ESampler: Boosting Sampling of Satisfying Assignments for Boolean Formulas via Derivation

Yongjie Xu[a], Fu Song[a], Taolue Chen[b]

[a]School of Information Science and Technology, ShanghaiTech University, Shanghai, China
[b]Department of Computer Science, Birkbeck, University of London, London, UK

Abstract

Boolean satisfiability (SAT) plays a key role in diverse areas such as spanning planning, inference, 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 past years. In practical applications, usually a large number of satisfying assignments for a given Boolean formula are needed, the generation of which turns out to be a computational 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 based on an insight that flipping the truth values of properly chosen variables of a satisfying assignment could result in satisfying assignments without invoking computationally expensive SAT solving. We propose a derivation algorithm to discover such variables for each given satisfying assignment. 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 using two representative state-of-the-art samplers (QuickSampler and UniGen3) as the underlying satisfying assignment generation engine. We conduct extensive experiments on various publicly available benchmarks and apply ESampler to solve Bayesian inference. The results show that ESampler can efficiently boost the sampling of satisfying assignments of both QuickSampler and UniGen3 on a large portion of the benchmarks and is at least comparable on the others. ESampler performs considerably better than QuickSampler and UniGen3, as well as another state-of-the-art sampler SearchTreeSampler.

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

*Corresponding author
Email address: songfu@shanghaitech.edu.cn (Fu Song)

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, 2]. Apart from the classical problem of checking Boolean satisfiability, generating random satisfying assignments has attracted significant theoretical and practical interests over the years [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. 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 [12, 13]. 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 [16, 17]. 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 [18, 19], adversarial images could be generated by encoding a binarized neural network as a Boolean formula based on which satisfying assignments are sampled [20, 21].

Sampling satisfying assignments for a given Boolean formula is, however, challenging. Cook has shown in 1971 that the SAT problem is **NP**-complete [22]. In recent years, we have seen a tremendous progress in SAT solving, supported by techniques such as conflict-driven clause learning (CDCL [23, 24, 25]), yielding powerful solvers such as CryptoMiniSAT [26]. However, generating a large number of satisfying assignments is still computationally prohibitive and often infeasible in practical settings [27, 28].

In this work, we develop ESampler, aiming for boosting the generation of 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 two types of state-of-the-art sampler QuickSampler [28] and UniGen3 [29]. We carry out extensive experiments on the publicly available benchmarks from UniGen [30] which include hundreds of Boolean formulas from real-world testing and verification applications, and apply ESampler to solve Bayesian inference of the plan recognition problems [31]. Our experimental results show that ESampler is able to effectively boost the sampling of satisfying assignments of both QuickSampler and UniGen3 on a large portion of the benchmarks and is at least comparable on the others. Consequently, ESampler considerably performs better than QuickSampler and UniGen3, as well as the another state-of-the-art sampler SearchTreeSampler (STS in short) [32]. The experimental results confirm the efficacy of our derivation 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 based on two state-of-the-art samplers. Our tool is available at `https://github.com/ESampler/Esampler`.
- We conduct extensive experiments on hundreds of real-world benchmarks. The results show that our derivation approach is effective and consequently 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 [33]. 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 [9, 34, 35]. 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 [36, 37, 34]. Another class of satisfying assignment generation techniques with theoretical guarantees of uniformity is based on hashing [38, 39, 40, 41, 42, 30, 43, 29]. 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.

SAT samplers aiming to quickly generate a large number of assignments have recently been proposed. Both QuickSampler [28] and STS [32] share the same goal as our work, namely, fast generation of a larger number of assignments. QuickSampler works as follows. Given a Boolean formula Φ, it first constructs a random assignment v and then uses the MaxSAT solver [44] to solve the MaxSAT problem with the hard constraint Φ and soft constraint Ψ, where Ψ is the conjunction of literals x if $v(x) = 1$ or $\neg x$ if $v(x) = 0$. Solving the MaxSAT problem yields a satisfying assignment v' of Φ that is close to the random assignment v. After that, QuickSampler iteratively flips the value of each variable x in the satisfying assignment v' to find another close satisfying assignment v'_x using the MaxSAT solver, where the soft constraint asserts the satisfying assignment v' except for the flipped variable x, and the original Boolean formula together with the flipped variable is used as hard constraint. For each flipped variable x, the difference δ_x between two satisfying assignments v' and v'_x is computed. All such differences are combined and applied to mutate the satisfying assignment v' to generate a large number of assignments. However, the assignments generated by QuickSampler may not satisfy the Boolean formula, hence follow-up checkings are 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 [45] 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 SAT solvers 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 and UniGen3.

Sampling 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 [46]) while satisfying assignment sampling techniques can also be used for model-counting (e.g., STS [32] and ApproxCount [35]).

This article is an extended version of [47], but with substantial new material. In particular, we apply ESampler to boost another uniform sampler UniGen3 and carry out more experiments (cf. Section 5.4), which show the generality and wide applicability of ESampler to diverse seed generation samplers. We also apply ESampler for inference of Bayesian networks and report experimental results on the real-world plan recognition problems (cf. Section 6), showing a significant improvement of our approach ESampler over the samplers QuickSampler, STS and UniGen3.

Outline. The remainder of this paper is organized as follows. In Section 2, we briefly revisit related concepts of Boolean formulas. We present our derivation procedure in Section 3, and show how to integrate it into existing SAT samplers in Section 4. We report evaluation results in Section 5. We apply ESampler to Bayesian inference in Section 6 and conclude this work in Section 7.

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 $\mathtt{var}(l)$ the variable x used in the literal l, namely, $\mathtt{var}(x) = \mathtt{var}(\neg x) = x$.

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 $v{:}\,\mathcal{V} \to \{0,1\}$ 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. A *partial assignment* is a partial function $v{:}\,\mathcal{V} \to \{0,1\}$ such that for each $x \in \mathcal{V}$, $v(x)$ is a Boolean value if x is defined in v, otherwise x is undefined in 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 $y \in \mathcal{V}$,

$$v[x \mapsto i](y) = \begin{cases} v(y), & \text{if } y \neq x; \\ i, & \text{otherwise.} \end{cases}$$

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 [22].

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 [44].

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* $\mathtt{Supp}$ of Φ [30], 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 $x \in \mathtt{Supp}$, then

$v(y) = v'(y)$ holds for all variables $y \in \mathcal{V} \setminus \texttt{Supp}$. 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 $y \in \mathcal{V} \setminus \texttt{Supp}$ 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` [48], 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 [49], 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 the 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.

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

Algorithm 1 Deriving satisfying assignments from a seed

1: **procedure** DERIVATION(Φ, v, `MaxNum`, `Supp`)
2:     `Derived` $= \{v\}$;
3:     `Queue` $= [v]$;
4:     **while** `Queue` $\neq \emptyset \wedge |$`Derived`$| \leq$ `MaxNum` **do**
5:         $v =$ `Queue`.DEQUEUE();
6:         $L = \{x \mid v(x) = 1\} \cup \{\neg x \mid v(x) = 0\}$;
7:         **for all** $l \in L \wedge \texttt{var}(l) \in \texttt{Supp}$ **do**
8:             **if** $\forall \bigvee_{i=1}^{m} l_i \in \Phi_l, \exists i. \ (1 \leq i \leq m \wedge l \neq l_i \wedge l_i \in L)$ **then**
9:                 $x = \texttt{var}(l)$;
10:                 $v' = v[x \mapsto \neg v(x)]$;
11:                 **if** $v' \notin$ `Derived` **then**
12:                     `Derived` $=$ `Derived` $\cup \{v'\}$;
13:                     `Queue`.ENQUEUE(v');
14:                 **end if**
15:             **end if**
16:         **end for**
17:     **end while**
18:     **return** `Derived`;
19: **end procedure**

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, 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 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 $x = \mathtt{var}(l)$ is also a satisfying assignment 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. Section 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 ways.

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 $x = \mathtt{var}(l)$ 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 $x = \mathtt{var}(l)$, 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.

$$
\begin{array}{lccccccc}
\Phi_e: & (\neg a \lor b \lor c) & \land & (a \lor \neg c \lor \neg d) & \land & (\neg b \lor c) & \land & (b \lor d) \\
 & & & & & & & \\
v_1: & (0 \lor 1 \lor 1) & \land & (1 \lor 0 \lor 0) & \land & (0 \lor 1) & \land & (1 \lor 1) \\
 & \multicolumn{3}{l}{\text{flip } b \text{ and } d \text{ respectively} \Downarrow} & & & & \\
v_2: & (0 \lor \mathbf{0} \lor 1) & \land & (1 \lor 0 \lor 0) & \land & (\mathbf{1} \lor 1) & \land & (\mathbf{0} \lor 1) \\
v_3: & (0 \lor 1 \lor 1) & \land & (1 \lor 0 \lor \mathbf{1}) & \land & (0 \lor 1) & \land & (1 \lor \mathbf{0}) \\
 & & & \text{flip } a \Downarrow & & & & \\
v_4: & (\mathbf{1} \lor 1 \lor 1) & \land & (\mathbf{0} \lor 0 \lor 1) & \land & (0 \lor 1) & \land & (1 \lor 0)
\end{array}
$$

Figure 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* `Supp` $=$ *$\{a, b, d\}$. The derivation steps are shown in Figure 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). Since the clause $a \lor \neg c \lor \neg d$ does not have any literals other than a that occur in L, Algorithm 1 will not flip the variable a.*
2. *Next, the variable b is chosen for flipping (Line 7). Since the clause $\neg a \lor b \lor c$ contains the literal c, the clause $b \lor d$ contains the literal d, and both literals c and d occur in L, 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). Since the clause $b \lor d$ contains literal b that occurs in L, 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, `Derived` $= \{v_1, v_2, v_3\}$ *and* `Queue` $= [v_2, v_3]$*. After entering the second iteration of the while-loop, $v = v_2$,* `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* `Derived` *or* `Queue`*.*

At the end of the second iteration of the while-loop, `Derived` $= \{v_1, v_2, v_3\}$ *and* `Queue` $= [v_3]$*. After entering the third iteration of the while-loop, $v = v_3$,* `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

Algorithm 2 Integrated our derivation procedure into an existing sampler

```
procedure IntegratedSampler(Sampler,Φ,T,MaxPerSeed,Supp,RT,DT)
    Solutions = ∅;
    Derivable = false;
    Round = 0;
    Iterator = Sampler(Φ, Supp);
    repeat
        v = Iterator.next();
        if v == Null then
            break;
        end if
        if v ∈ Solutions then
            continue;
        end if
        if Derivable == true ∨ Round<RT then
            Derived = Derivation(Φ, v, MaxPerSeed, Supp);
            Solutions = Solutions ∪ Derived;
            if |Derived| ≥ DT then
                Derivable = true;
            else
                Round = Round + 1;
            end if
        else
            Solutions = Solutions ∪ {v};
        end if
    until T is satisfied
    return Solutions;
end procedure
```

unique satisfying assignment each time. For each seed, we derive more satisfying 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`, it breaks the loop (Line 9). If v already exists in `Solutions`, it skips this loop (Line 12). 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 15–16). 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 18). 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. In fact, on some benchmarks, the derivation algorithm can derive a few satisfying solutions from a seed in the beginning, but no solution could be derived afterwards. Thus, a small `DT` value can be used to avoid `Derivable` being set to true on these benchmarks, while it will not change on other benchmarks. Based on these observations, we set `RT` $= 3$ and `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—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. To demonstrate the generic nature of Algorithm 1 for deriving satisfying assignments from a seed, we also implement Algorithm 2 with UniGen3 as the underlying seed generator in our tool ESampler. In contract to QuickSampler, UniGen3 only produces satisfying assignments for each given Boolean formula and the satisfying assignments are sampled uniformly at random with theoretical guarantees.

ESampler takes a Boolean formula in the DIMACS [50] 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.

In the rest of this work, we denote by ESampler+QS (resp. ESampler+UG) our tool ESampler using QuickSampler (resp. UniGen3) as the underlying seed generator.

We mainly compare ESampler+QS with three state-of-the-art tools QuickSampler, STS and UniGen3 [29]. As done by [28], for a fair comparison, we modify STS so that the additional independent support information can be used by STS. To show the generic nature of Algorithm 1, we also compare ESampler+UG with UniGen3.

Benchmarks. To evaluate the performance, we conducted extensive experiments. Industrial testing and verification instances are typically proprietary and unavailable for published research. Therefore, we conducted experiments on the publicly available benchmarks from UniGen [30], 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` [48] 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 [51]. 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,

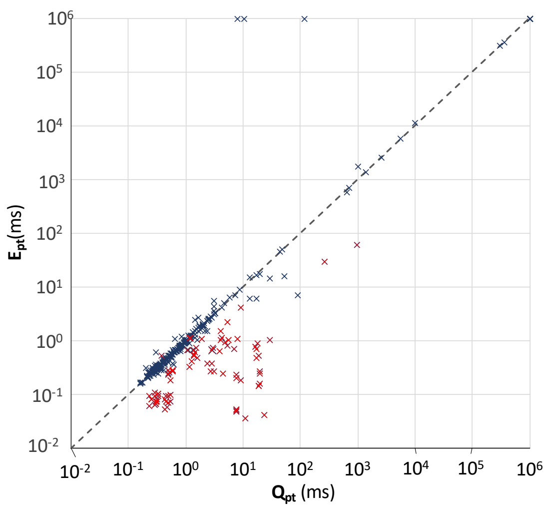


Figure 2: ESampler+QS vs. QuickSampler

xpose.sk_6_134, and listReverse.sk_11_43). These formulas are not considered here, leaving 364 Boolean formulas.

Experiment setup. In our experiments, for each sampler and each Boolean formula, we run the sampler once on the Boolean formula. Though samplers are non-deterministic, results on a large number of Boolean formulas are sufficient to demonstrate the performance of ESampler. 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 2.10GHz CPU with 256 RAM GB and the one-hour timeout.

5.1. *Comparison of* ESampler+*QS and* QuickSampler

Figure 2 shows the scatter plot comparing the average execution time per satisfying assignment between ESampler+QS 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+QS performs better (resp. worse) than QuickSampler.

The comparison of QuickSampler and ESampler+QS for a representative subset of the benchmarks is reported in Table 1. 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+QS, respec-

Table 1: Comparison of QuickSampler and ESampler+QS

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_case54	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_case126	302	1,129	0.34	1,007,411	0.34	0.34	1,022,991	0	0.33	1.03
blasted_case40	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_case107	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_case124	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_case207	824	2,128	1.02	1,149,017	0.89	0.98	1,149,017	0	0.86	1.04
blasted_case120	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

tively. Columns Q_n and E_n show the total numbers of satisfying assignments generated by QuickSampler and ESampler+QS, respectively. Column E_{dn} gives the numbers of satisfying assignments generated by our derivation procedure. The last column provides the ratio of execution time per satisfying assignment between QuickSampler and ESampler+QS, depicting the speedup of ESampler+QS. 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+QS may produce different satisfying assignments when our derivation procedure does not work, although ESampler+QS is built on QuickSampler.

Summary. ESampler+QS and QuickSampler respectively failed on 11 and 7 benchmarks due to the failures 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+QS determined that the derivation procedure can generate a large number of satisfying assignments, ESampler+QS performed better than QuickSampler on almost all the benchmarks. While ESampler+QS determined that our derivation procedure was not able to generate a large number of satisfying assignments, ESampler+QS was comparable to QuickSampler. Specifically, ESampler+QS was faster than QuickSampler on 227 benchmarks. It was 1.66× faster on average and more than 5× faster on 41 benchmarks, while it was 1.2 times slower on 16 benchmarks.

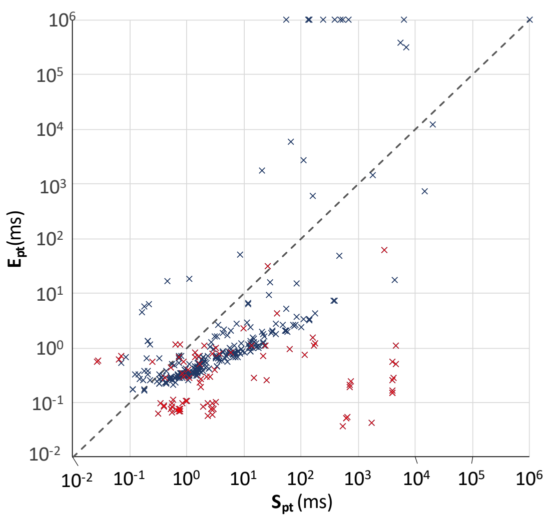


Figure 3: ESampler+QS vs. STS

5.2. *Comparison of* ESampler+*QS and* STS

Figure 3 shows the scatter plot comparing the average execution time per satisfying assignment between ESampler+QS and STS on all the 364 formulas. Recall that timeout occurred along the top or right border, the red color

Table 2: Comparison of STS and ESampler+QS

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

indicates that `Derivable` is set true by Algorithm 2, and points below the diagonal line indicate that ESampler+QS performs better than QuickSampler, and vice versa.

Table 2 reports the performance of STS and ESampler+QS 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+QS, depicting the speedup of ESampler+QS.

Summary. STS failed on 1 benchmark because the underlying SAT solver Minisat failed to solve the Boolean formula, while ESampler+QS failed on 11 benchmarks. In general, ESampler+QS 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 it was 1.2 times slower on only 45 benchmarks.

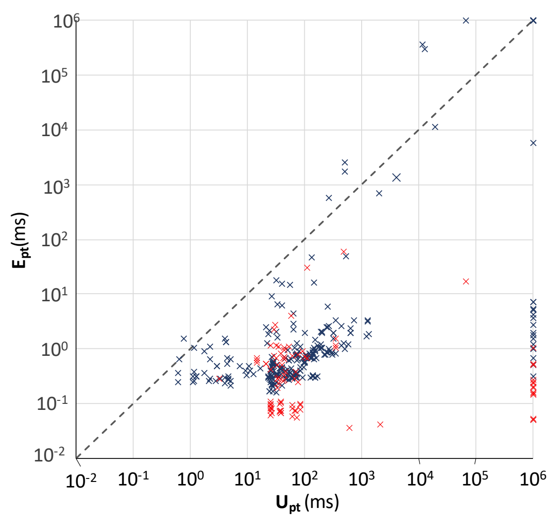


Figure 4: ESampler+QS vs. UniGen3

5.3. *Comparison of* ESampler+*QS and* UniGen3

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

Table 3 reports the performance of UniGen3 and ESampler+QS 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+QS, depicting the speedup of ESampler+QS.

Table 3: Comparison of UniGen3 and ESampler+QS

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	38.57
blasted_case.54	203	725	3.60	158,168	22.76	0.20	664,548	0	0.30	75.87
20.sk_1_51	15,475	60,994	3.60	70,312	51.21	1.67	1,520,152	∼1,520k	1.10	46.56
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	141.4
blasted_case.40	245	650	3.60	50,380	71.46	0.41	1,149,017	0	0.36	198.5
s349_3_2	198	469	3.60	144,279	24.95	0.07	1,142,757	∼1,088k	0.06	415.8
56.sk_6_38	4,842	17,828	3.60	104,149	34.57	1.18	1,093,080	∼1,092k	1.08	32.01
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	52.17
s420_new_7_4	312	770	3.60	98,934	36.39	0.08	1,048,576	∼1,043k	0.07	519.9
blasted_case.124	133	386	3.60	89,376	40.28	0.22	1,008,715	0	0.22	182.1
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	279
blasted_case.120	284	851	3.60	51,799	69.5	0.40	1,044,731	0	0.38	182.9
63.sk_3_64	7,242	24,379	3.60	48,004	75.01	0.30	1,200,120	∼1,200k	0.25	300
s420_7_4	312	770	3.60	95,260	37.79	0.10	1,366,784	∼1,363k	0.07	539.9

Summary. UniGen3 failed on 40 benchmarks. Recall that ESampler+QS failed on 11 benchmarks. No matter whether or not ESampler+QS determined that the derivation procedure was able to generate a large number of satisfying assignments, ESampler+QS performed significantly better than UniGen3 on almost all the benchmarks. Specifically, ESampler+QS was faster than Uni-

Gen3 on 348 benchmarks. It was 69.8× faster on average and more than 100× faster on 194 benchmarks, while it was 1.2 times slower on only 7 benchmarks.

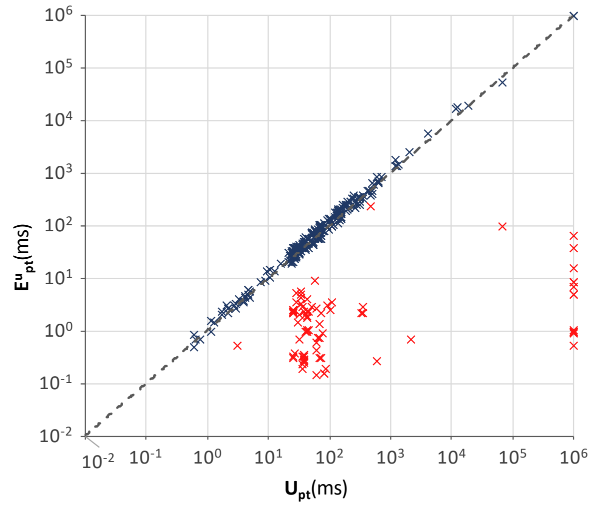


Figure 5: ESampler+UG vs. UniGen3

5.4. Comparison of ESampler+UG *and* UniGen3

Figure 5 shows the scatter plot comparing the average execution time per satisfying assignment between UniGen3 and ESampler+UG on all the 364 formulas. Almost all the red points are below the diagonal line while almost all the blue points are close to the diagonal line. indicating that ESampler+UG significantly outperforms UniGen3 on the benchmarks on which `Derivable` was set to true by Algorithm 2, while it was still comparable on other benchmarks.

Table 4 reports the performance of UniGen3 and ESampler+UG on the same representative subset of benchmarks. Column E^u_t gives the total execution time in thousand seconds (ks) of ESampler+UG, while column E^u_{pt} gives the execution time per satisfying assignment in milliseconds (ms) . Column E^u_n shows the total number of satisfying assignments generated by ESampler+UG for each Boolean formula, while column E^u_{dn} gives the numbers of satisfying assignments generated by the derivation procedure. The last column provides the ratio of execution time per satisfying assignment between UniGen3 and ESampler+UG, depicting the speedup of our algorithm.

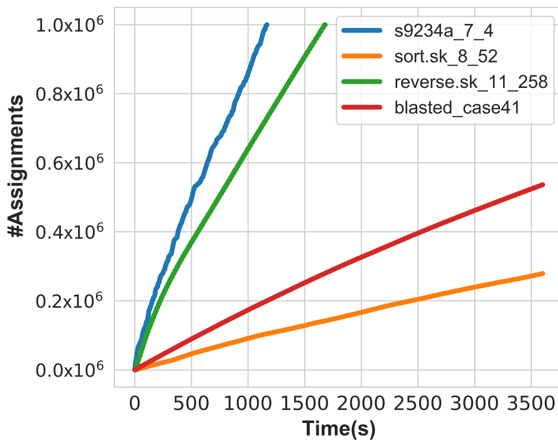
Summary. UniGen3 failed on 40 benchmarks while ESampler+UG failed on 25 benchmarks. Specifically, ESampler+UG was faster than UniGen3 on 207 benchmarks. It was 2.19× faster on average and more than 10× faster on 85 benchmarks, while there were only 12 benchmarks on which it was at least 1.2× slower. The results demonstrate the generic nature of Algorithm 1 for deriving satisfying assignments from a seed using UniGen3 as the underlying seed generator in our tool ESampler.

Table 4: Comparison of UniGen3 and ESampler+UG

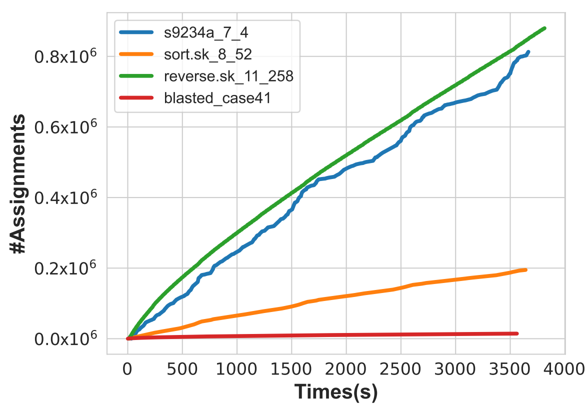
Benchmark	#Vars	#Cls	$U_t(ks)$	U_n	$U_{pt}(ms)$	$E^u_t(ks)$	E^u_n	E^u_{dn}	$E^u_{pt}(ms)$	$\frac{U_{pt}}{E^u_{pt}}$
s27_new_15_7	17	43	0.00	48	0.02	0.00	48	42	0.01	2.20
blasted_case_54	203	725	3.60	158,168	22.8	3.60	197,693	0	18.21	1.25
20_sk_1_51	15,475	60,994	3.60	70,312	51.2	3.87	1,320,000	~1,319k	2.93	17.46
s35932_7_4	17,849	44,425	3.60	0	-	4.03	60,005	~59k	67.16	-
blasted_case_126	302	1,129	3.60	77,185	46.7	3.60	74,111	0	56.01	0.96
blasted_case_40	245	650	3.60	50,380	71.46	3.60	46,327	0	48.59	0.92
s349_3_2	198	469	3.60	144,279	24.95	2.42	1,001,811	~974k	2.41	10.35
56_sk_6_38	4,842	17,828	3.60	104,149	34.57	3.82	1,194,765	~1,194k	3.2	10.81
blasted_case_107	618	1,661	3.60	0	-	3.60	0	0	-	-
s832a_15_7	693	2,017	3.60	132,705	27.13	3.60	130,075	3	33.4	0.98
s420_new_7_4	312	770	3.60	98934	36.39	0.26	1,086,720	~1,083k	0.236	154.47
blasted_case_124	133	386	3.60	89,376	40.27	3.60	94,080	0	38.27	1.05
s35932_15_7	17,918	44,709	3.60	55	65455	3.66	230,032	~229k	15.92	4109
blasted_case_207	824	2,128	3.61	15,026	239.9	3.61	15,054	0	239.9	1.00
blasted_case_120	284	851	3.60	51,799	69.5	3.60	64,745	0	55.6	1.25
63_sk_3_64	7,242	24,379	3.60	48,004	75.02	3.70	3,960,000	~3,959k	0.934	80.39
s420_7_4	312	770	3.60	95,260	37.79	0.32	1,096,960	~1,093k	0.292	129.95

5.5. 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 6(a) and Figure 6(b) respectively show the plots of results on the



(a) ESampler+QS



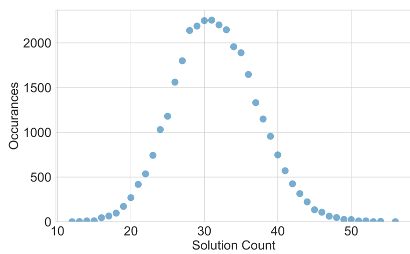
(b) ESampler+UG

Figure 6: Time vs. #assignments of ESampler

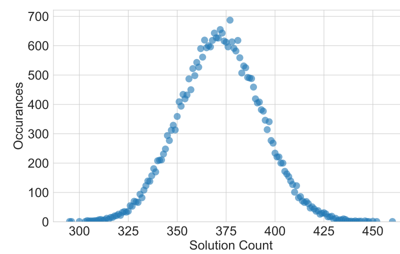
four randomly chosen benchmarks using ESampler+QS and ESampler+UG, where the x-axis is the execution time (in seconds) and the y-axis is number of satisfying assignments (#assignments). We can observe that the number of satisfying assignments for each benchmark is almost linear in the execution time. These results demonstrate the effectiveness of our derivation procedure.

5.6. Testing Uniformity

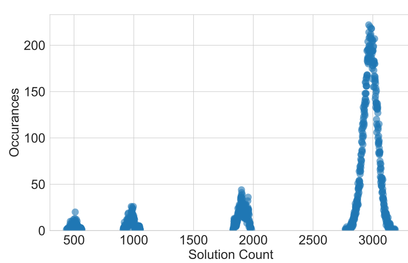
Similar to QuickSampler, ESampler does not provide a guarantee of uniformity. Remark that UniGen3 provides a theoretical guarantee of uniformity based on hashing, at the cost of sampling efficiency. We empirically show that the uniformity of the solutions can be controlled by adjusting the maximal number of solutions per seed, i.e., the parameter `MaxNumPerSeed`. We run both ESampler+QS and ESampler+UG 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.



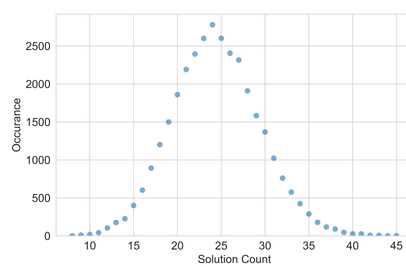
(a) ESampler+QS, `MaxNumPerSeed`=0



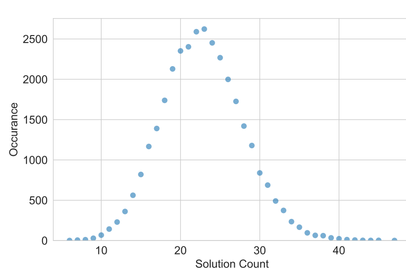
(b) ESampler+QS, `MaxNumPerSeed`=10



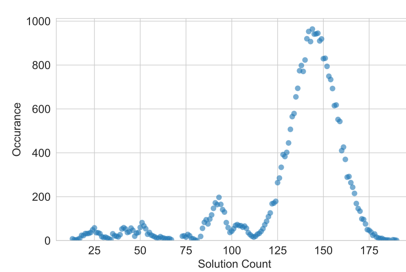
(c) ESampler+QS, `MaxNumPerSeed`=100



(d) ESampler+UG, `MaxNumPerSeed`=0



(e) ESampler+UG, `MaxNumPerSeed`=10



(f) ESampler+UG, `MaxNumPerSeed`=100

Figure 7: Distributions of solutions

Figure 7 depicts the distributions of solutions ESampler+QS and ESampler+UG when `MaxNumPerSeed` is set to 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 when `MaxNumPerSeed` is chosen properly.

6. Application to Bayesian Inference

In this section, to further show the effectiveness and efficiency of ESampler in real-world applications, we apply ESampler to Bayesian inference, namely, computing the posterior probability of a query given evidence in a Bayesian network [52, 53].

Bayesian inference. A *Bayesian network* is a tuple (V, E, T), where $V = \{X_1, \cdots, X_n\}$ is a finite set of nodes each of which represents a discrete random variable, $E \subseteq V \times V$ is a finite set of edges each of which represents dependence between two random variables. (V, E) forms a directed acyclic graph (DAG), and T is a finite set of conditional probability tables (CPTs) each of which encodes the conditional probability distribution of a random variable. Given a random variable X, value a, and a partial assignment v of some other random variables, the *Bayesian inference* is to compute the posterior probability $\Pr(X = a \mid v)$.

Bayesian inference is a well-known **#P**-complete problem [54]. Sang *et al.* [31] proposed an encoding from the Bayesian inference problem to the model-counting problem of Boolean formulas (#SAT), which we leverage to solve Bayesian inference.

Bayesian inference to #SAT. Given a Bayesian network (V, E, T) and a Bayesian inference query $\Pr(X = a \mid v)$, Sang *et al.* [31] use *chance* variables to encode entries in CPTs and *state* variables to encode the values of the nodes, based on which two Boolean formulas $\Phi_1 \wedge \Phi_2 \wedge \Phi_3$ and $\Phi_1 \wedge \Phi_2$ can be constructed, where Φ_1 encodes the Bayesian network (V, E, T), Φ_2 encodes the partial assignment v and Φ_3 encodes $X = a$. With satisfying assignments of $\Phi_1 \wedge \Phi_2 \wedge \Phi_3$ and $\Phi_1 \wedge \Phi_2$, one can calculate (or approximate) the posterior probability in the Bayesian network. Suppose n_1 (resp. n_2) denotes the number of the discovered satisfying assignments of $\Phi_1 \wedge \Phi_2 \wedge \Phi_3$ (resp. $\Phi_1 \wedge \Phi_2$), the approximate posterior probability is $\frac{n_1}{n_2}$. Furthermore, if n_1 is exact, then $\frac{n_1}{n_2}$ gives an upper-bound of the posterior probability; on the other hand, if n_2 is exact, then $\frac{n_1}{n_2}$ gives a lower-bound.

Performance of Bayesian inference. To evaluate ESampler for Bayesian inference, we use the QuickSampler sampler as the seed generation engine to solve Bayesian inference of the plan recognition problems provided by [31]. There are 11 plan recognition problems given as Bayesian networks on which we compute the posterior probability for each random variable, resulting in 11,326 Bayesian inference queries. For each query, we sample satisfying assignments until the 10 recently generated assignments already exist. Solved by ESampler in 65,122 seconds, the calculated probabilities of variables are shown in Table 5, where the columns (Var ID) show the indices of the random variables, and the columns (Prob) show the calculated posterior probabilities of the random variable. For the sake of brevity, we show first hundred variables in problem tire-3,

Table 5: Calculated probabilities of variables in problem tire-3.

Var ID	Prob	Var	Prob	Var ID	Prob	Var ID	Prob
5	0.9391	43	0.0335	64	0.4827	83	0.0000
12	0.0000	44	0.2407	66	0.0000	85	0.5720
19	0.0000	45	0.0000	67	0.0000	86	0.0000
20	0.6307	50	0.6962	68	0.2452	87	0.0000
21	0.0000	51	0.0000	69	0.0000	88	0.9085
25	0.8177	52	0.5173	71	0.0000	89	0.0000
26	0.4827	53	0.9645	72	0.2470	90	0.2159
27	0.9997	54	0.2358	74	0.1287	91	0.5366
31	0.9997	55	0.7548	75	0.5021	92	0.1862
33	0.5172	56	0.1253	76	0.2502	94	0.2452
35	0.0000	58	0.9837	77	0.0000	95	0.9342
36	0.0000	59	0.9977	78	0.0499	96	0.9903
38	0.5028	61	0.2670	79	0.1691	98	0.2981
40	0.3693	62	0.0000	80	0.0253	···	···
41	0.4972	63	0.0000	82	0.0000		

and the results are omitted if the posterior probability is 1. We notice that the reported probabilities in Table 5 are approximation of the exact posterior probabilities when the sampler fails to generate all the possible satisfying assignments of a Bayesian inference query. Remark that computing exact posterior probabilities are computational hard (**#P**-completeness).

Comparison of samplers on Bayesian inference. To compare the efficiency of ESampler (i.e., ESampler+QuickSampler), STS, QuickSampler and UniGen3 in solving Bayesian inference problems, we test them on 11 randomly chosen formulas from the plan recognition problems, each of which is aimed to compute 100,000 satisfying assignments within 10 minutes. The other settings are the same as in Section 5.

The results are reported in Table 6, where the last three columns provide the ratio of the execution time per satisfying assignment for QuickSampler, STS, UniGen3 to ESampler respectively, measuring the speedup of ESampler. All the samplers are able to generate satisfying assignments except that UniGen3 failed on 4 benchmarks (log-1, log-4, log-5 and tire-1). For the sake of brevity, we only report the number of satisfying assignments generated by ESampler. We can observe that ESampler outperforms the other three samples on Bayesian inference. On average, ESampler is 13.8, 18.7 and 556.3 times faster than QuickSampler, STS and UniGen3, respectively.

7. Conclusion

We have proposed a novel approach to derive 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 samplers QuickSampler and UniGen3

Table 6: Comparison of ESampler, QuickSampler, STS and UniGen3 on benchmarks derived from Bayesian inferences of plan recognition problems

Benchmark	#Vars	#Cls	$E_t(s)$	E_n	E_{dn}	$E_{pn}(ms)$	$\frac{Q_{pt}}{E_{pt}}$	$\frac{S_{pt}}{E_{pt}}$	$\frac{U_{pt}}{E_{pt}}$
4step	165	418	17.8	66,935	0	0.27	1.01	1.78	36.47
5step	177	475	3.01	80,033	∼80k	0.04	7.69	11.52	273.8
log-1	939	3,785	10.4	160,016	160k	0.07	16.31	72.99	-
log-2	1,377	24,777	53.9	110,011	110k	0.49	12.67	50.01	251.8
log-3	1,413	29,487	166	170,017	170k	0.98	4.78	24.14	511.3
log-4	2,303	20,963	18.2	120,012	120k	0.15	32.74	423.8	-
log-5	2,701	29,534	958	10,001	100k	95.81	4,355	2.99	-
tire-1	352	1,038	9.4	130,347	∼130k	0.07	6.81	17.01	-
tire-2	550	2,001	15.8	160,016	160k	0.10	6.63	10.72	731.7
tire-3	578	2,004	13.7	140,014	140k	0.10	11.31	16.65	3,496
tire-4	812	3,222	20.1	120,012	120k	0.17	5.92	13.64	5,009

as the seed generator. The extensive experiments on publicly available benchmarks and application on Bayesian inference confirmed the effectiveness and efficiency of our approach.

In future, we plan to further improve the performance of the tool ESampler and extend our derivation approach to SMT formulas, as well as their practical applications.

Acknowledgement

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 (EFFECT).

References

[1] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, Vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2009.

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

[3] F. Bacchus, S. Dalmao, T. Pitassi, Algorithms and complexity results for #SAT and bayesian inference, Proceedings of the 44th Symposium on Foundations of Computer Science, 11-14 October 2003, Cambridge, MA, USA, 2003, pp. 340–351.

[4] D. Roth, On the hardness of approximate reasoning, Artificial Intelligence 82 (1-2) (1996) 273–302.

[5] L. G. Valiant, The complexity of enumeration and reliability problems, SIAM Journal on Computing 8 (3) (1979) 410–421.

[6] D. Angluin, On counting problems and the polynomial-time hierarchy, Theoretical Computer Science 12 (1980) 161–173.

[7] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek, Constraint-based random stimuli generation for hardware verification, AI magazine 28 (3) (2007) 13–13.

[8] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek, Constraint-based random stimuli generation for hardware verification, Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference, 2006, pp. 1720–1727.

[9] J. Yuan, A. Aziz, C. Pixley, K. Albin, Simplifying boolean constraint solving for random simulation-vector generation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23 (3) (2004) 412–420.

[10] E. Guralnik, M. Aharoni, A. J. Birnbaum, A. Koyfman, Simulation-based verification of floating-point division, IEEE Transations on Computers 60 (2) (2011) 176–188.

[11] K. Vorobyov, P. Krishnan, Combining static analysis and constraint solving for automatic test case generation, Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation, 2012, pp. 915–920.

[12] R. Naveh, A. Metodi, Beyond feasibility: CP usage in constrained-random functional hardware verification, Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, Springer, 2013, pp. 823–831.

[13] Y. Zhao, J. Bian, S. Deng, Z. Kong, Random stimulus generation with self-tuning, Proceedings of the 13th International Conference on Computers Supported Cooperative Work in Design, IEEE, 2009, pp. 62–65.

[14] Y. Zhang, M. Zhang, G. Pu, F. Song, J. Li, Towards backbone computing: A greedy-whitening based approach, AI Communications 31 (3) (2018) 267–280.

[15] Y. Zhang, J. Li, M. Zhang, G. Pu, F. Song, Optimizing backbone filtering, in: Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering, 2017, pp. 1–8.

[16] Z. Zhao, G. Chen, J. Wang, Y. Yang, F. Song, J. Sun, Attack as defense: characterizing adversarial examples using robustness, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021, pp. 42–55.

[17] G. Chen, Z. Zhao, F. Song, S. Chen, L. Fan, Y. Liu, SEC4SR: A security analysis platform for speaker recognition, CoRR abs/2109.01766.

[18] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks, Proceedings of the Annual Conference on Neural Information Processing Systems, 2016, pp. 4107–4115.

[19] Y. Zhang, Z. Zhao, G. Chen, F. Song, T. Chen, BDD4BNN: A BDD-based quantitative analysis framework for binarized neural networks, in: Proceedings of the 33rd International Conference on Computer Aided Verification, 2021, pp. 175–200.

[20] S. Korneev, N. Narodytska, L. Pulina, A. Tacchella, N. Bjørner, M. Sagiv, Constrained image generation using binarized neural networks with decision procedures, Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing, 2018, pp. 438–449.

[21] N. Narodytska, Formal analysis of deep binarized neural networks, Proceedings of the 27th International Joint Conference on Artificial Intelligence, 2018, pp. 5692–5696.

[22] S. A. Cook, The complexity of theorem-proving procedures, Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 1971, pp. 151–158.

[23] J. P. M. Silva, K. A. Sakallah, Grasp-a new search algorithm for satisfiability, The Best of ICCAD, Springer, 2003, pp. 73–89.

[24] J. P. M. Silva, K. A. Sakallah, Grasp: A search algorithm for propositional satisfiability, IEEE Transactions on Computers 48 (5) (1999) 506–521.

[25] R. J. B. Jr., R. C. Schrag, 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, 1997, pp. 203–208.

[26] M. Soos, K. Nohl, C. Castelluccia, Extending SAT solvers to cryptographic problems, Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, 2009, pp. 244–257.

[27] N. Kitchen, A. Kuehlmann, Stimulus generation for constrained random simulation, Proceedings of the 2007 International Conference on Computer-Aided Design, 2007, pp. 258–265.

[28] R. Dutra, K. Laeufer, J. Bachrach, K. Sen, Efficient sampling of SAT solutions for testing, Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 549–559.

[29] M. Soos, S. Gocht, K. S. Meel, Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling, Proceedings of the 32nd International Conference on Computer Aided Verification, 2020, pp. 463–484.

[30] S. Chakraborty, K. S. Meel, M. Y. Vardi, Balancing scalability and uniformity in SAT witness generator, Proceedings of the 51st Annual Design Automation Conference, 2014, pp. 60:1–60:6.

[31] T. Sang, P. Beame, H. A. Kautz, Performing bayesian inference by weighted model counting, in: Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, 2005, pp. 475–482.

[32] S. Ermon, C. P. Gomes, B. Selman, Uniform solution sampling using a constraint solver as an oracle, Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, 2012, pp. 255–264.

[33] K. S. Meel, Constrained counting and sampling: Bridging the gap between theory and practice, CoRR abs/1806.02239.

[34] N. Kitchen, Markov chain monte carlo stimulus generation for constrained random simulation, Ph.D. thesis, University of California, Berkeley, USA (2010).

[35] W. Wei, B. Selman, A new approach to model counting, Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, 2005, pp. 324–339.

[36] J. H. Kukula, T. R. Shiple, Building circuits from relations, Proceedings of the 12th International Conference on Computer Aided Verification, 2000, pp. 113–123.

[37] W. Wei, J. Erenrich, B. Selman, Towards efficient sampling: Exploiting random walk strategies, Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence, 2004, pp. 670–676.

[38] M. Sipser, A complexity theoretic approach to randomness, Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 330–335.

[39] M. Bellare, O. Goldreich, E. Petrank, Uniform generation of np-witnesses using an np-oracle, Inf. Comput. 163 (2) (2000) 510–526.

[40] C. P. Gomes, A. Sabharwal, B. Selman, Near-uniform sampling of combinatorial spaces using XOR constraints, Proceedings of the 2th Annual Conference on Neural Information Processing Systems, 2006, pp. 481–488.

[41] S. Chakraborty, K. S. Meel, M. Y. Vardi, A scalable and nearly uniform generator of SAT witnesses, Proceedings of the 25th International Conference on Computer Aided Verification, 2013, pp. 608–623.

[42] S. Ermon, C. P. Gomes, A. Sabharwal, B. Selman, Embed and project: Discrete sampling with universal hashing, Proceedings of the 27th Annual Conference on Neural Information Processing Systems, 2013, pp. 2085–2093.

[43] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, M. Y. Vardi, On parallel scalable uniform SAT witness generation, Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the European Joint Conferences on Theory and Practice of Software, Springer, 2015, pp. 304–319.

[44] N. Bjørner, A. Phan, νz - maximal satisfaction with Z3, Proceedings of the 6th International Symposium on Symbolic Computation in Software Science, 2014, pp. 1–9.

[45] N. Sörensson, N. Eén, MiniSat: A SAT solver with conflict-clause minimization, Solver Description.

[46] D. Achlioptas, Z. S. Hammoudeh, P. Theodoropoulos, Fast sampling of perfectly uniform satisfying assignments, Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing, Springer, 2018, pp. 135–147.

[47] Y. Xu, F. Song, T. Chen, Esampler: Efficient sampling of satisfying assignments for boolean formulas, in: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications, Springer, 2021, pp. 279–298.

[48] A. Ivrii, S. Malik, K. S. Meel, M. Y. Vardi, On computing minimal independent support and its applications to sampling and counting, Constraints 21 (1) (2016) 41–58.

[49] C. H. Papadimitriou, Computational complexity, Addison-Wesley, 1994.

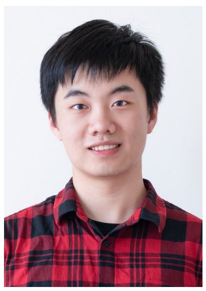
[50] DIMACS, Clique and coloring problems graph format, `http://archive.dimacs.rutgers.edu/pub/challenge/graph/doc/ccformat.tex` Accessed September 16, 2021 (1993).

[51] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 337–340.

[52] F. Y. Bois, Bayesian inference, Computational toxicology (2013) 597–636.

[53] D. Heckerman, A tutorial on learning with bayesian networks, in: Learning in Graphical Models, 1998, pp. 301–354.

[54] D. Roth, On the hardness of approximate reasoning, Artificial Intelligence 82 (1-2) (1996) 273–302.



Yongjie Xu is a M.S. student in ShanghaiTech University, supervised by Prof. Fu Song. He received the B.S. degree in Computer Science from ShanghaiTech University in 2019. His research interests are in SAT solving, program analysis and AI security.

Fu Song received the B.S. degree from Ningbo University, Ningbo, China, in 2006, the M.S. degree from East China Normal University, Shanghai, China, in 2009, and the Ph.D. degree in computer science from University Paris-Diderot, Paris, France, in 2013. From 2013 to 2016, he was a Lecturer and Associate Research Professor at East China Normal University. From August 2016 to July 2021, he is an Assistant Professor with ShanghaiTech University, Shanghai, China. Since July 2021, he is an Associate Professor with ShanghaiTech University. His research interests include formal methods and computer/AI security.

Taolue Chen received the B.S. and M.S. degrees from the Nanjing University, China, both in Computer Science. He was a junior researcher at the Centrum Wiskunde & Informatica (CWI) and acquired the Ph.D. degree from the Vrije Universiteit Amsterdam, The Netherlands. He is currently a lecturer at the Department of Computer Science and Information Systems, Birkbeck, University of London, United Kingdom. His research interests include formal verification and synthesis, program analysis, software security, software engineering and machine learning.