



WAPS: Weighted and Projected Sampling

Rahul Gupta¹(✉), Shubham Sharma¹, Subhajit Roy¹, and Kuldeep S. Meel²

¹ Indian Institute of Technology Kanpur, Kanpur, India
{grahul, smsharma, subhajit}@iitk.ac.in

² National University of Singapore, Singapore, Singapore
meel@comp.nus.edu.sg

Abstract. Given a set of constraints F and a user-defined weight function W on the assignment space, the problem of constrained sampling is to sample satisfying assignments of F conditioned on W . Constrained sampling is a fundamental problem with applications in probabilistic reasoning, synthesis, software and hardware testing. Consequently, the problem of sampling has been subject to intense theoretical and practical investigations over the years. Despite such intense investigations, there still remains a gap between theory and practice. In particular, there has been significant progress in the development of sampling techniques when W is a uniform distribution, but such techniques fail to handle general weight functions W . Furthermore, we are, often, interested in Σ_1^1 formulas, i.e., $G(X) := \exists Y F(X, Y)$ for some F ; typically the set of variables Y are introduced as auxiliary variables during encoding of constraints to F . In this context, one wonders *whether it is possible to design sampling techniques whose runtime performance is agnostic to the underlying weight distribution and can handle Σ_1^1 formulas?*

The primary contribution of this work is a novel technique, called WAPS, for sampling over Σ_1^1 whose runtime is agnostic to W . WAPS is based on our recently discovered connection between knowledge compilation and uniform sampling. WAPS proceeds by compiling F into a well studied compiled form, d-DNNF, which allows sampling operations to be conducted in linear time in the size of the compiled form. We demonstrate that WAPS can significantly outperform existing state-of-the-art weighted and projected sampler WeightGen, by up to 3 orders of magnitude in runtime while achieving a geometric speedup of $296\times$ and solving 564 more instances out of 773. The distribution generated by WAPS is statistically indistinguishable from that generated by an ideal weighted and projected sampler. Furthermore, WAPS is almost oblivious to the number of samples requested.

The original version of this chapter was revised: The acknowledgement was modified. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-17462-0_29

Appendix along with open source tool is available at <https://github.com/meelgroup/waps>.

This work was supported in part by NUS ODPRT Grant R-252-000-685-133 and National Research Foundation Singapore under its AI Singapore Programme [Award Number: AISG-RP-2018-005]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore <https://www.nscg.sg>.

© The Author(s) 2019

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 59–76, 2019.

https://doi.org/10.1007/978-3-030-17462-0_4

1 Introduction

Boolean satisfiability (SAT) has gathered applications in bounded model checking of hardware and software systems [5, 7, 51], classical planning [35] and scheduling [27]. Despite the worst-case hardness of SAT, the past few decades have witnessed a significant improvement in the runtime performance of the state-of-the-art SAT solvers [41]. This improvement has led to the usage of SAT solvers as oracles to handle problems whose complexity lies beyond NP. Among these problems, *constrained sampling*, that concerns with sampling from the space of solutions of a set of constraints F , subject to a user-defined weight function W , has witnessed a surge of interest owing to the wide range of applications ranging from machine learning, probabilistic reasoning, software and hardware verification to statistical physics [3, 32, 39, 45].

Not surprisingly, the problem of sampling is known to be computationally intractable. When the weight function W is fixed to a uniform distribution, the problem of constrained sampling is also known as uniform sampling. Uniform sampling has witnessed a long-standing interest from theoreticians and practitioners alike [4, 33, 38, 45]. The past few years, however, have witnessed a significant improvement in the runtime performance of the sampling tools when the weight function W is fixed to a uniform distribution owing to the rise of hashing-based paradigm [2, 11, 13, 22]. While the significant progress for uniform sampling has paved the way for its usage in constrained random simulation [45], the restriction of uniform distribution is limiting, and several applications of constrained sampling require the underlying techniques to be able to handle a wide variety of distributions and related problem formulations as listed below:

Literal-Weighted Sampling. In case of literal-weighted sampling, we consider the weight function over assignments defined as the product of the weight of literals, which is specified using a weight function $W(\cdot)$ that assigns a non-negative weight to each literal l in a boolean formula F . As argued in [12], literal-weighted weight function suffices for most of the practical applications ranging from constrained random simulation, probabilistic reasoning, and reliability of power-grids [10, 14, 21, 45].

Projected Sampling. Typically, users define constraints in high-level modeling languages such as Verilog [1], Bayesian networks [14] and configuration of grids [21] and then CNF encodings are employed to convert them into a CNF [6]. Commonly used schemes like *Tseitin encoding* [50] introduce auxiliary variables during encoding; though the encoded formulas are equisatisfiable, they typically do not preserve the number of satisfying assignments. In particular, given an initial set of constraints G expressed over a set of variables X , we obtain another formula F such that $G(X) = \exists Y F(X, Y)$. Therefore, we are concerned with sampling over solutions of F *projected over a subset of variables* (such as X in this case). In other words, we are concerned with sampling over Σ_1^1 formulas.

Conditioned Sampling. Given a boolean formula Φ and a partial assignment σ , conditioned sampling refers to sampling from the models of Φ that satisfy σ . Conditioning has interesting applications in testing where one is interested in fuzzing the system with inputs that satisfy certain patterns (pre-conditions). Conditioning has been applied in the past for fault diagnosis [23], conformant planning [46] and databases [15].

Typically, practical applications require sampling techniques that can handle all the above formulations. While techniques based on interval propagation, binary decision diagrams and random perturbation of solution space [22, 25, 44] cannot handle projection, conditioned, and weighted sampling efficiently, the hashing-based techniques have significantly improved the scalability of sampling techniques and are capable of handling projection and literal-weighted scheme [11, 42]. However, the performance of hashing-based techniques is extremely limited in their ability to handle literal-weighted sampling, and one observes a drastic drop in their performance as the weight distribution shifts away from uniform. In this context, one wonders: *whether it is possible to design techniques which can handle projection, conditioned, and literal-weighted sampling without degradation in their performance?*

In this work, we answer the above question in affirmative: we extend our previously proposed knowledge compilation framework in the context of uniform sampling to handle all the three variants. We have implemented a prototype of our framework, named WAPS, and demonstrate that within a time limit of 1800 s, WAPS performs better than the current state-of-the-art weighted and projected sampler WeightGen [10], by up to 3 orders of magnitude in terms of runtime while achieving a geometric speedup of $296\times$. Out of the 773 benchmarks available, WAPS was able to sample from 588 benchmarks while WeightGen was able to sample from only 24 benchmarks. Furthermore, WAPS is almost oblivious to the number of samples requested.

A significant advantage of our framework is its simplicity: we show that our previously proposed framework in the context of uniform sampling, KUS [49], can be lifted to handle literal-weighted, projection and conditioned sampling. We demonstrate that unlike hashing-based techniques, the runtime performance of WAPS is not dependent on the underlying weight distribution. We want to assert that the simplicity of our framework, combined with its runtime performance and its ability to be agnostic to the underlying distribution is a significant novel contribution to the area of constrained sampling. Besides, an important contribution of our work is the theoretical analysis of sampling techniques that employ knowledge compilation.

The rest of the paper is organized as follows. We first discuss the related work in Sect. 2. We then introduce notations and preliminaries in Sect. 3. In Sect. 4 we present WAPS and do theoretical analysis of WAPS in Sect. 5. We then describe the experimental methodology and discuss results in Sect. 6. Finally, we conclude in Sect. 7.

2 Related Work

Weighted sampling is extensively studied in the literature with the objective of providing scalability while ensuring strong theoretical guarantees. Markov Chain Monte Carlo (MCMC) sampling [32,40] is the most popular technique for weighted sampling; several algorithms like Metropolis-Hastings and simulated annealing have been extensively studied in the literature [36,40]. While MCMC based sampling is guaranteed to converge to a target distribution under mild requirements, convergence is often impractically slow [31]. The practical adaptations for MCMC-based sampling in the context of constrained-random verification has been proposed in [37]. Unfortunately, practical MCMC based sampling tools use heuristics that destroy the theoretical guarantees. Interval-propagation and belief networks have also been employed for sampling [20,26,29], but, though these techniques are scalable, the generated distributions can deviate significantly from the uniform distribution, as shown in [38].

To bridge the wide gap between scalable algorithms and those that give strong guarantees of uniformity several hashing-based techniques have been proposed [10,11,24,28] for weighted sampling. The key idea behind hashing-based techniques is to employ random parity constraints as pairwise independent hash functions to partition the set of satisfying assignments of CNF formula into cells. The hashing-based techniques have achieved significant runtime performance improvement in case of uniform sampling but their scalability suffers for weight distribution and depends strongly on parameters such as *tilt*, which are unlikely to be small for most practical distributions [42].

In recent past, a significant amount of work has been done to compile propositional theory, often represented as a propositional formula in CNF into tractable knowledge representations. One of the prominent and earliest representations is Ordered Binary Decision Diagrams (OBDDs), which have been effectively used for circuit analysis and synthesis [9]. Another family of representations known as Deterministic Decomposable Negation Normal Form (d-DNNF) [19] have proved to be influential in many probabilistic reasoning applications [14,17,18]. Recently, another representation called as Sentential Decision Diagram (SDD) [16] was proposed which maintains canonicity and polytime support for boolean combinations and bridged the gap of succinctness between OBDDs and d-DNNFs. In our recent work [49], we were able to tackle the problem of uniform sampling by exploiting the properties of d-DNNF. Specifically, we were able to take advantage of recent advancements made in the field of knowledge compilation and use the compiled structure to generate uniform samples while competing with the state-of-the-art tools for uniform sampling.

3 Notations and Preliminaries

A literal is a boolean variable or its negation. A clause is a disjunction of a set of literals. A propositional formula F in conjunctive normal form (CNF) is a conjunction of clauses. Let $Vars(F)$ be the set of variables appearing in F . The set $Vars(F)$

is called *support* of F . A *satisfying assignment* or *witness* of F , denoted by σ , is an assignment of truth values to variables in its support such that F evaluates to true. We denote the set of all witnesses of F as R_F . Let $var(l)$ denote the variable of literal l , i.e., $var(l) = var(\neg l)$ and $F|_l$ denotes the formula obtained when literal l is set to true in F . Given an assignment σ over $Vars(F)$ and a set of variables $P \subseteq Vars(F)$, define $\sigma_P = \{l \mid l \in \sigma, var(l) \in P\}$ and $R_{F \downarrow P}$ to be the projection of R_F onto P , i.e., $R_{F \downarrow P} = \{\sigma_P \mid \sigma \in R_F\}$.

Given a propositional formula F and a weight function $W(\cdot)$ that assigns a non-negative weight to every literal, the weight of assignment σ denoted as $W(\sigma)$ is the product of weights of all the literals appearing in σ , i.e., $W(\sigma) = \prod_{l \in \sigma} W(l)$. The weight of a set of assignments Y is given by $W(Y) = \sum_{\sigma \in Y} W(\sigma)$. Note that, we have overloaded the definition of weight function $W(\cdot)$ to support different arguments – a literal, an assignment and a set of assignments. We want to highlight that the assumption about weight distribution being generated solely by a literal-weighted function stands well, as many real-world applications like probabilistic inference can be efficiently reduced to literal-weighted sampling [14]. Also, for notational convenience, whenever the formula F , weight function W and sampling set P is clear from the context, we omit mentioning it.

3.1 Weighted and Projected Generators

A *weighted and projected probabilistic generator* is a probabilistic algorithm that generates a witness from $R_{F \downarrow P}$ with respect to weight distribution generated by weight function W . A *weighted and projected generator* $\mathcal{G}^{wp}(\cdot, \cdot, \cdot)$ is a probabilistic generator that guarantees

$$\forall y \in R_{F \downarrow P}, \Pr[\mathcal{G}^{wp}(F, P, W) = y] = \frac{W(y)}{W(R_{F \downarrow P})},$$

An *almost weighted and projected generator* $\mathcal{G}^{awp}(\cdot, \cdot, \cdot)$ relaxes this requirement, ensuring that: given a tolerance $\varepsilon > 0$, $\forall y \in R_{F \downarrow P}$ we have

$$\frac{W(y)}{(1 + \varepsilon)W(R_{F \downarrow P})} \leq \Pr[\mathcal{G}^{awp}(F, P, W) = y] \leq \frac{(1 + \varepsilon)W(y)}{W(R_{F \downarrow P})},$$

Probabilistic generators are allowed to occasionally “fail” in the sense that no witness may be returned even if $R_{F \downarrow P}$ is non-empty. The failure probability for such generators must be bounded by a constant strictly less than 1.

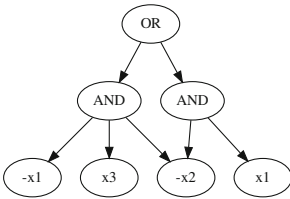


Fig. 1. Example of d-DNNF

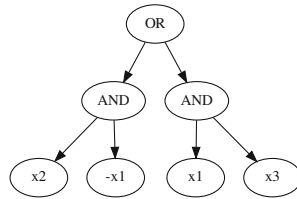


Fig. 2. The projected d-DNNF of Example 1

3.2 Deterministic Decomposable Negation Normal Form (d-DNNF)

To formally define d-DNNF, we first define the Negation Normal Form (NNF):

Definition 1 [19]. *Let X be the set of propositional variables. A sentence in NNF is a rooted, directed acyclic graph (DAG) where each leaf node i is labeled with true, false, x or $\neg x$, $x \in X$; and each internal node is labeled with \vee or \wedge and can have arbitrarily many children.*

d-DNNF further imposes that the representation is:

- **Deterministic:** An NNF is deterministic if the operands of \vee in all well-formed boolean formula in the NNF are mutually inconsistent.
- **Decomposable:** An NNF is decomposable if the operands of \wedge in all well-formed boolean formula in the NNF are expressed on a mutually disjoint set of variables.

The deterministic and decomposable properties are conveniently expressed by AND-OR graphs (DAGs) where a node is either an AND node, an OR node or a literal. The operands of AND/OR nodes appear as children of the node. Figure 1 shows an example of d-DNNF representation. For every node t , the subformula corresponding to t is the formula corresponding to d-DNNF obtained by removing all the nodes u such that there does not exist a path from t to u . $T(t)$ represents the set of all partial satisfying assignments for the subformula corresponding to t . The siblings of a node t are the children of the parent of t excluding t itself and the set of such children is given by $Siblings(t)$.

Decision-DNNF is a subset of d-DNNF where the deterministic OR nodes are decision nodes [18]. The state-of-the-art d-DNNF construction tools like C2D [18], DSHARP [43] and D4 [30], construct the Decision-DNNF representation where each OR node has exactly two children while an AND node may have multiple children. Since our framework WAPS employs modern d-DNNF compilers, we assume that the OR node has exactly two children. This assumption is only for the simplicity of exposition as our algorithms can be trivially adopted to the general d-DNNF representations.

4 Algorithm

In this section, we discuss our primary technical contribution: WAPS, weighted and projected sampler that samples from $R_{F \perp P}$ with respect to weight function W by employing the knowledge compilation techniques.

WAPS takes a CNF formula F , a set of sampling variables P , a function assigning weights to literals W and required number of samples s as inputs and returns `SampleList`, a list of size s which contain samples such that each sample is independently drawn from the weighted distribution generated by W over $R_{F \perp P}$.

Similar to KUS, WAPS (Algorithm 1) mainly comprises of three phases: Compilation, Annotation and Sampling. For d-DNNF compilation, WAPS invokes a specialized compilation routine `PCompile` over the formula F and the sampling

set P (line 1). This is followed by the normalization of weights such that for any literal l , $W'(l) + W'(\neg l) = 1$, where W' is the normalized weight returned in line 2. Then for annotation, `WAnnotate` is invoked in line 3 which uses the weight function W' to annotate weights to all the nodes of the d-DNNF tree. Finally, subroutine `Sampler` (line 4) is invoked which returns s independently drawn samples over P following the weighted distribution generated by W over $R_{F \downarrow P}$. We now describe these three phases in detail.

4.1 Compilation

The compilation phase is performed using the subroutine `PCompile`. `PCompile` is a modified procedure over the component caching and clause learning based algorithm of the d-DNNF compiler `DSHARP` [43,47]. It is presented in Algorithm 2. The changes from the existing algorithm are underlined. The rest of the procedure which is similar to `DSHARP` is mentioned here for completeness. The description of `PCompile` is as follows:

`PCompile` takes in a CNF formula F in the clausal form and a set of sampling variables P as input and returns a d-DNNF over P . If the formula does not contain any variable from P , `PCompile` invokes `SAT` (line 2) which returns a *True* node if the formula is satisfiable, else it returns a *False* node. Otherwise, `DecideLiteral` is invoked to choose a literal appearing in F such that $\text{var}(l) \in P$ (line 3). This decision is then recursively propagated by invoking `CompileBranch` to create t_1 , the d-DNNF of $F|_l$ (line 4) and t_2 , the d-DNNF of $F|_{\neg l}$ (line 5). `Disjoin` is invoked in line 6 which takes t_1 and t_2 as input and returns t_2 if t_1 is *False* node, t_1 if t_2 is *False* node otherwise a new tree composed by an OR node as the parent of t_1 and t_2 . The result of `Disjoin` is then stored in the cache (line 7) and returned as an output of `PCompile` (line 8).

We now discuss the subroutine `CompileBranch`. It is presented in Algorithm 3. It takes in a CNF formula F , set of sampling variables P and literal l as input and returns a d-DNNF tree of $F|_l$ on P . It first invokes `BCP` (Binary Constraint Propagation), with F , l and P which performs unit-propagation to return a tuple of reduced formula F' and a set of implied literals (*term*) projected over variables in P (line 1). Then `CompileBranch` checks if F' contains an empty clause and returns *False* node to indicate that $F|_l$ is not satisfiable, else the formula is solved using component decomposition as described below.

At line 6 it breaks the formula F' into separate components formed by disjoint set of clauses such that no two components share any variables. Then each component is solved independently (lines 8–15). For each component, it first examines the cache to see if this component has been solved earlier and is present in the cache (line 9). If cache lookup fails, it solves the component with a recursive call to `PCompile` (line 11). If any component is found to be unsatisfiable, *False* node is returned implying that the overall formula is unsatisfiable too, else `CompileBranch` simply conjoins the components' d-DNNFs together with the decided l and implied literals (*term*) and returns this after storing it in the cache for the formula $F|_l$ (lines 16–18).

We illustrate PCompile procedure on the following example formula F :

Example 1. $F = \{\{x_1, x_2\}, \{\neg x_3, \neg x_5, x_6\}, \{\neg x_2, x_4, \neg x_1\}, \{x_3, \neg x_6, \neg x_1\}, \{x_6, x_5, \neg x_1, x_3\}, \{x_3, x_6, \neg x_5, \neg x_1\}\}$

Figure 2 represents the d-DNNF of Example 1 on $P = \{x_1, x_2, x_3\}$. For detailed discussion about applying PCompile on F please refer to Appendix.

Algorithm 1. WAPS(F, P, W, s)	Algorithm 3. CompileBranch(F, P, l)
1: dag \leftarrow PCompile(F, P) 2: $W' \leftarrow$ Normalize(W) 3: WAnnotate(dag.root, W') 4: SampleList \leftarrow Sampler(dag.root, s) 5: return SampleList	1: $(F', term) \leftarrow$ BCP(F, l, P) 2: if $\emptyset \in F'$ then 3: CacheStore($F _l, False$) 4: return $False$ \triangleright CDCL is done 5: else 6: Comps \leftarrow DisjointComponents(F') 7: dcomps \leftarrow {} 8: for $C \leftarrow$ Comps do 9: ddnnf \leftarrow GetCache(C) 10: if ddnnf is not found then 11: ddnnf \leftarrow PCompile(C, P) 12: dcomps.Add(ddnnf) 13: if ddnnf = $False$ then 14: CacheStore($F _l, False$) 15: return $False$ 16: $t =$ Conjoin($l, term, dcomps$) 17: CacheStore($F _l, t$) 18: return t
Algorithm 2. PCompile(F, P)	
1: if $Vars(F) \cap P = \emptyset$ then 2: return SAT(F) 3: $l \leftarrow$ DecideLiteral(F, P) 4: $t_1 \leftarrow$ CompileBranch(F, P, l) 5: $t_2 \leftarrow$ CompileBranch($F, P, \neg l$) 6: $t \leftarrow$ Disjoin(t_1, t_2) 7: CacheStore(F, t) 8: return t	

4.2 Annotation

The subroutine WAnnotate is presented in Algorithm 4. WAnnotate takes in a d-DNNF dag and a weight function W as inputs and returns an annotated d-DNNF dag whose each node t is annotated with a weight, given by the sum of weights of all the partial assignments represented by subtree rooted at t . The weights subsequently annotated on children of an OR node indicate the probability with which it would be selected in the sampling phase.

WAnnotate performs a bottom up traversal on d-DNNF dag in a reverse topological order. For each node in the d-DNNF dag, WAnnotate maintains an attribute, weight, as per the label of the node given as follows:

Literal (lines 2–3): The weight of a literal node is taken as the weight of the literal given by the weight function W .

OR (lines 4–8): The weight of OR node is made equal to the sum of weights of both of its children.

AND (lines 9–13): The weight of AND node is made equal to the product of weights of all its children.

Algorithm 4. Bottom-Up Pass to annotate d-DNNF with weights on literals

```

1: function WAnnotate( $t, W$ )
2:   if  $label(t) = Literal$  then
3:      $t.weight \leftarrow W(t)$ 
4:   else if  $label(t) = OR$  then
5:      $t.weight \leftarrow 0$ 
6:     for  $c \in \{t.left, t.right\}$  do
7:       WAnnotate( $c$ )
8:      $t.weight \leftarrow t.weight + c.weight$ 
9:   else if  $label(t) = AND$  then
10:     $t.weight \leftarrow 1$ 
11:    for  $c \in Childrens(t)$  do
12:      WAnnotate( $c$ )
13:     $t.weight \leftarrow t.weight \times c.weight$ 

```

4.3 Sampling

Algorithm `Sampler` takes the annotated d-DNNF dag and the required number of samples s and returns `SampleList`, a list of s samples conforming to the distribution of their weights as governed by weight function W given to `WAnnotate`. The subroutine `Sampler` is very similar to the sampling procedure in our previous work [49] except that we take the annotated weight of the node instead of the annotated count in the previous work as the probability for Bernoulli trials. We refer the readers to Appendix for a detailed discussion.

4.4 Conditioned Sampling

The usage of samplers in testing environment necessitates sampling from F conditioned on fixing assignment to a subset of variables. The state of the art techniques, such as those based on universal hashing, treat every query as independent and are unable to reuse computation across different queries. In contrast, the compilation to d-DNNF allows WAPS to reuse the same d-DNNF. In particular, for a given conditioning expressed as conjunction of different literals, i.e., $\hat{C} = \bigwedge_i l_i$.

In particular, instead of modifying computationally expensive d-DNNF, we modify the weight function as follows:

$$\hat{W}(l) = \begin{cases} 0, & l \notin \hat{C} \\ W(l) & otherwise \end{cases}$$

5 Theoretical Analysis

We now present theoretical analysis of WAPS, which consists of two components: correctness of WAPS and analysis of behavior of `Sampler` on the underlying d-DNNF graph. First, we prove that WAPS is an exact weighted and projected

sampler in Theorem 1. To this end, we prove the correctness of our projected d-DNNF dag compilation procedure `PCompile` in Lemma 1. In Lemma 2, we show that `WAnnotate` annotates each node of the d-DNNF with weights that represent the weight of assignments represented by subtree rooted at that node. This enables us to sample as per the weight distribution in the sampling phase which is proved in Theorem 1 using Lemmas 1 and 2. Secondly, further probing into the behavior of subroutine `Sampler`, we provide an analysis of the probability of visiting any node in the d-DNNF dag while sampling. For this, we first find a probability of visiting a node by following a particular path in Lemma 3 and then we use this result to prove an upper bound for the general case of visiting a node from all possible paths in Theorem 2. We believe that this analysis will motivate the researchers to find new ways to speed up or device new methods to find exact or approximate sampling techniques over a given compiled representation.

The proofs of Theorem 1 and Lemmas 1, 2 and 3 can be found in Appendix.

Lemma 1. *Given a formula F and set of sampling variables P , the tree returned by `PCompile`(F, P) is a d-DNNF dag which represents the set of satisfying assignments of the formula F projected over the set of sampling variables P .*

Lemma 2. *Every node t in the d-DNNF dag returned by `WAnnotate` is annotated by $W(T(t))$, where $T(t)$ is the set of all the partial assignments corresponding to the subtree rooted at t .*

Theorem 1. *For a given F, P and s , `SampleList` is the list of samples generated by `WAPS`. Let `SampleList`[i] indicate the sample at the i^{th} index of the list. Then for each $y \in R_{F \downarrow P}$, $\forall i \in [s]$, we have $\Pr[y = \text{SampleList}[i]] = \frac{W(y)}{W(R_{F \downarrow P})}$.*

Lemma 3. *For a given F and P , let $\text{fol}(\rho)$ be the event of following a path ρ , which start at root and ends at node t , then $\Pr[\text{fol}(\rho)] = \frac{W(T(t)) \times c_\rho}{W(R_{F \downarrow P})}$ where c_ρ is the product of weight of all the OR nodes' siblings encountered in the path ρ from root to t and $T(t)$ is the set of all the partial satisfying assignments represented by subtree rooted at t .*

Theorem 2. *For a given F and P , let $\text{visit}(t)$ be the event of visiting a node t to fetch one sample as per subroutine `Sampler`, then $\Pr[\text{visit}(t)] \leq \frac{W(\Gamma(t))}{W(R_{F \downarrow P})}$ where $\Gamma(t) = \{\sigma \mid \sigma \in R_{F \downarrow P}, \sigma_{\downarrow \text{Vars}(t)} \in T(t)\}$ and $T(t)$ is a set of all the partial satisfying assignments represented by subtree rooted at t .*

Proof. In Lemma 3 we have calculated the probability of visiting a node t by taking a particular path from root to node t . So the probability of visiting a node t will be the sum of probability of visiting t by all possible paths. Let $\mathcal{P} = \{\rho_1, \rho_2, \dots, \rho_m\}$ be the set of all paths from root to node t and $\text{visit}(t)$ be the event of visiting a node t in subroutine `Sampler` then,

$$\Pr[\text{visit}(t)] = \sum_{\rho \in \mathcal{P}} \Pr[\text{visit}(t) \mid \text{fol}(\rho)] \times \Pr[\text{fol}(\rho)] = \sum_{\rho \in \mathcal{P}} 1 \times \Pr[\text{fol}(\rho)]$$

From Lemma 3, $\Pr[\text{visit}(t)] = \sum_{\rho \in \mathcal{P}} \frac{W(T(t)) \times c_\rho}{W(R_{F \downarrow P})}$ where c_ρ is the product of the weight of all the OR nodes' siblings encountered in a path ρ from root to t . For any such path, we call $\{t_\rho^1, t_\rho^2, \dots, t_\rho^n\}$ as the set of all the OR node siblings encountered on the path ρ . Now, let σ_ρ^{ext} be the set of assignments over P represented by path ρ . Therefore,

$$\sigma_\rho^{ext} = T(t_\rho^1) \times T(t_\rho^2) \cdots \times T(t_\rho^n) \times T(t)$$

where, $T(\cdot)$ are set of assignments and \times is a cross product. Now, any tuple from σ_ρ^{ext} represents a satisfying assignment in the d-DNNF. Therefore, $\sigma_\rho^{ext} \subseteq R_{F \downarrow P}$. Note that, from Lemma 2, it follows that weight annotated by WAnnotate at t is equal to $W(T(t))$. Therefore,

$$c_\rho = W(T(t_\rho^1)) \times W(T(t_\rho^2)) \cdots \times W(T(t_\rho^n))$$

And, $W(\sigma_\rho^{ext}) = W(T(t)) \times c_\rho$. Notice that, $\sigma_\rho^{ext} \subseteq \Gamma(t)$ as σ_ρ^{ext} represents satisfying extensions of partial assignments contained in $T(t)$ itself. This is true $\forall \rho \in \mathcal{P}$. Therefore as $W(\cdot)$ is an increasing function,

$$\bigcup_{\rho \in \mathcal{P}} \sigma_\rho^{ext} \subseteq \Gamma(t) \implies \sum_{\rho \in \mathcal{P}} W(T(t)) \times c_\rho \leq W(\Gamma(t))$$

Note that, the inequality indeed holds as the intersection of sets of partial assignments represented by t and any other node not lying on the path from root to t may not be ϕ (empty). Therefore,

$$\sum_{\rho \in \mathcal{P}} \frac{W(T(t)) \times c_\rho}{W(R_{F \downarrow P})} \leq \frac{W(\Gamma(t))}{W(R_{F \downarrow P})} \implies \Pr[\text{visit}(t)] \leq \frac{W(\Gamma(t))}{W(R_{F \downarrow P})}$$

6 Evaluation

In order to evaluate the runtime performance and analyze the quality of samples generated by WAPS, we implemented a prototype in Python. For d-DNNF compilation, our prototype makes use of DSHARP [43] when sampling set is available else we use D4 [30]. We would have preferred to use state-of-the-art d-DNNF compiler D4 but owing to its closed source implementation, we could not modify it as per our customized compilation procedure PCompile. Therefore, for projected compilation, we have modified DSHARP which has an open-source implementation. We have conducted our experiments on a wide range of publicly available benchmarks. In all, our benchmark suite consisted of 773 benchmarks arising from a wide range of real-world applications. Specifically, we used constraints arising from DQMR networks, bit-blasted versions of SMT-LIB (SMT) benchmarks, and ISCAS89 circuits [8] with parity conditions on randomly chosen subsets of outputs and nextstate variables [34, 48]. We assigned random weights to literals wherever weights were not already available in our benchmarks. All our experiments were conducted on a high performance compute cluster whose each node consists of E5-2690 v3 CPU with 24 cores and 96 GB of RAM. We utilized single core per instance of benchmark with a timeout of 1800 s.

Table 1. Run time (in seconds) for 1000 samples

Benchmark	Vars	Clauses	$ P $	WeightGen	WAPS			Speedup on WeightGen
					Compile	A+S	Total	
s526a_3_2	366	944	24	490.34	15.37	1.96	17.33	28.29
LoginService	11511	41411	36	1203.93	15.02	0.75	15.77	76.34
blockmap_05_02	1738	3452	1738	1140.87	0.04	5.30	5.34	213.65
s526_3_2	365	943	24	417.24	0.06	0.67	0.73	571.56
or-100-5-4-UC-60	200	500	200	1795.52	0.01	0.74	0.74	2426.38
or-50-5-10-UC-40	100	250	100	1292.67	0.01	0.36	0.36	3590.75
blasted_case35	400	1414	46	TO	0.57	1.46	2.03	-
or-100-20-4-UC-50	200	500	200	TO	0.19	2.48	2.67	-

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

1. How does WAPS perform in terms of runtime in comparison to WeightGen, the current state-of-the-art weighted and projected sampler?
2. How does WAPS perform for *incremental sampling* and scales when asked for different number of samples?
3. How does the distribution of samples generated by WAPS compare with the distribution generated by an ideal weighted and projected sampler?
4. How does WAPS perform for conditioning on arbitrary variables?
5. How does our knowledge compilation based sampling techniques perform in comparison to hashing based sampling techniques for the task of generalizing to arbitrary weight distributions?

Our experiment demonstrated that within a time limit of 1800 s, WAPS is able to significantly outperform existing state-of-the-art weighted and projected sampler WeightGen, by up to 3 orders of magnitude in terms of runtime while achieving a geometric speedup of $296\times$. Out of the 773 benchmarks available WAPS was able to sample from 588 benchmarks while WeightGen was able to sample from only 24 benchmarks. For *incremental sampling*, WAPS achieves a geometric speedup of 3.69. Also, WAPS is almost oblivious to the number of samples requested. Empirically, the distribution generated by WAPS is statistically indistinguishable from that generated by an ideal weighted and projected sampler. Also, while performing conditioned sampling in WAPS, we incur no extra cost in terms of runtime in most of the cases. Moreover, the performance of our knowledge compilation based sampling technique is found to be oblivious to weight distribution. We present results for only a subset of representative benchmarks here. Detailed data along with the expanded versions of all the tables presented here is available at <https://github.com/meelgroup/waps>.

Number of Instances Solved. We compared the runtime performance of WAPS with WeightGen [10] (state-of-the-art weighted and projected sampler) by generating 1000 samples from each tool with a timeout of 1800 s. Figure 3 shows the cactus plot for WeightGen and WAPS. We present the number of benchmarks on the

x -axis and the time taken on y -axis. A point (x, y) implies that x benchmarks took less than or equal to y seconds to sample. All our runtime statistics for WAPS include the time for the knowledge compilation phase (via D4 or DSHP). From all the 773 available benchmarks WeightGen was able to sample from only 24 benchmarks while WAPS was able to sample from 588 benchmarks. Table 1 shows the runtimes of some of the benchmarks on the two tools. The columns in the table give the benchmark name, number of variables, number of clauses, size of sampling set, time taken in seconds by WeightGen and WAPS divided into time taken by Compilation and A+S: Annotation and Sampling followed by speedup of WAPS with respect to WeightGen. Table 1 clearly shows that WAPS outperforms WeightGen by upto 3 orders of magnitude. For all the 24 benchmarks that WeightGen was able to solve WAPS outperformed WeightGen with a geometric speedup of $296\times$.

Incremental Sampling. Incremental sampling involves fetching multiple, relatively small-sized samples until the objective (such as desired coverage or violation of property) is achieved. We benefit from pre-compiled knowledge representations in this scenario, as they allow us to perform repeated sampling as per varied distributions. If weights are changed, we simply Annotate the tree again followed by sampling, else, we directly move to the sampling phase, thus saving a significant amount of time by bypassing the compilation phase.

In our experiments, we have evaluated the time taken by WAPS for 1000 samples in 10 successive calls with same weights. The results are presented in Table 2 for a subset of benchmarks. The first column mentions the benchmark name with the number of variables, clauses and size of sampling set in subsequent columns. The time taken by WAPS for first run to fetch 1000 samples is given in the fifth column while the overall time taken for first run together with the subsequent 9 incremental runs is presented in sixth column. The final column shows the average gain in terms of speedup calculated by taking the ratio of time taken by WAPS for first run with the average time taken by WAPS for subsequent 9 incremental runs thus resulting in a total of 10000 samples. Overall, WAPS achieves a geometric speedup of $3.69\times$ on our set of benchmarks.

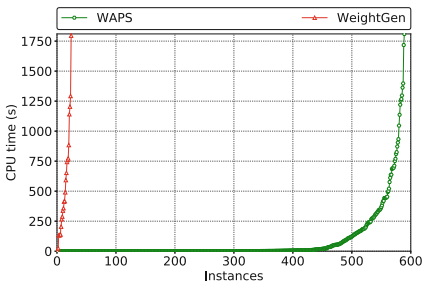


Fig. 3. Cactus Plot comparing WeightGen and WAPS.

Benchmark	Vars	Clauses	$ P $	WAPS		Speedup
				1000	10,000	
case110	287	1263	287	1.14	9.28	1.26
or-70-10-10-UC-20	140	350	140	2.75	9.02	6.56
s526_7_4	383	1019	24	60.38	143.16	13.20
or-60-5-2-UC-10	120	300	120	12.10	20.35	16.50
s35932_15_7	17918	44709	1763	69.01	106.65	20.73
case121	291	975	48	35.85	51.41	20.73
s641_15_7	576	1399	54	729.38	916.83	35.01
squaring7	1628	5837	72	321.95	365.13	67.10
LoginService	11511	41411	36	15.89	18.12	64.13
ProjectService	3175	11019	55	184.51	195.25	154.61

Table 2. Runtimes (in sec.) of WAPS for incremental sampling

Table 3. Runtime (in sec.) of WAPS to generate different size samples

Benchmark	Vars	Clauses	$ P $	Sampling Size				
				1000	2000	4000	8000	10000
s1488_7_4	872	2499	14	0.5	0.75	1.29	2.2	2.9
s444_15_7	377	1072	24	0.74	1.29	1.91	3.46	4.12
s526_3_2	365	943	24	0.84	1.03	1.86	3.71	4.22
s820a_3_2	598	1627	23	0.63	1.03	2.04	3.92	4.81
case35	400	1414	46	2.38	3.22	5.31	9.38	11.41
LoginService	11511	41411	36	15.8	16.12	16.68	18.3	18.36
ProjectService	3175	11019	55	184.22	184.99	188.33	191.16	193.92
or-60-20-6-UC-10	120	300	120	1465.34	1458.23	1494.46	1499.67	1488.23

Effect of Number of Samples. To check how WAPS scales with different number of samples, we invoked WAPS for fetching different number of samples: 1000, 2000, 4000, 8000, 10000 with a timeout of 1800s. Table 3 presents the runtime of WAPS for different samples on some benchmarks. The first column represents the benchmark name. Second, third and fourth columns represent the number of variables, clauses and size of sampling set. The next five columns represent the time taken by WAPS for 1000, 2000, 4000, 8000 and 10000 samples. Table 3 clearly demonstrates that WAPS is almost oblivious to the number of samples requested.

Uniform Sampling Generalized for Weighted Sampling. To explore the trend in performance between uniform and weighted sampling on the dimension of hashing based techniques pitched against our newly proposed sampling techniques based on knowledge compilation, we compared WAPS to KUS in a parallel comparison between WeightGen and UniGen2. Specifically, we ran WAPS for weighted sampling and KUS for uniform sampling without utilizing the sampling set as KUS does not support the sampling set. On the other hand, for hashing based sampling techniques, we compared WeightGen to UniGen2 while using the sampling set. Figure 4 shows the cactus plot for WeightGen and UniGen2 and Fig. 5 shows a cactus plot for WAPS and KUS. From all the 773 benchmarks, WeightGen was able to sample from only 24 benchmarks while UniGen2 was able to sample from 208 benchmarks. In comparison, WAPS was able to sample from 606 benchmarks while KUS was able to sample from 602 benchmarks. Our experiments demonstrated that the performance of hashing-based techniques is extremely limited in their ability to handle literal-weighted sampling and there is a drastic drop in their performance as the weight distribution shifts away from uniform. While for our knowledge compilation based sampling techniques we observe that their performance is oblivious to the weight distribution.

Distribution Comparison. We measure the distribution of WAPS vis-a-vis an *ideal weighted and projected sampler* (IS) and observed that WAPS is statistically indistinguishable from IS. Please refer to Appendix for more detailed discussion.

Effect of Conditioning on Variables. We evaluated the performance of WAPS in the context of conditioned sampling. We observed a slight improvement in average runtime as more and more variables get constrained. For detailed results, please refer to Appendix.

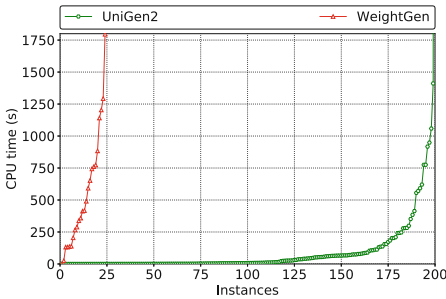


Fig. 4. Cactus Plot comparing WeightGen and UniGen2.

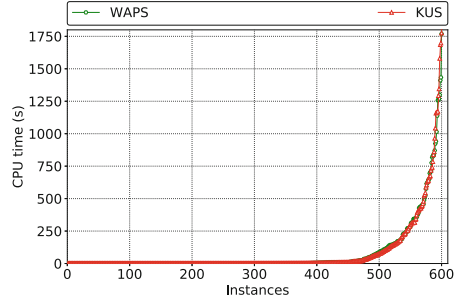


Fig. 5. Cactus Plot comparing WAPS and KUS.

7 Conclusion

In this paper, we designed a knowledge compilation-based framework, called WAPS, for literal-weighted, projected and conditional sampling. WAPS provides strong theoretical guarantees and its runtime performance upon the existing state-of-the-art weighted and projected sampler WeightGen, by up to 3 orders of magnitude in terms of runtime. Out of the 773 benchmarks available, WAPS is able to sample from 588 benchmarks while WeightGen is only able to sample from 24 benchmarks. WAPS achieves a geometric speedup of 3.69 for *incremental sampling*. It is worth noting that WeightGen has weaker guarantees than WAPS. Furthermore, WAPS is almost oblivious to the number of samples requested.

References

1. System Verilog (2015). <http://www.systemverilog.org>
2. Achlioptas, D., Hammoudeh, Z.S., Theodoropoulos, P.: Fast sampling of perfectly uniform satisfying assignments. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 135–147. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_9
3. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and Bayesian inference. In: Proceedings of FOCS, pp. 340–351 (2003)

4. Bellare, M., Goldreich, O., Petrank, E.: Uniform generation of NP-witnesses using an NP-oracle. *Inf. Comput.* **163**(2), 510–526 (2000)
5. Biere, A., Cimatti, A., Clarke, E., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: *Proceedings of DAC*, pp. 317–320 (1999)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T.: *Handbook of Satisfiability*. IOS Press, Amsterdam (2009)
7. Bjesse, P., Leonard, T., Mokedem, A.: Finding bugs in an Alpha microprocessor using satisfiability solvers. In: Berry, G., Comon, H., Finkel, A. (eds.) *CAV 2001*. LNCS, vol. 2102, pp. 454–464. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_44
8. Brglez, F., Bryan, D., Kozminski, K.: Combinational profiles of sequential benchmark circuits. In: *Proceedings of ISCAS*, pp. 1929–1934 (1989)
9. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. (CSUR)* **24**(3), 293–318 (1992)
10. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for SAT. In: *Proceedings of AAAI*, pp. 1722–1730 (2014)
11. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform SAT witness generation. In: Baier, C., Tinelli, C. (eds.) *TACAS 2015*. LNCS, vol. 9035, pp. 304–319. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_25
12. Chakraborty, S., Fried, D., Meel, K.S., Vardi, M.Y.: From weighted to unweighted model counting. In: *Proceedings of IJCAI*, pp. 689–695 (2015)
13. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable and nearly uniform generator of SAT witnesses. In: Sharygina, N., Veith, H. (eds.) *CAV 2013*. LNCS, vol. 8044, pp. 608–623. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_40
14. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. *Artif. Intell.* **172**(6), 772–799 (2008)
15. Dalvi, N.N., Schnaitter, K., Suciu, D.: Computing query probability with incidence algebras. In: *Proceedings of PODS*, pp. 203–214 (2010)
16. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: *Proceedings of 22nd International Joint Conference on Artificial Intelligence*, pp. 819–826 (2011)
17. Darwiche, A.: On the tractable counting of theory models and its application to belief revision and truth maintenance. *CoRR* (2000)
18. Darwiche, A.: New advances in compiling CNF to decomposable negation normal form. In: *Proceedings of ECAI*, pp. 318–322 (2004)
19. Darwiche, A., Marquis, P.: A knowledge compilation map. *J. Artif. Intell. Res.* **17**, 229–264 (2002)
20. Dechter, R., Kask, K., Bin, E., Emek, R.: Generating random solutions for constraint satisfaction problems. In: *Proceedings of AAAI*, pp. 15–21 (2002)
21. Duenas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: *Proceedings of AAAI* (2017)
22. Dutra, R., Laeuffer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: *Proceedings of ICSE*, pp. 549–559 (2018)
23. Elliott, P., Williams, B.: DNNF-based belief state estimation. In: *Proceedings of AAAI*, pp. 36–41 (2006)
24. Ermon, S., Gomes, C.P., Sabharwal, A., Selman, B.: Embed and project: discrete sampling with universal hashing. In: *Proceedings of NIPS*, pp. 2085–2093 (2013)

25. Ermon, S., Gomes, C.P., Selman, B.: Uniform solution sampling using a constraint solver as an Oracle. In: Proceedings of UAI, pp. 255–264 (2012)
26. Gogate, V., Dechter, R.: A new algorithm for sampling CSP solutions uniformly at random. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 711–715. Springer, Heidelberg (2006). https://doi.org/10.1007/11889205_56
27. Gomes, C.P., Selman, B., McAloon, K., Tretkoff, C.: Randomization in backtrack search: exploiting heavy-tailed profiles for solving hard scheduling problems. In: Proceedings of AIPS (1998)
28. Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. *Constraints* **21**, 1–18 (2015). <https://doi.org/10.1007/s10601-015-9204-z>
29. Iyer, M.A.: RACE: a word-level ATPG-based constraints solver system for smart random simulation. In: Proceedings of ITC, pp. 299–308 (2003)
30. Lagniez, J.-M., Marquis, P.: An improved decision-DNNF compiler. In: Proceedings of IJCAI, pp. 667–673 (2017)
31. Jerrum, M.R., Sinclair, A.: Approximating the permanent. *SIAM J. Comput.* **18**(6), 1149–1178 (1989)
32. Jerrum, M.R., Sinclair, A.: The Markov Chain Monte Carlo method: an approach to approximate counting and integration. In: Hochbaum, D.S. (ed.) *Approximation Algorithms for NP-Hard Problems*, pp. 482–520. ACM, New York (1996)
33. Jerrum, M.R., Valiant, L.G., Vazirani, V.V.: Random generation of combinatorial structures from a uniform distribution. *Theoret. Comput. Sci.* **43**(2–3), 169–188 (1986)
34. John, A.K., Chakraborty, S.: A quantifier elimination algorithm for linear modular equations and disequations. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 486–503. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_39
35. Kautz, H., Selman, B.: Pushing the envelope: planning, propositional logic, and stochastic search. In: Proceedings of AAAI (1996)
36. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. *Science* **220**(4598), 671–680 (1983)
37. Kitchen, N.: Markov Chain Monte Carlo stimulus generation for constrained random simulation. Ph.D. thesis, University of California, Berkeley (2010)
38. Kitchen, N., Kuehlmann, A.: Stimulus generation for constrained random simulation. In: Proceedings of ICCAD, pp. 258–265 (2007)
39. Madras, N., Piccioni, M.: Importance sampling for families of distributions. *Ann. Appl. Probab.* **9**, 1202–1225 (1999)
40. Madras, N.: *Lectures on Monte Carlo Methods*, Fields Institute Monographs, vol. 16. American Mathematical Society, Providence (2002)
41. Malik, S., Zhang, L.: Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* **52**(8), 76–82 (2009)
42. Meel, K.S.: Constrained counting and sampling: bridging the gap between theory and practice. Ph.D. thesis, Rice University (2017)
43. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: DSHARP: fast d-DNNF compilation with sharpSAT. In: Proceedings of AAAI, pp. 356–361 (2016)
44. Naveh, R., Metodi, A.: Beyond feasibility: CP usage in constrained-random functional hardware verification. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 823–831. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_60
45. Naveh, Y., et al.: Constraint-based random stimuli generation for hardware verification. In: Proceedings of IAAI, pp. 1720–1727 (2006)

46. Palacios, H., Bonet, B., Darwiche, A., Geffner, H.: Pruning conformant plans by counting models on compiled d-DNNF representations. In: Proceedings of ICAPS, pp. 141–150 (2005)
47. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: Proceedings of SAT (2004)
48. Sang, T., Beame, P., Kautz, H.: Performing Bayesian inference by weighted model counting. In: Proceedings of AAAI, pp. 475–481 (2005)
49. Sharma, S., Gupta, R., Roy, S., Meel, K.S.: Knowledge compilation meets uniform sampling. In: Proceedings of LPAR-22, pp. 620–636 (2018)
50. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) *Automation of Reasoning. Symbolic Computation (Artificial Intelligence)*. Springer, Berlin, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_28
51. Velev, M.N., Bryant, R.E.: Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *J. Symb. Comput.* **2**, 73–106 (2003)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

