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 assign 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 [Bachchus 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 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., 1997] 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

\(^1\)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. 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 $f : \{0, 1\}^X \rightarrow \mathbb{R}$, 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., $\sigma = \bigcup_{x \in X} \{x, \neg x\}$ where $x$ is either $x$ (assigned to true) or $\neg x$ (assigned to false). For a function $f : \{0, 1\}^X \rightarrow \mathbb{R}$ and a partial assignment $\sigma \in \{0, 1\}^Y$ such that $Y \subseteq X$, the function obtained by projecting $f$ on $\sigma$ is denoted as $f|_{\sigma} : \{0, 1\}^{X \setminus Y} \rightarrow \mathbb{R}$, 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 $\sigma$ to the variables in $Y$ as $\sigma|_Y$.

A Boolean formula $\varphi$ over variables $X$ represents a pseudo-Boolean function over $X$, denoted $[\varphi] : \{0, 1\}^X \rightarrow \mathbb{R}$, where for all $\sigma \in \{0, 1\}^X$, if $\varphi$ satisfies $\varphi$ i.e. $\sigma \models \varphi$, then $[\varphi](\sigma) = 1$ else $[\varphi](\sigma) = 0$. 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 $f : \{0, 1\}^X \rightarrow \mathbb{R}$ and $g : \{0, 1\}^Y \rightarrow \mathbb{R}$ is the function $f \cdot g : \{0, 1\}^{X \cup Y} \rightarrow \mathbb{R}$ 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 $\varphi$ and $\psi$ are propositional formulas, then $[\varphi] \cdot [\psi] = [\varphi \land \psi]$.

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 $f : \{0, 1\}^X \rightarrow \mathbb{R}$ w.r.t. $x$ is the function $\sum_x f : \{0, 1\}^{X \setminus \{x\}} \rightarrow \mathbb{R}$ defined for all $\sigma \in \{0, 1\}^{X \setminus \{x\}}$ by $\sum_x f(\sigma) \equiv f(\sigma \cup \{\neg x\}) + f(\sigma \cup \{x\})$.

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 $f : \{0, 1\}^X \rightarrow \mathbb{R}$. 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 $\sum_{\emptyset} f \equiv f$.

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, $\varphi$ be a Boolean formula over $X$, and $w : \{0, 1\}^X \rightarrow \mathbb{R}^{\geq 0}$ be a pseudo-Boolean function (called the weight function).

A random variable $S$ with sample space $\{0, 1\}^X$ is a weighted sample of $\varphi$ if, for all $\sigma \in \{0, 1\}^X$,$$
\Pr[S = \sigma] = \frac{w(\sigma)/w(\varphi)}{\text{if } \sigma \models \varphi} = \frac{0}{\text{if } \sigma \not\models \varphi}
$$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 $\varphi$ is also called a uniform sample of $\varphi$. The normalization factor $w(\varphi) = \sum_{\sigma \models \varphi} w(\sigma)$ is well-studied independently and is known as the weighted model count of $\varphi$ w.r.t. $w$.

We focus on sampling with respect to literal-weight functions:

**Definition 4 (Literal-Weight Function).** A pseudo-Boolean function $w : \{0, 1\}^X \rightarrow \mathbb{R}^{\geq 0}$ 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 $\sigma \in \{0, 1\}^X$,$$
w(\sigma) = \prod_{x \in X} w(x) \cdot \prod_{\sigma(x) = 0} w(\neg x).
$$

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

2.3 Graphs

For a graph $G$, we denote the set of vertices/nodes by $V(G)$ and set of edges by $E(G)$. We denote graphs that are trees by $T$ and the leaves of $T$ as $L(T)$. A rooted tree $(T, r)$ is a tree $T$ together with a distinguished root node $r \in V(T)$. The children of a node $n \in V(T)$ in a rooted tree are denoted $C(n)$.
and $\mathcal{C}(n) = \emptyset$ 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 $(X, S, p, G)$, where $X$ is a set of Boolean variables, $S$ is an arbitrary set (called the carrier set), $p : X \rightarrow \mathbb{N}$ 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 $\rho$. 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 $\Sigma^P_2$ oracle. [Bellare et al., 2000] improved this result to only require an NP-oracle. These approaches, however, are known to be impractical [Meel, 2018]. ADD-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 is currently the state-of-the-art exact weighted sampler. 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 ⊆ X
Output: σ': An assignment to Y = X \ Z

1: v ← root(f)
2: computeWeights(f, w, v, σ, ∅)
3: σ' ← ∅
4: while v ∉ leaves(f) do
5: /* x_v is the variable labeling node v */
6: if x_v ∈ σ then ▷ v ∈ Z and assigned True
7: v.next ← v.then
8: else if ¬x_v ∈ σ then ▷ v ∈ Z and assigned False
9: v.next ← v.else
10: else ▷ v ∈ Y i.e. unassigned
11: t_wt ← v.then.wt × w(x_v)
12: e_wt ← v.else.wt × w(¬x_v)
13: rand_bit ← weighted_sample(t_wt, e_wt)
14: if rand_bit == True then
15: σ' ← σ' ∪ {x_v} ▷ Assign x_v to True
16: v.next ← v.then
17: else
18: σ' ← σ' ∪ {¬x_v} ▷ Assign x_v to False
19: v.next ← v.else
20: /* Process skipped variables */
21: rand_bit ← weighted_sample(w(x_v), w(¬x_v))
22: if rand_bit == True then
23: σ' ← σ' ∪ {x_v} ▷ Assign x_v to True
24: else
25: σ' ← σ' ∪ {¬x_v} ▷ Assign x_v 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-ADDSs 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 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, ρ, G) with weight function w, an unvisited node v and an assignment σ to the variables Z ⊆ X. Let Y = X \ Z be the set of unassigned variables, Y ≥ v = {x ∈ Y | ρ(x) ≥ ρ(x_v)}, and Z ≥ v = {x ∈ Z | ρ(x) ≥ ρ(x_v)}. Then we have

\[
wt = \sum_{Y ≥ v} w(Y ≥ v) \cdot f_v[σ_{Z ≥ v}]
\]  

(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] = 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_e, using the corresponding literal weights.

Lemma 2. Let sampleFromADD be invoked on an ADD f = (X, S, ρ, G), weight function w, and an assignment σ to the variables Z ⊆ X. Let Y = X \ Z be the set of unassigned variables. Then sampleFromADD returns an assignment σ'
Algorithm 3 \texttt{DPSampler}(X, \varphi, w, n)

\begin{algorithm}
\begin{align*}
\textbf{Input:} & \ X: \text{ a set of variables; } \varphi: \text{ a CNF formula over } X \\
\textbf{Input:} & \ w: \text{ a literal-weight function over } X \\
\textbf{Input:} & \ n: \text{ The number of weighted samples to generate} \\
\textbf{Output:} & \sigma_1, \ldots, \sigma_n: \text{ for each } 1 \leq i \leq n, \sigma_i \text{ is an independent } w \text{-weighted sample of } \varphi. \\
1: & T \leftarrow \text{Plan}(\varphi) \quad \triangleright \text{ See Sec. 5.1} \\
2: & S \leftarrow \text{Compile}(T) \quad \triangleright \text{ See Sec. 5.2} \\
3: & \text{ for each } i \text{ in } 1 \leq i \leq n \text{ do} \\
4: & \quad \sigma_i \leftarrow \text{drawSample}(T, \text{root}(T), S, \emptyset) \quad \triangleright \text{ Alg. 5; } \text{ see Sec. 5.3} \\
5: & \text{ return } \sigma_1, \ldots, \sigma_n
\end{align*}
\end{algorithm}

to the variables in \( Y \) with probability
\[ \Pr[Y = \sigma' | Z = \sigma] = \frac{w(\sigma') \cdot f[\sigma', \sigma]}{\sum_{Y} w(Y) \cdot f[\sigma]} \quad (2) \]

5 Sampling from a Boolean Formula

We now present our algorithm \texttt{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 \texttt{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.

\textbf{Theorem 1.} Let \( X \) be a set of Boolean variables, \( \varphi \) be a CNF formula over \( X \), \( w \) be a literal-weight function over \( X \), and \( n \) be a positive integer. If \( \sigma_1, \ldots, \sigma_n \) is the sequence of random assignments returned by \texttt{DPSampler}(X, \varphi, w, n), then \( \sigma_1, \ldots, \sigma_n \) are i.i.d. \( w \)-weighted samples of \( \varphi \).

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 \texttt{DPMC} for model counting. The key idea is to represent the model counting computation as a rooted tree, called a \textit{project-join tree}, where leaves correspond to clauses, and internal nodes correspond to projections. Formally:

\textbf{Definition 5 (Project-Join Tree).} Let \( \varphi \) be a CNF formula over a set of variables \( X \). A project-join tree of \( \varphi \) is a tuple \( \mathcal{T} = (T, r, \gamma, X) \) where

\begin{itemize}
\item \( T \) is a tree with root \( r \in \mathbb{V}(T) \).
\end{itemize}

\end{itemize}
We seek to answer the following questions in our study:

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

<table>
<thead>
<tr>
<th>Benchmark Set</th>
<th>Compile Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPV</td>
<td>4.8</td>
<td>15.8</td>
</tr>
<tr>
<td>GSRM</td>
<td>13.75</td>
<td>19.99</td>
</tr>
<tr>
<td>All (Unique)</td>
<td>4.7</td>
<td>14.9</td>
</tr>
</tbody>
</table>

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

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

**Definition 6.** Let $T = (T, r, \gamma, X)$ be a project-join tree and let $w$ be a literal-weight function. A tree-of-ADDs for $T$ is a set of pseudo-Boolean functions $S = \{f^n : n \in V(T)\}$ defined recursively by, for each $n \in V(T)$:

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

To ease notation, within Def. 6 we define $X_\ell \equiv \varnothing$ for each $\ell \in L(T)$. 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 $n \in V(T) \setminus L(T)$ 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 $\text{Compile}(T, w)$ (Alg. 4) following Def. 6. Compile takes a project-join tree $T$ as input and recursively applies product and projection operations to construct a tree-of-ADDs for $T$. 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 $\text{drawSample}$ that generates samples from a tree-of-ADDs. $\text{drawSample}$ is invoked in DPSampler with parameters that include an empty assignment $\sigma$, and the root node of the project-join tree. A full assignment $\sigma$ to all variables in $X$ (the variable set of the input formula) is recursively sampled piece-wise by $\text{drawSample}$, 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 $X = \{X^n \mid c \in V(T) \setminus L(T)\}$ 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:
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 up to treewidth 55, is roughly equal between 55 – 80, and is worse thereafter.

**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 Järviselä, 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-2016666, DoD MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office. We thank the anonymous reviewers for extensive feedback.

**References**


