ESampler: Efficient Sampling of Satisfying Assignments for Boolean Formulas

Yongjie Xu[1], Fu Song[1(✉)], and Taolue Chen[2]

[1] ShanghaiTech University, Shanghai, China
songfu@shanghaitech.edu.cn
[2] Birkbeck, University of London, London, UK

Abstract. Boolean satisfiability (SAT) has played a key role in diverse areas spanning planning, inferencing, data mining, testing and optimization. Apart from the classical problem of checking Boolean satisfiability, generating random satisfying assignments has attracted significant theoretical and practical interests over the years. For practical applications, a large number of satisfying assignments for a given Boolean formula are needed, the generation of which turns out to be a hard problem in both theory and practice. In this work, we propose a novel approach to derive a large set of satisfying assignments from a given one in an efficient way. Our approach is orthogonal to the previous techniques for generating satisfying assignments and could be integrated into the existing SAT samplers. We implement our approach as an open-source tool ESampler and conduct extensive experiments on real-world benchmarks. Experimental results show that ESampler performs better than three state-of-the-art samplers on a large portion of the benchmarks, and is at least comparable on the others, showcasing the efficacy of our approach.

Keywords: Boolean satisfiability · Constraint-based sampling · SAT solving

1 Introduction

Boolean satisfiability, also known as SAT, concerns determining whether a given Boolean formula is satisfiable. There have been strong theoretical and practical interests in the SAT problem, which has played a key role in diverse areas spanning planning, inferencing, data mining, testing and optimization [1,6]. Apart from the classical problem of checking Boolean satisfiability, generating random satisfying assignments has attracted significant theoretical and practical interests

This work is supported by the National Natural Science Foundation of China (NSFC) under Grants No. 62072309 and No. 61872340, an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2018A16), and Birkbeck BEI School Project (ARTEFACT).

S. Qin et al. (Eds.): SETTA 2021, LNCS 13071, pp. 279–298, 2021.
https://doi.org/10.1007/978-3-030-91265-9_15

over the years [3,4,18,29–32,39,40,43,45–47]. In several practical applications, a large number of satisfying assignments for a given Boolean formula are needed. For instance, simulation-based verification is a commonly adopted technique to test hardware design. In this scenario, the simulated behavior is compared with the expected behavior where any mismatch is flagged as an indication of a bug [29,47]. It is a common practice to generate a large number of stimuli satisfying a given set of constraints in the form of Boolean formulas. These constraints typically arise from various sources such as application-specific knowledge and environmental requirements. Another application scenario is the generation of adversarial examples for adversarial training [11,48]. Adversarial training is a widely adopted technique to improve the robustness of neural networks against adversarial attacks where a large number of adversarial inputs (e.g., images) would be generated explicitly or implicitly. For instance, to adversarially train a binarized neural network [19,44], adversarial images were generated by encoding a binarized neural network as a Boolean formula based on which satisfying assignments were sampled [24,28].

Sampling satisfying assignments for a given Boolean formula is, however, challenging. It is well-known that the SAT problem is **NP**-complete [12]. In recent years, we have seen a tremendous progress in SAT solving, supported by techniques such as conflict-driven clause learning (CDCL [21,33,34]), yielding powerful solvers such as CryptoMiniSAT [37].

However, generating a large number of satisfying assignments is still computationally prohibitive and often infeasible in practical settings [14,23]. In this work, we develop ESampler, aiming for generating a large number of satisfying assignments efficiently for a given Boolean formula. The general strategy is to use an existing sampler to produce a seed sample as a satisfying assignment, from which we derive more satisfying assignments by flipping some variables of the given Boolean formula. Clearly, naively flipping variables may yield unsatisfying assignments. To tackle this problem, we propose a novel derivation procedure which explores the semantics of the Boolean formula under the seed sample, so that the resulting assignments can be guaranteed to satisfy the Boolean formula. The advantage of our approach lies in that it can be integrated with the existing SAT samplers, so would enjoy considerably wider applicability.

To demonstrate our approach, we implement a sampler ESampler based on the recent sampler QuickSampler [14]. We carry out extensive experiments on the publicly available benchmarks from UniGen [10] which include Boolean formulas from real-world testing and verification applications. Our experimental results show that ESampler performs considerably better than the three state-of-the-art samplers QuickSampler, SearchTreeSampler (STS in short) [16] and UniGen3 [36], indicating the effectiveness of our approach.

Our main contributions can be summarized as follows.

– We introduce a novel approach for deriving a large set of satisfying assignments from a given seed. It is generic and could be integrated with the existing samplers. To the best of our knowledge, it is the first work to generate satisfying assignments from a given seed.

- We implement an integrated sampler ESampler. Our tool is available at https://github.com/ESampler/Esampler.
- We conduct extensive experiments on hundreds of Boolean formulas from real-world applications and ESampler performs considerably better than the three state-of-the-art samplers QuickSampler, STS and UniGen3.

Related Work. Various techniques have been proposed to tackle the problem of the satisfying assignment generation for Boolean formulas [26]. Binary decision diagrams (BDD) and Markov Chain Monte Carlo (MCMC) algorithms such as simulated annealing and Metropolis-Hastings are widely used for generating satisfying assignments [22,42,43]. These techniques usually provide theoretical guarantees of uniformity but are limited in scalability and efficiency. Therefore, heuristics are proposed to speed up at the cost of theoretical guarantees of uniformity [22,25,41]. Another class of satisfying assignment generation techniques with theoretical guarantees of uniformity is based on hashing [5,8–10,15,17,35,36]. Hashing-based techniques add hash functions (e.g., XOR of a random subset of variables) to the Boolean formula in order to partition the search space uniformly and then randomly pick a satisfying assignment from a randomly chosen cell. These algorithms are also limited in scalability and efficiency. In comparison, our approach primarily aims for efficiency, using fewer solver calls to generate a large number of solutions. We also provide a parameter to balance the uniformity of the generated samples and the efficiency of the procedure. Although we do not provide a theoretical guarantee of uniformity, the experimental results demonstrate that our approach is able to produce solutions nearly uniformly when the maximal number of solutions per seed is set in a reasonable range.

Recently, SAT samplers aiming to quickly generate a large number of assignments have been proposed. Both QuickSampler [14] and STS [16] share the same goal as our work, namely, fast generation of a larger number of assignments. QuickSampler uses the MaxSAT solver [7] to generate random satisfying assignments, and then find more assignments that are close to satisfying assignments using the diffs of discovered satisfying assignments. However, the assignments generated by QuickSampler may not satisfy the Boolean formula, hence a follow-up checking is often needed. In contrast, our approach only mutates proper variables by which the formula is guaranteed to be satisfied. STS explores the tree of variable assignments in a breadth-first way with the MiniSat SAT solver [38] as an oracle. During this procedure, it generates *pseudosolutions*, which are partial assignments to the variables that can be completed to full satisfying assignments. However, it has to invoke the SAT solver multiple times during the breadth-first exploration. In contrast, ESampler does not require SAT solving when generating satisfying assignments from a seed.

Technically, our derivation procedure aims to generate a large set of satisfying assignments from a given seed, and is orthogonal to the existing SAT samplers. It can be integrated into the existing samplers to improve their efficiency as we demonstrated using QuickSampler.

Sampling of satisfying assignments is also closely related to the model-counting problem which counts the number of satisfying assignments for a

Boolean formula. Model-counting techniques have been used for sampling satisfying assignments (e.g., SPUR [2]) while satisfying assignment sampling techniques can also be used for model-counting (e.g., STS [16] and ApproxCount [42]).

Outline. The remainder of this paper is organized as follows. In Sect. 2, we briefly revisit related concepts of Boolean formulas. We present our derivation procedure in Sect. 3, and show how to integrate it into existing SAT samplers in Sect. 4. We report evaluation results in Sect. 5 and conclude this work in Sect. 6.

2 Preliminaries

We first recap some basic notions and notations which are used in this work.

Boolean Formulas. Let us fix a set of Boolean variables $\mathcal{V}$. A literal l is either a Boolean variable $x \in \mathcal{V}$ or its negation $\neg x$. We denote by $\texttt{var}(l)$ the variable x used in the literal l, namely, $\texttt{var}(x) = \texttt{var}(\neg x) = x$.

A *Boolean formula* Φ is a Boolean combination of literals using logical-AND ($\wedge$) and logical-OR ($\vee$) operators. As a convention, we assume that Boolean formulas are given in the conjunctive normal form (CNF) $\bigwedge_{j=1}^{m} \bigvee_{i=1}^{n_j} l_i^j$, where for each $1 \leq j \leq m$ and $1 \leq i \leq n_j$, l_i^j is a literal, and $\bigvee_{i=1}^{n_j} l_i^j$ is referred to a *clause* for each $1 \leq j \leq m$. Given a Boolean formula Φ and a literal l, let Φ_l denote the set of clauses that contain the literal l. For each clause $\phi = \bigvee_{i=1}^{n_j} l_i^j$, we assume that all literals in ϕ are distinct, and denote by $|\phi|$ the number n_j of literals in the clause ϕ.

Assignments. An *assignment* is a function $v \colon \mathcal{V} \to \{0,1\}$ which assigns a Boolean value to each Boolean variable $x \in \mathcal{V}$. Given a Boolean formula Φ and an assignment v, v is a *satisfying assignment* of Φ, denoted by $v \models \Phi$, if the Boolean formula Φ evaluates to 1 under the assignment v. For each assignment v, variable $x \in \mathcal{V}$ and value $i \in \{0,1\}$, we denote by $v[x \mapsto i]$ the assignment that agrees with v except for the variable x, i.e., for each variable $y \in \mathcal{V}$, $v[x \mapsto i](y) = v(y)$ if $y \neq x$, $v[x \mapsto i](y) = i$ otherwise.

Satisfiability and Maximum Satisfiability. Given a Boolean formula Φ, the *satisfiability problem* (SAT) is to determine whether a satisfying assignment of Φ exists or not. If Φ is satisfied, then a solution is produced as a witness. It is well-known that the SAT problem is **NP**-complete [12].

Given a pair of Boolean formulas (Φ, Ψ), the *maximum satisfiability problem* (MaxSAT) is to find a satisfying assignment that satisfies the Boolean formula Φ and meanwhile maximizes the number of satisfied clauses in Ψ. The clauses in Φ are usually called *hard* constraints, while the clauses in Ψ are called *soft* constraints. It is easy to see that the MaxSAT problem is at least **NP**-hard and can be solved by the state-of-the-art solvers such as Z3 [7].

In this work, by solvers we mean tools that are able to produce one satisfying assignment of the (Max)SAT problem whilst by samplers we mean those that are able to generate more than one satisfying assignments.

Independent Support. Given a Boolean formula Φ, an *independent support* $\texttt{Supp}$ of Φ [10], is a set of variables such that for each pair of satisfying assignments (v, v') of Φ, if $v(x) = v'(x)$ holds for all variables $x \in \texttt{Supp}$, then $v(y) = v'(y)$ holds for all variables $y \in \mathcal{V} \backslash \texttt{Supp}$. Intuitively, the truth values of the independent support $\texttt{Supp}_\Phi$ uniquely determine the truth values of the other variables. In other words, flipping the truth value of any variable $y \in \mathcal{V} \backslash \texttt{Supp}$ in the satisfying assignment v only will make the resulting assignment $v[y \mapsto \neg v(y)]$ fail to satisfy Φ.

It is easy to see that any superset of an independent support of Φ is also an independent support. There are tools, such as `MIS` and `SMIS` [20], that are able to compute minimal and minimum independent supports for Boolean formulas, where *minimal* means removing any variable from the independent support X will lead to a non-independent support, and *minimum* means there does not exist any independent support whose size is smaller. Remark that the problem of deciding whether a set of variables is a minimal independent support of a Boolean formula Φ is **DP**-complete, where $\mathbf{DP} := \{A - B \mid A, B \in \mathbf{NP}\}$.

3 Derivation Procedure

In this section, we first present a motivating example which exemplifies the key insight behind our approach for efficiently generating a large number of satisfying assignments. We then provide a derivation procedure which is able to derive more satisfying assignments from a seed by flipping the truth values of properly chosen variables without invoking computationally expensive SAT solving. The derivation procedure is the basis for efficiently generating a large number of satisfying assignments, and can be integrated into other samplers.

3.1 Motivating Example

To exemplify the key insight behind our approach, let us consider the following Boolean formula

$$\Phi_e \equiv (\neg a \vee b \vee c) \wedge (a \vee \neg c \vee \neg d) \wedge (\neg b \vee c) \wedge (b \vee d).$$

Suppose we have already obtained one satisfying assignment v (called *seed*) of Φ_e with $v(a) = v(b) = v(c) = v(d) = 1$. We can observe that the clause $\neg a \vee b \vee c$ (resp. $b \vee d$) contains two literals b and c (resp. b and d) whose values are 1 under the assignment v. Moreover, the common literal b does not appear in other clauses, namely, $a \vee \neg c \vee \neg d$ and $\neg b \vee c$. By flipping the value $v(b)$ of the variable b in the assignment v, we can obtain a new assignment $v[b \mapsto \neg v(b)]$, which is also a satisfying assignment of Φ_e.

However, by flipping the value $v(c)$ of the variable c in the assignment v, the new assignment $v[c \mapsto \neg v(c)]$ is not a satisfying assignment of Φ_e. This is because the clause $\neg b \vee c$ contains only one literal c whose value is 1 under the assignment v. After flipping the value $v(c)$ of the variable c in the assignment v, the clause $(\neg b \vee c)$ is no more satisfied.

Algorithm 1. Deriving satisfying assignments from a seed

```
1: procedure DERIVATION(Φ, v, MaxNum, Supp)
2:     Derived = {v};
3:     Queue = [v];
4:     while Queue ≠ ∅ ∧ |Derived| ≤ MaxNum do
5:         v = Queue.DEQUEUE();
6:         L = {x ∈ Supp | v(x) = 1} ∪ {¬x | v(x) = 0 ∧ x ∈ Supp};
7:         for all l ∈ L do
8:             if for each ⋁_{i=1}^{m} l_i ∈ Φ_l, there exists 1 ≤ i ≤ m. (l ≠ l_i ∧ l_i ∈ L) then
9:                 x = var(l);
10:                v' = v[x ↦ ¬v(x)];
11:                if v' ∉ Derived then
12:                    Derived = Derived ∪ {v'};
13:                    Queue.ENQUEUE(v');
14:                end if
15:            end if
16:        end for
17:    end while
18:    return Derived;
19: end procedure
```

This simple observation suggests that, for a seed v, we may identify proper variables (such as b but not c in the above example) so that when the value of one such variable is flipped it is still a satisfying assignment. Furthermore, the new satisfying assignments can be used as seeds to derive more satisfying assignments. This often allows generation of a larger number of satisfying assignments without invoking computationally expensive SAT solving.

3.2 Derivation Algorithm

In this subsection, we present a derivation procedure for deriving new satisfying assignments from a given seed. Given a Boolean formula Φ, a seed v and an independent support `Supp` of Φ, and the maximal number `MaxNum` of expected satisfying assignments, the procedure DERIVATION in Algorithm 1 iteratively derives new satisfying assignments from the seed v until no new satisfying assignment can be found or the number of generated satisfying assignments hits the threshold `MaxNum`. It returns the set of generated satisfying assignments including the original seed v.

To start, Algorithm 1 initializes the set `Derived` for recording all the generated satisfying assignments (Line 2) and the queue `Queue` for storing the seeds (Line 3). It then repeats the following procedure until no new satisfying assignments can be found or the number of the generated satisfying assignments exceeds the threshold `MaxNum` (While-loop).

For each seed v in `Queue` (Line 5), it first identifies all the literals l whose value is 1 under the assignment v(Line 6). After that, for each literal $l \in L$ whose variable $\texttt{var}(l) \in \texttt{Supp}$ (Line 7), it checks, for each clause $\bigvee_{i=j}^{m} l_j$ that

contains the literal l (i.e., $\bigvee_{i=j}^{m} l_j \in \Phi_l$), whether $\bigvee_{i=j}^{m} l_j$ contains a distinct literal l_i whose value is also 1, i.e., $l_i \in L$ (Line 8). If this is the case, we can deduce that the assignment $v[x \mapsto \neg v(x)]$ obtained from the assignment v by flipping the variable $x = \texttt{var}(l)$ is also a satisfying assignement of Φ. Therefore, we extract the variable x from the literal l (Line 9) and construct the assignment $v' = v[x \mapsto \neg v(x)]$ (Line 10). If the assignment v' has not been generated before, it is inserted to `Derived` and `Queue` (Lines 12 and 13).

One may notice that only variables in `Supp` are considered for flipping (Line 7). In general, we can take all the variables into account for flipping. However, as mentioned before (cf. Sect. 2), flipping variables outside of `Supp` will definitely lead to unsatisfying assignments. Therefore, it suffices to consider variables from `Supp` for flipping. Due to this, the values of each variable outside of `Supp` are the same in all the generated satisfying assignments from a given seed.

We remark that the derivation procedure Derivation could alternatively be presented as a recursive procedure which invokes itself when a new satisfying assignment is generated, or equivalently, use a stack rather than a queue to store the generated seeds. Intuitively, using the queue `Queue` to store the seeds, our algorithm works in a breadth-first fashion, while the other two ways would follow a depth-first fashion. We adopt the current way because it is more efficient than the other two.

Theorem 1. *Given a Boolean formula Φ, a seed v and an independent support* `Supp` *of Φ, the set* `Derived` *returned by Algorithm 1 contains only satisfying assignments of Φ. Moreover, these assignments agree on the variables outside of* `Supp`*.*

Proof. We show that the set `Derived` returned by Algorithm 1 contains only satisfying assignments of Φ by applying induction on the sequence $v_0 v_1 \cdots$ of the assignments added into `Derived`. The base case is trivial as the seed v_0 satisfies the Boolean formula Φ. We prove the inductive step below.

Suppose $v_0, v_1 \cdots v_{k-1}$ have been added into the set `Derived` and the inductive step adds the assignment v_k into the set `Derived`. Then, v_k must be added due to one v of the previously added satisfying assignments $v_0, v_1 \cdots v_{k-1}$. There necessarily exists a literal l such that $x = \texttt{var}(l)$ and $v_k = v[x \mapsto \neg v(x)]$.

To show that v_k satisfies Φ, it is sufficient to prove that v_k satisfies all the clauses of Φ. Let us consider a clause $\bigvee_{i=j}^{m} l_j$ of Φ,

- If $\bigvee_{i=j}^{m} l_j$ does not contain the literal l, then by applying induction hypothesis, v satisfies the Boolean formula Φ and hence v satisfies the clause $\bigvee_{i=j}^{m} l_j$. Since $v_k = v[x \mapsto \neg v(x)]$ and $x = \texttt{var}(l)$, the truth of the clause $\bigvee_{i=j}^{m} l_j$ does not change when the value of x in v is flipped. Therefore, we get that the assignment v_k satisfies the clause $\bigvee_{i=j}^{m} l_j$.
- If $\bigvee_{i=1}^{m} l_i$ contains the literal l, then there exists another literal $l_i \in \{l_1, \cdots, l_m\}$ such that $l_i \neq l$ and $l_i \in L = \{x \mid v(x) = 1\} \cup \{\neg x \mid v(x) = 0\}$. From $l_i \in L = \{x \mid v(x) = 1\} \cup \{\neg x \mid v(x) = 0\}$, we deduce that the literal l_i, hence the clause $\bigvee_{i=1}^{m} l_i$, holds under the assignment v_k.

$$
\begin{array}{ll}
\Phi_e: & (\neg a \vee b \vee c) \wedge (a \vee \neg c \vee \neg d) \wedge (\neg b \vee c) \wedge (b \vee d) \\
 & \\
v_1: & (\ 0 \vee 1 \vee 1) \wedge (1 \vee\ 0\ \vee\ 0) \wedge (\ 0 \vee 1) \wedge (1 \vee 1) \\
 & \text{flip } b \text{ and } d \text{ respectively} \Downarrow \\
v_2: & (\ 0 \vee \mathbf{0} \vee 1) \wedge (1 \vee\ 0\ \vee\ 0) \wedge (\ \mathbf{1} \vee 1) \wedge (\mathbf{0} \vee 1) \\
v_3: & (\ 0 \vee 1 \vee 1) \wedge (1 \vee\ 0\ \vee\ \mathbf{1}) \wedge (\ 0 \vee 1) \wedge (1 \vee \mathbf{0}) \\
 & \text{flip } a \Downarrow \\
v_4: & (\ \mathbf{1} \vee 1 \vee 1) \wedge (\mathbf{0} \vee\ 0\ \vee\ 1) \wedge (\ 0 \vee 1) \wedge (1 \vee 0)
\end{array}
$$

Fig. 1. Derivation steps of the motivating example

Example 1. Recall the motivating example Φ_e. Suppose the input seed is v_1 with $v_1(a) = v_1(b) = v_1(c) = v_1(d) = 1$ and the independent support $\texttt{Supp} = \{a, b, d\}$. The derivation steps are shown in Fig. 1. At the beginning of the first iteration of the while-loop, $v = v_1$ and $L = \{a, b, c, d\}$.

1. Suppose the variable a is chosen for flipping (Line 7), the clause $a \vee \neg c \vee \neg d$ does not have any literals other than a that occur in L, then Algorithm 1 will not flip the variable a.
2. Next, the variable b is chosen for flipping (Line 7), both clauses $\neg a \vee b \vee c$ and $b \vee d$ contain literals c and d that occur in L, then Algorithm 1 will flip the variable b (Line 9) and produce a new satisfying assignment $v_2 = v_1[b \mapsto 0]$ (Line 10).
3. Finally, the variable d is chosen for flipping (Line 7), the clause $b \vee d$ contains literal b that occurs in L, then Algorithm 1 will flip the variable d (Line 9) and produce a new satisfying assignment $v_3 = v_1[d \mapsto 0]$ (Line 10).

At the end of the first iteration of the while-loop, $\texttt{Derived} = \{v_1, v_2, v_3\}$ and $\texttt{Queue} = [v_2, v_3]$. After entering the second iteration of the while-loop, $v = v_2$, $\texttt{Queue}$ (resp. L) becomes $[v_3]$ (resp. $\{a, \neg b, c, d\}$). By applying similar steps as above, the satisfying assignment v_2 is regenerated but will not be inserted to $\texttt{Derived}$ or $\texttt{Queue}$.

At the end of the second iteration of the while-loop, $\texttt{Derived} = \{v_1, v_2, v_3\}$ and $\texttt{Queue} = [v_3]$. After entering the third iteration of the while-loop, $v = v_3$, $\texttt{Queue}$ (resp. L) becomes $\emptyset$ (resp. $\{a, b, c, \neg d\}$). By applying similar steps as above, Algorithm 1 will flip the variable a and produce a new satisfying assignment $v_4 = v_3[a \mapsto 0]$. In the end, no more new satisfying assignments can be generated and Algorithm 1 returns the set $\{v_1, v_2, v_3, v_4\}$. □

4 ESampler

In this section, we show that our derivation procedure is of generic nature in the sense that it can be integrated with other samplers. The basic idea is to generate seeds by invoking an existing sampler as an iterator, which returns a unique satisfying assignment each time. For each seed, we derive more satisfying

Algorithm 2. Integrated our derivation procedure into an existing sampler

```
1: procedure IntegratedSampler(Sampler, Φ, T, MaxPerSeed, Supp, RT, DT)
2:     Solutions = ∅;
3:     Derivable = false;
4:     Round = 0;
5:     Iterator = Sampler(Φ, Supp);
6:     repeat
7:         v = Iterator.next();
8:         if v == Null ∨ v ∈ Solutions then
9:             break;
10:        end if
11:        if Derivable == true ∨ Round<RT then
12:            Derived = Derivation(Φ, MaxPerSeed, v, Supp);
13:            Solutions = Solutions ∪ Derived;
14:            if |Derived| ≥ DT then
15:                Derivable = true;
16:            else
17:                Round = Round + 1;
18:            end if
19:        else
20:            Solutions = Solutions ∪ {v};
21:        end if
22:    until T is satisfied
23:    return Solutions;
24: end procedure
```

assignments by invoking our derivation procedure. However, our derivation procedure may not be effective on some Boolean formulas. Therefore, we propose a heuristic to determine whether our derivation procedure is able to derive a large number of satisfying assignments or not. If it can derive a large number of satisfying assignments, we apply the derivation procedure for each satisfying assignment generated by the sampler, otherwise we disable it.

Our idea is formalized as the procedure IntegratedSampler in Algorithm 2, which takes, as input, an off-the-shelf sampler `Sampler`, a Boolean formula Φ, a threshold `T` as the termination condition, the maximum number `MaxPerSeed` of satisfying assignments per seed, an independent support `Supp` of the Boolean formula Φ, two thresholds `RT` and `DT` to determine whether our derivation procedure is able to derive a large number of satisfying assignments, and returns a set `Solutions` of satisfying assignments of the formula Φ.

The procedure IntegratedSampler first initializes the set `Solutions`, the Boolean flag `Derivable`, the counter `Round` and the iterator `Iterator` of the sampler using the independent support `Supp` and Boolean formula Φ (Lines 2–5), where the Boolean flag `Derivable` and counter `Round` are used to determine if our derivation procedure is able to derive a large number of satisfying assignments. Then, it repeats the following procedure until the threshold `T` is hit.

During each iteration, IntegratedSampler first invokes the iterator to get a satisfying assignment v, where v is `Null` if Φ is unsatisfiable or the iterator cannot find new satisfying assignments. If v is Null or already exists in `Solutions`, it breaks the loop (Line 9). Otherwise it checks if the Boolean flag `Derivable` is true or the number `Round` of iterations is less than the threshold `RT`.

- If neither holds, the derivation procedure is considered to be not able to derive a large number of satisfying assignments and will be skipped;
- Otherwise, the derivation procedure is invoked to generate more satisfying assignments which are added to the set `Solutions` (Lines 12–13). If the number of satisfying assignments generated by the derivation procedure exceeds the threshold `DT`, we consider that the derivation procedure is able to derive a large number of satisfying assignments and set the Boolean flag `Derivable` to true (Line 15). Otherwise, we increase the counter `Round` by one. In general, we probe the effectiveness of the derivation procedure by checking the number of satisfying assignments generated by the derivation procedure in the first `RT` iterations. In our experiments, we found few rounds are sufficient to detect for each benchmark whether a large number of satisfying assignments can be derived from a seed. Therefore, we set $\texttt{RT} = 3$ and $\texttt{DT} = 16$.

By Theorem 1, we obtain that

Theorem 2. *The set* `Solutions` *returned by Algorithm 2 contains only satisfying assignments of* Φ.

5 Implementation and Evaluation

We implement Algorithms 1 and 2 as an open-source tool ESampler in C++, with QuickSampler as the underlying seed generator. QuickSampler takes a Boolean formula and its independent support as inputs, and outputs a set of assignments. However, as mentioned above, assignments produced by QuickSampler may be duplicated or not satisfy the formula. As we focus on satisfying assignments of each Boolean formula in this work, we modify it so that duplicated and unsatisfying assignments are omitted.

ESampler takes a Boolean formula in the DIMACS [13] format and other required options as inputs, and outputs a set of satisfying assignments for the given Boolean formula. To reduce the memory usage of storing the satisfying assignments, we only store and output the satisfying assignments for the variables in the given independent support. Indeed, the truth values of the independent support determine those of the other variables, thereby the satisfying assignments can be easily completed.

We compare ESampler with three state-of-the-art tools QuickSampler, STS and UniGen3 [36]. As done by [14], for a fair comparison, we modify STS so that the additional independent support information can be used by STS.

Benchmarks. To evaluate the performance, we conducted extensive experiments. Industrial testing and verification instances are typically proprietary and

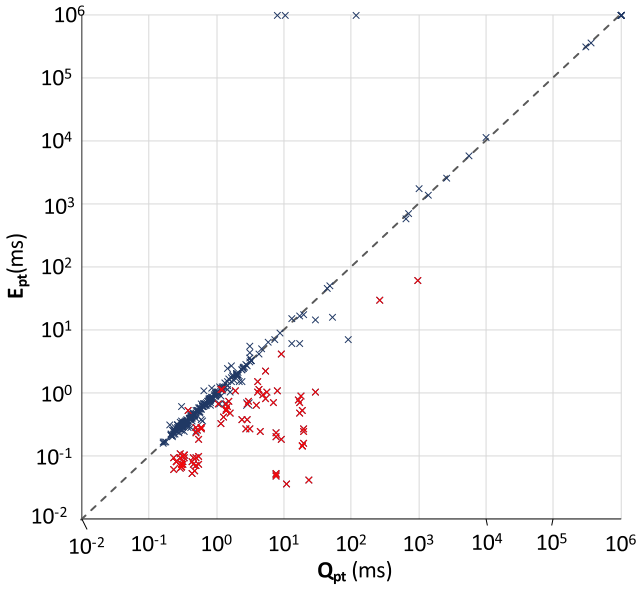


Fig. 2. ESampler vs. QuickSampler

unavailable for published research. Therefore, we conducted experiments on the publicly available benchmarks from UniGen [10], which consist of 370 Boolean formulas in the DIMACS format and the independent supports thereof. Indeed, the independent supports of most Boolean formulas could be computed using `MIS` [20] in few seconds. These benchmarks come from four classes of problem instances:

1. ISCAS89: constraints arising from ISCAS89 circuits with parity conditions on randomly chosen subsets of outputs and next-state variables;
2. SMTLib: bit-blasted versions of SMTLib benchmarks;
3. ProgSyn: constraints arising from automated program synthesis; and
4. BMC: constraints arising in bounded model checking of circuits.

Note that the accompanied independent supports of these benchmarks may contain variables that are not involved in the corresponding Boolean formulas; such variables are removed from the independent supports in our experiments. We remark that our approach also works without the given independent supports, in which case the independent support of a Boolean formula contains all the involved variables. Since it does not make any sense to compute solutions for unsatisfiable Boolean formulas or the satisfiability cannot be solved, we checked the satisfiability of all these Boolean formulas with a timeout of one hour per Boolean formula using Z3 [27]. There are two unsatisfiable formulas (79.sk_4_40 and 36.sk_3_77), and four unsolvable formulas (logcount.sk_16_86, log2.sk_72_391, xpose.sk_6_134, and listReverse.sk_11_43). These formulas are not considered here, leaving 364 Boolean formulas.

Table 1. Comparison of QuickSampler and ESampler

Benchmark	#Vars	#Cls	$Q_t(ks)$	Q_n	$Q_{pt}(ms)$	$E_t(ks)$	E_n	E_{dn}	$E_{pt}(ms)$	$\frac{Q_{pt}}{E_{pt}}$
s27_new_15_7	17	43	0.00	48	1.39	0.00	48	42	0.54	2.56
blasted_case.54	203	725	0.20	691,127	0.30	0.20	664,548	0	0.30	0.99
20.sk_1_51	15, 475	60,994	3.94	491,074	8.02	1.67	1, 520, 152	∼1,520k	1.10	7.31
s35932_7_4	17,849	44,425	4.22	245,506	17.17	0.63	1,270,247	∼1,270k	0.50	34
blasted_case.126	302	1,129	0.34	1,007,411	0.34	0.34	1,022,991	0	0.33	1.03
blasted_case.40	245	650	0.41	1,149,017	0.35	0.41	1,149,017	0	0.36	0.99
s349_3_2	198	469	0.24	1,008,386	0.24	0.07	1,142,757	∼1,088k	0.06	3.81
56.sk_6_38	4,842	17,828	1.97	1,004,037	1.96	1.18	1,093,080	∼1,092k	1.08	1.81
blasted_case.107	618	1,661	0.82	1,149,017	0.72	0.84	1,149,017	0	0.73	0.98
s832a_15_7	693	2,017	0.53	1,001,732	0.53	0.52	1,000,093	4	0.52	1.01
s420_new_7_4	312	770	0.35	1,117,085	0.31	0.08	1,048,576	∼1,043k	0.07	4.18
blasted_case.124	133	386	0.23	1,039,563	0.22	0.22	1,008,715	0	0.22	1.02
s35932_15_7	17,918	44,709	4.29	145,499	29.46	1.34	1,270,247	∼1,270k	1.06	27
blasted_case.207	824	2,128	1.02	1,149,017	0.89	0.98	1,149,017	0	0.86	1.04
blasted_case.120	284	851	0.41	1,113,780	0.37	0.40	1,044,731	0	0.38	0.97
63.sk_3_64	7,242	24,379	4.04	917,681	4.41	0.30	1,200,120	∼1,200k	0.25	17
s420_7_4	312	770	0.32	1,058,100	0.31	0.10	1,366,784	∼1,363k	0.07	4.14

Experiment Setup. In our experiments, the maximal number `MaxPerSeed` of satisfying assignments per seed is 10,000 and the maximal number `T` of satisfying assignments to compute is 1,000,000, unless the recent 10 assignments/pseudosolutions already exist. As aforementioned, we set `RT` = 3 and `DT` = 16 for ESampler. For STS and QuickSampler, we use their default parameter settings. All the experiments were conducted on Intel Xeon E5-2620 v4 2.10 GHz CPU with 256 RAM GB and the one-hour timeout.

5.1 Comparison with QuickSampler

Figure 2 shows the scatter plot comparing the average execution time per satisfying assignment between ESampler and QuickSampler on all the 364 formulas. Timeout occurred along the top or right border; the red color indicates that `Derivable` is set true by Algorithm 2, namely, it determines that our derivation procedure is able to derive a large number of satisfying assignments. Points below (resp. above) the diagonal line indicate that ESampler performs better (resp. worse) than QuickSampler.

Table 1 reports the performance of QuickSampler and ESampler for a representative subset of the benchmarks. Columns benchmark, #Vars and #Cls respectively show the name, numbers of variables and clauses in each Boolean formula. Columns Q_t and E_t (resp. Q_{pt} and E_{pt}) give the total execution time in thousand seconds (ks) (resp. execution time per satisfying assignment in milliseconds (ms)) of QuickSampler and ESampler, respectively. Columns Q_n and E_n show the total numbers of satisfying assignments generated by QuickSampler and ESampler, respectively. Column E_{dn} gives the numbers of satisfying assignments generated by our derivation procedure. The last column provides

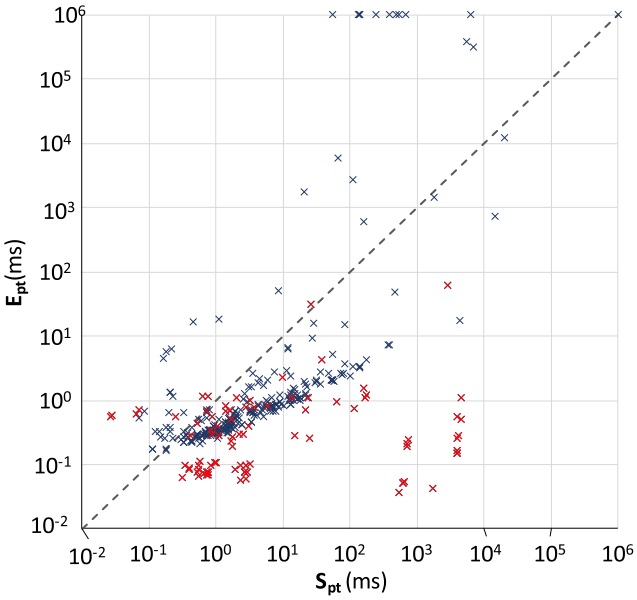


Fig. 3. ESampler vs. STS

the ratio of execution time per satisfying assignment between QuickSampler and ESampler, depicting the speedup of ESampler. We can observe when our derivation procedure works, it can produce more satisfying assignments (e.g., 20.sk_1_51 and s35932_7_4) than QuickSampler in the same time budget, while when it does not work well, it often does not produce any satisfying assignments (e.g., blasted_case.54 and blasted_case.40). Note that, since QuickSampler is a randomized approach, QuickSampler and ESampler may produce different satisfying assignments when our derivation procedure does not work, although ESampler is built on QuickSampler.

Summary. ESampler and QuickSampler respectively failed on 11 and 7 benchmarks due to the failure of MaxSAT solving. The difference between the numbers of the failed benchmarks indicates that the soft constraints generated randomly slightly affect MaxSAT solving. When ESampler determined that the derivation procedure can generate a large number of satisfying assignments, ESampler performed better than QuickSampler on almost all the benchmarks. While ESampler determined that our derivation procedure was not able to generate a large number of satisfying assignments, ESampler was comparable to QuickSampler. Specifically, ESampler was faster than QuickSampler on 227 benchmarks. It was 1.66× faster on average and more than 5× faster on 41 benchmarks, while is 1.2 times slower on 16 benchmarks.

Table 2. Comparison of STS and ESampler

Benchmark	#Vars	#Cls	$S_t(ks)$	S_n	$S_{pt}(ms)$	$E_t(ks)$	E_n	E_{dn}	$E_{pt}(ms)$	$\frac{S_{pt}}{E_{pt}}$
s27_new_15_7	17	43	0.00	48	0.85	0.00	48	42	0.54	1.57
blasted_case.54	203	725	1.45	961,782	1.51	0.20	664,548	0	0.30	5.06
20.sk_1_51	15,475	60,994	3.60	151,948	23.69	1.67	1,520,152	∼1,520k	1.10	21
s35932_7_4	17,849	44,425	3.49	800	4,361	0.63	1,270,247	∼1,270k	0.50	8,757
blasted_case.126	302	1,129	0.92	1,000,006	0.92	0.34	1,022,991	0	0.33	2.78
blasted_case.40	245	650	1.53	1,000,000	1.53	0.41	1,149,017	0	0.36	4.30
s349_3_2	198	469	0.31	1,000,028	0.31	0.07	1,142,757	∼1,088k	0.06	4.94
56.sk_6_38	4,842	17,828	1.99	1,000,048	1.99	1.18	1,093,080	∼1,092k	1.08	1.84
blasted_case.107	618	1,661	3.60	558,950	6.44	0.84	1,149,017	0	0.73	8.82
s832a_15_7	693	2,017	1.55	1,000,018	1.55	0.52	1,000,093	4	0.52	2.97
s420_new_7_4	312	770	0.72	1,000,001	0.72	0.08	1,048,576	∼1,043k	0.07	9.68
blasted_case.124	133	386	0.32	1,000,013	0.32	0.22	1,008,715	0	0.22	1.47
s35932_15_7	17,918	44,709	3.50	800	4,380	1.34	1,270,247	∼1,270k	1.06	4,140
blasted_case.207	824	2,128	3.60	276,250	13.03	0.98	1,149,017	0	0.86	15
blasted_case.120	284	851	1.59	1,000,000	1.59	0.40	1,044,731	0	0.38	4.13
63.sk_3_64	7,242	24,379	3.60	148,050	24.31	0.30	1,200,120	∼1,200k	0.25	97
s420_7_4	312	770	0.74	1,000,038	0.74	0.10	1,366,784	∼1,363k	0.07	9.93

5.2 Comparison with STS

Figure 3 shows the scatter plot comparing the average execution time per satisfying assignment between ESampler and STS on all the 364 formulas. Recall that timeout occurred along the top or right border, the red color indicates that `Derivable` is set true by Algorithm 2, and points below the diagonal line indicate that ESampler performs better than QuickSampler, and vice versa.

Table 2 reports the performance of STS and ESampler for the same representative subset of the benchmarks. Column S_t (resp. S_{pt}) gives the total execution time in thousand seconds (ks) (resp. execution time per satisfying assignment in milliseconds (ms)) of STS. Column S_n shows the total number of satisfying assignments generated by STS for each Boolean formula. The last column provides the ratio of execution time per satisfying assignment between STS and ESampler, depicting the speedup of ESampler.

Summary. STS failed on 1 benchmark because the underlying SAT solver Minisat failed to solve the Boolean formula, while ESampler failed on 11 benchmarks. In general, ESampler performed better than STS on most benchmarks. It was faster on 316 benchmarks (5.47× faster on average and more than 10× faster on 93 benchmarks), while was 1.2 times slower on only 45 benchmarks.

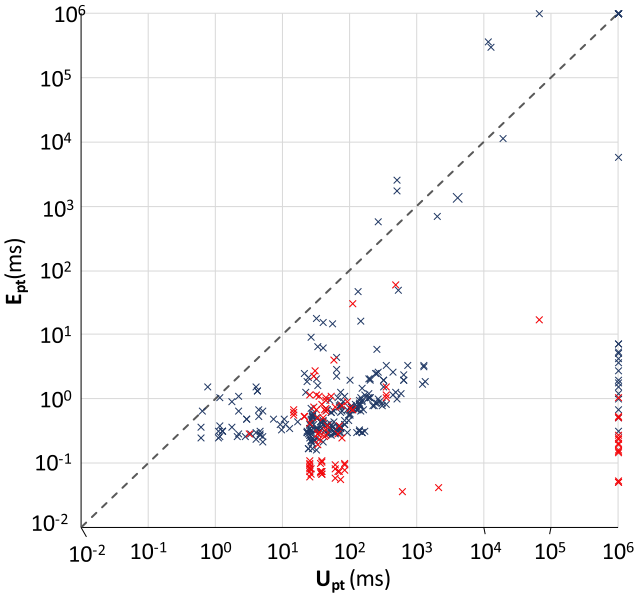


Fig. 4. ESampler vs. UniGen3

5.3 Comparison with UniGen3

Figure 4 shows the scatter plot comparing the average execution time per satisfying assignment between ESampler and UniGen3 on all the 364 formulas. Almost all the points are below the diagonal line, indicating ESampler significantly outperforms UniGen3.

Table 3 reports the performance of UniGen3 and ESampler on the same representative subset of benchmarks. Column U_t (resp. U_{pt}) gives the total execution time in thousand seconds (ks) (resp. execution time per satisfying assignment in milliseconds (ms)) of UniGen3. Column U_n shows the total number of satisfying assignments generated by UniGen3 for each Boolean formula. The last column provides the ratio of execution time per satisfying assignment between UniGen3 and ESampler, depicting the speedup of ESampler.

Summary. UniGen3 failed on 40 benchmarks. Recall that ESampler failed on 11 benchmarks. No matter whether or not ESampler determined that the derivation procedure was able to generate a large number of satisfying assignments, ESampler performed significantly better than UniGen3 on almost all the benchmarks. Specifically, ESampler was faster than STS on 348 benchmarks. It was 69.8× faster on average and more than 100× faster on 194 benchmarks, while was 1.2 times slower on only 7 benchmarks.

Table 3. Comparison of UniGen3 and ESampler

Benchmark	#Vars	#Cls	$U_t(ks)$	U_n	$U_{pt}(ms)$	$E_t(ks)$	E_n	E_{dn}	$E_{pt}(ms)$	$\frac{U_{pt}}{E_{pt}}$
s27_new_15_7	17	43	0.00	48	20.83	0.00	48	42	0.54	19.83
blasted_case.54	203	725	3.60	158,168	22.76	0.20	664,548	0	0.30	3.33
20.sk_1_51	15,475	60,994	3.60	70,312	51.21	1.67	1,520,152	∼1,520k	1.10	57.83
s35932_7_4	17,849	44,425	3.60	0	–	0.63	1,270,247	∼1,270k	0.50	–
blasted_case.126	302	1,129	3.60	77,185	46.67	0.34	1,022,991	0	0.33	9.33
blasted_case.40	245	650	3.60	50,380	71.46	0.41	1,149,017	0	0.36	5.78
s349_3_2	198	469	3.60	144,279	24.95	0.07	1,142,757	∼1,088k	0.06	1,643
56.sk_6_38	4,842	17,828	3.60	104,149	34.57	1.18	1,093,080	∼1,092k	1.08	30.63
blasted_case.107	618	1,661	3.60	0	–	0.84	1,149,017	0	0.73	–
s832a_15_7	693	2,017	3.60	132,705	27.13	0.52	1,000,093	4	0.52	2.02
s420_new_7_4	312	770	3.60	98,934	36.39	0.08	1,048,576	∼1,043k	0.07	3,966
blasted_case.124	133	386	3.60	89,376	40.28	0.22	1,008,715	0	0.22	14.92
s35932_15_7	17,918	44,709	3.60	0	–	1.34	1,270,247	∼1,270k	1.06	–
blasted_case.207	824	2,128	3.60	15,026	239.92	0.98	1,149,017	0	0.86	16.89
blasted_case.120	284	851	3.60	51,799	69.5	0.40	1,044,731	0	0.38	3.12
63.sk_3_64	7,242	24,379	3.60	48,004	75.01	0.30	1,200,120	∼1,200k	0.25	1,133
s420_7_4	312	770	3.60	95,260	37.79	0.10	1,366,784	∼1,363k	0.07	3,990

5.4 Execution Time vs Number of Satisfying Assignments

To see the relation between the execution time and the number of satisfying assignments, we evaluate ESampler on four randomly chosen benchmarks by varying the execution time and counting the number of satisfying assignments. Figure 5 shows the plot for the four randomly chosen benchmarks, where the x-axis is the execution time (in seconds) and the y-axis is number of satisfying assignments (#assignments). We can observe that the number of satisfying assignments for each benchmark is almost linear in the execution time. These results demonstrate the effectiveness of our derivation procedure.

5.5 Testing Uniformity

Since QuickSampler does not provide a guarantee of uniformity, neither does ESampler. We empirically show that the uniformity of the solutions can be controlled by adjusting the maximal number of solutions per seed, i.e., the parameter `MaxNumPerSeed`. We run ESampler on a randomly selected benchmark (i.e., 27.sk_3_32) on which our derivation procedure works, where duplicated solutions are recorded to measure uniformity and the mutation phase of QuickSampler is disabled to be more precise.

Figure 6 depicts the distributions of solutions when `MaxNumPerSeed` is 0, 10 and 100, where (x, y) denotes that there are y unique solutions each of which occurs x times. We can observe that the smaller the parameter `MaxNumPerSeed` is, the closer the distribution is to the normal distribution, meaning that the solutions generated by our tool are actually close to uniform.

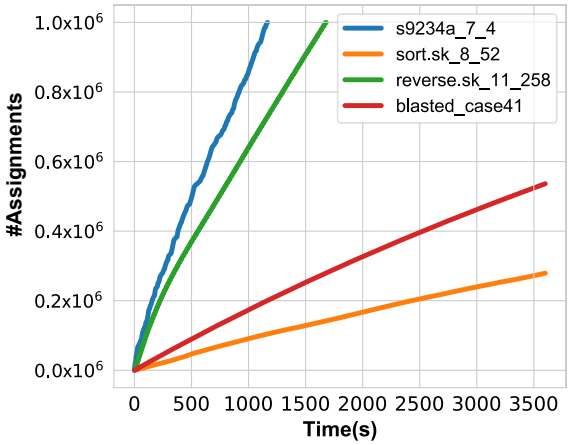
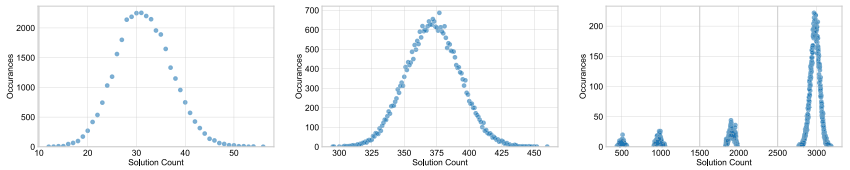


Fig. 5. Time vs. #assignments



(a) `MaxNumPerSeed`=0 (b) `MaxNumPerSeed`=10 (c) `MaxNumPerSeed`=100

Fig. 6. Distribution of solutions

6 Conclusion

We have proposed a novel approach to generate a large set of satisfying assignments from a seed assignment without invoking computationally expensive SAT solving. Our approach is orthogonal to the previous techniques and could be integrated into the existing SAT samplers. We have also developed a new tool ESampler, based on the recent sampler QuickSampler as the seed generator. We have carried out extensive experiments on real-world benchmarks. The experimental results confirmed the effectiveness and efficiency of our approach.

In future, we plan to further improve the performance of our tool ESampler, which will be applied in emerging practical scenarios such as adversarial training of binaried neural networks and constrained hardware design fuzz testing.

References

1. Abed, S., Abdelaal, A.A., Alshayeji, M.H., Ahmad, I.: Sat-based and CP-based declarative approaches for top-rank-k closed frequent itemset mining. Int. J. Intell. Syst. **36**(1), 112–151 (2021)

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. Angluin, D.: On counting problems and the polynomial-time hierarchy. Theoret. Comput. Sci. **12**, 161–173 (1980)
4. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and Bayesian inference. In: Proceedings of the 44th Symposium on Foundations of Computer Science, 11–14 October 2003, Cambridge, MA, USA, pp. 340–351 (2003)
5. Bellare, M., Goldreich, O., Petrank, E.: Uniform generation of np-witnesses using an NP-oracle. Inf. Comput. **163**(2), 510–526 (2000)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
7. Bjørner, N., Phan, A.: νz - maximal satisfaction with Z3. In: Proceedings of the 6th International Symposium on Symbolic Computation in Software Science, pp. 1–9 (2014)
8. 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
9. 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
10. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: Proceedings of the 51st Annual Design Automation Conference (DAC), pp. 60:1–60:6 (2014)
11. Chen, G., Zhao, Z., Song, F., Chen, S., Fan, L., Liu, Y.: SEC4SR: a security analysis platform for speaker recognition. CoRR abs/2109.01766 (2021)
12. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, pp. 151–158 (1971)
13. DIMACS: Clique and coloring problems graph format (1993). http://archive.dimacs.rutgers.edu/pub/challenge/graph/doc/ccformat.tex. Accessed 16 Sept 2021
14. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: Proceedings of the 40th International Conference on Software Engineering, pp. 549–559 (2018)
15. Ermon, S., Gomes, C.P., Sabharwal, A., Selman, B.: Embed and project: discrete sampling with universal hashing. In: Proceedings of the 27th Annual Conference on Neural Information Processing Systems, pp. 2085–2093 (2013)
16. Ermon, S., Gomes, C.P., Selman, B.: Uniform solution sampling using a constraint solver as an oracle. In: Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, pp. 255–264 (2012)
17. Gomes, C.P., Sabharwal, A., Selman, B.: Near-uniform sampling of combinatorial spaces using XOR constraints. In: Proceedings of the 2th Annual Conference on Neural Information Processing Systems, pp. 481–488 (2006)
18. Guralnik, E., Aharoni, M., Birnbaum, A.J., Koyfman, A.: Simulation-based verification of floating-point division. IEEE Trans. Comput. **60**(2), 176–188 (2011)
19. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: Proceedings of the Annual Conference on Neural Information Processing Systems, pp. 4107–4115 (2016)

20. 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), 41–58 (2015). https://doi.org/10.1007/s10601-015-9204-z
21. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, 27–31 July 1997, Providence, Rhode Island, USA, pp. 203–208 (1997)
22. Kitchen, N.: Markov chain Monte Carlo stimulus generation for constrained random simulation. Ph.D. thesis, University of California, Berkeley, USA (2010)
23. Kitchen, N., Kuehlmann, A.: Stimulus generation for constrained random simulation. In: Proceedings of the 2007 International Conference on Computer-Aided Design, pp. 258–265 (2007)
24. Korneev, S., Narodytska, N., Pulina, L., Tacchella, A., Bjorner, N., Sagiv, M.: Constrained image generation using binarized neural networks with decision procedures. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 438–449. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_27
25. Kukula, J.H., Shiple, T.R.: Building circuits from relations. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 113–123. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_12
26. Meel, K.S.: Constrained counting and sampling: bridging the gap between theory and practice. CoRR abs/1806.02239 (2018)
27. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
28. Narodytska, N.: Formal analysis of deep binarized neural networks. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, pp. 5692–5696 (2018)
29. 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
30. Naveh, Y., et al.: Constraint-based random stimuli generation for hardware verification. In: Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference, pp. 1720–1727 (2006)
31. Naveh, Y., et al.: Constraint-based random stimuli generation for hardware verification. AI Mag. **28**(3), 13 (2007)
32. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**(1–2), 273–302 (1996)
33. Silva, J.P.M., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
34. Silva, J.P.M., Sakallah, K.A.: Grasp–a new search algorithm for satisfiability. In: Kuehlmann, A. (ed.) The Best of ICCAD, pp. 73–89. Springer, Boston (2003). https://doi.org/10.1007/978-1-4615-0292-0_7
35. Sipser, M.: A complexity theoretic approach to randomness. In: Proceedings of the 15th Annual ACM Symposium on Theory of Computing, pp. 330–335 (1983)
36. Soos, M., Gocht, S., Meel, K.S.: Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 463–484. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_22

37. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24
38. Sörensson, N., Eén, N.: MiniSat: a SAT solver with conflict-clause minimization. Solver Description (2005)
39. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979)
40. Vorobyov, K., Krishnan, P.: Combining static analysis and constraint solving for automatic test case generation. In: Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation, pp. 915–920 (2012)
41. Wei, W., Erenrich, J., Selman, B.: Towards efficient sampling: exploiting random walk strategies. In: Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence, pp. 670–676 (2004)
42. Wei, W., Selman, B.: A new approach to model counting. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 324–339. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_24
43. Yuan, J., Aziz, A., Pixley, C., Albin, K.: Simplifying boolean constraint solving for random simulation-vector generation. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **23**(3), 412–420 (2004)
44. Zhang, Y., Zhao, Z., Chen, G., Song, F., Chen, T.: BDD4BNN: a BDD-based quantitative analysis framework for binarized neural networks. In: Proceedings of the 33rd International Conference on Computer Aided Verification, pp. 175–200 (2021)
45. Zhang, Y., Li, J., Zhang, M., Pu, G., Song, F.: Optimizing backbone filtering. In: Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering, pp. 1–8 (2017)
46. Zhang, Y., Zhang, M., Pu, G., Song, F., Li, J.: Towards backbone computing: a greedy-whitening based approach. AI Commun. **31**(3), 267–280 (2018)
47. Zhao, Y., Bian, J., Deng, S., Kong, Z.: Random stimulus generation with self-tuning. In: Proceedings of the 13th International Conference on Computers Supported Cooperative Work in Design, pp. 62–65. IEEE (2009)
48. Zhao, Z., Chen, G., Wang, J., Yang, Y., Song, F., Sun, J.: Attack as defense: characterizing adversarial examples using robustness. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 42–55 (2021)