DPSampler: Exact Weighted Sampling Using Dynamic Programming*

Jeffrey M. Dudek, Aditya A. Shrotri, Moshe Y. Vardi
Rice University
{jmd11, as128, vardi}@rice.edu

Abstract

The problem of exact weighted sampling of solutions of Boolean formulas has applications in Bayesian inference, testing, and verification. The state-of-the-art approach to sampling involves carefully *decomposing* the input formula and compiling a data structure called d-DNNF in the process. Recent work in the closely connected field of model counting, however, has shown that smartly *composing* different subformulas using dynamic programming and Algebraic Decision Diagrams (ADDs) can outperform d-DNNF-style approaches on many benchmarks. In this work, we present a modular algorithm called `DPSampler` that extends such dynamic-programming techniques to the problem of exact weighted sampling.

`DPSampler` operates in three phases. First, an execution plan in the form of a project-join tree is computed using tree decompositions. Second, the plan is used to compile the input formula into a succinct tree-of-ADDs representation. Third, this tree is traversed to generate a random sample. This decoupling of planning, compilation and sampling phases enables usage of specialized libraries for each purpose in a black-box fashion. Further, our novel ADD-sampling algorithm avoids the need for expensive dynamic memory allocation required in previous work. Extensive experiments over diverse sets of benchmarks show `DPSampler` is more scalable and versatile than existing approaches.

1 Introduction

Given a Boolean formula F 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 F 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 [Bacchus *et al.*, 2003], testing and verification [Roy *et al.*, 2018; Naveh *et al.*, 2007; Chakraborty *et al.*, 2020].

There is a deep connection between sampling and the problem of model counting [Jerrum *et al.*, 1986], which has informed the design of modern sampling algorithms. The most popular approach to sampling is to carefully *decompose* the the input formula using heuristics and Boolean reasoning developed from research in model counting. A data structure called d-DNNF [Darwiche and Marquis, 2002], which is a succinct representation of the solution space, is compiled in the process and is used for generating samples quickly on demand through Markovian random walks. The study of data structures like d-DNNF in the field of knowledge compilation (KC) has contributed extensively to the theory and practice of counting and sampling. For example, the state-of-the-art sampler `WAPS` [Gupta *et al.*, 2019] uses the d-DNNF-compiler `d4` [Lagniez and Marquis, 2017]. `d4` doubles as a model counter and was highly placed in the recent Model Counting Competition[1]. Thus, the synergistic interplay between counting, sampling and KC has proven to be profitable for d-DNNF-based approaches.

Recently, a line of model-counting work that leverages a different data structure called Algebraic Decision Diagram (ADD) [Bahar *et al.*, 97] and dynamic programming has evolved in parallel to d-DNNF-based approaches [Dudek *et al.*, 2020a; Dudek *et al.*, 2020b]. It was shown that by smartly *composing* the solution spaces of different subformulas of the input formula using ADDs, it is possible to perform model counting extremely efficiently on formulas with low treewidth [Samer and Szeider, 2010; Dudek *et al.*, 2020b]. Real-world benchmarks in various domains often have low treewidth [Wang *et al.*, 2001]. For low-treewidth instances, the tool `DPMC` [Dudek *et al.*, 2020b] was shown to outperform state-of-the-art d-DNNF-based tools including `d4`. Unlike d-DNNF-based approaches, however, the interplay between counting, sampling and KC has not been leveraged for ADD-based techniques. The question left unanswered is: can we perform sampling by exploiting tree decompositions, using dynamic programming and ADDs?

In this work, we answer this question in the positive. Our algorithm, `DPSampler`, operates in three phases. In the first

*Code, results and full version of the text is available at https://www.gitlab.com/Shrotri/dpsampler

[1] https://mccompetition.org/2021/mc_description

phase, an execution plan in the form of a project-join tree is computed, based on the tree decomposition of the input formula. In the second phase, `DPSampler` compiles the input CNF into a succinct tree-of-ADDs representation based on the plan generated in first phase. In the 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. Further, our novel ADD-sampling algorithm avoids the need for expensive dynamic memory allocation required in previous work [Chakraborty *et al.*, 2020]. Extensive experiments over diverse sets of benchmarks arising from applications in AI show `DPSampler` is more scalable and versatile than existing approaches. In summary, our contributions are as follows:

1. `DPSampler`, the first weighted and uniform sampler based on dynamic programming that is able to exploit tree decompositions[2] ;
2. A new "in-place" ADD-sampling algorithm that is comprehensively faster than the previous approach; and
3. An empirical study that demonstrates that `DPSampler` has strong performance on diverse benchmarks.

2 Preliminaries

2.1 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 $\sigma \in \{0,1\}^X$ to be a set of literals i.e., $\sigma = \bigcup_{x\in X}\{lit_x\}$ where lit_x is either x (x assigned to true) or $\neg x$ (x assigned to false). For a function $f : \{0,1\}^X \to \mathbb{R}$ and a partial assignment $\sigma \in \{0,1\}^Y$ such that $Y \subseteq X$, the function obtained by projecting f on σ is denoted as $f[\sigma] : \{0,1\}^{X\setminus Y} \to \mathbb{R}$, defined for all $\sigma' \in \{0,1\}^{X\setminus Y}$ by $f[\sigma](\sigma') = f(\sigma \cup \sigma')$. If $Y \subseteq X$, and $\sigma \in \{0,1\}^X$ then we denote the restriction of σ to the variables in Y as σ_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 $\sigma \in \{0,1\}^X$, if σ satisfies φ i.e. $\sigma \models \varphi$, then $[\varphi](\sigma) \equiv 1$ else $[\varphi](\sigma) \equiv 0$. 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 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 $\sigma \in \{0,1\}^{X\cup Y}$ by $(f \cdot g)(\sigma) \equiv f(\sigma_X) \cdot g(\sigma_Y)$.*

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

[2]Works like [Bova *et al.*, 2015; Fichte *et al.*, 2022] can be extended to sampling in theory. but this has not been implemented and the practical performance is unknown.

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

Definition 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 $\sigma \in \{0,1\}^{X\setminus\{x\}}$ by $\left(\sum_x f\right)(\sigma) \equiv f(\sigma \cup \{\neg x\}) + f(\sigma \cup \{x\})$.*

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 $\sum_X f \equiv \sum_{x_1} \sum_{x_2} \cdots \sum_{x_n} f$. Our convention is that $\sum_\varnothing f \equiv f$.

2.2 Weighted Sampling and Counting

This paper is concerned with the problem of weighted sampling:

Definition 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 $\sigma \in \{0,1\}^X$,*

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

where $w(\varphi) \equiv \sum_{\sigma\models\varphi} w(\sigma)$ 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 $w(\varphi) = \sum_{\sigma\models\varphi} w(\sigma)$ 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 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 $\sigma \in \{0,1\}^X$,*

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

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 $\sigma \in \{0,1\}^{X'}$ by $w(X')(\sigma) = \prod_{\substack{x\in X' \\ \sigma(x)=1}} w(x) \cdot \prod_{\substack{x\in X' \\ \sigma(x)=0}} w(\neg x)$.

2.3 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 that 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.

2.4 Algebraic Decision Diagrams

An *algebraic decision diagram (ADD)* is a compact representation of a pseudo-Boolean function as a directed acyclic graph [Bahar *et al.*, 97]. For functions with logical structure, ADD representations can be exponentially smaller than the explicit representation. ADDs have been used for various applications such as matrix multiplication and shortest path algorithms [Bahar *et al.*, 97], Bayesian inference [Chavira and Darwiche, 2007; Gogate and Domingos, 2011] and stochastic planning [Hoey *et al.*, 1999], besides model counting.

Formally, an ADD is a tuple (X, S, ρ, G), where X is a set of Boolean variables, S is an arbitrary set (called the *carrier set*), $\rho : X \to \mathbb{N}$ is an injection (called the *diagram variable order*), and G is a rooted directed acyclic graph satisfying the following three properties. First, every leaf node of G is labeled with an element of S. Second, every internal node v 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 v (denoted $v.then$) and the node at the other end of the 0-edge is called the 'else-child' (denoted $v.else$). Finally, for every path in G, the labels of internal nodes must occur in increasing order under ρ. In this work, we consider ADDs with the carrier set $S = \mathbb{R}$. Each node v in an ADD represents a pseudo-Boolean function f_v. The function f represented by the ADD is same as the function represented by the root node r, i.e. $f \equiv f_r$.

3 Related Work

[Jerrum *et al.*, 1986] showed that uniform sampling can be done in probabilistic polynomial time relative to a Σ_2^P oracle. [Bellare *et al.*, 2000] improved this result to only require an NP-oracle. These approaches, however, are known to be impractical [Meel, 2018]. BDD-based sampling techniques [Yuan *et al.*, 2004] are also known to suffer from performance issues on real-world instances [Kitchen, 2010].

The first exact uniform sampling tool shown to perform well on standard benchmarks was `SPUR` [Achlioptas *et al.*, 2018], which generated samples on-the-fly without explicit compilation. Subsequently, the sampler `KUS` [Sharma *et al.*, 2018], which relied on d-DNNF compilation, was shown to significantly outperform `SPUR`. The tool `WAPS` [Gupta *et al.*, 2019] extended `KUS` to support weighted and projected sampling, and was shown to convincingly outperform even the approximate weighted-sampling tool `WeightGen` [Chakraborty *et al.*, 2015]. To the best of our knowledge, `WAPS` is currently the state-of-the-art exact weighted sampler. In Sec. 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 [Chakraborty *et al.*, 2014b; Chakraborty *et al.*, 2015; Soos *et al.*, 2020] provides strong guarantees on the "almost uniformity" of generated samples, while tools such as `QuickSampler` [Dutra *et al.*, 2018] and `XOR-Sample` [Gomes *et al.*, 2006] provide weak or no guarantees on the output distribution. In this work, we focus exclusively on exact (that is, no approximation) sampling.

Starting with the seminal work of [Darwiche and Marquis, 2002], a wide variety of tractable representations of Boolean functions such as d-DNNFs and SDDs [Darwiche, 2011], along with variants of OBDDs [Bryant, 1986], have been explored in literature under the umbrella of *knowledge compilation*. Additionally, [Fargier *et al.*, 2014] analyzed pseudo-Boolean representations including Algebraic Decision Diagrams [Bahar *et al.*, 97]. A number of compilers have also been developed such as `d4` [Lagniez and Marquis, 2017], `c2d` [Darwiche, 2004], `dSharp` [Muise *et al.*, 2012] etc. The compiled form generated by `DPSampler` is closely related to the tree-of-BDDs (ToB) language developed in [Subbarayan, 2005; Subbarayan *et al.*, 2007] and further analyzed in [Fargier and Marquis, 2009]. We note, however, that `DPSampler` actually compiles a tree-of-ADDs with some stark differences to the variants of ToB analyzed in [Fargier and Marquis, 2009]: (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 [Fargier and Marquis, 2009] is an interesting direction for future work. Recently, [de Colnet and Mengel, 2021] pointed out inherent limitations of compositional approaches to knowledge compilation when the target language is *str-DNNF*, which includes monolithic BDDs as a subset. However, the tree-of-ADDs language used in the current work is not known to be a subset of str-DNNF, and hence the results are not directly applicable here.

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 [Chakraborty *et al.*, 2020], in the context of a different problem of trace-sampling. While the algorithm of [Chakraborty *et al.*, 2020] could be used as-is for our purpose, it suffers from serious drawbacks in practice. 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* [Brayton *et al.*, 1984]. 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 our 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

Algorithm 1 sampleFromADD(f, w, σ)

Input: f: An ADD (X, S, ρ, G)
Input: w: A literal-weight function over X
Input: σ: An assignment to $Z \subseteq X$
Output: σ': An assignment to $Y = X \setminus Z$

```
1:  v ← root(f)
2:  computeWeights(f, w, v, σ, ∅)
3:  σ' ← ∅
4:  while v ∉ leaves(f) do
    /* x_v is the variable labeling node v */
5:      if x_v ∈ σ then                      ▷ x_v ∈ Z and assigned True
6:          v_next ← v.then
7:      else if ¬x_v ∈ σ then                ▷ x_v ∈ Z and assigned False
8:          v_next ← v.else
9:      else                                 ▷ x_v ∈ Y i.e. unassigned
10:         t_wt ← v.then.wt × w(x_v)
11:         e_wt ← v.else.wt × w(¬x_v)
12:         rand_bit ← weighted_sample(t_wt, e_wt)
13:         if rand_bit == True then
14:             σ' ← σ' ∪ {x_v}              ▷ Assign x_v to True
15:             v_next ← v.then
16:         else
17:             σ' ← σ' ∪ {¬x_v}             ▷ Assign x_v to False
18:             v_next ← v.else
19:     for x ∈ X \ Z s.t. ρ(x_v) < ρ(x) < ρ(x_{v_next}) do
20:         /*Process skipped variables*/
21:         rand_bit ← weighted_sample(w(x), w(¬x))
22:         if rand_bit == True then
23:             σ' ← σ' ∪ {x}                ▷ Assign x to True
24:         else
25:             σ' ← σ' ∪ {¬x}               ▷ Assign x to False
26:     v ← v_next
27: return σ'
```

cost amortization through compilation.

We present here a faster top-down algorithm for ADD-sampling. Procedure sampleFromADD (Alg. 1), takes as input an ADD f along with a partial assignment σ 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. 1) 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. 2) 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 1). 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

Algorithm 2 computeWeights$(f, w, v, \sigma, visited)$

Input: f: An ADD (X, S, ρ, G)
Input: w: A literal-weight function over X
Input: v: A node in f
Input: σ: 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 1)

```
1:  if v ∈ visited then
2:      return v.wt
3:  visited ← visited ∪ {v}
4:  if v ∈ leaves(f) then
5:      return v.val
    /* x_v is the variable labeling node v */
6:  if x_v ∈ σ then                          ▷ x_v ∈ Z and assigned True
7:      v.wt ← computeWeights(f, w, v.then, σ, visited)
8:  else if ¬x_v ∈ σ then                    ▷ x_v ∈ Z and assigned False
9:      v.wt ← computeWeights(f, w, v.else, σ, visited)
10: else                                     ▷ x_v is unassigned
11:     t_wt ← computeWeights(f, w, v.then, σ, visited)
12:     e_wt ← computeWeights(f, w, v.else, σ, visited)
13:     v.wt ← t_wt × w(x_v) + e_wt × w(x_v)
14: return v.wt
```

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 1. *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 σ 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*

$$wt = \sum_{Y_{\geq v}} w(Y_{\geq v}) \cdot f_v[\sigma_{Z_{\geq v}}] \quad (1)$$

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

Algorithm 3 DPSampler(X, φ, w, n)

Input: X: A set of variables; φ: A CNF formula over X
Input: w: A literal-weight function over X
Input: n: The number of weighted samples to generate
Output: $\sigma_1, \cdots, \sigma_n$: for each $1 \leq i \leq n$, σ_i is an independent w-weighted sample of φ.
1: $\mathcal{T} \leftarrow \mathsf{Plan}(\varphi)$ ▷ See Sec. 5.1
2: $S \leftarrow \mathsf{Compile}(\mathcal{T})$ ▷ See Sec. 5.2
3: **for** each i in $1 \leq i \leq n$ **do**
4: $\quad\sigma_i \leftarrow \mathsf{drawSample}(\mathcal{T}, root(\mathcal{T}), S, w, \varnothing)$ ▷ Alg. 5; see Sec. 5.3
5: **return** $\sigma_1, \cdots, \sigma_n$

to the variables in Y with probability

$$\Pr[Y = \sigma' | Z = \sigma] = \frac{w(\sigma') \cdot f[\sigma', \sigma]}{\sum_Y w(Y) \cdot f[\sigma]} \tag{2}$$

5 Sampling from a Boolean Formula

We now present our algorithm DPSampler, a three-phase algorithm for exact weighted sampling, in Alg. 3. While we use existing techniques [Dudek *et al.*, 2020b] 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 [Dudek *et al.*, 2020b] 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 4 on each ADD in the tree.

The following theorem asserts the correctness of Alg. 3.

Theorem 1. *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 1 appears in the full version.

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 [Dudek *et al.*, 2020b] 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 5 (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 $x \in$ Vars(c), then the leaf node $\gamma^{-1}(c)$ is a descendant of n.*

The model counting algorithm DPMC that was presented by [Dudek *et al.*, 2020b] 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 **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 [Dudek *et al.*, 2020b] 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.

Algorithm 4 $\mathsf{Compile}(\mathcal{T}, w)$

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
1: $S \leftarrow$ empty map
2: **procedure** $\mathsf{Compile_rec}(T, n, S, w)$
$\quad$**Input:** n: A node in $\mathcal{V}(T)$
$\quad$**Output:** f^n: an ADD corresponding to $n \in \mathcal{V}(T)$
3: $\quad$**if** $n \in \mathcal{L}(T)$ **then**
4: $\quad\quad f^n \leftarrow [\gamma(n)]$
5: $\quad$**else**
6: $\quad\quad f^n \leftarrow \top$
7: $\quad\quad$**for** $c \in C(n)$ **do**
8: $\quad\quad\quad f^c \leftarrow \mathsf{Compile_rec}(\mathcal{T}, c, S, w)$
9: $\quad\quad\quad f^n \leftarrow f^n \times \sum_{X^c} f^c \cdot w(X^c)$
10: $\quad S[n] \leftarrow f^n$ ▷ S is modified here
11: $\quad$**return** f^n
12: **end procedure**
13: $\mathsf{Compile_rec}(\mathcal{T}, r, S, w)$
14: **return** S

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 [Dudek *et al.*, 2020b]. Our goal is to construct a *tree-of-ADDs* using $\mathcal{T}$.

Algorithm 5 drawSample$(\mathcal{T}, n, S, w, \sigma)$

Input: $\mathcal{T} = (T, r, \gamma, \pi)$: A project-join tree
Input: n: A node in $\mathcal{V}(T)$; w: A literal-weight function
Input: S: A map from each $n \in \mathcal{V}(T)$ to ADD f^n
Input: σ: A preexisting assignment to some variables
Output: σ': A sampled assignment to all variables
1: $f^n \leftarrow S[n]$
2: $\sigma \leftarrow \sigma \cup \mathsf{sampleFromADD}(f^n, w, \sigma)$
3: **for** $c \in C(n)$ **do**
4: $\quad \sigma \leftarrow \sigma \cup \mathsf{drawSample}(\mathcal{T}, c, S, w, \sigma)$
5: **return** σ

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

Definition 6. *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 $n \in \mathcal{V}(T)$:*

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

To ease notation, within Def. 6 we define $X_\ell \equiv \varnothing$ for each $\ell \in \mathcal{L}(T)$. Recall that $[\gamma(n)]$ is the pseudo-Boolean function where $[\gamma(n)](\sigma) = 1$ if $\sigma \models \gamma(n)$ and 0 otherwise.

Note that, in contrast to the w-evaluation of [Dudek *et al.*, 2020b] 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$(\mathcal{T}, w)$ (Alg. 4) following Def. 6. Compile takes a project-join tree $\mathcal{T}$ as input and recursively applies product and projection operations to construct a tree-of-ADDs for $\mathcal{T}$. 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 [Dudek *et al.*, 2020b]).

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. 5 presents a procedure drawSample that generates samples from a tree-of-ADDs. drawSample is invoked in `DPSampler` with parameters that include an empty assignment σ, and the root node of the project-join tree. A full assignment σ 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.

6 Empirical Evaluation

We seek to answer the following questions in our study:

Benchmark Set	Compile Time	Total Time
DPV	4.8	15.8
GSRM	13.75	19.99
All (Unique)	4.7	14.9

Table 1: Average (Geometric mean) speedup of `DPSampler` relative to `WAPS` across each benchmark set

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. 1) perform compared to the bottom-up procedure of [Chakraborty *et al.*, 2020]?
3. How does `DPSampler` perform compared to the state-of-the-art, especially on low-treewidth instances?

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 [Somenzi, 2012] library version 3.0. `WAPS` as well as `DPSampler` with CUDD are single threaded. We used the tree-decomposition solver `FlowCutter` [Strasser, 2017] for the planning phase. The modularity of our framework yielded a parallel version of `DPSampler` by substituting CUDD with the library Sylvan [van Dijk and van de Pol, 2017] and no additional effort. For Q(1), we report that the distribution generated by `DPSampler` was indistinguishable from the ideal one, with Jensen-Shannon distance of 0.003. Additional details about the experimental setup, benchmarks, results on the parallel version and analysis of the output distribution are discussed in the full version.

6.1 Comparison with Bottom-Up Sampling

In order to answer Q(2), we implemented two versions of `DPSampler`: one with top-down ADD-Sampling as presented in Sec. 4, and another with the bottom-up sampling procedure of [Chakraborty *et al.*, 2020]. We tested the two implementations on the suite of 1945 benchmarks used in [Dudek *et al.*, 2020b], and compared the times taken by each to generate 5000 samples (excluding the time required for planning and compilation). We observed that the speedup offered by top-down sampling – i.e. the ratio of the time taken by bottom-up sampling to the time taken by top-down sampling on the same instance – was 19.6 on average (geometric mean), and the (geometric) standard deviation was 1.93. Top-down sampling was never slower than bottom-up, and also completed 23.8% more benchmarks overall. Henceforth, we exclusively use top-down sampling.

6.2 Comparison with the State-of-the-Art

Setup. We compared `DPSampler` with the state-of-the-art exact weighted sampling tool `WAPS` [Gupta *et al.*, 2019]. To the best of our knowledge, there are no other exact tools, and `WAPS` comprehensively outperforms the approximate sampler, WeightGen [Chakraborty *et al.*, 2014a]. We use the benchmark sets of weighted CNF formulas from both [Dudek

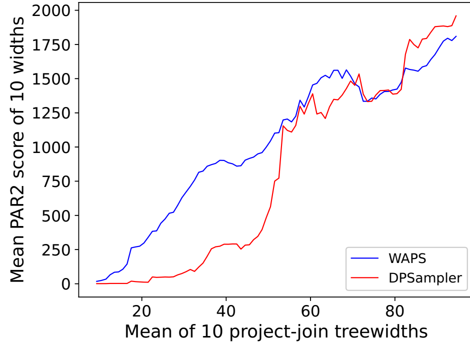

Figure 1: Average PAR2 Score vs. Average Treewidths

Benchmark Set	Both	Only WAPS	Only DPSampler
DPV	1078	180	324
GSRM	580	2	165
All (Unique)	1153	182	331

Table 2: Number of benchmarks successfully solved

et al., 2020b] (labeled 'DPV' with 1945 instances) and [Gupta *et al.*, 2019] (labeled '$GSRM$' with 773 instances). Note that 664 formulas are common to both sets. For ease of comparison with prior work, we present results on each benchmark set separately, as well as cumulative results over unique instances. A benchmark is "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.

Results. The results are shown in Tables 1, 2 and 3. In Table 1, it can be seen that `DPSampler` is consistently faster than `WAPS` in terms of the total time taken to generate 5000 samples. We also compare the time taken to compile the d-DNNF by `WAPS` and the tree-of-ADDs by `DPSampler`. For compiling the d-DNNF, `WAPS` relies on `d4`, a mature tool written in C++. However, the code for sampling from the compiled d-DNNF is written in Python and involves expensive disk reads and writes, which can adversely affect the running time. Thus comparing the compile times better shows the true potential of a d-DNNF-based approach. The performance of `WAPS` is unsurprisingly better in terms of compile time than total time. Nevertheless, `DPSampler` is still faster overall. Tables 2 and 3 show that `DPSampler` is able to uniquely solve more instances overall and is faster on the majority of instances.

Treewidths. We compare the performance of `WAPS` and `DPSampler` against treewidths in Fig. 1. Similar to [Dudek *et al.*, 2021], we plot mean PAR-2 scores (in seconds) against mean project-join tree widths. 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. $w_1 \leq w \leq w_{10}$. The performance of `DPSampler` is significantly better than `WAPS` upto treewidth 55, is roughly equal between 55 – 80, and is worse thereafter.

Benchmark Set	`WAPS`	`DPSampler`
DPV	221	1361
GSRM	2	745
All (Unique)	223	1443

Table 3: Number of benchmarks each tool is the fastest on

Failure Analysis. Out of the 182 benchmarks where `WAPS` succeeded but `DPSampler` failed, 137 failures were because a tree-decomposition could not be constructed. Further, out of the 1762 unique benchmarks where a tree-decomposition was obtained, `DPSampler` was successful in solving 1484 (84%). Thus, the planning phase is the biggest bottleneck.

7 Discussion

It is clear from Sec. 6.1 that top-down sampling is unequivocally superior to a bottom-up approach, and is crucial for overall scalability of `DPSampler`. Further, Fig. 1 confirms our hypothesis that `DPSampler` performs extremely well in the regime of low treewidths. We emphasize that unlike CDCL-based tools, which have enjoyed significant engineering effort over the years, `DPSampler` is an early prototype. Nevertheless, because of its modularity, `DPSampler` can easily be extended and enhanced in the future. For example, it can also be used with other tree-decomposition tools, or with off-the-shelf heuristics for constructing project-join trees such as those studied in [Dudek *et al.*, 2020b]. This can potentially address the bottleneck in the planning phase. `DPSampler` can also be used with *graded* project-join trees [Dudek *et al.*, 2021], for projected sampling. Further, the use of *dynamic variable ordering* for ADD construction was shown to greatly enhance sampling performance in [Chakraborty *et al.*, 2020], and can also be used with `DPSampler` in the future. We note that tree-decomposition-based heuristics were recently shown to significantly improve CDCL and d-DNNF-style model counting [Korhonen and Järvisalo, 2021]. Nevertheless, the factored compositional approach taken here has regularly proven to be an important part of the portfolio [Fichte *et al.*, 2022; Dudek *et al.*, 2019], and has ample room for algorithmic innovation.

Acknowledgements

Work supported in part by NSF grants IIS-1527668, CCF-1704883, IIS-1830549, CNS-2016656, DoD MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office. We thank the anonymous reviewers for extensive feedback.

References

[Achlioptas *et al.*, 2018] Dimitris Achlioptas, Zayd S Hammoudeh, and Panos Theodoropoulos. Fast sampling of perfectly uniform satisfying assignments. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 135–147. Springer, 2018.

[Bacchus *et al.*, 2003] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results

for# sat and bayesian inference. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 340–351. IEEE, 2003.

[Bahar *et al.*, 97] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Form Method Sys Des*, 10(2-3):171–206, '97.

[Bellare *et al.*, 2000] Mihir Bellare, Oded Goldreich, and Erez Petrank. Uniform generation of np-witnesses using an np-oracle. *Information and Computation*, 163(2):510–526, 2000.

[Bova *et al.*, 2015] Simone Bova, Florent Capelli, Stefan Mengel, and Friedrich Slivovsky. On compiling cnfs into structured deterministic dnnfs. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 199–214. Springer, 2015.

[Brayton *et al.*, 1984] Robert K Brayton, Gary D Hachtel, Curt McMullen, and Alberto Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*, volume 2. Springer Science & Business Media, 1984.

[Bryant, 1986] Randal E Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE TC*, 100(8):677–691, 1986.

[Chakraborty *et al.*, 2014a] Supratik Chakraborty, Daniel Fremont, Kuldeep Meel, Sanjit Seshia, and Moshe Vardi. Distribution-aware sampling and weighted model counting for sat. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[Chakraborty *et al.*, 2014b] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. Balancing scalability and uniformity in sat witness generator. In *2014 51st acm/edac/ieee design automation conference (dac)*, pages 1–6. IEEE, 2014.

[Chakraborty *et al.*, 2015] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. On parallel scalable uniform sat witness generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 304–319. Springer, 2015.

[Chakraborty *et al.*, 2020] Supratik Chakraborty, Aditya A Shrotri, and Moshe Y Vardi. On Uniformly Sampling Traces of a Transition System. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[Chavira and Darwiche, 2007] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks using variable elimination. In *IJCAI*, pages 2443–2449, 2007.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[Darwiche, 2004] Adnan Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332. Citeseer, 2004.

[Darwiche, 2011] Adnan Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[de Colnet and Mengel, 2021] Alexis de Colnet and Stefan Mengel. Lower bounds on intermediate results in bottom-up knowledge compilation. *arXiv preprint arXiv:2112.12430*, 2021.

[Dudek *et al.*, 2019] Jeffrey M Dudek, Leonardo Dueñas-Osorio, and Moshe Y Vardi. Efficient contraction of large tensor networks for weighted model counting through graph decompositions. *preprint arXiv:1908.04381*, 2019.

[Dudek *et al.*, 2020a] Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *AAAI*, volume 34, pages 1468–1476, 2020.

[Dudek *et al.*, 2020b] Jeffrey M Dudek, Vu HN Phan, and Moshe Y Vardi. Dpmc: Weighted model counting by dynamic programming on project-join trees. In *International Conference on Principles and Practice of Constraint Programming*, pages 211–230. Springer, 2020.

[Dudek *et al.*, 2021] Jeffrey M Dudek, Vu HN Phan, and Moshe Y Vardi. Procount: Weighted projected model counting with graded project-join trees. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 152–170. Springer, 2021.

[Dutra *et al.*, 2018] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of sat solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 549–559. IEEE, 2018.

[Fargier and Marquis, 2009] Hélène Fargier and Pierre Marquis. Knowledge compilation properties of trees-of-bdds, revisited. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[Fargier *et al.*, 2014] Hélene Fargier, Pierre Marquis, Alexandre Niveau, and Nicolas Schmidt. A knowledge compilation map for ordered real-valued decision diagrams. In *AAAI*, 2014.

[Fichte *et al.*, 2022] Johannes K Fichte, Markus Hecher, Patrick Thier, and Stefan Woltran. Exploiting database management systems and treewidth for counting. *Theory and Practice of Logic Programming*, 22(1):128–157, 2022.

[Gogate and Domingos, 2011] Vibhav Gogate and Pedro Domingos. Approximation by Quantization. In *UAI*, pages 247–255, 2011.

[Gomes *et al.*, 2006] Carla P Gomes, Ashish Sabharwal, and Bart Selman. Near-uniform sampling of combinatorial spaces using xor constraints. In *NIPS*, pages 481–488, 2006.

[Gupta *et al.*, 2019] Rahul Gupta, Shubham Sharma, Subhajit Roy, and Kuldeep S Meel. Waps: Weighted and projected sampling. In *TACAS (1)*, pages 59–76, 2019.

[Hoey *et al.*, 1999] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: stochastic planning using decision diagrams. In *UAI*, pages 279–288, 1999.

[Jerrum *et al.*, 1986] Mark R Jerrum, Leslie G Valiant, and Vijay V Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical computer science*, 43:169–188, 1986.

[Kitchen, 2010] Nathan Boyd Kitchen. *Markov Chain Monte Carlo stimulus generation for constrained random simulation*. University of California, Berkeley, 2010.

[Korhonen and Järvisalo, 2021] Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[Lagniez and Marquis, 2017] Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *IJCAI*, pages 667–673, 2017.

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

[Muise *et al.*, 2012] Christian Muise, Sheila A McIlraith, J Christopher Beck, and Eric I Hsu. D sharp: fast d-dnnf compilation with sharpsat. In *Canadian Conference on Artificial Intelligence*, pages 356–361. Springer, 2012.

[Naveh *et al.*, 2007] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI magazine*, 28(3):13–13, 2007.

[Roy *et al.*, 2018] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu 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*, pages 224–234, 2018.

[Samer and Szeider, 2010] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.

[Sharma *et al.*, 2018] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. Knowledge compilation meets uniform sampling. In *LPAR*, pages 620–636, 2018.

[Somenzi, 2012] Fabio Somenzi. Cudd: Cu decision diagram package-release 2.4. 0. *University of Colorado at Boulder*, 2012.

[Soos *et al.*, 2020] Mate Soos, Stephan Gocht, and Kuldeep S Meel. Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling. In *International Conference on Computer Aided Verification*, pages 463–484. Springer, 2020.

[Strasser, 2017] Ben Strasser. Computing tree decompositions with flowcutter: Pace 2017 submission. *arXiv preprint arXiv:1709.08949*, 2017.

[Subbarayan *et al.*, 2007] Sathiamoorthy Subbarayan, Lucas Bordeaux, and Youssef Hamadi. Knowledge compilation properties of tree-of-bdds. In *AAAI*, pages 502–507, 2007.

[Subbarayan, 2005] Sathiamoorthy 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*, pages 351–365. Springer, 2005.

[van Dijk and van de Pol, 2017] Tom van Dijk and Jaco van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19(6):675–696, 2017.

[Wang *et al.*, 2001] Dong Wang, Edmund Clarke, Yunshan Zhu, and James Kukula. Using cutwidth to improve symbolic simulation and boolean satisfiability. In *Sixth IEEE International High-Level Design Validation and Test Workshop*, pages 165–170. IEEE, 2001.

[Yuan *et al.*, 2004] Jun Yuan, Adnan Aziz, Carl Pixley, and Ken Albin. Simplifying boolean constraint solving for random simulation-vector generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):412–420, 2004.