as prescribed by the project-join tree. Lastly, in the sampling phase a random assignment to all variables is sampled by recursively invoking the ADD-sampling algorithm from Section 9.4 on each ADD in the tree.

The following theorem asserts the correctness of Alg. 21.

Theorem 9.7. *Let X be a set of Boolean variables, φ be a CNF formula over X, w be a literal-weight function over X, and n be a positive integer. If $\sigma_1, \cdots, \sigma_n$ is the sequence of random assignments returned by* `DPSampler`*(X, φ, w, n), then $\sigma_1, \cdots, \sigma_n$*

are i.i.d. w-weighted samples of φ.

A full proof of Theorem 9.7 appears in the appendix.

9.5.1 Planning

In the planning phase, the goal is to compute a project-join tree from an input CNF formula.

Project-join trees were originally used in [59] as part of a unifying framework called `DPMC` for model counting. The key idea is to represent the model counting computation as a rooted tree, called a *project-join tree*, where leaves correspond to clauses, and internal nodes correspond to projections. Formally:

Definition 9.8 (Project-Join Tree)**.** *Let φ be a CNF formula over a set of variables X. A* project-join tree *of φ is a tuple $\mathcal{T} = (T, r, \gamma, \boldsymbol{X})$ where*

- *T is a tree with root $r \in \mathcal{V}(T)$,*
- *$\gamma : \mathcal{L}(T) \to \varphi$ is a bijection from the leaves of T to the clauses of φ, and*
- *$\boldsymbol{X} : \{X^c \mid \forall c \in \mathcal{V}(T) \setminus \mathcal{L}(T),\ X^c \subseteq X\}$ labels each internal node c with subsets of X.*

$\mathcal{T}$ must satisfy the following two properties.

1. *The set $\boldsymbol{X}$ is a partition of X.*
2. *Let $n \in \mathcal{V}(T)$ be an internal node, x be a variable in $X^n \in \boldsymbol{X}$, and c be a clause of φ. If $x \in \texttt{Vars}(c)$, then the leaf node $\gamma^{-1}(c)$ is a descendant of n.*

The model counting algorithm `DPMC` that was presented by [59] for model counting of an input CNF formula φ is similarly modular to our algorithm and consists of two phases. Firstly, in the planning phases `DPMC` constructs a project-join tree $\mathcal{T}$ of φ. Secondly, in the execution phases `DPMC` computes the model count of φ by traversing $\mathcal{T}$

from leaves to root, multiplying clauses according to the tree structure and additively projecting out variables according to $\mathbf{X}$. We use the same planning phase from `DPMC` as the first phase of `DPSampler`. Since we are not interested in computing the complete model count, we do not need the execution phase of `DPMC` for `DPSampler`.

`DPMC` supports two major ways to construct project-join trees in the planning phase: either from tree-decompositions of the primal graph, or from various heuristics. Since [59] found that tree decompositions of the primal graph were the most efficient technique in practice for the planning phase in the case of model counting, we also use tree decompositions to generate project-join trees in `DPSampler`.

9.5.2 Compilation

In the compilation phase, we assume that a project-join tree $\mathcal{T}$ has already been constructed using any of the methods presented in [59]. Our goal is to construct a a *tree-of-ADDs* using $\mathcal{T}$.

A tree-of-ADDs is a compiled representation of φ from which solutions of φ can be sampled. Formally:

Definition 9.9. *Let $\mathcal{T} = (T, r, \gamma, \boldsymbol{X})$ be a project-join tree and let w be a literal-weight function. A* tree-of-ADDs *for $\mathcal{T}$ is a set of pseudo-Boolean functions $S = \{f^n : n \in \mathcal{V}(T)\}$ defined recursively by, for each $n \in \mathcal{V}(T)$:*

$$f^n \equiv \begin{cases} [\gamma(n)] & \text{if } n \in \mathcal{L}(T) \\ \prod_{c \in C(n)} \sum_{X^c} f^c \cdot w(X^c) & \text{if } n \in \mathcal{V}(T) \setminus \mathcal{L}(T) \end{cases}$$

To ease notation, within Def. 9.9 we define $X_\ell \equiv \varnothing$ for each $\ell \in \mathcal{L}(T)$. Recall that $[\gamma(n)]$ is the pseudo-Boolean function where $[\gamma(n)](\omega) = 1$ if $\omega \models \gamma(n)$ and 0 otherwise.

Note that, in contrast to the w-evaluation of [59] used for model counting, at each internal node $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$ the variables in X^n are not abstracted out at

the function f^n within the tree-of-ADDs. This is because we use f^n in the sampling phase in order to sample values for the variables in X^n.

We define a procedure Compile($\mathcal{T}, w$) following Def. 9.9. Compile takes a project-join tree $\mathcal{T}$ as input and recursively applies product and projection operations to construct a tree-of-ADDs for $\mathcal{T}$. The full algorithm for Compile appears in Appendix C.1. While we represent the functions f^n as ADDs in our implementation, one could in principle use any data-structure that can represent pseudo-Boolean functions and supports product and projection operations (including tensors, as was done in [59].

9.5.3 Sampling

Finally, in the sampling phase we assume that a tree-of-ADDs has been previously constructed. Our goal is to use these ADDs in order to generate samples.

Alg. 22 presents a procedure drawSample that generates samples from a tree-of-ADDs. drawSample is invoked in `DPSampler` with parameters that include an empty assignment ω, and the root node of the project-join tree. A full assignment ω to all variables in X (the variable set of the input formula) is recursively sampled piece-wise by drawSample, through a top-down traversal of the project-join tree. At each node n in the tree, the values for the variables X^n are sampled using the variable values already sampled at the ancestors of n. Since $\mathbf{X} = \{X^c \mid c \in \mathcal{V}(T) \setminus \mathcal{L}(T)\}$ is a partition of X, this samples a value for every variable exactly once.

9.6 Empirical Evaluation

The objective of our empirical study was to answer the following questions

1. How close is the distribution generated by `DPSampler` to that of an ideal sampler?

2. How does the new top-down ADD-sampling algorithm (Alg. 19) perform compared to the bottom-up procedure of [62]?

Algorithm 22 drawSample$(\mathcal{T}, n, S, w, \omega)$

Input: $\mathcal{T} = (T, r, \gamma, \pi)$: A project-join tree

Input: n: A node in $\mathcal{V}(T)$

Input: S: A map from each $n \in \mathcal{V}(T)$ to ADD f^n

Input: w: A literal-weight function

Input: ω: A preexisting assignment to some variables

Output: ω': A sampled assignment to all variables

1: $f^n \leftarrow S[n]$

2: $\omega \leftarrow \omega \cup$ sampleFromADD(f^n, w, ω)

3: **for** $c \in C(n)$ **do**

4: $\quad \omega \leftarrow \omega \cup$ drawSample$(\mathcal{T}, c, S, w, \omega)$

5: **return** ω

3. How does `DPSampler` perform compared to the state-of-the-art sampling tools?

We defer a detailed discussion of results as well as an exposition on question (2) to Appendix C.2. We report that the 'in-place' top-down sampling approach presented in Sec. 9.4 comprehensively outperformed the bottom-up approach of [62].

Experimental Setup We ran all experiments on a high performance cluster. Each experiment had exclusive access to one node comprising of 16 cores (32 threads) with an Intel Xeon E5-2650 v2 processor running at 2.6 GHz, with memory capped at 30 GB. We used GCC 9.4.0 for compiling `DPSampler` with 'Ofast' flag enabled, along with CUDD library version 3.0. `WAPS` as well as `DPSampler` with CUDD are single threaded.

9.6.1 Distribution generated by `DPSampler`

While Thm. 9.7 guarantees that the output of `DPSampler` adheres to the desired distribution, we performed an experiment to compare the actual distribution of samples

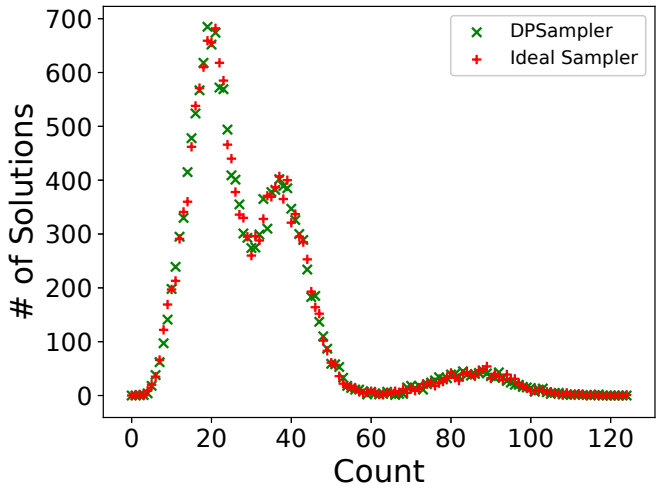


Figure 9.1 : Distribution of Generated Samples

generated by `DPSampler` with that generated by an ideal sampler. For this comparison we selected the unweighted CNF benchmark 'blasted_case110.cnf' with 16384 solutions and added random weights to a small subset of literals. We implemented a simple ideal sampler that first enumerates all solutions of the input CNF and samples assignments proportionate to the weight of each assignment. We generated 5×10^5 samples using both `DPSampler` and the ideal sampler, computed the frequency of occurrence of each of the 16384 assignments, and grouped samples occurring with the same frequency. This setup is similar to that used in previous works on sampling [192, 31]. The result is shown in Fig. 9.1 where a point (x, y) indicates that x distinct traces were generated y times. It can be seen that the distributions generated by `DPSampler` and the ideal sampler are practically indistinguishable, with Jensen-Shannon distance 0.003.

9.6.2 Comparison with State-of-the-Art Tools

We compared the performance of `DPSampler` with the state-of-the-art weighted sampling tool `WAPS` [31], which has been shown to significantly outperform other samplers

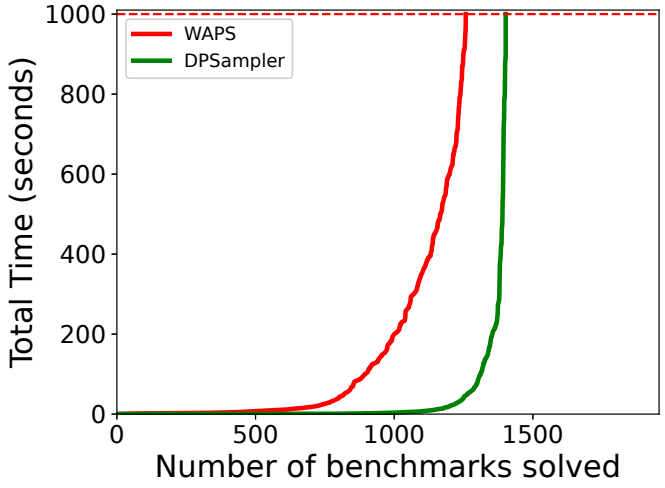


Figure 9.2 : Performance of `WAPS` vs `DPSampler` on all benchmarks

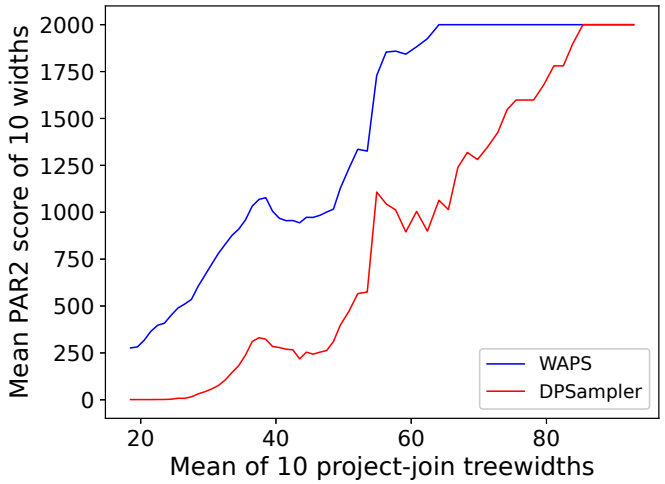


Figure 9.3 : Average PAR2 Score vs. Average Treewidths on Bayes

	Compile Time	Total Time
Bayes	11.86	31.78
Pseudo-weighted	0.89	4.81
All	4.8	15.8

Table 9.1 : Avg. (Geometric Mean) Speedups offered by `DPSampler` over `WAPS`

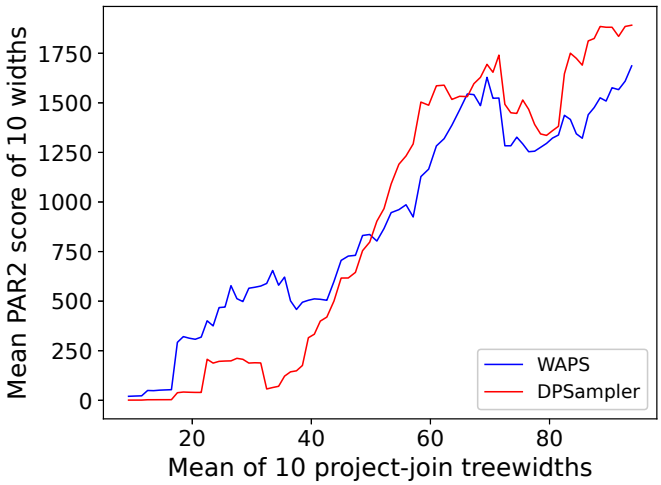


Figure 9.4 : Average PAR2 Score vs. Average Treewidths on Pseudoweighted

(see Sec. 9.3). We use the benchmark sets of weighted CNF formulas "Bayes" and "Pseudo-weighted" for this comparison, which were also used in previous works [49, 59]. A benchmark is considered "solved" by a tool if the tool is able to generate 5000 samples within a timeout of 1000 seconds. We treat both timeouts and memouts as failures.

The results are shown in Fig. 9.2. A point (x, y) in the plot, implies that x instances took less than or equal to y seconds to solve. `DPSampler` is able to solve 144 more instances as compared to `WAPS`. On instances where both `WAPS` and `DPSampler` succeeded, we found the average (geometric mean) speedup offered by `DPSampler` over `WAPS` was a factor of 31 for Bayes benchmarks, and a factor of 4 for Pseudo-weighted (see Tab. 9.1). Further for benchmarks that were solved by at least one tool, `DPSampler` was faster than `WAPS` on all 943 such Bayes benchmarks, while for Pseudo-weighted benchmarks, `DPSampler` was faster on 418 benchmarks, while `WAPS` was faster on 221.

Scalability wrt Treewidths Similar to [199], we plot mean PAR-2 scores (in seconds) against mean project-join tree widths in Fig. 9.3 and 9.4. A point (x, y)

indicates that x is the central moving average of 10 consecutive project-join tree widths $1 \leq w_1 < w_2 < \ldots < w_{10} \leq 99$, and y is the average PAR-2 score of the benchmarks whose project-join trees have widths w s.t. $w_1 \leq w \leq w_{10}$. We observe that the performance of `DPSampler` degrades as the project-join tree width increases in general as expected. Nevertheless, it is the best performer on average, for all treewidths we encountered for the Bayes set, and for treewidths upto roughly 50 for Pseudo-weighted set. Further, on the Pseudoweighted set, from 50 to 100, the performance is only marginally worse than that of `WAPS`.

9.6.3 Discussion

`DPSampler` comprehensively outperforms `WAPS` on Bayes benchmarks, as evident from Fig. C.2 and from the fact that `DPSampler` is more than 31× faster on average on benchmarks that were solved by both. On Pseudoweighted benchmarks, the differences in performance are less pronounced. `WAPS` is able to solve more benchmarks while `DPSampler` is more than 4× faster on average (see Tab. 9.1).

Figs. 9.3 and 9.4 confirm our hypothesis that `DPSampler` performs extremely well in the regime of low treewidths, and is thus a valuable addition to a portfolio of samplers. Surprisingly however, it was also faster than `WAPS` on Bayes benchmarks with treewidths upto 80. This indicates that treewidth alone is not a good predictor of performance, which is similar to what was observed in [59]. It is known that diagram variable order also adversely affects running times of ADD-based algorithms [49]. In the context of a different problem of trace-sampling, [62] reported dynamic variable ordering for ADDs to improve both memory requirements and running times. A comprehensive evaluation of different variable ordering techniques is beyond the scope of this work, but we hypothesize both dynamic variable ordering as well as instance-specific variable ordering to significantly impact the performance of `DPSampler`.

Limitations `WAPS` is based on the d-DNNF compiler `d4`, which is a well-engineered and mature tool written in C++. The code for annotating and sampling the d-DNNF is, however, written in Python, involves expensive disk-reads and writes, and is not as efficient as well-written C++ code. Therefore it is possible that the preceding results may not be truly reflective of the true capabilities of the d-DNNF approach. In order to level the playing field, we also compared the times taken for compiling the d-DNNF and Tree-of-ADDs (including project-join tree construction). The performance of `WAPS` (i.e. `d4`) showed marked improvement, as expected. Nevertheless, `DPSampler` was still competitive, with a 12× avg. speedup on Bayes benchmarks, and was able to compile 25 instances from Pseudoweighted that `d4` timed-out on. We give more details in Appendix C.2.

9.7 Chapter Summary

We presented a modular algorithm, `DPSampler`, for weighted sampling of CNF formulas which leverages and extends the dynamic programming framework of [59]. Our algorithm employs a novel top-down and 'in-place' ADD-sampling sub-procedure that convincingly outperforms the existing bottom-up SAT-based approach. In the broader context of Constrained Sampling, the success of our algorithm demonstrates that SAT-based samplers like `WAPS` may not be the last word, and that ADD-based approaches that exploit factored representations are an important part of the portfolio.

Part IV

Epilogue

Chapter 10

Conclusion

Constrained Counting and Sampling are two fundamental problems intersecting a variety of areas in computer science such as complexity theory, AI and Formal Methods. Research in the theory and practice of these problems has come a long way since the formal definition of the counting class #P by Valiant [9]. Significant advances have been made in both theory and practice, such as the resolution of the dichotomy conjecture for #CSPs [208, 209, 210] and latest iteration of counting and sampling tools scaling to constraints with tens of thousands of variables [35]. Nevertheless, this research effort is splintered across disjoint communities and directions, leading to large gaps in literature. Research into the practice of counting and sampling is narrowly focused on general-purpose approaches, while on the theoretical side there is an overemphasis on establishing polynomial upper bounds at the expense of fine-grained analysis. Consequently, many domain-specific problems are left in no-man's-land: poly-time algorithms from the theory community often fail to scale in practice, while purportedly powerful general-purpose counting and sampling tools are surprisingly unable to exploit the hidden structure in these problems.

The primary contributions of this dissertation are to illuminate this gap in literature, and offer balanced algorithmic solutions for both the exact and approximate counting and sampling problems. Unlike the generic SAT-based approach, we proposed to tailor powerful but flexible techniques to each constraint type instead of tailoring constraints to suit rigid techniques. On the approximation front, our techniques combined the strengths of Monte Carlo and hashing-based frameworks yielding improved theoretical and practical performance over the state-of-the-art in the con-

text of the problems of DNF-Counting and conditional counting. In the context of exact counting and sampling, we showed novel ways of applying an ADD-based counting and sampling framework to the problems of computing the permanent of a 0-1 matrix, sampling traces of a transition system and weighted sampling solutions of low-treewidth CNF formulas. Our works represent a first-step towards the goal of obtaining fast practical algorithms for different constraint types, without sacrificing theoretical guarantees. We are optimistic that this dissertation will serve as a starting point for more research into techniques that balance generality of the constraint types and the specificity of the problem structure.

10.1 Strategies for Domain-Specific Counting and Sampling

In this dissertation, we designed solutions for five domain-specific counting and sampling problems. For this, we tailored general techniques like hashing, and factored ADDs to operate directly on domain-specific constraint types so as to better exploit the problem structure. This yielded significant gains in performance over existing approaches. This is in stark contrast to the general-purpose SAT-based pipeline, where all constraint types are first converted to CNF and then given as input to a SAT-based counter or sampler. The conversion to CNF often leads to loss of structure inherent in the native constraints [45], which hampers the performance of the SAT-based approach. Nevertheless, the translation to CNF can be done mechanically using standard encodings which allows the SAT-based approach to be used as-is without explicit knowledge of its inner workings. In contrast, our approach requires some domain knowledge and familiarity with counting and sampling techniques. This begs the question of how the lessons learnt from our work can be applied to new domain-specific counting and sampling problems, such as counting the number of linear extensions of posets [211]. In this section we outline some high-level strategies that may be useful for this purpose.

A good starting point and baseline for any such endeavor is to use the SAT-based

approach, making full use of the wealth of CNF encodings available in literature (c.f. [23]) as well as experimenting with the various tunable parameters of each tool. In a wide variety of domains, especially those with heterogeneous constraints, the efficiency of modern SAT-based counters and samplers cannot be overstated [35]. Given that the startup cost of using this pipeline is low and the potential benefits are high, it is advisable to start here.

If the resulting performance is not acceptable, the next step can be to tailor other existing techniques to the specific problem. We highlight that we found it best to meld a general flexible technique having high-performance implementations with known problem-specific approaches. The benefits of doing this are twofold. Using a flexible technique with mature implementations ensures that there are no low-level inefficiencies that often plague new hand-crafted code. Leveraging problem-specific approaches allows the exploitation of the hidden structure in each instance. We found this strategy to be fruitful for both approximate and exact scenarios. On the approximate side, we melded the techniques of hashing and Monte Carlo to achieve better performance for both DNF-Counting and conditional counting. On the exact side we melded the general technique of factored ADDs with problem-specific approaches viz., Ryser's formula for the matrix permanent, iterative-squaring for trace sampling and tree decomposition tools for low-treewidth CNF sampling.

Finally, it may be necessary to make low-level algorithmic improvements to fully realize the gains from the previous step. For example, in the case of trace sampling, we augmented the naive iterative squaring procedure with pruning (encapsulated in the procedure computeReachableSets), while the novel top-down ADD sampling subprocedure was crucial for `DPSampler`. In each of these cases, practice informed the theory, i.e., we identified practical inefficiencies in the algorithm through experimentation and devised new techniques to circumvent the bottlenecks. It is worth noting that worst-case complexity analysis alone would have been insufficient to illuminate these issues.

10.2 Future Work

We now discuss potential research directions that can potentially yield further improvements in both approximate and exact scenarios.

10.2.1 Approximate Counting and Sampling

In our works we enhanced and extended the techniques of naive Monte Carlo sampling and hashing for DNF-Counting and conditional counting. It will be interesting to see if the powerful technique of Markov Chain Monte Carlo (MCMC) can yield similar benefits for other approximation problems. The theory behind MCMC has been extensively researched [212], and majority of polynomial-time approximation algorithms from the theory community are based on it. However, it is known that these algorithms often do not scale well in practice [211, 17]. Generic MCMC-based approaches like Gibbs sampling that do scale well often give up theoretical guarantees in the process. Thus more work needs to be done to achieve scalability without losing PAC-style guarantees.

DNF-Counting

The Reverse Search technique presented in this work is an enhancement of the hashing-based framework and is not tied to the DNF constraint language in any way. For general CNF benchmarks, the number of XOR constraints required is typically too low to justify searching in the reverse direction. Nevertheless, with increasing applications of approximate counting, it would be interesting to investigate other potential application domains where reverse search could yield theoretical or practical improvements.

To the best of our knowledge, our work was the first explicit investigation into the practical aspects of DNF-Counting. It would be interesting to see how our results inform the design choices for future algorithms on DNF-Counting, especially once real-world benchmarks from applications areas like probabilistic databases become

available.

Conditional Counting and Explainable AI

The need for making explainable AI more rigorous and evidence-based has been highlighted in the past [213], and we believe our constraint-driven framework takes a concrete step in this direction. Our tool, CLIME, can also be readily extended in numerous ways. Helping the user with defining relevant subspaces by mining constraints from data is an interesting direction. Richer constraint languages like SMT [214] can provide even more flexibility, once sampling technology matures. Construction of CLIME's explainer model can also potentially be extended to incorporate Shapley values as in [96]. It will also be interesting to investigate other applications of our approximate conditional counting algorithm such as in the areas of verification and probabilistic inference.

10.2.2 Exact Counting and Sampling

We leveraged factored ADDs to tackle the problems of computing the matrix permanent, sampling traces of a transition system and weighted sampling from low-treewidth CNF formulas. It will be interesting to explore what other general techniques besides factored ADDs can lend themselves to exact domain-specific problems. For instance, Affine Algebraic Decision Diagrams (AADDs) [215] offer better compression compared to ADDs with little overhead. However, currently there is a lack of mature libraries implementing AADDs. Another potential direction is to extend existing tools from Constraint Programming to counting and sampling. Solvers from the CP community such as Picat [216] and Gecode [217] offer a lot of flexibility in tuning the constraint solving framework to each constraint type, and are not limited to CNF formulas. This framework, once extended to exhaustively explore the solution space, is thus likely to be beneficial for domain-specific counting and sampling as well.

0-1 Matrix Permanent

Our algorithm for the matrix permanent was shown to scale on matrices with 'similar rows', which includes both dense and sparse matrices as special cases. It is an interesting open problem to obtain a complete characterization of the class of matrices for which ADD representation of Ryser's formula is succinct. Our experimental results for dense matrices hint at the possibility of improved theoretical bounds similar to those obtained in earlier work on sparse matrices. Developing an algorithm for general matrices that is exponentially faster than Ryser's approach remains a long-standing open problem [151], and obtaining better bounds for non-sparse matrices would be an important first step in this direction. For sparse matrices, using a factored representation was crucial to the success of our approach. In fact, it may be possible to optimize the algorithm in this regime even further, by evaluating other heuristics used in [49].

Sampling Traces of a Transition System

Our experience indicates that there is significant potential to improve the performance of TraceSampler through engineering optimizations, as ADDs offer a lot of trade-offs between space and time. For instance, in out current implementation, all the ADDs are part of a single manager since it saves space by enabling sharing of nodes. However, this makes it harder to find a good variable order, and we observed that most of the time spent during ADD construction was on dynamic variable reordering. Another interesting direction for further research is to combine the strengths of decision diagram based techniques (like TraceSampler) with SAT-based techniques (like UniGen2) to build trace samplers that have the best of both worlds.

Weighted Sampling of Low-Treewidth CNF Formulas

Sampling plays a crucial role in many applications in AI and Machine Learning. In this context, the speedups offered by `DPSampler` over the current state-of-the-

art sampler, especially on benchmarks encoding Bayesian networks, are encouraging and warrant further investigation. The Tree-of-ADDs datastructure compiled by `DPSampler` may be useful in other applications of Knowledge Compilation as well, and a full investigation as in [131] is likely to be fruitful.

Bibliography

[1] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of reasoning*, pp. 466–483, Springer, 1983.

[2] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, 1971.

[3] A. Darwiche, "The quest for efficient probabilistic inference," *Invited Talk, IJCAI-05*, 2005.

[4] F. Biondi, M. Enescu, A. Heuser, A. Legay, K. S. Meel, and J. Quilbeuf, "Scalable approximation of quantitative information flow in programs," in *Proc. of VMCAI*, 1 2018.

[5] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," *The VLDB Journal*, vol. 16, no. 4, pp. 523–544, 2007.

[6] D. R. Karger, "A Randomized Fully Polynomial Time Approximation Scheme for the All-Terminal Network Reliability Problem," *SIAM Review*, 2001.

[7] S. Arora and B. Barak, *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[8] S. Toda, "Pp is as hard as the polynomial-time hierarchy," *SIAM Journal on Computing*, vol. 20, no. 5, pp. 865–877, 1991.

[9] L. G. Valiant, "The complexity of computing the permanent," *Theoretical computer science*, vol. 8, no. 2, pp. 189–201, 1979.

[10] J.-Y. Cai and X. Chen, *Complexity dichotomies for counting problems: Volume 1, Boolean domain*. Cambridge University Press, 2017.

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

[12] M. Sipser, "A complexity theoretic approach to randomness," in *Proc. of the 15th Annual ACM Symposium on Theory of Computing*, pp. 330–335, 1983.

[13] L. Stockmeyer, "The complexity of approximate counting," in *Proceedings of the Annual ACM Symposium on Theory of Computing*, pp. 118–126, 1983.

[14] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani, "Random generation of combinatorial structures from a uniform distribution," *Theoretical computer science*, vol. 43, pp. 169–188, 1986.

[15] M. Bellare, O. Goldreich, and E. Petrank, "Uniform generation of np-witnesses using an np-oracle," *Information and Computation*, vol. 163, no. 2, pp. 510–526, 2000.

[16] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable and nearly uniform generator of sat witnesses," in *International Conference on Computer Aided Verification*, pp. 608–623, Springer, 2013.

[17] J. Newman, *FPRAS Approximation of the Matrix Permanent in Practice*. PhD thesis, RICE UNIVERSITY, 2020.

[18] F. Bacchus, S. Dalmao, and T. Pitassi, "Algorithms and complexity results for# sat and bayesian inference," in *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pp. 340–351, IEEE, 2003.

[19] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 26th ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 224–234, 2018.

[20] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI magazine*, vol. 28, no. 3, pp. 13–13, 2007.

[21] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, 2001.

[22] J. P. M. Silva and K. A. Sakallah, "Graspa new search algorithm for satisfiability," in *The Best of ICCAD*, pp. 73–89, Springer, 2003.

[23] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*, vol. 185. IOS press, 2009.

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

[25] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi, "Combining component caching and clause learning for effective model counting.," *SAT*, vol. 4, p. 7th, 2004.

[26] A. Darwiche, "New advances in compiling cnf to decomposable negation normal form," in *Proc. of ECAI*, pp. 328–332, Citeseer, 2004.

[27] J.-M. Lagniez and P. Marquis, "An improved decision-DNNF compiler," in *IJCAI*, pp. 667–673, 2017.

[28] S. Sharma, S. Roy, M. Soos, and K. S. Meel, "Ganak: A scalable probabilistic exact model counter.," in *IJCAI*, vol. 19, pp. 1169–1176, 2019.

[29] D. Achlioptas, Z. S. Hammoudeh, and P. Theodoropoulos, "Fast sampling of perfectly uniform satisfying assignments," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 135–147, Springer, 2018.

[30] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, "Knowledge compilation meets uniform sampling.," in *LPAR*, pp. 620–636, 2018.

[31] R. Gupta, S. Sharma, S. Roy, and K. S. Meel, "WAPS: Weighted and projected sampling," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4 2019.

[32] L. G. Valiant, "A theory of the learnable," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pp. 436–445, ACM, 1984.

[33] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings* (C. Schulte, ed.), vol. 8124 of *Lecture Notes in Computer Science*, pp. 200–216, Springer, 2013.

[34] M. Soos and K. S. Meel, "BIRD: Engineering an efficient CNF-XOR SAT Solver and its applications to approximate model counting," in *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 1 2019.

[35] M. Soos, S. Gocht, and K. S. Meel, "Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling," in *International Conference on Computer Aided Verification*, pp. 463–484, Springer, 2020.

[36] K. S. Meel and S. Akshay, "Sparse hashing for scalable approximate model counting: theory and practice," in *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 728–741, 2020.

[37] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pp. 244–257, 2009.

[38] J. Rintanen, "Engineering efficient planners with sat," in *ECAI 2012*, pp. 684–689, IOS Press, 2012.

[39] P. Großmann, S. Hölldobler, N. Manthey, K. Nachtigall, J. Opitz, and P. Steinke, "Solving periodic event scheduling problems with sat," in *International conference on industrial, engineering and other applications of applied intelligent systems*, pp. 166–175, Springer, 2012.

[40] A. Biere and K. Claessen, "Hardware model checking competition," in *Hardware Verification Workshop*, 2010.

[41] C. R. Morris and C. H. Ferguson, "How architecture wins technology wars.," *Harvard Business Review*, vol. 71, no. 2, pp. 86–96, 1993.

[42] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1167–1176, 1996.

[43] E. Freuder, "In pursuit of the holy grail," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4es, pp. 63–es, 1996.

[44] M. Jerrum, A. Sinclair, and E. Vigoda, "A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries," *Journal of the ACM (JACM)*, vol. 51, no. 4, pp. 671–697, 2004.

[45] C. Thiffault, F. Bacchus, and T. Walsh, "Solving non-clausal formulas with dpll search," in *International Conference on Principles and Practice of Constraint Programming*, pp. 663–678, Springer, 2004.

[46] N.-F. Zhou, M. Tsuru, and E. Nobuyama, "A comparison of cp, ip, and sat solvers through a common interface," in *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, vol. 1, pp. 41–48, IEEE, 2012.

[47] S. Abed, A. A. Abdelaal, M. H. Al-Shayeji, and I. Ahmad, "Sat-based and cp-based declarative approaches for top-rank-k closed frequent itemset mining," *International Journal of Intelligent Systems*, vol. 36, no. 1, pp. 112–151, 2021.

[48] D. Suciu, D. Olteanu, C. Ré, and C. Koch, "Probabilistic databases," *Synthesis Lectures on Data Management*, vol. 3, no. 2, pp. 1–180, 2011.

[49] J. M. Dudek, V. H. N. Phan, and M. Y. Vardi, "ADDMC: weighted model counting with algebraic decision diagrams," in *AAAI*, vol. 34, pp. 1468–1476, 2020.

[50] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," *Form Method Syst Des*, vol. 10, no. 2-3, pp. 171–206, 1997.

[51] T.-C. Wei and S. Severini, "Matrix permanent and quantum entanglement of permutation invariant states," *Journal of Mathematical Physics*, vol. 51, no. 9, p. 092203, 2010.

[52] H. Liang and F. Bai, "A partially structure-preserving algorithm for the permanents of adjacency matrices of fullerenes," *Computer physics communications*, vol. 163, no. 2, pp. 79–84, 2004.

[53] J.-C. Régin, "A filtering algorithm for constraints of difference in CSPs," in *AAAI*, vol. 94, pp. 362–367, 1994.

[54] A. Zanarini and G. Pesant, "Solution counting algorithms for constraint-centered search heuristics," *Constraints*, vol. 14, no. 3, pp. 392–413, 2009.

[55] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, pp. 1–6, 2015.

[56] `https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf`. Accessed: 2020-05-17.

[57] S. Iman and S. Joshi, *The e hardware verification language*. Springer Science & Business Media, 2007.

[58] A. Yehia, "The top most common systemverilog constrained random gotchas," in *Proceedings of 2014 Design and Verification Conference and Exhibition United States (DVCon 2014)*, 2014.

[59] J. M. Dudek, V. H. Phan, and M. Y. Vardi, "Dpmc: Weighted model counting by dynamic programming on project-join trees," in *International Conference on Principles and Practice of Constraint Programming*, pp. 211–230, Springer, 2020.

[60] K. S. Meel, A. A. Shrotri, and M. Y. Vardi, "Not all fprass are equal: demystifying fprass for dnf-counting," *Constraints*, vol. 24, no. 3, pp. 211–233, 2019.

[61] S. Chakraborty, A. A. Shrotri, and M. Y. Vardi, "On symbolic approaches for computing the matrix permanent," in *International Conference on Principles and Practice of Constraint Programming*, pp. 71–90, Springer, 2019.

[62] S. Chakraborty, A. A. Shrotri, and M. Y. Vardi, "On Uniformly Sampling Traces of a Transition System," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2020.

[63] B. Courcelle, "Graph rewriting: An algebraic and logic approach," in *Formal Models and Semantics*, pp. 193–242, Elsevier, 1990.

[64] M. Jerrum and A. Sinclair, "The markov chain monte carlo method: an approach to approximate counting and integration," *Approximation Algorithms for NP-hard problems, PWS Publishing*, 1996.

[65] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan, "An introduction to mcmc for machine learning," *Machine learning*, vol. 50, no. 1, pp. 5–43, 2003.

[66] D. Roth, "On the hardness of approximate reasoning," *Artificial Intelligence*, vol. 82, no. 1-2, pp. 273–302, 1996.

[67] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proc. of STOC*, pp. 106–112, ACM, 1977.

[68] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls," in *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 7 2016.

[69] L. Babai, "Monte-carlo algorithms in graph isomorphism testing," *Université tde Montréal Technical Report, DMS*, pp. 79–10, 1979.

[70] R. Motwani and P. Raghavan, *Randomized algorithms*. Cambridge university press, 1995.

[71] P. Dagum, R. Karp, M. Luby, and S. Ross, "An optimal algorithm for Monte Carlo estimation," *SIAM Journal on Computing*, vol. 29, no. 5, pp. 1484–1496, 2000.

[72] R. Karp, M. Luby, and N. Madras, "Monte Carlo approximation algorithms for enumeration problems," *Journal of Algorithms*, vol. 10, no. 3, pp. 429–448, 1989.

[73] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2013.

[74] K. S. Meel, A. A. Shrotri, and M. Y. Vardi, "On hashing-based approaches to approximate DNF-counting," in *Proc. of FSTTCS*, 12 2017.

[75] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman, "Taming the curse of dimensionality: Discrete integration by hashing and optimization," in *Proc. of ICML*, pp. 334–342, 2013.

[76] K. S. Meel, "Constrained counting and sampling: bridging the gap between theory and practice," *arXiv preprint arXiv:1806.02239*, 2018.

[77] M. Luby and B. Veličkovic, "On deterministic approximation of DNF," in *Proc. of STOC*, pp. 430–438, ACM, 1991.

[78] L. Trevisan, "A note on approximate counting for k-DNF," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pp. 417–425, Springer, 2004.

[79] P. Gopalan, R. Meka, and O. Reingold, "DNF sparsification and a faster deterministic counting algorithm," *Computational Complexity*, 2013.

[80] M. Ajtai and A. Wigderson, "Deterministic simulation of probabilistic constant depth circuits," in *Proc. of FOCS*, pp. 11–19, IEEE, 1985.

[81] N. Nisan, "Pseudorandom bits for constant depth circuits," *Combinatorica*, vol. 11, no. 1, pp. 63–70, 1991.

[82] A. De, O. Etesami, L. Trevisan, and M. Tulsiani, "Improved pseudorandom generators for depth 2 circuits," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pp. 504–517, Springer, 2010.

[83] D. Olteanu, J. Huang, and C. Koch, "Approximate confidence computation in probabilistic databases," in *ICDE*, pp. 145–156, IEEE, 2010.

[84] R. Fink and D. Olteanu, "On the optimal approximation of queries using tractable propositional languages," in *Proc. of ICDT*, ACM, 2011.

[85] W. Gatterbauer and D. Suciu, "Oblivious bounds on the probability of Boolean functions," *ACM TODS*, vol. 39, no. 1, p. 5, 2014.

[86] Q. Tao, S. Scott, N. Vinodchandran, and T. T. Osugi, "SVM-based generalized multiple-instance learning via approximate box counting," in *Proceedings of the twenty-first international conference on Machine learning*, p. 101, ACM, 2004.

[87] M. Albrecht and G. Bard, *The M4RI Library – Version 20121224*, 2012.

[88] J. Huang, L. Antova, C. Koch, and D. Olteanu, "MayBMS: a probabilistic database management system," in *Proc. of SIGMOD*, ACM, 2009.

[89] "TPC Benchmark H." `http://www.tpc.org/`.

[90] D. Mitchell, B. Selman, and H. Levesque, "Hard and easy distributions of SAT problems," in *Proc. of AAAI*, pp. 459–465, 1992.

[91] X. Hu, C. Rudin, and M. Seltzer, "Optimal sparse decision trees," in *Advances in Neural Information Processing Systems 32*, pp. 7267–7275, Curran Associates, Inc., 2019.

[92] E. Angelino, N. Larus-Stone, D. Alabi, M. Seltzer, and C. Rudin, "Learning certifiably optimal rule lists for categorical data," *Journal of Machine Learning Research*, vol. 18, pp. 234:1–234:78, 2018.

[93] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," 2019.

[94] F. Avellaneda, "Efficient inference of optimal decision trees," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020,*

The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, pp. 3195–3202, AAAI Press, 2020.

[95] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why should I trust you?": Explaining the predictions of any classifier," in *KDD*, pp. 1135–1144, 2016.

[96] S. M. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *NIPS*, pp. 4765–4774, 2017.

[97] A. Adadi and M. Berrada, "Peeking inside the black-box: A survey on explainable artificial intelligence (XAI)," *IEEE Access*, vol. 6, pp. 52138–52160, 2018.

[98] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 3145–3153, PMLR, 2017.

[99] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *CoRR*, vol. abs/1312.6034, 2013.

[100] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I* (D. J. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), vol. 8689 of *Lecture Notes in Computer Science*, pp. 818–833, Springer, 2014.

[101] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017* (D. Precup and

Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 3319–3328, PMLR, 2017.

[102] S. Wachter, B. Mittelstadt, and C. Russell, "Counterfactual explanations without opening the black box: Automated decisions and the gdpr," *Harv. JL & Tech.*, vol. 31, p. 841, 2017.

[103] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," *Advances in neural information processing systems*, vol. 32, p. 9240, 2019.

[104] D. Slack, S. Hilgard, E. Jia, S. Singh, and H. Lakkaraju, "Fooling LIME and SHAP: Adversarial attacks on post hoc explanation methods," in *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, AIES '20, (New York, NY, USA), p. 180186, Association for Computing Machinery, 2020.

[105] G. Van den Broeck, A. Lykov, M. Schleich, and D. Suciu, "On the tractability of shap explanations," in *Proceedings of AAAI*, 2021.

[106] M. Arenas, P. Barceló, L. Bertossi, and M. Monet, "The tractability of shap-score-based explanations for classification over deterministic and decomposable boolean circuits," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 6670–6678, 2021.

[107] D. Deutch and N. Frost, "Constraints-based explanations of classifications," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 530–541, IEEE, 2019.

[108] M. Cadoli and A. Schaerf, "Compiling problem specifications into sat," *Artificial Intelligence*, vol. 162, no. 1-2, pp. 89–120, 2005.

[109] M. T. Ribeiro, S. Singh, and C. Guestrin, "Anchors: High-precision model-agnostic explanations," in *AAAI*, 2018.

[110] M. Chavira and A. Darwiche, "On probabilistic inference by weighted model counting," *Artificial Intelligence*, vol. 172, no. 6-7, pp. 772–799, 2008.

[111] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Form. Methods Syst. Des.*, vol. 19, p. 734, July 2001.

[112] S. Chakraborty, D. Fried, K. S. Meel, and M. Y. Vardi, "From weighted to unweighted model counting," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[113] S. Moro, P. Cortez, and P. Rita, "A data-driven approach to predict the success of bank telemarketing," *Decision Support Systems*, vol. 62, pp. 22–31, 2014.

[114] J. Amann, A. Blasimme, E. Vayena, D. Frey, and V. I. Madai, "Explainability for artificial intelligence in healthcare: a multidisciplinary perspective," *BMC Medical Informatics and Decision Making*, vol. 20, no. 1, pp. 1–9, 2020.

[115] N. Narodytska, A. A. Shrotri, K. S. Meel, A. Ignatiev, and J. Marques-Silva, "Assessing heuristic machine learning explanations with model counting," in *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings* (M. Janota and I. Lynce, eds.), vol. 11628 of *Lecture Notes in Computer Science*, pp. 267–278, Springer, 2019.

[116] R. Kohavi, "Scaling up the accuracy of naive-Bayes classifiers: A decision-tree hybrid," in *KDD*, pp. 202–207, 1996.

[117] R. C. Fong and A. Vedaldi, "Interpretable explanations of black boxes by meaningful perturbation," in *Proceedings of the IEEE international conference on computer vision*, pp. 3429–3437, 2017.

[118] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[119] H. N. Temperley and M. E. Fisher, "Dimer problem in statistical mechanics-an exact result," *Philosophical Magazine*, vol. 6, no. 68, pp. 1061–1063, 1961.

[120] P. W. Kasteleyn, "The statistics of dimers on a lattice: I. the number of dimer arrangements on a quadratic lattice," *Physica*, vol. 27, no. 12, pp. 1209–1225, 1961.

[121] E. Birnbaum and E. L. Lozinskii, "The good old davis-putnam procedure helps counting models," *Journal of Artificial Intelligence Research*, vol. 10, pp. 457–477, 1999.

[122] A. Darwiche and P. Marquis, "A knowledge compilation map," *Journal of Artificial Intelligence Research*, vol. 17, pp. 229–264, 2002.

[123] H. Ryser, "Combinatorial mathematics, the carus mathematical monographs," *Math. Assoc. Amer*, vol. 4, 1963.

[124] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE TC*, vol. 100, no. 8, pp. 677–691, 1986.

[125] A. Darwiche, "On the tractable counting of theory models and its application to truth maintenance and belief revision," *Journal of Applied Non-Classical Logics*, vol. 11, no. 1-2, pp. 11–34, 2001.

[126] J. Huang and A. Darwiche, "The language of search," *Journal of Artificial Intelligence Research*, vol. 29, pp. 191–219, 2007.

[127] C. Muise, S. A. McIlraith, J. C. Beck, and E. Hsu, "DSHARP: Fast d-DNNF Compilation with sharpSAT," in *Canadian Conference on Artificial Intelligence*, 2012.

[128] M. Chavira and A. Darwiche, "Compiling Bayesian networks using variable elimination," in *IJCAI*, pp. 2443–2449, 2007.

[129] V. Gogate and P. Domingos, "Approximation by Quantization," in *UAI*, pp. 247–255, 2011.

[130] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, "SPUDD: stochastic planning using decision diagrams," in *UAI*, pp. 279–288, 1999.

[131] H. Fargier, P. Marquis, A. Niveau, and N. Schmidt, "A knowledge compilation map for ordered real-valued decision diagrams," in *AAAI*, 2014.

[132] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebric decision diagrams and their applications," *Formal methods in system design*, vol. 10, no. 2-3, pp. 171–206, 1997.

[133] C. Boutilier, T. Dean, and S. Hanks, "Decision-theoretic planning: Structural assumptions and computational leverage," *Journal of Artificial Intelligence Research*, vol. 11, pp. 1–94, 1999.

[134] S. Subbarayan, "Integrating csp decomposition techniques and bdds for compiling configuration problems," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pp. 351–365, Springer, 2005.

[135] S. Subbarayan, L. Bordeaux, and Y. Hamadi, "Knowledge compilation properties of tree-of-bdds," in *AAAI*, pp. 502–507, 2007.

[136] H. Fargier and P. Marquis, "Knowledge compilation properties of trees-of-bdds, revisited," in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[137] C. Wallace, K. B. Korb, and H. Dai, "Causal discovery via mml," in *ICML*, vol. 96, pp. 516–524, 1996.

[138] Y. Huo, H. Liang, S.-Q. Liu, and F. Bai, "Computing monomer-dimer systems through matrix permanent," *Physical Review E*, vol. 77, no. 1, p. 016706, 2008.

[139] L. Valiant, "The complexity of enumeration and reliability problems," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 410–421, 1979.

[140] J.-Y. Cai, A. Pavan, and D. Sivakumar, "On the hardness of permanent," in *Annual Symposium on Theoretical Aspects of Computer Science*, pp. 90–99, Springer, 1999.

[141] H. Dell, T. Husfeldt, D. Marx, N. Taslaman, and M. Wahlén, "Exponential time complexity of the permanent and the tutte polynomial," *ACM Transactions on Algorithms (TALG)*, vol. 10, no. 4, p. 21, 2014.

[142] H. W. Kroto, J. R. Heath, S. C. O'Brien, R. F. Curl, and R. E. Smalley, "C60: Buckminsterfullerene," *Nature*, vol. 318, no. 6042, p. 162, 1985.

[143] G. G. Cash, "A fast computer algorithm for finding the permanent of adjacency matrices," *Journal of mathematical chemistry*, vol. 18, no. 2, pp. 115–119, 1995.

[144] Q. Chou, H. Liang, and F. Bai, "Computing the permanental polynomial of the high level fullerene C70 with high precision," *MATCH Commun. Math. Comput. Chem*, vol. 73, pp. 327–336, 2015.

[145] G. Pesant, C.-G. Quimper, and A. Zanarini, "Counting-based search: Branching heuristics for constraint satisfaction problems," *Journal of Artificial Intelligence Research*, vol. 43, pp. 173–210, 2012.

[146] A. Nijenhuis and H. S. Wilf, *Combinatorial algorithms: for computers and calculators*. Elsevier, 2014.

[147] N. Robertson and P. D. Seymour, "Graph minors. iii. planar tree-width," *Journal of Combinatorial Theory, Series B*, vol. 36, no. 1, pp. 49–64, 1984.

[148] B. Courcelle, J. Engelfriet, and G. Rozenberg, "Handle-rewriting hypergraph grammars," *Journal of computer and system sciences*, vol. 46, no. 2, pp. 218–270, 1993.

[149] B. Courcelle, J. A. Makowsky, and U. Rotics, "On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic," *Discrete Applied Mathematics*, vol. 108, no. 1-2, pp. 23–52, 2001.

[150] R. A. Servedio and A. Wan, "Computing sparse permanents faster," *Information Processing Letters*, vol. 96, no. 3, pp. 89–92, 2005.

[151] T. Izumi and T. Wadayama, "A new direction for counting perfect matchings," in *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pp. 591–598, IEEE, 2012.

[152] B. Yue, H. Liang, and F. Bai, "Improved algorithms for permanent and permanental polynomial of sparse graph," *MATCH Commun. Math. Comput. Chem*, vol. 69, pp. 831–842, 2013.

[153] A. Z. Broder, "How hard is it to marry at random?(on the approximation of the permanent)," in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pp. 50–58, ACM, 1986.

[154] M. Grohe, "Descriptive and parameterized complexity," in *International Workshop on Computer Science Logic*, pp. 14–31, Springer, 1999.

[155] F. Somenzi, "CUDD package, release 2.4.1." `http://vlsi.colorado.edu/~fabio/CUDD/`.

[156] T. van Dijk and J. van de Pol, "Sylvan: multi-core framework for decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 6, pp. 675–696, 2017.

[157] M. Fujita, P. C. McGeer, and J.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *Formal methods in system design*, vol. 10, no. 2, pp. 149–169, 1997.

[158] Y. Okamoto, R. Uehara, and T. Uno, "Counting the number of matchings in chordal and chordal bipartite graph classes," in *International Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 296–307, Springer, 2009.

[159] S. H. Sæther, J. A. Telle, and M. Vatshelle, "Solving #SAT and MaxSAT by dynamic programming," *Journal of Artificial Intelligence Research*, vol. 54, pp. 59–82, 2015.

[160] J. Brault-Baron, F. Capelli, and S. Mengel, "Understanding model counting for β-acyclic CNF-formulas," *arXiv preprint arXiv:1405.6043*, 2014.

[161] E. Bax and J. Franklin, "A permanent algorithm with exp [(n1/3/2ln (n))] expected speedup for 0-1 matrices," *Algorithmica*, vol. 32, no. 1, pp. 157–162, 2002.

[162] H. Liang, S. Huang, and F. Bai, "A hybrid algorithm for computing permanents of sparse matrices," *Applied mathematics and computation*, vol. 172, no. 2, pp. 708–716, 2006.

[163] G. Pan and M. Y. Vardi, "Search vs. symbolic techniques in satisfiability solving," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 235–250, Springer, 2004.

[164] R. Dechter, "Bucket elimination: A unifying framework for reasoning," *Artificial Intelligence*, vol. 113, no. 1-2, pp. 41–85, 1999.

[165] F. Bouquet, *Gestion de la dynamicité et énumération d'impliquants premiers: une approche fondée sur les Diagrammes de Décision Binaire.* PhD thesis, Aix-Marseille 1, 1999.

[166] R. E. Tarjan and M. Yannakakis, "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs," *SIAM Journal on computing*, vol. 13, no. 3, pp. 566–579, 1984.

[167] A. M. Koster, H. L. Bodlaender, and S. P. Van Hoesel, "Treewidth: computational experiments," *Electronic Notes in Discrete Mathematics*, vol. 8, pp. 54–57, 2001.

[168] D. E. Knuth, "Generating all n-tuples," *The Art of Computer Programming*, vol. 4, 2004.

[169] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," in *CP*, pp. 827–831, 2005.

[170] I. P. Gent and P. Nightingale, "A new encoding of alldifferent into SAT," in *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pp. 95–110, 2004.

[171] C. Ansótegui and F. Manya, "Mapping problems with finite-domain variables to problems with boolean variables," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 1–15, Springer, 2004.

[172] T. Ogawa, Y. Liu, R. Hasegawa, M. Koshimura, and H. Fujita, "Modulo based CNF encoding of cardinality constraints and its application to MaxSAT solvers," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pp. 9–17, IEEE, 2013.

[173] R. Martins, S. Joshi, V. Manquinho, and I. Lynce, "Incremental cardinality constraints for MaxSAT," in *International Conference on Principles and Practice of Constraint Programming*, pp. 531–548, Springer, 2014.

[174] J. Burchard, T. Schubert, and B. Becker, "Laissez-faire caching for parallel #SAT solving," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 46–61, Springer, 2015.

[175] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, pp. 428–437, 2018.

[176] J. E. Hopcroft and R. M. Karp, "An n^5/2 algorithm for maximum matchings in bipartite graphs," *SIAM Journal on computing*, vol. 2, no. 4, pp. 225–231, 1973.

[177] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[178] L. Wang, H. Liang, F. Bai, and Y. Huo, "A load balancing strategy for parallel computation of sparse permanents," *Numerical Linear Algebra with Applications*, vol. 19, no. 6, pp. 1017–1030, 2012.

[179] L. Bening and H. Foster, *Principles of verifiable RTL design*. Springer, 2001.

[180] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based verification*. Springer Science & Business Media, 2006.

[181] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," *IEEE Design & Test of Computers*, vol. 24, no. 2, pp. 0112–122, 2007.

[182] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *2007 International Conference on Computer-Aided Design, ICCAD 2007, San Jose, CA, USA, November 5-8, 2007* (G. G. E. Gielen, ed.), pp. 258–265, IEEE Computer Society, 2007.

[183] H. M. Benito, M. Boreale, D. Gorla, and D. Clark, "Output sampling for output diversity in automatic unit test generation," *IEEE Transactions on Software Engineering*, 2020.

[184] `https://www.systemverilog.io/randomization`. Accessed: 2020-05-17.

[185] C. Spear, "Randomization," in *System Verilog for Verification*, pp. 161–216, Springer, 2008.

[186] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-based design*. Springer Science & Business Media, 2004.

[187] N/A, "Ieee standard for property specification language (psl)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.

[188] B. Wile, J. Goss, and W. Roesner, *Comprehensive functional verification: The complete industry cycle*. Morgan Kaufmann, 2005.

[189] G. Hamon, L. De Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.*, pp. 261–270, IEEE, 2004.

[190] M. Arenas, L. A. Croquevielle, R. Jayaram, and C. Riveros, "Efficient logspace classes for enumeration, counting, and uniform generation," in *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pp. 59–73, 2019.

[191] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Balancing scalability and uniformity in sat witness generator," in *2014 51st acm/edac/ieee design automation conference (dac)*, pp. 1–6, IEEE, 2014.

[192] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform sat witness generation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 304–319, Springer, 2015.

[193] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10/sup 20/states and beyond," in *Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pp. 428–429, 1990.

[194] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking," in *27th ACM/IEEE Design Automation Conference*, pp. 46–51, IEEE, 1990.

[195] O. Coudert, J. C. Madre, and C. Berthet, "Verifying temporal properties of sequential machines without building their state diagrams," in *International Conference on Computer Aided Verification*, pp. 23–32, Springer, 1990.

[196] F. Brglez, D. Bryan, and K. Kozminski, "Notes on the iscas'89 benchmark circuits," *North-Carolina State University*, 1989.

[197] `http://fmv.jku.at/hwmcc17/Biere-HWMCC17-talk.pdf`. Accessed: 2020-05-17.

[198] J. M. Dudek, L. Dueñas-Osorio, and M. Y. Vardi, "Efficient contraction of large tensor networks for weighted model counting through graph decompositions," *arXiv preprint arXiv:1908.04381*, 2019.

[199] J. M. Dudek, V. H. Phan, and M. Y. Vardi, "Procount: Weighted projected model counting with graded project-join trees," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 152–170, Springer, 2021.

[200] M. Samer and S. Szeider, "Algorithms for propositional model counting," *Journal of Discrete Algorithms*, vol. 8, no. 1, pp. 50–64, 2010.

[201] S. Maniu, P. Senellart, and S. Jog, "An experimental study of the treewidth of real-world graph data (extended version)," *arXiv preprint arXiv:1901.06862*, 2019.

[202] J. Yuan, A. Aziz, C. Pixley, and K. Albin, "Simplifying boolean constraint solving for random simulation-vector generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 3, pp. 412–420, 2004.

[203] N. B. Kitchen, *Markov Chain Monte Carlo stimulus generation for constrained random simulation*. University of California, Berkeley, 2010.

[204] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of sat solutions for testing," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 549–559, IEEE, 2018.

[205] C. P. Gomes, A. Sabharwal, and B. Selman, "Near-uniform sampling of combinatorial spaces using xor constraints," in *NIPS*, pp. 481–488, 2006.

[206] A. Darwiche, "Sdd: A new canonical representation of propositional knowledge bases," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[207] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*, vol. 2. Springer Science & Business Media, 1984.

[208] A. A. Bulatov, "The complexity of the counting constraint satisfaction problem," in *International Colloquium on Automata, Languages, and Programming*, pp. 646–661, Springer, 2008.

[209] M. E. Dyer and D. M. Richerby, "On the complexity of# csp," in *Proceedings of the forty-second ACM symposium on Theory of computing*, pp. 725–734, 2010.

[210] J.-Y. Cai and X. Chen, "Complexity of counting csp with complex weights," in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pp. 909–920, 2012.

[211] T. Talvitie, K. Kangas, T. Niinimäki, and M. Koivisto, "Counting linear extensions in practice: Mcmc versus exponential monte carlo," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[212] M. Jerrum, "Mathematical foundations of the markov chain monte carlo method," in *Probabilistic methods for algorithmic discrete mathematics*, pp. 116–165, Springer, 1998.

[213] F. Doshi-Velez and B. Kim, "A roadmap for a rigorous science of interpretability," *CoRR*, vol. abs/1702.08608, 2017.

[214] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*, pp. 305–343, Springer, 2018.

[215] S. Sanner and D. McAllester, "Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference," in *IJCAI*, vol. 2005, pp. 1384–1390, 2005.

[216] N.-F. Zhou, H. Kjellerstrand, and J. Fruhman, *Constraint solving and planning with Picat*. Springer, 2015.

[217] C. Schulte, G. Tack, and M. Z. Lagerkvist, "Modeling and programming with gecode," *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, vol. 1, 2010.

[218] R. R. Hoffman and G. Klein, "Explaining explanation, part 1: Theoretical foundations," *IEEE Intelligent Systems*, vol. 32, no. 3, pp. 68–73, 2017.

[219] R. R. Hoffman, S. T. Mueller, and G. Klein, "Explaining explanation, part 2: Empirical foundations," *IEEE Intelligent Systems*, vol. 32, no. 4, pp. 78–86, 2017.

[220] Z. C. Lipton, "The mythos of model interpretability," *Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[221] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 93:1–93:42, 2019.

[222] P. Schwab and W. Karlen, "Cxplain: Causal explanations for model interpretation under uncertainty," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlche Buc, E. Fox, and R. Garnett, eds.), pp. 10220–10230, Curran Associates, Inc., 2019.

[223] H. Lakkaraju, E. Kamar, R. Caruana, and J. Leskovec, "Faithful and customizable explanations of black box models," in *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pp. 131–138, 2019.

[224] A. Björklund, A. Henelius, E. Oikarinen, K. Kallonen, and K. Puolamäki, "Sparse robust regression for explaining classifiers," in *International Conference on Discovery Science*, pp. 351–366, Springer, 2019.

[225] H. Lakkaraju, N. Arsov, and O. Bastani, "Robust and stable black box explanations," *ICML 2020*, 2020.

[226] A. Shih, A. Choi, and A. Darwiche, "A symbolic approach to explaining Bayesian network classifiers," in *IJCAI*, pp. 5103–5111, 2018.

[227] A. Shih, A. Choi, and A. Darwiche, "Compiling Bayesian network classifiers into decision graphs," in *AAAI*, pp. 7966–7974, 2019.

[228] A. Ignatiev, N. Narodytska, and J. Marques-Silva, "Abduction-based explanations for machine learning models," in *AAAI*, pp. 1511–1519, 2019.

[229] A. Darwiche and A. Hirth, "On the reasons behind decisions," *CoRR*, vol. abs/2002.09284, 2020.

[230] A. Darwiche, "Three modern roles for logic in AI," *CoRR*, vol. abs/2004.08599, 2020.

[231] A. Ghorbani, A. Abid, and J. Y. Zou, "Interpretation of neural networks is fragile," in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI*

2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pp. 3681–3688, AAAI Press, 2019.

[232] D. Alvarez-Melis and T. S. Jaakkola, "On the robustness of interpretability methods," *CoRR*, vol. abs/1806.08049, 2018.

[233] Y. Zhang, K. Song, Y. Sun, S. Tan, and M. Udell, "" why should you trust my explanation?" understanding uncertainty in lime explanations," *arXiv preprint arXiv:1904.12991*, 2019.

[234] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018.

[235] A. Albarghouthi, L. D'Antoni, S. Drews, and A. V. Nori, "Fairsquare: probabilistic verification of program fairness," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[236] O. Bastani, X. Zhang, and A. Solar-Lezama, "Probabilistic verification of fairness properties via concentration," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

Appendix A

DNF-Counting

For obtaining a concrete algorithm from the framework described in Algorithm 4, we need to instantiate the sub-procedures SampleHashFunction, GetLowerBound, GetUpperBound, EnumerateNextSol, ExtractSlice and ComputeIncrement for a particular counting problem. We now show how SymbolicDNFApproxMC [74], 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.

A.1 SampleHashFunction

One can directly invoke the procedure SampleBase described in Algorithm 4 of [74] with minor modifications. This is shown in Algorithm 23. Note that the hash function $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}$ so obtained belongs to the Row Echelon XOR family.

Algorithm 23 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** $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}, \mathsf{q}$;

A.2 Lower and Upper Bounds

As shown in [74], 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.

A.3 Extracting a prefix slice

Procedure ExtractSlice required for ReverseSearch is shown in Algorithm 24. If flip is false, ExtractSlice returns the result of the procedure Extract (described in [74]) directly. Otherwise, the p-th bit of $\boldsymbol{y}$ is negated before being passed to Extract.

Algorithm 24 ExtractSlice$(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{y}, \mathsf{p}, flip)$

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

A.4 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 [74]).

A.5 ComputeIncrement

Procedure CheckSAT (Algorithm 26 adapted from [74]) can be used to compute the increments to Y_{cell} as shown in Algorithm 25. The assignment s is divided into a solution $\boldsymbol{x}$ and a cube φ^i using the same Interpret function used in line 7 of Algorithm 6 in [74]. 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 [72], and can be used to compute an accurate probabilistic estimate Y_{cell} of the true number of solutions in the cell [74].

Algorithm 25 ComputeIncrement$(s, Y_{cell}, \mathsf{threshold})$

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

Algorithm 26 CheckSAT$(\boldsymbol{x}, \varphi^i, Y_{cell}, \mathsf{threshold})$

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** $\boldsymbol{x} \models \varphi^j$ **then**
6: **return** $\mathsf{c}_{\boldsymbol{x}}/\mathsf{m}$;
7: **return** $\mathsf{c}_{\boldsymbol{x}}/\mathsf{m}$

Lemma A.1. *The complexity of* SaturatingCounter *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 SaturatingCounter after a call to ComputeIncrement and CheckSAT. Since SaturatingCounter 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 [74] takes $\mathcal{O}(n)$ time. As a result, the complexity of SaturatingCounter is $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot \mathsf{threshold})$. □

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

Proof: In ReverseSearch, SaturatingCounter is invoked with different thresholds (say $T_1, T_2, T_3 \ldots$) in each iteration of the for loop in line 9 (Algorithm 8) 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 SaturatingCounter 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 ExtractSlice in line 12 is $\mathcal{O}(\mathsf{n}(\log \mathsf{m} + \log(1/\varepsilon^2))^2)$ [74], and the loop in line 9 can be executed at most $\mathcal{O}(\log \log \mathsf{m})$ times. Therefore, the complexity of 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 4.1.

Theorem 4.1. *The complexity of* SymbolicDNFApproxMC*, when invoked with* ReverseSearch *is* $\mathcal{O}(\mathsf{mn} \log(1/\delta)/\varepsilon^2)$

Proof: In Algorithm 4, ApproxMCCore is invoked $\mathcal{O}(\log(1/\delta))$ times, which in turn makes a call to ReverseSearch. The complexity of SampleHashFunction is $\mathcal{O}(\mathsf{n}(\log \mathsf{m} + \log(1/\varepsilon^2)))$ [74]. Since $\mathsf{hiThresh} = \mathcal{O}(1/\varepsilon^2)$, the complexity of Algorithm 4 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 $\mathcal{O}(\mathsf{m} \cdot \mathsf{n} \cdot (1/\varepsilon^2) \cdot \log(1/\delta))$. □

Appendix B

Conditional Counting for Explainable AI

B.1 Related work

Model explainability is one of the most important problems in machine learning. Therefore, there are a large number of recent surveys on the topic, e.g. [218, 219, 220, 97, 221, 93]. To overview, we partition approaches to generate explanations for ML models into two groups based on whether they provide theoretical guarantees on the quality of the generated explanations.

Explanations without theoretical guarantees. There were a number of approaches proposed to compute (model-agnostic) local explanations. We have overviewed LIME [95] in Section 5.2. Anchor is a successor of LIME [109]. The main contribution of Anchor is to produce explanations that hold globally, for the entire distribution of inputs. SHAP [96] is another popular model-agnostic explainer to produce local explanations. Similar to other explainers, SHAP does not provide any theoretical justification for the sampling procedure. However, SHAP employs game theoretic principles to produce an explainable model. Our work focuses on model-agnostic, local explanations, however, we produce explanations with provable guarantees. CXPlain proposes to train an additional 'explanation model' to provide explanations for a given ML model [222]. Learning of the explanation model involves an estimation of feature importance using a causal objective. The causal objective captures how input features cause a marginal improvement in the predictive performance of the ML model. In our work, we do not consider each feature individually and reason about the space of features as a whole. Moreover, our framework allows us to work

with constrained spaces. Finally, works such as [223] provide limited capabilities for customizing global explanations by letting the user supply a set of features that the they deem important. Similar to [224], they avoid sampling a neighbourhood around a given point by using original data points to construct an explainer. While avoiding sampling helps scalability, it also undermines applicability. For instance, dealing with user-defined constraints, as well as unbalanced or skewed input datasets can be problematic. In both cases, the input data may be too sparse to yield meaningful explanations. Recently, [225] demonstrated that these explanations are less robust compared to LIME, for example.

Another line of work in on gradient-based explainers, for example, saliency maps [100], Integrated Gradient [101], DeepLIFT [98]. Gradient-based methods assume full knowledge about the ML model and, also, require these models to be differentiable. While these methods are very efficient, they do not provide theoretical guarantees on the produced explanations. On top of that these approaches are not model-agnostic.

Explanations with theoretical guarantees. Recently, a formal approach to analyze explanation of ML models was proposed. If an ML model allows a formal representation in restricted fragment of the first order logic, then one can (a) define a formal notion of an explanation and (b) design an algorithm to produce these explanations [226, 227, 228, 229, 230]. One of the formal approaches is built on powerful knowledge compilation techniques, e.g. [226, 227]. The other approach employees very efficient formal reasoners, like SAT, SMT or ILP solvers, as a part of explanation generation algorithms [228, 115]. If the process of ML model compilation into a tractable structure is feasible then the first approach is very effective and allows the user to analyse the ML model efficiently. However, the compilation can be computationally expensive and resource demanding, so the second approach is more efficient in some applications. There are some limitations of these approaches. First, similar to gradient-based methods, they require full knowledge of the original ML model.

Second, in practice, these approaches face scalability issues as reasoning about ML models formally is computationally expensive.

Quality of the explanations. Finally, we consider a recent line of work on analysis of the quality of explanations. [109] proposed several heuristic measures to evaluate quality of explanations including fidelity and coverage, but do not provide a way to estimate the true value of these metrics. In [231, 232], it was shown using perturbation-based methods that explanations are susceptible to adversarial attacks and lack robustness property. For example, [233] investigated several sources of uncertainty in LIME, like sampling variance in explaining a sample. The authors experimentally demonstrated that LIME often fails to capture the most important features locally. However, the paper does not propose a solution to remedy identified issues. Moreover, [104] showed that it is easy to fool an explainer, like LIME and SHAP, as we discussed in detail in Section 5.5.3. [115] presented a technique for evaluation quality of explanations based on model counting, but their approach suffers from scalability issues (as shown in Sec. 5.4) and is only applicable to BNNs. [225] proposed to use adversarial training [234] to improve robustness of the explanations. While the proposed approach improves robustness to adversarial attacks it cannot be easily extended to work in constraint environments and does not provide theoretical guarantees on the fidelity of the explanations. A related line of work on probabilistic verification of ML models has seen a surge in interest. [235] encoded the underlying model and fairness properties as formulas in SMT over real arithmentic, and relied on symbolic integration techniques. However, this approach is known not to scale, eg. it can only handle neural networks with a single hidden layer containing just three hidden units. [236] present an alternative approach that uses Monte Carlo sampling and adaptive concentration inequalities. However, unlike Alg. 10, their method only returns a yes/no answer and does not provide a quantitative estimate. Further, their algorithm may fail to terminate on some inputs, and the sample complexity is not

proven to be close-to-optimal.

B.2 Certifying Constraint-Driven Explanations (Additional materials)

B.2.1 AA' Algorithm

In this section, we present our proofs for Thm. 5.2, in the context estimation framework of Algs. 10 and 11, but we use a more general setting. In particular, we observe that our technique is abstract in the sense that it can estimate the density of samples satisfying any property (not just fidelity), in any given domain (not just Boolean), so long as it is possible to sample (almost) uniformly from the domain (encapsulated in the procedure getSamples). We assume access to a procedure checkProperty, that given a sample s, returns 1 if the property of interest is satisfied by s and 0 otherwise. This entails, that the check on line 6 of Alg.11 will be replaced by a call to checkProperty, and that for measuring fidelity, the procedure checkProperty will simply return the value of $\mathcal{I}[l_f(z) = l_g(z')]$.

For completeness, we first present the AA' algorithm in full, which is a simple adaptation of AA algorithm by [71] which uses almost-uniform samples instead of perfectly uniform. AA', takes as input 3 parameters, ε_1,ε_2 and δ. It uses ε_2 as the tolerance parameter in calls to an almost-uniform sampler (encapsulated in procedure *getSamples*). For ease of exposition, we use φ to denote the subspace that *getSamples* generates samples from.

$$\Pr[\frac{(1-\varepsilon_1)}{1+\varepsilon_2}\rho \leq \widehat{\rho} \leq (1+\varepsilon_1)(1+\varepsilon_2)\rho] \geq (1-\delta) \tag{B.1}$$

The guarantees provided by AA' are similar to Eqn. 5.4 and are precisely captured in Eqn. B.1. See Lemmas B.1 and B.4 for the proof.

Algorithm 27 $AA'(\varepsilon_1, \varepsilon_2, \delta)$

Output: $\widehat{\rho}$: Estimate of ρ satisfying Eqn. B.1

$\tau \leftarrow 4(e-2)\ln(2/\delta)/\varepsilon_1^2$
$\tau_2 \leftarrow 1.1\tau$
$\mathring{\rho} \leftarrow stoppingRule(1/2, \varepsilon_2, \delta/3, \tau)$
$N \leftarrow \tau_2 \cdot \varepsilon_1/\mathring{\rho}$
$a \leftarrow 0$
for $i \in \{1, \ldots, N\}$ **do**
    $s_1 \leftarrow getSamples(\varphi, \varepsilon_2, 1)$
    $c_1 \leftarrow checkProperty(s_1)$
    $s_2 \leftarrow getSamples(\varphi, \varepsilon_2, 1)$
    $c_2 \leftarrow checkProperty(s_2)$
    $a \leftarrow a + (c_1 - c_2)^2/2$
$\xi \leftarrow max(a/N, \mathring{\rho} \cdot \varepsilon_1)$
$N \leftarrow \tau_2 \cdot \xi/\mathring{\rho}^2$
$a \leftarrow 0$
for $i \in \{1, \ldots, N\}$ **do**
    $s \leftarrow getSamples(\varphi, \varepsilon_2, 1)$
    $c \leftarrow checkProperty(s)$
    $a \leftarrow a + c$
$\widehat{\rho} \leftarrow a/N$
return $\widehat{\rho}$

Algorithm 28

$stoppingRule(\varepsilon_1, \varepsilon_2, \delta, \tau)$

Output: $\mathring{\rho}$ (weak estimate)

$\tau_1 \leftarrow 1 + (1+\varepsilon_1)\tau$
$N \leftarrow 0$
$a \leftarrow 0$
while $a < \tau_1$ **do**
    $s \leftarrow getSamples(\varphi, \varepsilon_2, 1)$
    $a \leftarrow a + checkProperty(s)$
$\mathring{\rho} \leftarrow \tau_1/N$
return $\mathring{\rho}$

B.2.2 Proof of Theorem

Mean deviation due to almost uniform sampling

Lemma B.1. *Let ρ be the density of instances that satisfy some property P in a universe $\mathcal{U}^{\mathcal{Z}'}$, that is,*

$$\rho = \frac{\sum_{z' \in \mathcal{U}^{\mathcal{Z}'}} \mathcal{I}[P(z, z')]}{|\mathcal{U}^{\mathcal{Z}'}|}$$

Suppose we sample each instance $z \in \mathcal{U}^{\mathcal{Z}'}$ almost-uniformly, that is

$$\frac{1}{(1+\varepsilon)|\mathcal{U}^{\mathcal{Z}'}|} \leq Pr[z^* = z'] \leq \frac{1+\varepsilon}{|\mathcal{U}^{\mathcal{Z}'}|}$$

Then we have

$$\frac{\rho}{(1+\varepsilon)} \leq \sum_{z' \in \mathcal{U}^{\mathcal{Z}'}} \mathcal{I}[P(z, z')] \cdot \Pr[z^* = z'] \leq (1+\varepsilon) \cdot \rho$$

Proof. In the worst cases, each sample z' s.t. $\mathcal{I}[P(z, z')] = 1$ will be sampled with

1. probability $\frac{1+\varepsilon}{|\mathcal{U}^{\mathcal{Z}'}|}$, in which case $\sum_{z' \in \mathcal{U}^{\mathcal{Z}'}} \mathcal{I}[P(z, z')] \cdot \Pr[z^* = z'] = (1+\varepsilon) \cdot \rho$
2. probability $\frac{1}{(1+\varepsilon)|\mathcal{U}^{\mathcal{Z}'}|}$ in which case$\sum_{z' \in \mathcal{U}^{\mathcal{Z}'}} \mathcal{I}[P(z, z')] \cdot \Pr[z^* = z'] = \frac{\rho}{(1+\varepsilon)}$

□

Lemma B.2. *Let $\rho \leq \gamma - \varepsilon$. Then the probability that checkThreshold returns True is at-least $1 - \delta$.*

Proof. Note that $\mu_C \leq (1 + \varepsilon/2)\rho$ as *getSamples* may return almost uniformly distributed samples in line 3 of *checkThreshold*. We will first prove that $\mu_C + \nu \leq \gamma$. We have

$$\begin{aligned} \gamma &\leq \gamma \\ \implies \gamma\,(1 + \varepsilon/2 - \varepsilon/2) &\leq \gamma \\ \implies \gamma\,(1 + \varepsilon/2) - \gamma\varepsilon/2 &\leq \gamma \end{aligned}$$

But we have $\gamma \geq \rho + \varepsilon$. Therefore,

$$\begin{aligned} (\rho + \varepsilon)(1 + \varepsilon/2) - \gamma\varepsilon/2 &\leq \gamma \\ \implies \rho + \rho\varepsilon/2 + \varepsilon + \varepsilon^2/2 - \gamma\varepsilon/2 &\leq \gamma \\ \implies \mu_C + \nu &\leq \gamma \end{aligned}$$

The last equation follows from the fact that $\mu_C \leq (1+\varepsilon/2)\rho$ and $\nu \leq \varepsilon + \varepsilon^2/2 - \gamma\varepsilon/2$ from line 1 of *checkThreshold*. Now since $\mu_C + \nu \leq \gamma$, we have $\Pr[C \leq \gamma] \geq \Pr[C \leq \mu_C + \nu] = 1 - \Pr[C \geq \mu_C + \nu]$. C is the average of N independent 0/1 random variables c (line 6). Therefore applying Chernoff bound,

$$\Pr[C \geq \mu_C + \nu] \leq exp\{-2\nu^2 N\}$$

But $N = \frac{1}{2\nu^2}\log(\frac{1}{\delta})$. Therefore, $\Pr[C \geq \mu_C + \nu] \leq \delta$. Substituting back, we get $\Pr[C \leq \gamma] \geq 1 - \delta$. Therefore, in line 8, the probability that *checkThreshold* returns True, is at least $1 - \delta$.

□

Lemma B.3. *Let $\rho \geq \gamma + \varepsilon$. Then the probability that checkThreshold returns False is at-least $1 - \delta$.*

Proof. Similar to preceding Lemma. □

Lemma B.4. *Algorithm AA' returns an estimate $\widehat{\rho}$ such that $\Pr[(1-\varepsilon)\rho_\psi \leq \widehat{\rho}_\psi \leq (1+\varepsilon)\rho_\psi] \geq (1-\delta)$*

Proof. Algorithm AA' takes as input 3 parameters, ε_1, ε_2 and δ. It invokes the AA Algorithm by Dagum et al. with parameters ε_1 and δ on samples generated by an almost-uniform sampler with parameter ε_2. By Lemma B.1, the population mean can shift at most by a factor of $(1 + \varepsilon_2)$ due to almost-uniform sampling (instead of perfectly uniform). Combined with the approximation guarantees of AA algorithm, the resulting tolerance has an upper-bound of $(1 + \varepsilon_1) \times (1 + \varepsilon_2)$ and a lower bound

of $(1-\varepsilon_1)/(1+\varepsilon_2)$. In line 4 of Alg. 10, the AA' algorithm is invoked with $\varepsilon_1 = \varepsilon_2 = 0.4 * \varepsilon$. Substituting these values in the expressions for the upper and lower bounds on tolerance, we get the result. □

Theorem 5.2. *If $\rho \leq \gamma - \varepsilon$, then* computeFidelity *returns $\perp$ with high probability (i.e. at least $1-\delta$). If $\rho \geq \gamma + \varepsilon$, w.h.p., it returns an estimate $\widehat{\rho}$ such that* $\Pr[(1-\varepsilon)\rho \leq \widehat{\rho} \leq (1+\varepsilon)\rho] \geq (1-\delta)$.

Proof. Follows from preceding lemmas. □

Note that the bounds and number of samples used for proving the preceding theorem were computed assuming we only have access to Almost-Uniform samples. The bounds can be made significantly tighter or the number of samples can be reduced, if we have access to perfectly uniform samples.

B.2.3 Applicability of the Estimation framework

We emphasize that our estimation framework is general enough to compute any metric for any universe (so long as one can sample from it almost uniformly) according to the guarantees provided by Thm. 5.2 in any setting (not just explainability). Further, for the specific application explainability, our estimation framework can be used for measuring properties like fidelity of any explainer model (not just the ones crafted by CLIME), and on any subspace of inputs (not just the one that the explainer was trained on). For example, the fidelity of a CLIME explainer model trained on a subspace defined by one set of constraints (say φ) maybe evaluated on another subspace defined by ψ. If the fidelity on ψ is found to be high enough, it can save the cost of having to generate a separate explanation for ψ. This can be especially useful in model debugging where users may refine constraints frequently.

B.2.4 Empirical Evaluation of Estimation Algorithm

In order to test the scalability of our estimation algorithm (Algs. 10, we evaluated

its performance on the same set of benchmarks used by [115].

Benchmarks The benchmarks are CNF formulas that encode fidelity of Anchor [109] explanations for Binarized Neural Networks with upto 3 hidden layers and 100 neurons in total, and are generated from Adult, Recidivism and Lending datasets. There were 50 CNF formulas from each dataset, for a total of 150 benchmarks, with number of variables ranging between 20,000 to 80,000 and number of clauses ranging between 80,000 and 290,000. The projected model count of each formula represents the number of inputs on which the class-label for Anchor's explanation matches the true label of the instance being explained. The fidelity of an explanation can thus be computed as the ratio of the solution-count of the formula, to the size of the universe.

Parameters We used the same tolerance ($\varepsilon = 0.8$) and confidence ($\delta = 0.2$) used in [115], for the main experiment. Additionally, we set the threshold to $\gamma = 0.05$. We also compared the running times for tighter tolerance and confidence (see Discussion below).

Experimental Setup We set a time out of 3 minutes (180 seconds), and ran each experiment on Intel Xeon E5-2650 CPU running at 2.20GHz, with 4GB main memory. We compiled our code using GCC 6.4 with O3 flag enabled. For ApproxMC we used the latest publicly available version (4.01).

Results The results are presented in Fig. 5.1. Each point in blue corresponds to one benchmark, and the x-coordinate represents the time taken by ApproxMC while the y-coordinate represents the time taken by our approach. It can be seen that our approach completes all benchmarks in under 25 seconds with majority taking less than 10 seconds. In contrast, ApproxMC is able to finish only 10 benchmarks out of 150 in under 25 seconds, with a majority taking around 75 seconds. The average (geometric mean) speedup factor offered by our tool relative to ApproxMC was 7.5.

Discussion Fig. 5.1 conclusively demonstrates the efficiency of our approach as compared to hashing and SAT based approaches like ApproxMC. Our tool was able to return estimates with two-sided bounds for all benchmarks. Our tool offers an average (geometric mean) speedup by a factor of 7.5 relative to ApproxMC. In addition, our approach is able to scale far better than ApproxMC for tighter tolerance and confidence parameters. For a representative benchmark, we evaluated the time taken by our tool and ApproxMC for the default tolerance ($\varepsilon = 0.8$) and confidence ($\delta = 0.2$). We then computed the slow-down in both tools after setting $\varepsilon = 0.05$ and confidence $\delta = 0.1$. We found that our tool slowed down by a factor of 10 while ApproxMC slowed down by a factor of 30. Lastly, we computed the error in the estimate returned by our algorithm (with default tolerance and confidence) on small benchmarks where it was possible to compute the true count. We consistently observed that the error was less than 0.1, which is much smaller than the specified tolerance of 0.8. Thus our approach is both sound and efficient in practice.

B.3 Detecting Adversarial Attacks(Extended)

Suppose a biased classifier makes decisions solely based on some sensitive feature; for ex: a classifier for accepting / rejecting loan applications that decides purely based on an individual's race. LIME's explanations for the predictions made by such a classifier would clearly indicate race to be a major deciding factor. [104] devised a technique that takes such an arbitrary biased classifier and constructs another classifier that closely mimics the biased classifier on in-distribution data, yet is able to evade bias-detection by preventing the sensitive feature from appearing in post-hoc explanations. Note that accurately measuring the fidelity of the explainer model alone is not sufficient to detect such an attack as the problem lies in the way the opaque classifier is constructed and not in the resulting explanation. We first discuss the construction of the adversarial classifier, and then see how constraints can help in foiling it.

Background

The attack makes use of the fact that LIME's perturbation procedure creates samples that are distinguishable from those of the original distribution. The first step is to train a classifier such as a Random Forest to detect whether a given sample is in or out of distribution. This classifier is trained on a synthetic data-set consisting of instances from the original data-set labeled as in-distribution and LIMEs perturbations of each of those instances as OOD. This OOD detecting classifier is able to achieve very good accuracy, which is a testament to the severity of the OOD problem of LIME.

The next step in the adversarial attack is to create an innocuous classifier that makes decisions based on some unrelated feature. For example, in case of credit card applications, the innocuous classifier could make decisions based on either an existing unimportant feature such as card number or on a new feature added to the dataset whose values are randomly sampled. This unrelated feature will be made to appear in place of the sensitive feature in explanations.

The last step of the attack is construct the adversarial classifier which first applies the OOD-detecting classifier on a given instance x. If the OOD-detecting classifier predicts x to be in-distribution, the the adversarial classifier outputs the prediction of the biased classifier on x. If x is predicted to be OOD, then the adversarial classifier outputs the prediction of the innocuous classifier on x.

Clearly, the predictions made by the adversarial classifier on in-distribution instances are biased, as the adversarial classifier simply outputs the biased classifiers prediction. Suppose LIME is used for explaining the prediction made by the adversarial classifier on some in-distribution instance x. LIME first perturbs x. However, due to LIME's OOD sampling problem, most of the perturbed samples are OOD and are detected by the OOD-detecting classifier as such. Therefore, the output of the adversarial model on the perturbed samples is that of the innocuous classifier. Thus for vast majority of the samples on which LIME's explainer model is trained, the unrelated feature appears as the deciding factor for the adversarial model's prediction,

whereas the sensitive feature is successfully hidden.

Experimentally, the authors generated explanations for the adversarial model on multiple test instances belonging to Recidivism, Communities and Crime and German datasets. They showed that the sensitive features appeared with very low frequency among the top 3 explanations for all the test points, which indicates that the attack was successful.

Experimental Setup

We hypothesized that for discrete data, instances with low Hamming distance to points in the given dataset should tend to be in-distribution, whereas instances with a high hamming distance should tend to be OOD. Note that the recommended way for using LIME is on data which is discretized. LIME itself uses Hamming distance as a distance metric for weighting the samples it generates. However, it is unable to specifically sample proportionately from high-weight regions which leads to a lot of samples with low weights. With CLIME one can, however, use constraints to generate samples that would be weighted highly by LIME. For a given Hamming distance h and a set of instances S, a formula φ^h can be constructed such that the the solutions of φ^h are exactly those instances that are at most at a distance h from each instance in S. If our hypothesis is correct, the sensitive features should appear as the top CLIME explanation of the adversarial classifier for low values of h, while innocuous features should dominate for larger h.

Note that the Hamming distance between two instances doesn't take into account the degree to which the value of a feature in one instance differs from the other. Specifically, for discrete features that are the quantized versions of continuous features, values of adjacent quantiles are clearly 'closer' that those of non-adjacent quantiles. This is also true for natively discrete features like the star rating of a product. One can design metrics smarter than the hamming distance, that also take into account these differences in values and are even more robust against adversar-

ial attacks. In this work however, we focus on the Hamming distance as a proof of concept and leave experimentation with other metrics to future work.

B.3.1 Practical Considerations

The tool by [104], as of the time of writing this paper, was only able to create an adversarial attack on continuous features (for LIME), and did not have the ability to train the OOD-detecting classifier on discrete data. For our experiments, we augmented their tool to handle purely discrete data by using LIME's native perturbation procedure for generating training data for the OOD-detecting classifier. For discrete data, the parameter 'perturbation_multiplier' in their tool plays an important role. Its default value of 30 creates a very imbalanced dataset and the accuracy of the OOD-detector is dramatically affected and it classifies most instances as OOD. We found that a value of 2 led to the best test accuracy for the OOD-detector.

With discrete data, we also observed that the efficacy of the adversarial attack on LIME was drastically reduced. The sensitive feature appeared as the 2nd from top explanation almost 100% of the time. Additionally, for attacks that used 2 uncorrelated features, the sensitive feature also appeared as the top explanation with a considerable frequency. However, for attacks that used 1 uncorrelated feature, the top explanation was the uncorrelated feature with almost 100% frequency. Hence in our experiments, we focus on the top explanation ([104] considered top 3 explanations) and use only one unrelated feature. For CC and German experiments, we selected the top 12 most important features of the 3 datasets using scikit-learn's Random Forest classifier, as there were far too many features in the original datasets (100 for CC and 28 for German). The test accuracy of the Random Forest with 100 estimators was 0.83 and 0.75 for CC and German respectively even after restriction to 12 features, down from 0.875 and 0.82, respectively, before removing features.

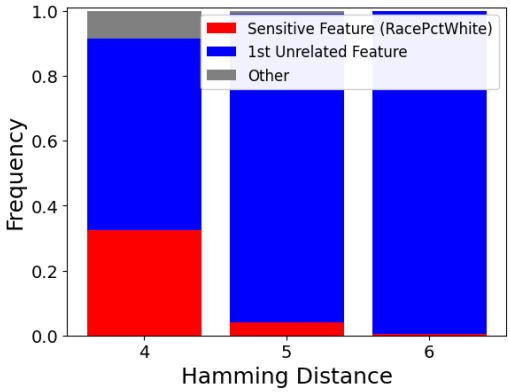
	Recidivism	CC	German
OOD detector trained on LIME's samples	0.635	0.32	0.42
OOD detector trained on samples from Hamming-constraints	0.63	0.1	0.42

Table B.1 : Frequency of sensitive feature in top explanation

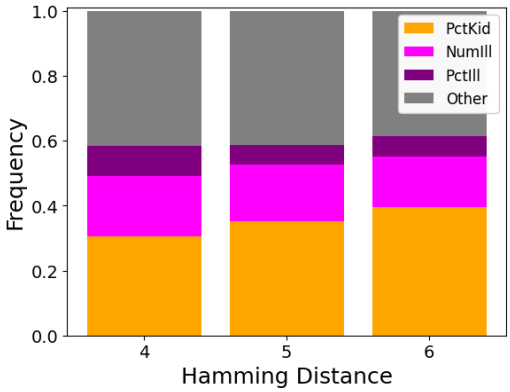
B.3.2 Results

The results for the Recidivism dataset were depicted in Figure 5.2. The results for CC dataset are depicted in Fig. B.1 and those for German in Fig. B.2. It can be seen that the sensitive feature appears prominently for low hamming distance while the unrelated feature dominates higher for higher distance, which confirms the efficacy of CLIME.

In general, it is harder to craft attacks similar to [104] against CLIME explanations, as the sampling procedure is dependent on user-provided constraints and is not fixed like LIME. However, for the current experiment, an adversary could conceivably train the OOD-detector on instances with low hamming distance from the original dataset to specifically foil our method. We tested whether the sensitive feature still appeared in the top CLIME explanation when the OOD-detector was trained in this manner (instead of being trained on LIME generated perturbations). We compare the frequency of seeing the sensitive feature as the top CLIME explanation of the adversarial classifier in the two scenarios in Table B.1. While the frequency of seeing the sensitive feature drops as compared to the case when the OOD detector is trained on LIME perturbations, the drop is not very steep and the sensitive feature still figures prominently as the top explanation. This shows that our hamming distance based technique is not only able to detect attacks, but is also robust against attacks itself.

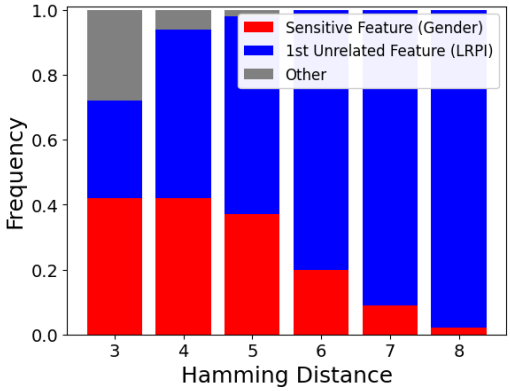


(a) Adversarial Classifier

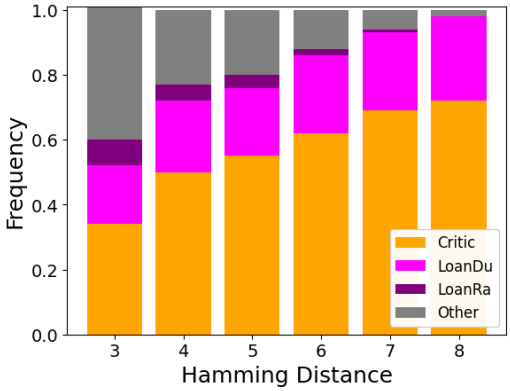


(b) Non-Adversarial Classifier

Figure B.1 : CC Dataset: Top CLIME explanation vs. Hamming Distance



(a) Adversarial Classifier



(b) Non-Adversarial Classifier

Figure B.2 : German Dataset: Top CLIME explanation vs. Hamming Distance

Appendix C

Sampling Solutions of Low-Treewidth CNF Formulas

C.1 Proofs of Correctness

We introduce some additional notation for ease of exposition.

We define the *level* of a node v in an ADD f as follows:

Definition C.1. *Let v be a node in an ADD $f = (X, S, \rho, G)$ with associated variable x_v. Then the level of v in f, denoted as $level_f(v)$ is defined as*

$$level_f(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf node} \\ i & \text{if } |S| = i - 1 \text{ where } S = \{x \in X \mid \rho(x) > \rho(x_v)\} \end{cases}$$

We drop f and simply write $level(v)$ when clear from context. Intuitively, $level(v)$ is the height at which v appears in the DAG, assuming that leaves are at the bottom and the root node is at the top. Note that all nodes labeled with the same variable in f have the same value of *level*, *level*'s are numbered sequentially from leaves to root, and that for a node v and child v_c, it is possible that $level(v) > level(v_c) + 1$ as there may be skipped nodes between v and v_c, under the semantics of ADDs.

For an ADD $f = (X, S, \rho, G)$, a node v in f and an assignment ω to the variables $Z \subseteq X$, and $Y = X \setminus Z$ i.e. the set of unassigned variables, we define the following sets:

- $Y_{\geq v} = \{x \in Y \mid \rho(x) \geq \rho(x_v)\}$
- $Y_{<v} = \{x \in Y \mid \rho(x) < \rho(x_v)\}$

Algorithm 29 $\mathsf{Compile}(\mathcal{T}, w)$

Input: $\mathcal{T} = (T, r, \gamma, \pi)$: a project-join tree

Input: w: A literal-weight function $w : \{0,1\}^X \to \mathbb{R}$

Output: S: a map from each $n \in \mathcal{V}(T)$ to ADD f^n

$S \leftarrow$ empty map

procedure $\mathsf{Compile\_rec}(T, n, S, w)$

Input: n: A node in $\mathcal{V}(T)$

Output: f^n: an ADD corresponding to $n \in \mathcal{V}(T)$

    if $n \in \mathcal{L}(T)$ **then**

        $f^n \leftarrow [\gamma(n)]$

    else

        $f^n \leftarrow \top$

        for $c \in C(n)$ **do**

            $f^c \leftarrow \mathsf{Compile\_rec}(\mathcal{T}, c, S, w)$

            $f^n \leftarrow f^n \times \sum_{X^c} f^c \cdot w(X^c)$

    $S[n] \leftarrow f^n$ ▷ S is modified here

    return f^n

end procedure

$\mathsf{Compile\_rec}(\mathcal{T}, r, S, w)$

return S

- $Y_{>v} = \{x \in Y \mid \rho(x) > \rho(x_v)\}$

The sets $Z_{\geq v}$, $Z_{<v}$ and $Z_{>v}$ are defined analogously. Intuitively, the set $Y_{\geq v}$ consists of unassigned variables in the support of f that label nodes 'at or below' the *level* of node v in f, while $Y_{>v}$ and $Y_{<v}$ are strictly 'below' and 'above' v. We use the shorthand v_t and v_e to denote the then and else children i.e. $v.then$ and $v.else$ respecitvely.

Lemma 9.5. *Let wt be the return value of* computeWeights *invoked on an ADD $f = (X, S, \rho, G)$ with weight function w, an unvisited node v and an assignment ω to the variables $Z \subseteq X$. Let $Y = X \setminus Z$ be the set of unassigned variables, $Y_{\geq v} = \{x \in Y \mid \rho(x) \geq \rho(x_v)\}$, and $Z_{\geq v} = \{x \in Z \mid \rho(x) \geq \rho(x_v)\}$. Then we have*

$$wt = \sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\omega_{Z_{\geq v}}] \tag{9.1}$$

Proof. We will prove by induction on $level(v)$.

Base case: If v is a leaf then $Z_{\geq v} = \varnothing$ and $Y_{\geq v} = \varnothing$. Therefore, $\sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\omega_{Z_{\geq v}}] = f_v = value(v)$. In line 5 in Alg. 20, the value of v is returned which agrees with Eqn. 9.1.

Assume Eqn. 9.1 holds for all v such that $level(v) \leq k$.

We now prove that Eqn. 9.1 also holds when $level(v) = k + 1$. Two cases arise: (1) $x_v \in Y$ (2) $x_v \in Z$.

Case 1: x_v is assigned under ω. W.L.O.G. assume that $\omega(x_v) = 1$ (the analysis is similar for when $\omega(x_v) = 0$). Let v_t be the 'then' child of v. Then by inductive assumption, we have that the value ($v_t.wt$) returned by the call to computeWeights with argument v_t on line 7 of Alg. 20 is $v_t.wt = \sum_{Y_{\geq v_t}} w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}]$. Note that this value is directly stored as $v.wt$ on line 7, and is also returned as the weight for node v on line 14. Therefore, we need to show that $v.wt = \sum_{Y_{\geq v_t}} w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}] = \sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\omega_{Z_{\geq v}}]$ to complete the proof for this case. To see this, first note that since $x_v \in Z$, and x_v is assigned True, we have $f_v[\omega_{Z_{\geq v}}] \equiv f_v[\omega_{Z_{\geq v_t}}, \omega(x_v) = 1] \equiv$

$f_{v_t}[\omega_{Z_{\geq v_t}}]$. Then we have,

$$\begin{aligned}
v.wt &= \sum_{Y_{\geq v_t}} w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}] \quad \text{(from line 7 of Alg. 20)} \\
&= \prod_{x \in Y_{\geq v} \setminus Y_{\geq v_t}} (w(x) + w(\neg x)) \sum_{Y_{\geq v_t}} w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}] \\
&= \sum_{Y_{\geq v} \setminus Y_{\geq v_t}} w(Y_{\geq v} \setminus Y_{\geq v_t}) \sum_{Y_{\geq v_t}} w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}] \\
&= \sum_{Y_{\geq v} \setminus Y_{\geq v_t}} \sum_{Y_{\geq v_t}} w(Y_{\geq v} \setminus Y_{\geq v_t}) \cdot w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}] \\
&= \sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\omega_{Z_{\geq v}}]
\end{aligned}$$

where the second equality follows from the fact that $w(x) + w(\neg x) = 1$ for all x, from our definition of literal-weight functions. This completes the proof for Case 1.

Case 2: $x_v \in Y$. Similar to the analysis in Case 1, using the inductive assumption we get from lines 10-13 of computeWeights that

$$\begin{aligned}
v.wt = &\left(w(x_v) \sum_{Y_{\geq v_t}} w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}] \right) \\
&+ \left(w(\neg x_v) \sum_{Y_{\geq v_e}} w(Y_{\geq v_e}) \cdot f_{v_e}[\omega_{Z_{\geq v_e}}] \right)
\end{aligned}$$

Note, as in Case 1, that $\sum_{Y_{>v} \setminus Y_{\geq v_t}} w(Y_{>v} \setminus Y_{\geq v_t}) = 1$ because literal weights sum to

1, and similarly $\sum_{Y_{>v} \setminus Y_{\geq v_e}} w(Y_{>v} \setminus Y_{\geq v_e}) = 1$. Therefore, we get

$$\begin{aligned}
v.wt &= \left(w(x_v) \sum_{Y_{>v} \setminus Y_{\geq v_t}} w(Y_{>v} \setminus Y_{\geq v_t}) \sum_{Y_{\geq v_t}} w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}] \right) \\
&+ \left(w(\neg x_v) \sum_{Y_{>v} \setminus Y_{\geq v_e}} w(Y_{>v} \setminus Y_{\geq v_e}) \sum_{Y_{\geq v_e}} w(Y_{\geq v_e}) \cdot f_{v_e}[\omega_{Z_{\geq v_e}}] \right) \\
&= \left(\sum_{Y_{>v}} w(x_v) w(Y_{>v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}] \right) \\
&+ \left(\sum_{Y_{>v}} w(\neg x_v) w(Y_{>v_e}) \cdot f_{v_e}[\omega_{Z_{\geq v_e}}] \right)
\end{aligned}$$

Note that $f_{v_e}[\omega_{Z_{\geq v_e}}] = f_v[\omega_{Z_{\geq v}}, \omega(x_v) = 0]$ since all variables in $Z_{\geq v_e}$ appear at a lower *level* than x_v, and only the variables below the level of x_v matter in the function f_{v_e}. Similarly $f_{v_t}[\omega_{Z_{\geq v_t}}] = f_v[\omega_{Z_{\geq v}}, \omega(x_v) = 1]$. Therefore, we get

$$\begin{aligned}
v.wt &= \left(\sum_{Y_{>v}} w(x_v) w(Y_{>v_t}) \cdot f_v[\omega_{Z_{\geq v}}, \omega(x_v) = 1] \right) \\
&+ \left(\sum_{Y_{>v}} w(\neg x_v) w(Y_{>v_e}) \cdot f_v[\omega_{Z_{\geq v}}, \omega(x_v) = 0] \right) \\
&= \sum_{x_v} \sum_{Y_{>v}} w(\{x_v\}) w(Y_{>v}) \cdot f_v[\omega_{Z_{\geq v}}] \\
&= \sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\omega_{Z_{\geq v}}]
\end{aligned}$$

□

Lemma 9.6. *Let* sampleFromADD *be invoked on an ADD* $f = (X, S, \rho, G)$*, weight function* w*, and an assignment* ω *to the variables* $Z \subseteq X$*. Let* $Y = X \setminus Z$ *be the set of unassigned variables. Then* sampleFromADD *returns an assignment* ω' *to the variables in* Y *with probability*

$$\Pr[Y = \omega' | Z = \omega] = \frac{w(\omega') \cdot f[\omega', \omega]}{\sum_Y w(Y) \cdot f[\omega]} \tag{9.2}$$

Proof. First, note that the conditions on lines 5 and 7 ensure that the assignment ω to the variables Z is followed while traversing the ADD f. On line 10, similar to the analysis in Lem. 9.5, we can infer that $t\_wt = w(x_v) \sum_{Y_{\geq v_t}} w(Y_{\geq v_t}) \cdot f_{v_t}[\omega_{Z_{\geq v_t}}]$, and correspondingly for e_wt. On lines 13-18, x_v is set to True with probability $t\_wt/(t\_wt + e\_wt)$. We note that the denominator $(t\_wt + e\_wt)$ can be simplified (similar to Lem. 9.5) to $\sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\omega_{Z_{\geq v}}]$. Thus, in general, for a node v with child v_c corresponding to the assingment $\omega'(x_v)$ of variable x_v, with variables in $Y_{<v}$ already sampled, we have

$$\Pr[x_v = \omega'(x_v) | Z = \omega(Z_n), Y_{<v} = \omega'(Y_{<v})] = \frac{w(\omega'(x_v)) \cdot \sum_{Y_{>v}} w(Y_{>v}) \cdot f_{v_c}[\omega_{Z_{\geq v_c}}]}{\sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\omega_{Z_{\geq v}}]} \tag{C.1}$$

Similar to Lemma 9.5, we can infer the term in the numerator of Eqn. C.1

$$\sum_{Y_{>v}} w(Y_{>v}) \cdot f_{v_c}[\omega_{Z_{\geq v_c}}] = \sum_{Y_{\geq v_c}} w(Y_{\geq v_c}) \cdot f_{v_c}[\omega_{Z_{\geq v_c}}] \tag{C.2}$$

Let $v_{\omega,\omega'}$ be the leaf node in f reached after following assignments ω and ω'. Let L be the set of all skipped *level*'s when traversing f under ω, ω' and for all $l \in L$ let x_l denote the variable at that *level*. Note that in lines 19-25, each x_l is set to $\omega'(x_l)$ with probability $w(\omega(x^l))/(w(\omega(x^l)) + w(\neg\omega(x^l))) = w(\omega'(x_l))$ since literal weights sum to 1. Then, LHS of Eqn. 9.2 can be written as

$$\begin{aligned} &\Pr[Y = \omega' | Z = \omega] = \\ &\prod_{v=root(f)}^{v_{\omega,\omega'}} Pr[x_v = \omega'(x_v) | Z = \omega(Z_n), Y_{<v} = \omega'(Y_{<v})] \\ &\times \prod_{l \in L} w(\omega'(x_l)) \end{aligned} \tag{C.3}$$

We can substitute Eqn. C.1 in Eqn. C.3 and infer from Eqn. C.2 that the summation in the numerator of the term in Eqn. C.1 corresponding to a node v, will cancel out with the denominator of the term corresponding to its child v_c reached

under the assignments ω, ω' to x_v. Thus the residual equation will be

$$\begin{aligned}
&\Pr[Y = \omega' | Z = \omega] \\
&= \frac{f_{v_{\omega,\omega'}} \cdot \prod_{l \in L} w(\omega'(x_l)) \prod_{v=root(f_n)}^{v_{\omega,\omega'}} w(\omega'(x^v))}{\sum_{Y_{\geq root}} w(Y_{\geq root}) \cdot f_{root}[\omega_{Z_{\geq root}}]} \\
&= \frac{w(\omega') \cdot f[\omega', \omega]}{\sum_Y w(Y) \cdot f[\omega]}
\end{aligned} \tag{C.4}$$

□

Theorem 9.7. *Let X be a set of Boolean variables, φ be a CNF formula over X, w be a literal-weight function over X, and n be a positive integer. If $\sigma_1, \cdots, \sigma_n$ is the sequence of random assignments returned by* `DPSampler`*(X, φ, w, n), then $\sigma_1, \cdots, \sigma_n$ are i.i.d. w-weighted samples of φ.*

Proof. First note, that there are no shared variable values between calls to drawSample and sampleFromADD, and that computeWeights computes all weights afresh in each call. Therefore the samples generated are i.i.d. It remains to be shown, from the definition of weighted sampling, that $\Pr[X = \omega] = \frac{w(\omega)}{w(\varphi)}$. We can write LHS using chain rule as:

$$\Pr[X = \omega] = \prod_{n \in V(T)} \Pr[X^n = \omega_{X^n} | Z^n = \omega_{Z^n}]$$

where the variables $Z^n = \bigcup_{a \in \mathcal{A}(n)} X^a$ is the set of all variables labeling the ancestors of n in $\mathcal{T}$, and are already have an assignment under ω, when sampleFromADD is called on f^n by drawSample_rec.

By Lemma 9.6, an assignment for X^n is sampled in Alg. 19 with probability

$$\Pr[X^n = \omega_{X^n} | Z^n = \omega_{Z^n}] = \frac{w(\omega_{X^n}) \cdot f^n[\omega_{X^n}, \omega_{Z^n}]}{\sum_{X^n} w(X^n) \cdot f^n[\omega_{Z^n}]}$$

Substituting back, we get

$$\Pr[X = \omega] = \prod_{n \in V(T)} \frac{w(\omega_{X^n}) \cdot f^n[\omega_{X^n}, \omega_{Z^n}]}{\sum_{X^n} w(X^n) \cdot f^n[\omega_{Z^n}]} \tag{C.5}$$

By Definition of Tree-of-ADDs, we have

$$f^n[\omega_{X^n}, \omega_{Z^n}] = \prod_{c \in C(n)} \sum_{X^c} f^c[\omega_{Z^c}] \cdot w(X^c)$$

Thus in Eqn. (C.5), in the numerator the term corresponding to a node n with non-empty set of children $C(n)$ cancels out with the product of the denominators of the terms for the children. Also, $Z^r = \varnothing$. Noting that $X^n = \varnothing$ for all $n \in \mathcal{L}(T)$, the residual equation is thus

$$\Pr[X = \omega] = \frac{w(\omega) \cdot \prod_{n \in \mathcal{L}(T)} f^n[\omega_{Z^n}]}{\sum_{X^r} w(X^r) \cdot f^r} \tag{C.6}$$

A leaf node n in the project-join tree corresponds to a clause. If $\omega^* \not\models \varphi$, then at least one clause will be violated, i.e. $f^n[\omega_{Z^n}] = 0$, yielding $\Pr[X = \omega] = 0$. But for all ω such that $\omega \models \varphi$, each term $f^n[\omega_{Z^n}] = 1$ in the numerator of Eqn. (C.6). Therefore, we get

$$\Pr[X = \omega] = \frac{w(\omega)}{\sum_{X^r} w(X^r) \cdot f^r}$$

But we have $\sum_{X^r} w(X^r) \cdot f^r = w(\varphi)$ (Theorem 2 in [59]). The result follows. □

C.2 Experiments: Additional Results and Details

C.2.1 Experimental Setup

A comprehensive experimental comparison of all algorithm parameters requires significant computation time. Due to limited resources, we had to arrive at informed choices for various settings which we outline here. We invoked `DPSampler` with the following parameters, which were seen yield the best performance in the context of model-counting [59]. We used the MCS variable order as the diagram variable order and used arbitrary join priority. We used the treedecomposing tool LG with Flowcutter in the planning phase. These are anytime tools, in that they are capable of generating trees with lower treewidths the longer they run. However, we used the first project-join tree returned, in all experiments.

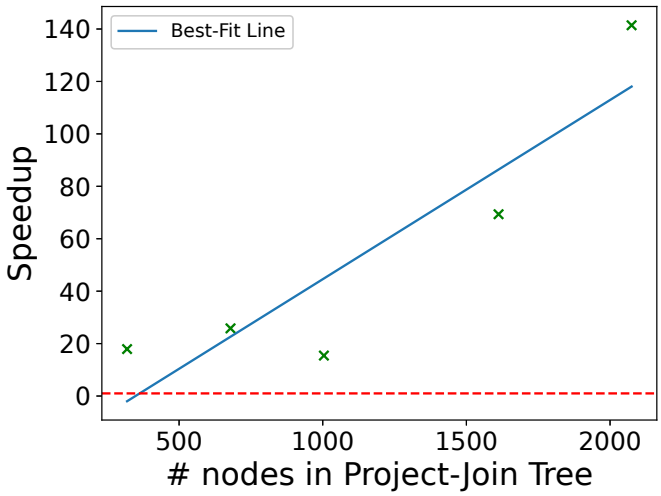


Figure C.1 : Speedup offered by Top-Down Sampling over Bottom-up

We also implemented a parallel version of `DPSampler` using the Sylvan library version 1.5. By plugging in Sylvan instead of CUDD as the ADD library, we get parallelization for 'free', which is a benefit derived from the modular nature of our algorithm. Note that only the Tree-of-ADD compilation phase was parallelized; the plannin and sample generation phases were still single-threaded. `DPSampler` with Sylvan was configured to use 15 threads. We used log-counting during ADD-compilation for both CUDD and Sylvan, as well as for computing sampling weights. `DPMC` with log-counting was seen to yield accurate model counts in the Model Counting Competition, and we did not encounter underflows in our experiments.

C.2.2 Comparison of ADD-Sampling algorithms

The aim of this experiment was to quantify the benefits of the new top-down ADD-sampling approach as described in Sec. 9.4, to the bottom-up sampling approach of [62]. In particular, we tested the hypothesis that the performance of the bottom-up procedure would degrade significantly as the number of nodes in the project-join tree increased, owing to the need for cofactoring the ADD at each node of the project-

	Both	Only WAPS	Only DPSampler
Bayes	679	0	264
Pseudoweighted	399	180	60
Total	1078	180	324

Table C.1 : Number of Benchmarks Successfully Solved

join tree, for each sample. We implemented two versions of `DPSampler` – one with the top-down subprocedure and another with the bottom-up one. For each tool, we attempted to generate 5000 samples in 1000 seconds. We computed the speedup offered by top-down sampling relative to bottom-up, i.e. the ratio of the time taken by bottom-up to the time taken by top-down. A ratio greater than 1 indicates that top-down is faster than bottom-up by that factor, and vice versa. In Fig. C.1, we plot the speedpup against the number of nodes in the project-join tree of 5 representative benchmarks. It can be seen that top-down sampling is faster than bottom-up by a factor ranging from 15 to 140, strongly depending on the size of project-join tree, as hypothesized.

These results clearly show the efficiency of top-down sampling vis-a-vis bottom-up. We observed a similar correlation between speedup and size of project-join tree on other benchmarks as well. Coupled with the fact that the median number of nodes in all the project-join trees constructed for Bayes benchmarks is 666, and 2656 for Pseudo-weighted, it is clear that bottom-up sampling is prohibitively expensive in practice.

C.2.3 Additonal Results on Comparison with WAPS

The individual cactus plots for Bayes and Pseudo-weighted benchmarks are shown in Figs. C.2 and C.3 respectively. A point (x, y) in the plot, implies that x instances took less than or equal to y seconds to solve. `DPSampler` is able to solve 264 more

	Both	Only WAPS	Only DPSampler
Bayes	838	19	122
Pseudoweighted	451	221	25
Total	1289	240	147

Table C.2 : Number of Benchmarks Successfully Compiled

	Compile Time	Total Time
Bayes	11.86	31.78
Pseudo-weighted	0.89	4.81
All	4.8	15.8

Table C.3 : Avg. (Geometric Mean) Speedups offered by `DPSampler` over `WAPS`

instances from the Bayes set as compared to `WAPS`, while `WAPS` is able to solve 120 more instances from the Pseudo-weighted set, as compared to `DPSampler`.

Tables C.1 and C.2 depict the number of benchmarks each sampler was successfully able to solve and compile respectively. Recall that we consider a benchmark to be solved, if the sampler is able to generate 5000 samples within a timeout of 1000s, and a benchmark is considered successfully compiled, if the sampler finishes constructing the d-DNNF (`WAPS` or `d4`) or the Tree-of-ADDs (`DPSampler`) from the input formula. We highlight that the compilation time includes the time taken for constructing the project-join tree. It can be seen that `DPSampler` is significantly superior to `WAPS` on Bayes benchmarks both in terms of compilation and completion times. This is also evident from the speedups offered by `DPSampler` (Table 9.1) For Pseudo-weighted benchmarks, `WAPS` and `d4` are able to solve and compile more benchmarks than `DPSampler`, but the speedups (which are computed on benchmarks that both samplers could solve/compile), are not very pronounced.

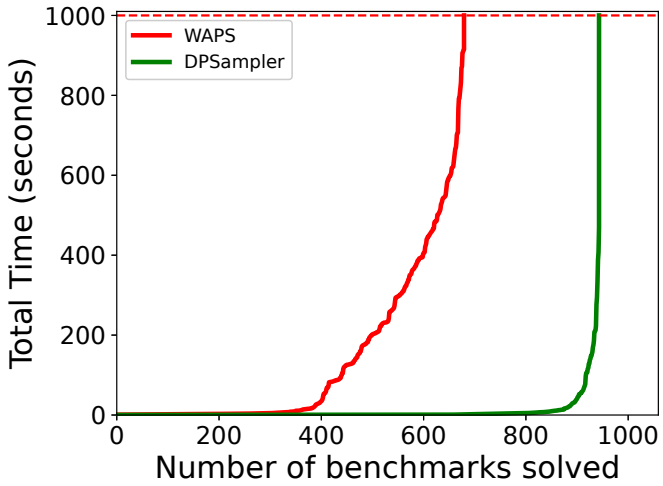


Figure C.2 : Performance comparison on 'Bayes' benchmark set

Out of the 221 Pseudo-weighted benchmarks that `WAPS` (`d4`) was able to compile but `DPSampler` could not, `DPSampler` failed on 165 because a project-join could not be constructed. Further, on the 1663 total benchmarks where a project-join tree could be computed, `DPSampler` succeeded in compilation on 84.7% of the benchmarks, and fully solved 82.7%. Project-join tree computation failed on 282 Pseudo-weighted benchmarks and 0 Bayes benchmarks. This shows that for `DPSampler`, the planning phase is the biggest bottleneck, and that finding better heuristics and tools for constructing project-join trees is likely to yield significant gains in the future. For instance, finding the right trade-off between waiting for a better tree-decomposition vs. spending that time on ADD construction is an interesting direction for future work.

In terms of compile time, `DPSampler` with Sylvan offered an average speedup of 1.99× on Bayes and 2.75× on Pseudo-weighted relative to `DPSampler` with CUDD. `DPSampler` with Sylvan was able to solve 5 more benchmarks from Pseudo-weighted as compared to `DPSampler` with CUDD, but failed on 1 Bayes benchmark that `DPSampler` with CUDD was able to solve.

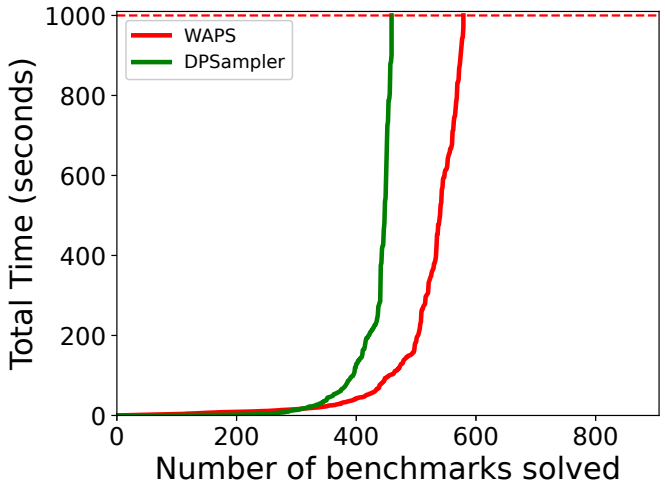


Figure C.3 : Performance comparison on 'Pseudo-weighted' benchmark set

Other observations We observed consistently that node-sharing among the different ADDs in a Tree-of-ADDs was very low. This indicates that using different variable orders for each ADD is likely to yield benefits. We note that the benefits from parallelization, while present (speedup by a factor of 2) are not very pronounced especially in terms of number of benchmarks solved. This was because most benchmarks that timed out for CUDD ended up running out of memory for Sylvan. Thus also indicates that more research into variable ordering and even using dynamic variable ordering can potentially help the parallelized a lot.