Quantifier Elimination in Stochastic Boolean Satisfiability

Hao-Ren Wang
Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan

Kuan-Hua Tu
Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan

Jie-Hong Roland Jiang
Graduate Institute of Electronics Engineering / Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

Christoph Scholl
Department of Computer Science, Universität Freiburg, Germany

Abstract

Stochastic Boolean Satisfiability (SSAT) generalizes quantified Boolean formulas (QBFs) by allowing quantification over random variables. Its generality makes SSAT powerful to model decision or optimization problems under uncertainty. On the other hand, the generalization complicates the computation in its counting nature. In this work, we address the following two questions: 1) Is there an analogy of quantifier elimination in SSAT, similar to QBF? 2) If quantifier elimination is possible for SSAT, can it be effective for SSAT solving? We answer them affirmatively, and develop an SSAT decision procedure based on quantifier elimination. Experimental results demonstrate the unique benefits of the new method compared to the state-of-the-art solvers.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Constraint and logic programming

Keywords and phrases Stochastic Boolean Satisfiability, Quantifier Elimination, Decision Procedure

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.23

Supplementary Material *Software (Repository)*: `https://github.com/NTU-ALComLab/elimssat`
archived at `swh:1:rev:076babed38a25f78805750192e9a54eba6ce9ef6`

Funding This work was supported in part by the Ministry of Science and Technology of Taiwan under grants MOST 108-2221-E-002-144-MY3 and MOST 110-2224-E-002-011.

1 Introduction

Stochastic Boolean satisfiability (SSAT) [27] generalizes the satisfiability of the well-known quantified Boolean formula (QBF). In addition to the standard exist-quantifiers, a variable in SSAT can be specified as a random variable with a probability for it to be valuated to TRUE through the *random-quantifier*. The SSAT formulas provide a convenient language to encode decision or optimization problems under uncertainty. Various applications have been studied, e.g., probabilistic planning [25], trust management [9], belief network inference [25], probabilistic design verification [20], fairness evaluation of machine learning models [10], solving *partially observable Markov decision processes* (POMDPs) [31], etc. Although the decision version of SSAT is PSPACE-complete, the same computational complexity as QBF, it is considered more challenging than pure propositional reasoning due to its intrinsic characteristics of counting.

Despite their broad applications, SSAT solvers are much underdeveloped compared to QBF solvers. Among the existing SSAT solvers, there are special-form solvers, such as `reSSAT` [22] for solving random-exist quantified SSAT formulas, `ComPlan` [13], `maxcount` [8] and `erSSAT` [23] for solving the exist-random quantified SSAT formulas, a.k.a., the E-


© Hao-Ren Wang, Kuan-Hua Tu, Jie-Hong Roland Jiang, and Christoph Scholl;
licensed under Creative Commons License CC-BY 4.0

MAJSAT problem [24]. There are also general-form solvers, which impose no restriction on the quantification levels, such as `DC-SSAT` [26] and `ClauSSAT` [6]. In this work, we are primarily concerned with solving general SSAT formulas.

The existing SSAT solvers exploit various techniques from SAT, QBF and knowledge compilation domains. E.g., DPLL SAT search is adopted in [26], QBF clause selection [14] is applied in [6], knowledge compilation is used [13]. To the best of our knowledge, quantifier elimination in SSAT has not been studied previously. Quantifier elimination is a common technique in QBF rewriting and preprocessing. Essentially, the quantifiers of a QBF can be eliminated via formula expansion or formula composition [15]. `AIGSolve` [29] is a QBF solver relying on quantifier elimination. However, extending quantifier elimination to SSAT is not as trivial because of the random-quantifiers. In this work, we present a framework to perform quantifier elimination in SSAT. Through the proposed quantifier elimination, an SSAT formula with an arbitrary number of quantification levels can be rewritten into a quantifier-free formula such that the satisfying probability of the SSAT formula can be derived by a linear-time model counting on the quantifier-free formula. We further develop an SSAT solver, named `ElimSSAT`, based on quantifier elimination. Experiments on a variety of benchmarks demonstrate the strength of `ElimSSAT` compared to the state-of-the-art solvers.

The rest of the paper is organized as follows. Section 2 first provides the essential preliminaries. After the motivation and intuition are explained in Section 3, the quantifier elimination in SSAT is formulated in Section 4. The overall algorithm of quantifier elimination for SSAT solving is presented in Section 5. Section 6 discusses implementation issues and enhancements. The experimental evaluation is performed in Section 7, and conclusions are drawn in Section 8.

2 Preliminaries

As notational convention, the Boolean domain $\mathbb{B} = \{\top, \bot\}$, or $\{1, 0\}$, where $\top$, or 1, and $\bot$, or 0, denote Boolean values TRUE and FALSE, respectively. Boolean connectives are denoted with "$\wedge$" (sometimes omitted) for conjunction, "$\vee$" for disjunction, "$\rightarrow$" for implication, "$\leftrightarrow$" for biconditional, and "$\neg$" for negation. A *literal* is a variable or the negation of a variable. A *clause* is a disjunction of literals; a *cube* is a conjunction of literals. A *conjunctive normal form* (CNF) formula is a conjunction of clauses. As a Boolean formula uniquely determines a Boolean function, in this work we refer to Boolean formulas and Boolean functions interchangeably.

An *assignment* α over a set of variables X is a mapping from X to $\mathbb{B}$ and alternatively represented as a cube, e.g., the assignment $x_1 = 1, x_2 = 0$ is alternatively represented as $x_1 \neg x_2$. The set of all assignments over X is denoted as $[\![X]\!]$. An assignment is *full* if every $x \in X$ is mapped to some Boolean value; otherwise, it is *partial*. The value of variable $x \in X$ in an assignment α is denoted by $\alpha(x)$. Given a formula φ and a full assignment α over variables X, let $\varphi[\alpha]$ denote the valuation of φ by substituting every occurrence of variables $x \in X$ in φ with $\alpha(x)$. For a full assignment α, if $\varphi[\alpha] = \top$, then α is a *satisfying assignment* of φ. A satisfying assignment of a given Boolean formula is often referred to as a *model*. A Boolean formula is *satisfiable* if it has a model; otherwise, it is *unsatisfiable*. Given a formula φ and a (full or partial) assignment α, substituting every $x \in X$ with $\alpha(x)$ in φ yields a new formula, denoted as $\varphi|_\alpha$, which is referred to as the *cofactor* of φ with respect to α.

2.1 Model Counting

Given a Boolean formula (often in CNF) φ over variable X, the *model counting* problem asks to count the number, denoted $\#\varphi$, of solutions of φ.

▶ **Definition 1.** *Two Boolean formulas/functions are called* counting equivalent *if they possess the same number of satisfying assignments.*

Two extended variants of model counting, namely, *weighted model counting* and *projected model counting*, are closely related to SSAT. The former allows different solutions being counted differently by imposing different weights on different variables. The latter permits counting the number of different solutions with respect to a specified subset of variables.

2.2 Stochastic Boolean Satisfiability

A *stochastic Boolean satisfiability* (SSAT) formula can be expressed in the prenex form as

$$\Phi = Q_1 X_1, \ldots, Q_n X_n . \varphi, \tag{1}$$

where quantifier $Q_i \in \{\exists, \mathrm{R}\}$, $Q_i \neq Q_{i+1}$, variable set $X_i \neq \emptyset$, and φ is a quantifier-free Boolean formula. The quantifier part is referred to as the *prefix* and the Boolean formula is referred to as the *matrix*, which is commonly expressed in CNF. In addition to the well-known existential quantifier $\exists$, the random quantifier R^{p_i} on variable x_i indicates $x_i = \top$ (resp. $\bot$) with the probability $p_i \in [0, 1]$ (resp. $1 - p_i$). For variable $x \in X_i$, the *quantification level* of x equals i. The formula Φ of Eq. (1) is said to have n quantification levels.

For the semantics of an SSAT formula Φ, it is interpreted with the satisfying probability computed by the following rules. Let x be the outermost quantified variable in Φ.

1. $\Pr[\top] = 1$,
2. $\Pr[\bot] = 0$,
3. $\Pr[\Phi] = \max\{\Pr[\Phi|_{\neg x}], \Pr[\Phi|_x]\}$, for x being existentially quantified, and
4. $\Pr[\Phi] = (1 - p)\Pr[\Phi|_{\neg x}] + p \Pr[\Phi|_x]$, for x being randomly quantified with probability p.

Note that the cofactor $\Phi|_x$ on an SSAT formula Φ with prefix π and matrix φ corresponds to a new SSAT formula Φ' with prefix π' same as π expect for the quantifier of x being removed and matrix $\varphi' = \varphi|_x$.

We remark that an SSAT formula Φ in which the probabilities of random quantifiers represented in binary fractions can be converted in linear time to a new SSAT formula Φ' such that $\Pr[\Phi'] = \Pr[\Phi]$ and all random quantifiers are specified with probability 0.5. This conversion can be done similar to converting a weighted model counting problem into an unweighted model counting problem as was done in [20]. E.g.,

$$\begin{aligned}&\exists x_1, \mathrm{R}^{0.75} x_2, \exists x_3, \mathrm{R}^{0.25} x_4 . \varphi \\ &= \exists x_1, \mathrm{R}^{0.5} y_2, z_2, \exists x_2, x_3, \mathrm{R}^{0.5} y_4, z_4, \exists x_4 . \varphi \wedge (\neg x_2 \leftrightarrow (y_2 \wedge z_2)) \wedge (x_4 \leftrightarrow (y_4 \wedge z_4))\end{aligned}$$

Although an SSAT formula with irrational probability values cannot be precisely converted into one with a probability value of 0.5 only, the conversion can be approximated within a given precision. In the sequel, unless otherwise stated we assume that the probabilities of random quantifiers have been converted to 0.5 and omit specifying the probability.

SSAT is closely related to model counting. Specifically, for an SSAT formula $\mathrm{R} X . \varphi$ of only one level of random quantifiers with arbitrary probability values, it corresponds to the weighted model counting problem. Furthermore, if the probabilities are normalized to 0.5, it is equivalent to (unweighted) model counting. Also, for an SSAT formula $\mathrm{R} X_1, \exists X_2 . \varphi$ of two levels of random-exist quantifiers, it corresponds to the projected (weighted) model counting problem.

3 Motivation and Intuition

By Rule 4 of the SSAT semantics, when $p = 0.5$, the probability $\Pr[\Phi] = 0.5(\Pr[\Phi|_{\neg x}] + \Pr[\Phi|_x])$. For $\Phi = \text{ꓤ}X.\varphi$ with $X = \{x_1, \ldots, x_n\}$, the probability $\Pr[\Phi]$ is simply $(0.5)^n \cdot \#\varphi$, that is, a model counting problem.

To eliminate random quantifiers of an SSAT formula, one important task is to represent the counting results after quantifier elimination. One approach may be to use binary numbers of $n+1$ bits to represent the $2^n + 1$ possible counting results, i.e., $0, 1, \ldots, 2^n$, for a formula with n variables. However, as an SSAT formula involves not only random quantifiers but also exist quantifiers, the binary number representation may not be ideal to perform the subsequent elimination of exist quantifiers. Rather than representing the model count with a binary number, we devise a mechanism of bookkeeping the counting results using a n-variable Boolean function with the settlement property defined as follows.

▶ **Definition 2.** *A Boolean function f over variables $X = \{x_1, \ldots, x_n\}$ is called* settled *with respect to the variable order $(x_1, \ldots, x_n)$, for x_1 (resp. x_n) being the most (resp. least) significant bit, if the implication $f|_\alpha \to f|_\beta$ holds for any assignments α and β on X with their respective binary coded numbers N_α and N_β satisfying $N_\alpha > N_\beta$.*

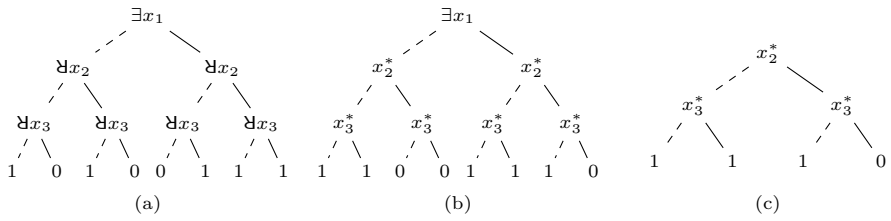

Figure 1 Decision trees under the process of quantifier elimination.

To illustrate, consider the SSAT formula

$$\Phi = \exists x_1, \text{ꓤ}x_2, x_3.\varphi, , \tag{2}$$

for $\varphi = (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$. The decision tree of Φ is shown in Figure 1(a), where the dashed and solid edges correspond to ELSE and THEN branches, respectively, of a decision node.

First, by eliminating the innermost random quantifiers, the two cofactors $\varphi|_{\neg x_1}$ and $\varphi|_{x_1}$ are settled with respect to the variable order (x_2, x_3) while their model counts are preserved. The corresponding decision tree is shown in Figure 1(b). In the sequel, we reuse the names of the random quantified variables and annotate them with superscript "*" for the variables of the settled functions. The resulting SSAT formula becomes

$$\Phi' = \exists x_1.\varphi', \tag{3}$$

for $\varphi' = (x_1 \vee \neg x_2^*) \wedge (\neg x_1 \vee \neg x_2^* \vee \neg x_3^*)$, where x_2^* and x_3^* are free variables.

Second, we eliminate the remaining exist quantifier of variable x_1 from Φ' of Eq. (3). Note that the max operation, i.e., $\max\{\Pr[\Phi'|_{\neg x}], \Pr[\Phi'|_x]\}$, of Rule 3 of the SSAT semantics reduces to the standard disjunction operation, i.e., $\varphi'|_{\neg x} \vee \varphi'|_x$, when the inner random quantifiers has been eliminated and represented with settled functions. It yields the final quantifier-free formula

$$\varphi'' = \neg x_2^* \vee \neg x_3^*, \tag{4}$$

of a settled function, whose decision tree is shown in Figure 1(c).

Lastly, the model count of φ'' in Eq. (4) can be determined by a binary search strategy to find the number N_α of assignment α such that $\varphi''|_\alpha = \top$ and $\varphi''|_\beta = \bot$ for $N_\beta = N_\alpha + 1$. That is, $\alpha = x_2^* \neg x_3^*$ which represents number 3. Therefore, $\Pr[\Phi] = (0.5)^2 \cdot 3$.

4 Quantifier Elimination of SSAT Formula

We elaborate the elimination of random- and exist-quantifiers for SSAT solving.

4.1 Elimination of Random-Quantifier

In the above exposition, one of the most critical issues to be addressed is how to settle a function while maintaining its model count. We eliminate random-quantifiers via function settlement with in-place sorting as follows.

▶ **Problem Statement 1.** *Given a Boolean function $f(X,Y)$ over variable sets X and Y, we are concerned with settling f to f' with respect to Y under a fixed order such that $f|_\alpha$ and $f'|_\alpha$ are counting equivalent for any assignment α on variables X.*

To achieve the stated function settlement, we rely on an implicit approach to sorting the satisfying solutions inside function f. Let $Sort(f,Y)$ be the procedure that sorts in an ascending order of the function values of $f|_{\alpha\wedge\beta}$ for any assignment α on X variables with respect to the binary numbers N_β coded by the assignment β of Y variables.

▶ **Proposition 3.** *Given a function $f(X,Y)$ and an order of Y variables, $Sort(f,Y)$ yields a new function f' such that $f'|_\alpha$ is settled and counting equivalent to $f|_\alpha$ for any assignment α on the X variables.*

We note that $Sort(f,Y)$ can done by iteratively sorting f with respect to the Y variables one at a time in the order of $y_m, y_{m-1}, \ldots, y_1$. For example, the Boolean function f of formula

$$(x \vee y \vee z)(x \vee \neg y \vee \neg z)(\neg x \vee y \vee z)(\neg x \vee y \vee z)(\neg x \vee y \vee \neg z)$$

can be settled with respect to the ordered variables (x,y,z) through three iterations of sorting from z, to y, and then x. The corresponding truth tables in sorting f are shown in Table 1. In Columns 3-5, each block is sorted independently, and the satisfying assignments are moved to the top of each block.

Table 1 Function settlement through sorting.

x	y	z	f	$Sort(f,\{z\})$	$Sort(f,\{y,z\})$	$Sort(f,\{x,y,z\})$
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	0	0	0	1
1	0	0	0	0	1	0
1	0	1	0	0	1	0
1	1	0	1	1	0	0
1	1	1	1	1	0	0

As eliminating random-quantifiers of an SSAT formula correspond to model counting, function settlement provides an effective representation of the counting results.

4.1.1 Function Settlement with Implicit Sorting

To compute $Sort(f, Y)$ for a given function $f(X, Y)$, we resort to an implicit approach based on the data-oblivious sorting algorithm *Bitonic Sort* [2], a well-known sorting network. Before introducing the implicit construction, we first present the explicit counterpart for illustration. In our context, the bitonic sorting network is composed of 1-bit Boolean sorters, whose operations are depicted in Figure 2, where a sorter or comparator (represented by a vertical wire that connects two horizontal wires) has two inputs y_1, y_2 (the left two terminals) and two outputs z_1, z_2 with $z_1 = y_1 \vee y_2$ (the upper-right terminal) and $z_2 = y_1 \wedge y_2$ (the lower-right terminal). To sort the function values of an n-variable Boolean function, a 2^n-input and 2^n-output bitonic sorter is required. We adopt the *normalized bitonic sorting network* [28] for our implementation. Figure 3 shows a normalized bitonic sorter for sorting the function values of a 3-input Boolean function $f(x, y, z)$. After the sorting, the TRUE function values are pushed upward above the FALSE function values while the number of TRUE values at the outputs remains the same as that at the inputs. It, therefore, does achieve the desired properties of function settlement and counting equivalence.

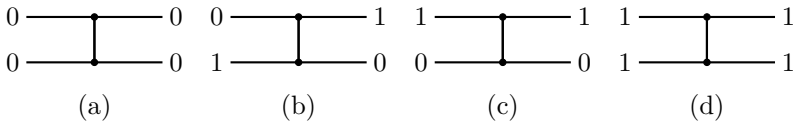

Figure 2 Operations of the 1-bit Boolean sorter.

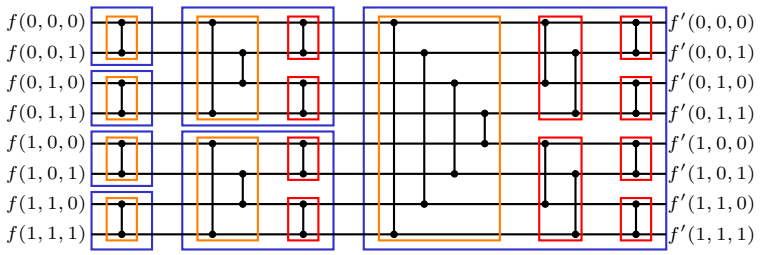

Figure 3 Bitonic sorter for sorting the function values of a 3-input Boolean function.

The above exposition provides an explicit way of sorting the function values of a given function for the purpose of function settlement. However, it is impractical to enumerate the 2^n function values of an n-variable Boolean function and sorting them explicitly. To avoid the exponential blow-up, we rely on implicit sorting instead. In general, an implicit bitonic sorter for sorting f with respect to the ordered variable set $(x_1, \ldots, x_n)$ can be constructed by Algorithm 1 with $\lambda = n$ for the third input argument. By observing the repetitive structure of the normalized bitonic sorting network, we can construct a bitonic sorter using the building blocks marked in orange (resp. red) as shown in Figure 3. We refer to their corresponding operations as the *OrangeBlockTransform* (resp. *RedBlockTransform*) in Algorithm 1. Figures 4 and 5 provide the implicit constructions of *OrangeBlockTransform* and *RedBlockTransform*, respectively. In Figure 4, the orange-block transformation on variable y (resp. x) is shown on the left (resp. right). In Figure 5, the orange-block transformation on variable z (resp. y) is shown on the left (resp. right). The four formulas listed below these four sorters implicitly summarize the corresponding transformations. These formulas in their general form for function $f(x_1, \ldots, x_n)$ can be expressed as follows.

Algorithm 1 $BSort(f, (x_1, \ldots, x_n), \lambda)$.

```
1 for i ← λ to 1 do
2 |   f := OrangeBlockTransform(f, x_i);
3 |   for j ← i + 1 to n do
4 |   |   f := RedBlockTransform(f, x_j);
5 |   end
6 end
7 return f;
```

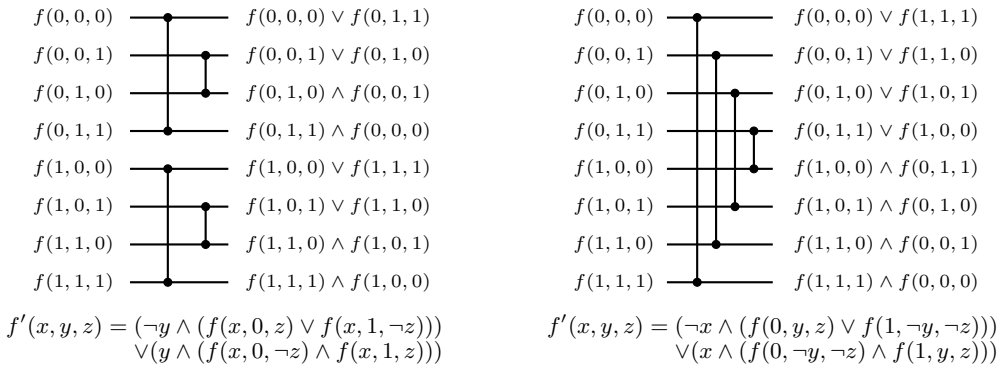

Figure 4 Orange-block transformation on variable y (left) and x (right) of function $f(x, y, z)$.

OrangeBlockTransform(f, x_i) yields $f'(x_1, \ldots, x_n)$ equal to

$$(\neg x_i \wedge (f(x_1, \ldots, x_{i-1}, x_i = 0, x_{i+1}, \ldots, x_n) \vee f(x_1, \ldots, x_{i-1}, x_i = 1, \neg x_{i+1}, \ldots, \neg x_n))) \vee (x_i \wedge (f(x_1, \ldots, x_{i-1}, x_i = 0, \neg x_{i+1}, \ldots, \neg x_n) \wedge f(x_1, \ldots, x_{i-1}, x_i = 1, x_{i+1}, \ldots, x_n))) \quad (5)$$

RedBlockTransform(f, x_i) yields $f'(x_1, \ldots, x_n)$ equal to

$$(\neg x_i \wedge (f(x_1, \ldots, x_{i-1}, x_i = 0, x_{i+1}, \ldots, x_n) \vee f(x_1, \ldots, x_{i-1}, x_i = 1, x_{i+1}, \ldots, x_n))) \vee (x_i \wedge (f(x_1, \ldots, x_{i-1}, x_i = 0, x_{i+1}, \ldots, x_n) \wedge f(x_1, \ldots, x_{i-1}, x_i = 1, x_{i+1}, \ldots, x_n))) \quad (6)$$

▶ **Proposition 4.** *Eq.* (5) *and Eq.* (6) *correctly implement OrangeBlockTransform and RedBlockTransform, respectively.*

Notice that in quantifier elimination for SSAT formulas with multiple quantification levels, we have to resort the variables for inner quantification levels. For example, eliminating the quantifiers of formula $\forall X, \exists Y, \forall Z.f(X, Y, Z)$ corresponds to $Sort(\exists Y.Sort(f, Z), (X, Z^*))$. That is, first eliminate Z by $Sort(f, Z)$ yielding a quantifier-free formula $f'(X, Y, Z^*)$. Then eliminate Y for $\exists Y.f'(X, Y, Z^*)$ yielding a quantifier-free formula $f''(X, Z^*)$. Finally, eliminate X for $\forall X.f''(X, Z^*)$ by $Sort(f'', (X, Z^*))$. Observe that f'' is already partially sorted on variables Z^*. To skip redundant sorting, the argument λ in Algorithm 1 is herein used to explicitly specify the starting point for the loop in line 1.

For implicit sorting for function f over n variables, Algorithm 1 requires $O(n^2)$ transformations. Because each transformation has four appearances of f, in the worse case the formula or circuit of f may grow four times after each transformation. Therefore, the formula may grow in $O(4^{n^2})$. However, in practice the four copies of f share a significant amount of structural similarity and often can be simplified substantially.

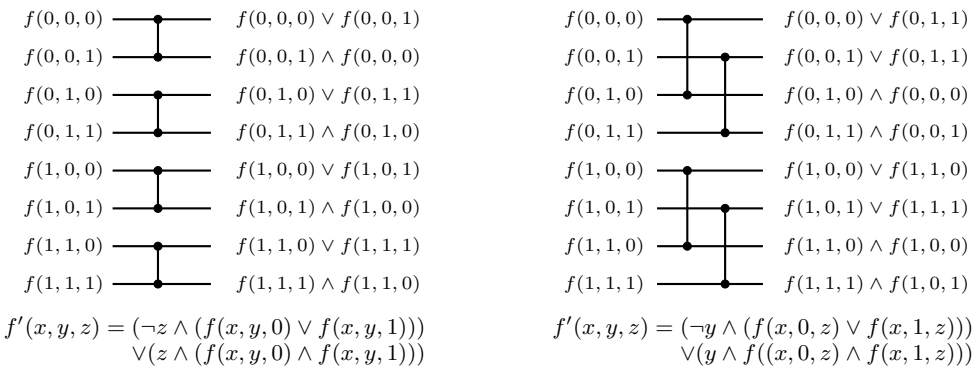

Figure 5 Red-block transformation on variable z (left) and y (right) of function $f(x, y, z)$.

4.2 Elimination of Exist-Quantifier

Function settlement paves a convenient way to existential quantification as the following proposition asserts.

▶ **Proposition 5.** *Given a function* $f(X, Y)$*, let* $f'(X, Y^*) = Sort(f, Y)$*. Then* $\exists X.f'(X, Y^*) = \bigvee_{\alpha \in [\![X]\!]} f'|_\alpha$*.*

That is, the max operation of exist-quantification in SSAT can be reduced to the simple disjunction operation.

For quantifying n exist-variables, in the worse case the formula or circuit of f may grow exponentially in $O(2^n)$. However, in practice the 2^n copies of f may share a significant amount of structural similarity and often can be simplified substantially.

5 Algorithm

With the aforementioned quantifier elimination techniques, we can combine them for SSAT solving based on the following proposition.

▶ **Proposition 6.** *Given the SSAT formula* $\Phi = \exists X_1, \text{Ƃ} Y_1, \ldots, \exists X_k, \text{Ƃ} Y_k.\varphi$ *for non-empty* X_i*'s and* Y_i*'s with the exception of* X_1 *and* Y_k *possibly empty, the probability*

$$\begin{aligned}
\Pr[\Phi] &= \Pr[\exists X_1, \text{Ƃ} Y_1, \ldots, \text{Ƃ} Y_{k-1}, \exists X_k.\varphi^{(k)}], \\
&= \Pr[\exists X_1, \text{Ƃ} Y_1, \ldots, \exists X_{k-1}, \text{Ƃ} Y_{k-1}.\psi^{(k)}], \\
&\vdots \\
&= \Pr[\exists X_1.\varphi^{(1)}], \\
&= \Pr[\psi^{(1)}], \\
&= \frac{\#\psi^{(1)}}{2^{|Y|}}
\end{aligned}$$

where $\varphi^{(i)} = Sort(\psi^{(i+1)}, (Y_i, Y_{i+1}^* \ldots, Y_k^*))$*,* $\psi^{(i)} = \bigvee_{\alpha \in [\![X_i]\!]} \varphi^{(i)}|_\alpha$*,* $\psi^{(k+1)} = \varphi$*, and* $Y = \bigcup_i Y_i$*.*

The proposition naturally translates into the procedure of Algorithm 2. For the input SSAT formula Φ, recall our assumption that all the probabilities of the random quantifiers in Φ

have been converted to 0.5. Also, we assume the variable sets X_i's are ordered. The n levels of quantifiers are iteratively eliminated from the innermost level to the outermost one. Depending on the quantifier type, the corresponding quantifier elimination technique presented in Section 4 is applied. In line 6, λ of Algorithm 1 is set to $|X_i|$ to avoid resorting the partially sorted variables Y as discussed in Section 4.1.1. In line 7, Y is updated by appending Y to X_i^*. After all quantifiers being eliminated, the final quantifier-free formula that corresponds to a settled function can be model counted to derive the satisfying probability of Φ. We further elaborate how model counting can be done on settled functions.

Algorithm 2 Quantifier Elimination Based SSAT Solving.

input : SSAT formula $\Phi = Q_1 X_1, \ldots, Q_n X_n.\phi$
output : Satisfying probability of Φ

1 $Y := \emptyset$;
2 **for** $i \leftarrow n$ *to* 1 **do**
3 **if** $Q_i = \exists$ **then**
4 $\varphi := \mathit{ExistElim}(\varphi, X_i)$;
5 **else**
6 $\varphi := \mathit{BSort}(\varphi, (X_i, Y), |X_i|)$;
7 $Y := \mathit{Append}(X_i^*, Y)$;
8 **end**
9 **end**
10 $\Pr := \dfrac{\mathit{ModelCount}(\varphi)}{2^{|Y|}}$;
11 **return** Pr;

5.1 Model Counting for Settled Function

Given a settled function f over ordered variables $(y_1, \ldots, y_n)$, the model counting problem corresponding to finding the satisfying assignment α largest in terms of the corresponding binary coded number N_α. Finding the minimum or maximum satisfying assignment in a general setting is also known as the *lexicographic Boolean satisfiability* (LEXSAT) problem [18]. For a settled function, LEXSAT can be done efficiently as shown in Algorithm 3. It first tests whether f is unsatisfiable by checking $f|_\alpha = \bot$ for $N_\alpha = 0$ in lines 1-4. It then iteratively flips variable y_i for i from 1 to n to locate the largest N_α making $f|_\alpha = \top$ in lines 5-10. The final count $N_\alpha + 1$ is returned in line 11. The computation is done in $O(mn)$ for m being the formula size of f.

6 Implementation Issues and Enhancement Techniques

We detail implementation issues and enhancement techniques for the proposed SSAT solving.

6.1 Boolean Formula Representation

We exploit And-Inverter Graphs (AIGs) [19] and Binary Decision Diagrams (BDDs) as the main data structures to represent Boolean functions. BDDs are very efficient when computing existential elimination and generating sorted functions using Algorithm 1, while AIGs are more scalable in handling complex and large Boolean functions. Consequently, when solving an SSAT instance, we start by building a BDD of the matrix with an empirically

Algorithm 3 Model Counting for Settled Boolean Function.

input : A settled function f over ordered variable set $Y = (y_1, \ldots, y_n)$
output : Model count of f

$\alpha := \bigwedge_i \neg y_i$;
if $f|_\alpha = \bot$ **then**
  return 0;
end
for $i \leftarrow 1$ *to* n **do**
  $\alpha := \mathit{Flip}(\alpha, y_i)$;
  if $f|_\alpha = \bot$ **then**
    $\alpha := \mathit{Flip}(\alpha, y_i)$;
  end
end
return $N_\alpha + 1$;

decided number of nodes as the limit. If the number of nodes exceeds the limit in the process of building BDDs, we assume the formula is too complicated and AIGs are used as an alternative. After eliminating a level of exist-quantifiers or random-quantifiers, the new formula may potentially become easier to represent by BDDs because of the removal of some variables or because of the simplification of the function. Since building BDDs with the pre-defined limit on node counts is not a very time-consuming task, we tried converting AIGs into BDDs after each quantifier elimination. We remark that `AIGSolve` [29] also adopts this strategy. We observe that most solvable cases are convertible to BDDs after some iterations of quantifier elimination and those that cannot be converted often struggle in the later stage of elimination and thus exceed the time-out limit.

Since most SSAT formulas are in the prenex CNF form, directly translating them into AIGs may result in inefficient circuit representation because many variables in CNF formulas are functionally defined by other variables but they are treated as primary inputs of the AIG. In our implementation, we take advantage of the interpolation-based gate extraction [32] to alleviate the problem. It helps derive compact AIGs from CNF formulas and thus reduce the runtime for quantifier elimination.

6.2 Elimination of Exist-Quantifiers using Boolean Functional Synthesis

When representing the matrix formula with an AIG, elimination of exist-quantifiers by disjunctive expansion may cause the AIG size to increase rapidly. Our practical experience suggests that disjunctive expansion often has poor performance and only works for small instances. We solve this problem by using function composition for exist-quantifier elimination [15], and exploit Boolean functional synthesis tools for good scalability.

Given an existentially quantified Boolean formula $\exists Y.F(X, Y)$ over the set of variables X and Y, the problem of *Boolean functional synthesis* [16, 17] is to compute a vector of Boolean functions $\Psi = (\psi_1, \ldots, \psi_{|Y|})$, referred to as the Skolem function vector, such that $\exists Y.F(X, Y) \leftrightarrow F(X, \Psi(X))$. The existentially quantified formula can be computed by substituting the original variables in the set $y_i \in Y$ by the generated Skolem function $\psi_i \in \Psi$.

Several approaches have been proposed to address the scalability issue of this problem. We mainly consider three prominent tools in this paper. `R2F` [16] uses AIGs as the underlying data structure and exploits interpolation to determinize the given Boolean relation. `CADET` [30]

lifts the QBF solving algorithm *incremental determinization* to perform Boolean functional synthesis. `Manthan` [11, 12] uses a data-driven method by generating a set of training data with the help of constraint sampling and learns the candidate Skolem functions with decision trees. While the candidate functions might be incorrect, `Manthan` applies proof-guided refinement until all generated functions can be used as a valid Skolem function vector. As both `Manthan` and `CADET` require converting AIGs to CNF formulas, the conversion may deteriorate their effectiveness. For future work, we plan to try other Boolean functional synthesis tools, e.g., [1], that work directly on AIGs.

Among the three considered tools, `R2F` is a natural choice for us since it shares the same data structure as our solver. On the other hand, the performance of `R2F` highly depends on the size of the interpolant which may still cause the size of the AIG to increase out of control. For the other two tools working on CNF formulas, we observe that `CADET` is very efficient on easier benchmarks while `Manthan` can handle hard instances and produce smaller Skolem functions. Our implementation for exist-quantifier elimination is a hybrid approach combining these tools. We first try to solve the formula using `CADET` with a short time-out limit. If the runtime of `CADET` exceeds the limit, we switch to `R2F` and carefully monitor the size of the AIG. `Manthan` is used only if we observe that the AIG size becomes too large in the process of `R2F` computation. Nevertheless, most of the unsolvable instances fail because `Manthan` requires too many iterations of refinement. We notice that the current publicly available version of `Manthan` does not use self substitution to handle the Skolem function after a certain number of refinements. In future work, we plan to integrate `R2F` into the refinement procedure of `Manthan` to achieve better scalability.

6.3 Bitonic Sorting

In general, computing partially sorted functions using bitonic sort is not the bottleneck of the proposed algorithm because most SSAT instances have fewer random-variables than exist-variables. Nevertheless, there still exist families of benchmarks that require carefully implemented bitonic sorting to be easily solvable. For example, for the conformant planning benchmark set `Sand-Castle`, the BDDs of the formulas can be built in the early quantifier elimination stage but the node count of the BDD quickly grows in the process of performing Algorithm 1. We solve this problem by heuristically adjusting the ordering of the BDD. Observe that in the i^{th} iteration of the loop in Algorithm 1, the function values of f are sorted with respect to variables $x_{n-i+1}, \ldots, x_n$. The heuristics for BDD reordering is to group sorted variables to remove identical nodes. Accordingly, we move the sorted variables on top of the BDD variable ordering after each outer-loop iteration in Algorithm 1, then perform the traditional BDD reordering algorithm to further reduce the number of nodes. We observe that the number of nodes can be greatly reduced compared to performing reordering directly. As for bitonic sorting on AIG, we notice that we may create a large amount of functionally equivalent nodes in the process of bitonic sort. Consequently, we use SAT sweeping with a small number of conflict limits on the satisfiability solver to reduce the AIG size by merging some of the easily identified functionally equivalent nodes.

6.4 Projected Model Counting on Partially Sorted Function

Observe that if the outermost level is existentially quantified, the problem is reduced to projected model counting if all other quantification levels have been eliminated. Surely we can perform existential elimination on the last quantification level, but as mentioned in Section 6.2, performing exist-quantifier elimination on AIGs can be costly. Hence, we may directly solve the projected model counting problem on a partially sorted function of the form $\exists X.Sort(f, Y)$.

Table 2 Performance comparison of SSAT solvers.

	DC-SSAT	ClauSSAT	ElimSSAT
Solved	193	203	242
PAR-2	3378.12	3192.46	2356.06
Uniquely Solved	18	16	46

The target here is trying to find the assignment α of variable set Y with the largest binary coded value N_α such that $f|_\alpha$ has a satisfying solution. This can solved by checking the satisfiability of $f|_\alpha$, in contrast to checking the *value* of $f|_\alpha$ in Algorithm 3. The process requires $|Y|+1$ calls to a satisfiability checker. Based on our experience, this strategy can be more efficient compared to solving a Boolean functional synthesis problem. This enhancement works particularly well for the E-MAJSAT instances, where projected model counting can be applied after the innermost random-quantifiers being eliminated.

7 Experimental Results

The proposed SSAT solver, named `ElimSSAT`, was implemented in C++ within the `ABC` system [4] and is available at `https://github.com/NTU-ALComLab/elimssat`. We used the ABC built-in AIG package, `CUDD` [33], and `Minisat-2.2` [7] for AIG manipulation, BDD manipulation, and SAT solving, respectively. We compared the performance of `ElimSSAT` with the state-of-the-art general SSAT solvers `ClauSSAT` [6] (under its best performing option `-sguwc`) and `DC-SSAT` [26]. All experiments were conducted on a Linux machine with Intel Xeon 2.1 GHz CPU and 256 GB RAM. We used `Benchexec`[3] as the benchmarking framework for reliable resource measurement. A runtime limit of 3600 seconds and memory limit of 32 GB were imposed on each instance for every solver. In the rest of the discussion, we use the Penalized Average Runtime PAR-2 score to compare each solver, where an unsolved instance is given the penalty of 2 times the time-out limit.

We evaluate the solver performance on 20 different benchmark families with a total of 357 SSAT formulas, which were taken from those used in `ClauSSAT` [6]. Because `ElimSSAT` requires SSAT formulas with normalized probability 0.5 on random variables, we approximated the SSAT formulas for `ElimSSAT` using *WMCRewriting* [20] with 4-bit precision while `DC-SSAT` and `ClauSSAT` took the original formulas for solving. (We note that `DC-SSAT` and `ClauSSAT` performed similarly on formulas with original and approximated probabilities.) Table 2 reports the number of solved cases within runtime and memory limit and the PAR-2 score of each solver. As seen, `ElimSSAT` outperformed the state-of-the-art solvers by solving 242 out of 357 instances while `DC-SSAT` and `ClauSSAT` solved 193 and 203 cases, respectively. `ElimSSAT` also dominated previous work in terms of the PAR-2 score 2356.06 in contrast to 3378.12 of `DC-SSAT` and 3192.46 of `ClauSSAT`. For the number of uniquely solved instances, `ElimSSAT` can solve 46 cases that are not solvable by any of prior solvers `DC-SSAT` and `ClauSSAT`. Figure 6 shows the cactus plot that compares the runtime performance of different SSAT solvers. The x-axis represents the number of solved benchmarks and the y-axis represents the runtime in seconds. The plot suggests that `ElimSSAT` can solve easier benchmarks within a short time while having better scalability than other solvers.

A pairwise comparison with the other solvers is summarized in Table 3, where the row `Less` shows the number of formulas that can be solved by the competing solver but not `ElimSSAT`, the row `More` shows the number of formulas that can be solved by `ElimSSAT` but not the competing solver. As can be seen, `DCSSAT` and `ClauSSAT` can solve 28 and 26

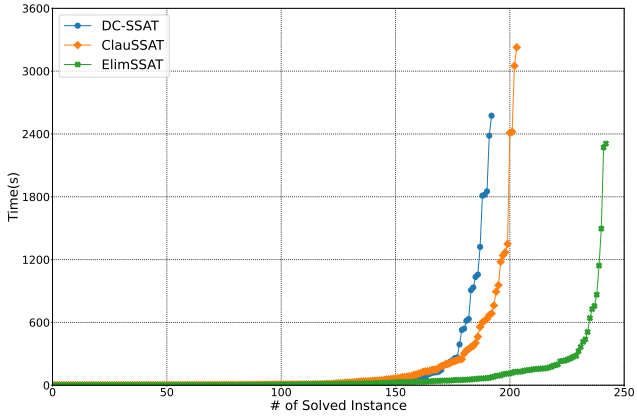

Figure 6 `ElimSSAT` versus state-of-the-art solvers in runtime behavior.

Table 3 ElimSSAT versus state-of-the-art solvers in relative solving performance.

		DCSSAT	ClauSSAT
ElimSSAT	Less	28	26
	More	75	66

formulas, respectively, that `ElimSSAT` cannot solve. Most of these cases are multi-level SSAT formulas where `ElimSSAT` struggled to perform existential elimination by Boolean functional synthesis [11]. On the other hand, `ElimSSAT` can solve 75 and 66 formulas that were not solved by `DCSSAT` and `ClauSSAT`, respectively.

It is worth mentioning that in the above experiments `ElimSSAT` actually solved harder instances than the other two solvers because *WMCRewriting* complicated the formulas. To assess the effects of formula complication due to *WMCRewriting*, we modified the original 357 formulas by making all the random variables have probability 0.5 so that no *WMCRewriting* is required and all solvers work on exactly the same set of benchmarks. Table 4 shows the new results. `ElimSSAT` can solve up to 266 cases with PAR-2 score as 2067.89. Comparing Tables 2 and 4, we found that the efficiency of `ElimSSAT` was improved while the other two solvers did not benefit from the simplified probability values. This phenomenon suggests the potential advantage of `ElimSSAT` on solving instances of unbiased probabilities.

Table 4 Performance comparison of SSAT solvers on modified 0.5-probability benchmarks.

	DC-SSAT	ClauSSAT	ElimSSAT
Solved	193	203	266
PAR-2	3368.66	3192.46	2067.89

For the experiment of Table 2, we examine the solver performance on specific benchmarks in Table 5. For each of the benchmarks, the prefix information is given in the `#blk` column, where we use Σ_i (resp. Π_i) to denote the outermost quantifier is exist (resp. random) and i quantification levels in total. The runtime and the computed satisfying probability are given for each solver in columns `T` and `Pr`, where symbol "-" indicates a time-out or memory-out case. Since `ClauSSAT` can report the upper bound and lower bound even if it fails to give the exact answers before time-out, we report the upper bound in column `UB` and lower bound in column `LB`. Some interesting aspects of `ElimSSAT` are also shown in the table, with

Table 5 Solver comparison on specific benchmarks.

benchmarks		DC-SSAT		ClauSSAT			ElimSSAT			
formula	#blk	Pr	T	UB	LB	T	Pr	T	E	R
ev-pr-4x4										
5-...-lg	Σ_5	1	**0.93**	1	1	3.36	1	640.66	0	BDD
7-...-lg	Σ_7	1	**4.41**	1	0	-	-	-	0	AIG
Connect2										
3x3_w	Σ_{11}	0.2689	**0.05**	0.2689	0.2689	2.09	0.2828	89.86	11	BDD
3x4_w	Σ_{13}	0.1474	**0.09**	0.1474	0.1474	7.24	0.1706	281.07	13	BDD
Adder										
2-unsat	Σ_3	0.9998	0.98	0.9998	0.9998	43.86	0.9999	**0.58**	3	BDD
4-unsat	Σ_3	-	-	1	1	**226.69**	1	1142.73	3	BDD
Arbiter										
depth-23	Π_{48}	-	-	1	0.9999	-	-	-	0	AIG
depth-24	Π_{50}	-	-	1	0.2607	-	-	-	0	AIG
Counter										
cnt03e	Σ_7	-	-	1	0.9942	-	1	**1.45**	7	BDD
cnt03r	Σ_7	-	-	1	0.9930	-	1	**3.69**	7	BDD
k_ph_p										
4	Σ_5	-	-	0.9681	0.9681	562.17	0.9656	**3.91**	5	BDD
5	Σ_5	-	-	1	0	-	0.9993	**76.40**	5	BDD
QIF										
pwd	Σ_3	-	-	1	0	-	1	**1.91**	3	BDD
reverse2	Σ_3	-	-	1	0	-	1	**232.63**	3	BDD
conformant										
empty...-10	Σ_3	-	-	1	0.5469	-	1	**58.77**	2	AIG
empty...-19	Σ_3	-	-	1	0.75	-	0.875	**162.46**	2	AIG
ToiletA										
10_01.12	Σ_3	-	-	1	0.0313	-	0.0313	**154.59**	2	AIG
10_01.13	Σ_3	-	-	1	0.0625	-	0.0625	**142.38**	2	AIG
sand-castle										
14	Σ_3	0.9918	**0.68**	1	0.9912	-	0.9926	2199.20	3	BDD
15	Σ_3	0.9999	**1.38**	1	0.9934	-	-	-	1	BDD
PEC										
c1908-re	Π_2	0.0007	242.88	-	-	-	0.0007	**0.43**	2	BDD
c3540-re	Π_2	-	-	0.0034	0.0034	4.61	0.0034	**0.55**	2	BDD
stracomp										
x75.9	Π_2	-	-	1	1	**601.26**	-	-	0	AIG
x75.14	Π_2	-	-	1	1	**463.05**	-	-	0	AIG

the column `E` indicates the number of eliminated quantifier blocks and `R` shows the final representation for Boolean function when complete solving or terminated due to runtime or memory limit. We only reported certain families with at most 2 formulas due to the page limit; other families are either too hard or too easy for all solvers. Families 1-6 listed in the table are multi-level benchmarks converted from QBFLIB by substituting random quantifiers for universal quantifiers with p randomly chosen in $[0, 1]$. The next 5 families and the last 2 families are E-MAJSAT and random-exist SSAT families, respectively, where family 7 encodes the quantitative information flow (QIF) problem; families 8-10 encode the conformant planning problems; families 11 encode the *probability equivalence checking*[20] problem and family 12 encodes the strategic companies problem[5].

Table 6 Comparison between ElimSSAT with projected model counting and ElimSSAT without projected model counting.

	w/ projected MC	w/o projected MC
Solved	121	106
PAR-2	2163.42	2745.11

As can be seen, ElimSSAT struggled the most in the multi-level benchmarks where existential elimination on AIG spent too much time. For families such as ev-pr-4x4 and arbiter, ElimSSAT cannot even finish the innermost quantifier elimination. More advanced preprocessing techniques may be further development to assist ElimSSAT to solve these challenging benchmarks. If the BDD of a formula can be built after some levels of quantifier elimination, such as families Counter and k_ph_p, ElimSSAT may solve it efficiently. On the other hand, search-based solvers like DC-SSAT can deal with some of these families quite well by outperforming other solvers on families ev-pr-4x4 and Connect2. For the E-MAJSAT benchmarks, DC-SSAT performed very well on the sand-castle families. On the other hand, ElimSSAT dominates other solvers in terms of runtime and solved several cases that could not be solved previously. One of the reasons is the enhancement proposed in Section 6.4. We can observe that for the families ToiletA and conformant, ElimSSAT can solve the formulas after two levels of quantifiers being eliminated and by leaving the last level of exist-quantifiers for projected model counting. In contrast, ClauSSAT performs the best in some of the random-exist SSAT families such as stracomp. In addition, for a large percentage of the benchmarks, ClauSSAT can achieve quite a tight bound even for the unsolvable instances for all solvers such as family arbiter. ClauSSAT can be a preferred approximate solver for hard instances.

We evaluate the benefit of performing projected model counting, mentioned in Section 6.4, on the outermost exist-quantification level by evaluating the 164 E-MAJSAT instances. From Table 6, we can see that with projected counting turned on, we can solve 15 more E-MAJSAT instances and the PAR-2 score dropped significantly from 2745.11 to 2163.42.

Table 7 shows the usage of different tools when performing existential elimination in ElimSSAT. The row Used indicates the number of instances used with the tools and Solved indicates the number of successfully solved instances. We note that the number of failed cases in CADET (resp. R2F) may not be exactly the same as the number of Used cases in R2F (resp. Manthan) because a case may be time-out or memory-out in the CADET (resp. R2F) phase. We use CADET with a 100-second time-out to filter out the easy instances. Despite given a short period of runtime, CADET still performed well by solving 159 out of the 314 instances. R2F is used in 154 instances, while 81 instances solved successfully, others exceed total time limit or the AIG size becomes too large to handle. Manthan can only handle 2 of the 58 instances, but these instances should be exceptionally hard since they also failed in R2F and CADET. As we expect, existential elimination using BDDs is the most efficient and is used the most in ElimSSAT.

Table 7 Performance of different tools used for exist-quantifier elimination in ElimSSAT.

	BDD	CADET	R2F	Manthan
Used	488	314	154	58
Solved	488	159	81	2

8 Conclusions

We developed a new approach to solving SSAT formulae using quantifier elimination. With the usage of BDDs and AIGs and the help of modern Boolean functional synthesis tools, the prototype solver `ElimSSAT` demonstrates the superiority of the proposed framework compared to prior works. For future work, we would like to develop SSAT preprocessing techniques and hybrid methods for existential elimination. Also, it would be interesting to extend the current framework to dependency SSAT (DSSAT) [21] solving.

References

1 Sundararaman Akshay, Supratik Chakraborty, Shubham Goel, Sumith Kulal, and Shetal Shah. What's hard about Boolean functional synthesis? In *Proceedings of the International Conference on Computer Aided Verification*, pages 251–269, 2018.

2 Kenneth Batcher. Sorting networks and their applications. In *Proceedings of Spring Joint Computer Conference*, pages 307–314, 1968.

3 Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, pages 1–29, 2019.

4 Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *Proceedings of the International Conference on Computer Aided Verification*, pages 24–40, 2010.

5 Marco Cadoli, Thomas Eiter, and Georg Gottlob. Default logic as a query language. *IEEE Transactions on Knowledge and Data Engineering*, pages 448–463, 1997.

6 Pei-Wei Chen, Yu-Ching Huang, and Jie-Hong R. Jiang. A sharp leap from quantified boolean formula to stochastic Boolean satisfiability solving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3697–3706, 2021.

7 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 502–503, 2003.

8 Daniel Fremont, Markus Rabe, and Sanjit Seshia. Maximum model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3885–3892, 2017.

9 Eric Freudenthal and Vijay Karamcheti. QTM: Trust management with quantified stochastic attributes. *NYU Computer Science Technical Report TR2003-848*, pages 1–14, 2003.

10 Bishwamittra Ghosh, Debabrota Basu, and Kuldeep S. Meel. Justicia: A stochastic SAT approach to formally verify fairness. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7554–7563, 2021.

11 Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Manthan: A data driven approach for Boolean function synthesis. In *Proceedings of the International Conference on Computer Aided Verification*, pages 611–633, 2020.

12 Priyanka Golia, Friedrich Slivovsky, Subhajit Roy, and Kuldeep S. Meel. Engineering an efficient Boolean functional synthesis engine. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–9, 2021.

13 Jinbo Huang. Combining knowledge compilation and search for conformant probabilistic planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 253–262, 2006.

14 Mikolás Janota and Joao Marques-Silva. Solving QBF by clause selection. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 325–331, 2015.

15 Jie-Hong R. Jiang. Quantifier elimination via functional composition. In *Proceedings of the International Conference on Computer Aided Verification*, pages 383–397, 2009.

16 Jie-Hong R. Jiang, Hsuan-Po Lin, and Wei-Lun Hung. Interpolating functions from large Boolean relations. In *Proceedings of the International Conference on Computer-Aided Design*, pages 779–784, 2009.

17 Ajith K. John, Shetal Shah, Supratik Chakraborty, Ashutosh Trivedi, and Sundararaman Akshay. Skolem functions for factored formulas. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 73–80, 2015.

18 Donald E. Knuth. *The Art of Computer Programming. Fascicle 6: Satisfiability.* Addison-Wesley, 2015.

19 Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the Design Automation Conference*, pages 232–237, 2001.

20 Nian-Ze Lee and Jie-Hong R. Jiang. Towards formal evaluation and verification of probabilistic design. In *Proceedings of the International Conference on Computer-Aided Design*, pages 340–347, 2014.

21 Nian-Ze Lee and Jie-Hong R. Jiang. Dependency stochastic Boolean satisfiability: A logical formalism for NEXPTIME decision problems with uncertainty. *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3877–3885, 2021.

22 Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R. Jiang. Solving stochastic Boolean satisfiability under random-exist quantification. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 688–694, 2017.

23 Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R. Jiang. Solving exist-random quantified stochastic Boolean satisfiability via clause selection. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1339–1345, 2018.

24 Michael Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, pages 1–36, 1998.

25 Michael Littman, Stephen Majercik, and Toniann Pitassi. Stochastic Boolean satisfiability. *Journal of Automated Reasoning*, pages 251–296, 2001.

26 Stephen M. Majercik and Byron Boots. DC-SSAT: A divide-and-conquer approach to solving stochastic satisfiability problems efficiently. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 416–422, 2005.

27 Christos H. Papadimitriou. Games against nature. *Journal of Computer and System Sciences*, pages 288–301, 1985.

28 Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place, comparison-based sorting with CUDA: A study with bitonic sort. *Concurrency and Computation: Practice and Experience*, pages 681–693, 2011.

29 Florian Pigorsch and Christoph Scholl. An AIG-based QBF-solver using SAT for preprocessing. In *Proceedings of the Design Automation Conference*, pages 170–175, 2010.

30 Markus N. Rabe. Incremental determinization for quantifier elimination and functional synthesis. In *Proceedings of the International Conference on Computer Aided Verification*, pages 84–94, 2019.

31 Ricardo Salmon and Pascal Poupart. On the relationship between satisfiability and Markov decision processes. In *Proceedings of the Uncertainty in Artificial Intelligence Conference*, pages 1105–1115, 2020.

32 Friedrich Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In *Proceedings of the International Conference on Computer Aided Verification*, pages 508–528, 2020.

33 Fabio Somenzi. CUDD: CU decision diagram package release 2.4.2, 2005.