SharpSSAT: A Witness-Generating Stochastic Boolean Satisfiability Solver

Yu-Wei Fan[1], **Jie-Hong R. Jiang**[1,2]

[1]Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan
[2]Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
{r11943096, jhjiang}@ntu.edu.tw

Abstract

Stochastic Boolean satisfiability (SSAT) is a formalism allowing decision-making for optimization under quantitative constraints. Although SSAT solvers are under active development, existing solvers do not provide Skolem-function witnesses, which are crucial for practical applications. In this work, we develop a new witness-generating SSAT solver, `SharpSSAT`, which integrates techniques, including component caching, clause learning, and pure literal detection. It can generate a set of Skolem functions witnessing the attained satisfying probability of a given SSAT formula. We also equip the solver `ClauSSat` with witness generation capability for comparison. Experimental results show that `SharpSSAT` outperforms current state-of-the-art solvers and can effectively generate compact Skolem-function witnesses. The new witness-generating solver may broaden the applicability of SSAT to practical applications.

1 Introduction

Efficient solving techniques for Boolean satisfiability (SAT) problems have led to tremendous success in formal verification, synthesis, electronic design automation, and other domains that rely on Boolean reasoning. The growing complexity of verification and variety of applications have motivated researchers to investigate logical formalisms and problems beyond SAT. Recently, stochastic Boolean satisfiability (SSAT) has drawn attention from the field of probabilistic planning (Salmon and Poupart 2019), verification for probabilistic design (Lee and Jiang 2018), and fairness analysis of machine-learning models (Ghosh, Basu, and Meel 2021). It provides a convenient language to compactly encode decision and optimization problems under uncertainty. As more applications of SSAT emerge, developing effective solving techniques becomes crucial.

Solving a given SSAT formula involves two important tasks: One is to determine the maximum satisfying probability of the formula. The other is to generate a Skolem-function witness, i.e., Skolem functions for existential variables, that witnesses the maximum satisfying probability. To date, existing SSAT solvers, e.g., `DC-SSAT` (Majercik and Boots 2005), `Prime` (Salmon and Poupart 2019), `ClauSSat` (Chen, Huang, and Jiang 2021), and `ElimSSAT` (Wang et al. 2022), mostly focus on the first task. Specifically, `DC-SSAT` is a divide-and-conquer SSAT solver that solves an SSAT formula by decomposing it into smaller sub-problems. `Prime` exploits model counting techniques, including component analysis and advanced caching schemes, for SSAT solving. `ClauSSat` tackles SSAT solving based on the clause selection (Janota and Marques-Silva 2015) framework. `ElimSSAT` proposes a quantifier-elimination approach to SSAT solving. Regarding the second task, the witness can be crucial for practical application. For example, the witness may correspond to an access control policy between organizations (Freudenthal and Karamcheti 2003), and a feasible plan for planning (Littman, Majercik, and Pitassi 2001). Unfortunately, none of the existing SSAT solvers can provide a Skolem-function witness. We note that although `DC-SSAT` can maintain solution tree structures during solving, it still lacks a complete procedure to generate a full Skolem-function witness.

In this paper, a new SSAT solver, named `SharpSSAT`, is developed based on the model counter `SharpSAT` (Thurley 2006). Different from `Prime`, in addition to component analysis and caching, `SharpSSAT` further combines clause learning and pure literal detection. To bridge the gap between SSAT solving and witness genreration, we propose two different witness generation algorithms under `SharpSSAT` and `ClauSSat`. To the best of our knowledge, it is the first time that witness generation is supported for SSAT solvers. Experiments on various benchmarks show the superiority in performance `SharpSSAT` to state-of-the-art solvers, including `DC-SSAT`, `Prime`, `ClauSSat`, and `ElimSSAT`. The witness generation overhead and witness quality are compared between `SharpSSAT` and `ClauSSat`. The results suggest that `SharpSSAT` can generate more compact strategies with less run-time but more memory overhead than `ClauSSat`.

The rest of this paper is organized as follows. The required preliminaries are provided in Section 2 The SSAT solving and witness generation algorithms of `SharpSSAT` are detailed in Section 3 and Section 4, respectively. Section 5 elaborates the witness generation algorithm for `ClauSSat`. Section 6 evaluates `SharpSSAT` with experimental results. Section 7 concludes this work and outline some future work.

2 Preliminaries

We use the symbols "$\top$" and "$\bot$" to represent Boolean values TRUE and FALSE, respectively. Symbols "$\neg$," "$\vee$," "$\wedge$," and "$\equiv$" denote Boolean connectives negation, disjunction, conjunction, and equivalence, respectively. For brevity, in a formula, a conjunction $\wedge$ is sometimes omitted, and a negation is sometimes denoted with an overline. A *literal* l of a variable x is either a positive-phase literal x or a negative-phase literal $\neg x$. We denote the variable corresponding to l by $var(l)$. A *clause* is a disjunction of literals. A *cube* is a conjunction of literals. A Boolean formula f is in the *conjunctive normal form* (CNF) if it is a conjunction of a set clauses.

For a CNF formula ϕ, we use $Vars(\phi)$ and $Cls(\phi)$ to denote the set of variables and clauses appear in f. A Boolean function f over variables X is a mapping $f : \mathbb{B}^{|X|} \to \mathbb{B}$. An *assignment* τ is a mapping $\tau : X \to \mathbb{B}$. The *induced* formula over an assignment τ, denoted as $f|_\tau$, is obtained by replacing each variable with its value mapped by τ. The *onset* of a Boolean function f, denoted as f^+, is the set of assignments $\{\tau \mid f|_\tau = \top\}$. The *offset* of f, denoted as f^-, is the set of assignments $\{\tau \mid f|_\tau = \bot\}$.

Model Counting

Given a CNF formula ϕ, the unweighted model counting problem is to find the number of satisfying assignments of ϕ, denoted as $\#(\phi)$. The weighted model counting problem can be easily extended from the unweighted model counting by assigning each literal a weight. The weight of an assignment is the product of the weight of the literals that appear in the assignment. The weighted model counting is to find the summation of the weight of the assignments.

Stochastic Boolean Satisfiability

An SSAT formula Φ over variables $X = X_1 \cup X_2 \cup \cdots \cup X_n$ with $X_i \neq \emptyset$ and $X_i \cap X_j = \emptyset$ for $i \neq j$ and $i, j \in \{1, \cdots, n\}$, can be expressed in the *prenex conjunctive normal form* (PCNF)

$$Q_1 X_1, \cdots, Q_n X_n . \varphi \tag{1}$$

where $Q_1 X_1, \cdots, Q_n X_n$ is called the *prefix*, for $Q_i \in \{\exists, \mathbb{R}\}$ being either an existential quantifier $\exists$ or a randomized quantifier $\mathbb{R}$, and φ being a CNF formula is called the *matrix*. The *quantification level* of a variable x, denoted by $qlev(x)$, is i if $x \in X_i$. We refer to the set $X_\exists = \{x \in X_i \mid Q_i = \exists\}$ as the *existential variables* and the set $X_\mathbb{R} = \{x \in X_i \mid Q_i = \mathbb{R}\}$ as the *randomized variables*. When a variable x is quantified by randomized quantifier $\mathbb{R}^{p_x}$, it means that x is assigned to $\top$ (resp. $\bot$) with probability p_x (resp. $1 - p_x$). Given a variable x at the outermost quantifier, the satisfying probability of the SSAT formula Φ is computed by the following rules.

1. $\Pr[\top] = 1$,
2. $\Pr[\bot] = 0$,
3. $\Pr[\Phi] = \max\{ \Pr[\Phi|_{\neg x}], \Pr[\Phi|_x] \}$ if $x \in X_\exists$,
4. $\Pr[\Phi] = (1 - p_x) \Pr[\Phi|_{\neg x}] + p_x \Pr[\Phi|_x]$ if $x \in X_\mathbb{R}$

In solving an SSAT formula, we aim to compute its maximum satisfying probability. On the other hand, for witness generation, the objective is to derive *Skolem functions* for the existential variables to attain the claimed satisfying probability. Formally, given an SSAT formula Φ in the form of Eq. (1), we say the set of Skolem functions $F = \{f_x \mid x \in X_\exists\}$, for f_x depending on the set of randomized variables $Y_x = \{y \mid y \in X_\mathbb{R}, qlev(y) < qlev(x)\}$, is a witness for probability p if the SSAT formula Φ' obtained from Φ by substituting each existential variable x in the matrix of Φ with f_x has satisfying probability $\Pr[\Phi'] = p$.

Example 1. *Consider the SSAT formula*

$$\mathbb{R}^{0.4} x_1, \exists y_1, \mathbb{R}^{0.3} x_2, \exists y_2. \underbrace{(\bar{y}_1 \vee x_1)(y_1 \vee \bar{x}_1)}_{y_1 \equiv x_1} \underbrace{(\bar{y}_2 \vee x_1)(\bar{y}_2 \vee \bar{x}_2)(y_2 \vee \bar{x}_1 \vee x_2)}_{y_2 \equiv (x_1 \wedge \bar{x}_2)}.$$

The Skolem functions $F = \{f_{y_1}(x_1) = \top, f_{y_2}(x_1, x_2) = x_1 \wedge \bar{x}_2\}$ *is a witness for probability* 0.4 *since by substituting* y_1 *with* $\top$ *and* y_2 *with* $x_1 \wedge \bar{x}_2$*, the matrix becomes* (x_1)*, and* $\Pr[\mathbb{R}^{0.4} x_1 . (x_1)] = 0.4$*. We note that, in fact, the maximum satisfying probability of this formula is* 1.

For an SSAT formula Φ, a witness is associated with a probability p. Because p is not necessarily equal to the maximum satisfying probability $\Pr[\Phi]$, a witness may not be strong enough to certify the maximum satisfying probability. However, it certifies a lower bound of $\Pr[\Phi]$. In this work, when a solver concludes a probability p for an SSAT formula, we are concerned with deriving the witness for the claimed probability p.

3 SSAT Solving via Component Analysis and Clause Learning

We demonstrate the SSAT solving algorithm incorporating advanced model counting techniques as shown in Algorithm 1. The algorithm differs from `Prime` (Salmon and Poupart 2019) in the highlighted lines. Algorithm 1 takes an SSAT formula Φ as input and returns the satisfying probability p_{ret}. It starts with a Boolean constraint propagation on the formula in Line 1. After that, it decomposes the current formula into several sub-components in Line 2. For each sub-component Φ_i, it first checks if the component is in the cache. If the current component is in the cache, it will update p_{ret} with the cached probability and continue to the next component. Otherwise, it chooses a branching literal x that appears in Φ_i and then performs constant propagation to simplify the positive cofactor $\Phi_i|_x$ and the negative cofactor $\Phi_i|_{\neg x}$. In Lines 11 and 12, it recursively calls the procedure to get the satisfying probabilities, p_0 and p_1, of the two branches. According to the quantifier of the branching literal, it concludes that the satisfying probability of Φ_i is either the maximum or weighted sum of p_0 and p_1. Upon solving the probability of Φ_i, it updates the return probability p_{ret} accordingly. When all the sub-components are processed, it adds the cache entry (Φ, p_{ret}) to the cache to avoid re-computation.

Algorithm 1: solveSSAT

Input: An SSAT formula Φ
Output: satisfying probability p_{ret}

1: *bcp*(Φ)
2: *componentAnalysis*(Φ)
3: *pureElimination*(Φ)
4: $p_{\text{ret}} \leftarrow 1$
5: **for** $\Phi_i \in$ *Components*(Φ) **do**
6: $p \leftarrow 0$
7: **if** *inCache*(Φ_i) **then**
8: $p_{\text{ret}} \leftarrow p_{\text{ret}} \cdot$ *getProbFromCache*(Φ_i)
9: continue
10: $x \leftarrow$ *chooseBranchLit*(Φ_i)
11: $p_0 \leftarrow$ *solveSSAT*($\Phi_i|_x$)
12: $p_1 \leftarrow$ *solveSSAT*($\Phi_i|_{\neg x}$)
13: **if** $x \in X_\exists$ **then**
14: $p \leftarrow \max(p_0, p_1)$
15: **else**
16: $p \leftarrow \Pr[x] \cdot p_0 + \Pr[\neg x] \cdot p_1$
17: **if** $p = 0$ **then**
18: *removePollutedCache*(Φ_i)
19: *addLearnedClause*()
20: $p_{\text{ret}} \leftarrow 0$
21: break
22: $p_{\text{ret}} \leftarrow p_{\text{ret}} \cdot p$
23: *addToCache*(Φ, p_{ret})
24: **return** p_{ret}

Component Analysis and Caching

Component analysis is a technique that is widely used in search-based model counter (Thurley 2006; Sang et al. 2004). It sees a formula Φ as a connectivity graph and applies breadth-first search to identify connected components, say $\Phi_1, \Phi_2, \ldots, \Phi_n$, such that $Vars(\Phi_i) \cap Vars(\Phi_j) = \emptyset$. Then the model count $\#(\Phi)$ of Φ equals the products of the model counts of its components, i.e.,

$$\#(\Phi) = \prod_{i=1}^{n} \#(\Phi_i). \tag{2}$$

Salmon and Poupart observe that such a property also holds in SSAT:

Theorem 1. *Given an SSAT formula Φ in the form of Eq. (1), if $\varphi = \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_k$ such that $Vars(\varphi_i) \cap Vars(\varphi_j) = \emptyset$, and let $\Phi_i = Q_1 X_1 \cdots Q_n X_n, \varphi_i$, then* $\Pr[\Phi] = \prod_{i=1}^{k} \Pr[\Phi_i]$.

Hence, by Theorem 1, we can solve the original SSAT formula Φ by decomposing it into smaller sub-components $\Phi_1, \Phi_2, \cdots, \Phi_n$ and solve each of them independently.

Caching is another technique that records the previously solved sub-problems to avoid solving the same sub-problem repetitively (Sang et al. 2004). It has been shown that different coding of the component will significantly affect the cache miss rate and memory usage (Thurley 2006). In our implementation, we use the *hybrid-coding scheme* proposed in (Thurley 2006).

Clause Learning

Clause learning records the conflict information as a learned clause to effectively prune the UNSAT search space (Marques-Silva and Sakallah 1999). Adding learned clauses to the original formula can strengthen the power of Boolean constraint propagation. In `SharpSSAT`, the learned clause would be added to the matrix of the SSAT formula whenever a conflict occurs. However, as pointed out by (Sang et al. 2004), we may cache some components with the wrong model count when encountering an UNSAT component in model counting. The same problem occurs in SSAT solving. Therefore, when a sub-component Φ_i is UNSAT, all the cached sub-components derived from Φ_1 to Φ_{i-1} need to be removed from the cache. In Algorithm 1, the function *removePollutedCache* does this job in Line 18.

Pure Literal Detection

A literal l is called *pure* in the formula Φ if l appears in Φ but not $\neg l$. If l is pure and $var(l) \in X_\exists$, we can immediately assign l to $\top$ and eliminate all the clauses containing l, called the occurrence clauses. In QBF solving, the detection of pure literal is commonly implemented using a clause-watching data structure (Gent et al. 2003) as it does not need to be updated during backtrack. In `SharpSSAT`, we observe that pure literals can be detected during component analysis since all literals appearing in the formula will be visited once without the clause-watching data structure. Although more pure literals may appear and can be detected by iteratively applying component analysis after eliminating the clauses involving a detected pure literal, the extra component analysis steps can be too costly. Therefore, we implement the detection and elimination in one pass. Empirical results show that such implementation is still effective and introduces little run-time overhead.

Early Return at Existential Quantifier

With the component analysis scheme, the update of satisfying probability is monotonically decreasing since p_{ret} is multiplied by a probability, which is smaller than 1, in every sub-component iteration (Line 22 in Algorithm 1). Since at existential quantifier the returned probability is the maximum among the two branches, given the probability of the first branch $\Pr[\Phi|_{\neg v}]$, whenever an update makes the probability of the current branch $p_{\text{ret}} \leq \Pr[\Phi|_{\neg v}]$, `SharpSSAT` will immediately conclude $\Pr[\Phi] = \Pr[\Phi|_{\neg v}]$ without solving the remaining sub-components.

Example 2. *Consider the SSAT formula Φ:*

$$\mathfrak{R}^{0.4} x_1, \mathfrak{R}^{0.5} x_2, x_3, \exists y_1, y_2, y_3, \mathfrak{R}^{0.6} x_4.$$
$$(x_1 \vee \bar{y}_1)(\bar{y}_1 \vee \bar{y}_2 \vee y_3)(y_1 \vee \bar{y}_2 \vee \bar{x}_4)(\bar{y}_2 \vee \bar{y}_3)$$
$$(y_2 \vee y_3 \vee \bar{x}_4)(y_3 \vee x_4)(x_2 \vee x_3),$$

Algorithm 1 starts from the x_1 branch. $\Phi|_{x_1}$ is then decomposed into $\Phi_1 = (\bar{y}_1 \vee \bar{y}_2 \vee y_3)(y_1 \vee \bar{y}_2 \vee \bar{x}_4)(\bar{y}_2 \vee \bar{y}_3)(y_2 \vee y_3 \vee \bar{x}_4)(y_3 \vee x_4)$ and $\Phi_2 = (x_2 \vee x_3)$. Now Φ_1 is solved first. At the $\bar{y}_1 y_2$ branch, since $\Phi_1|_{\bar{y}_1 y_2}$ is UNSAT*, a learned clause $(y_1 \vee \bar{y}_2)$ is added to the matrix. At the $\bar{y}_1 \bar{y}_2$ branch, where $\Phi_1|_{\bar{y}_1 \bar{y}_2} = (y_3 \vee \bar{x}_4)(y_3 \vee x_4)$, literal y_3 is detected as pure and $\Pr[\Phi_1|_{\bar{y}_1 \bar{y}_2}]$ is concluded to be* 1. *Since y_2*

is existential, $\Pr[\Phi_1|_{\bar{y}_1}]= \max(0,1) = 1$. *As* $\Pr[\Phi_1|_{\bar{y}_1}]= 1$, $\Pr[\Phi_1]$ *is directly concluded to be 1 without solving* $\Phi_1|_{y_1}$. *Similarly,* $\Pr[\Phi_2]$ *is valuated to* 0.75 *and* $\Pr[\Phi_{x_1}]= 1 \cdot 0.75 = 0.75$. *Algorithm 1 then explores the* $\bar{x}_1$ *branch, where* $\Phi|_{\bar{x}_1} = (y_3 \vee \bar{x}_4)(y_3 \vee x_4)(x_2 \vee x_3)$. *After a component analysis, two cache hits occur for* $(y_3 \vee \bar{x}_4)(y_3 \vee x_4)$ *and* $(x_2 \vee x_3)$. $\Pr[\Phi_{\bar{x}_1}]$ *equals* 0.75. *Since* x_1 *is randomized, the returned probability* $\Pr[\Phi]= 0.4 \cdot 0.75 + 0.6 \cdot 0.75 = 0.75$.

4 SharpSSAT Witness Generation

In this section, we focus on witness generation for `SharpSSAT`.

The algorithm for `SharpSSAT` witness generation proceeds in two phases: It first records a *trace* corresponding to the solving steps and second constructs a Skolem-function witness from the trace.

Trace Representation

A *trace* is represented as a single-source directed-acyclic graph (DAG). Each node n corresponds to a formula $n.f$. We use $Parents(n)$ and $Children(n)$ to denote the set of the parents and children of n, respectively. A trace has two constant sink nodes, the $zero$-node and the one-node with the associated formula $\bot$ and $\top$, respectively. On the other hand, there are two different types of non-leaf nodes: the *decision-node* and the *and-node*, which correspond to a branching step and a component analysis step, respectively, during SSAT solving. Each decision-node n is associated with a decision variable $n.v$ and has exactly two children, a *positive* one and a *negative* one with respect to their parents n. Note that for a non-root node, it can be a negative child of a parent and, at the same time, a positive one of another parent. For a decision-node n with its variable $n.v$, the formula of its positive child and negative child are $n.f|_v$ and $n.f|_{\neg v}$, respectively, where $n.f|_v$ and $n.f|_{\neg v}$ can be obtained from $n.f$ by constant propagation replacing variable $n.v$ with $\top$ and $\bot$, respectively. For each decision `SharpSSAT` made, the corresponding decision-node and the existential literals implied by BCP due to the decision are recorded for later witness extraction. On the other hand, an and-node corresponds to a component analysis step and can have one or more children, each representing one sub-component.

Consider the case where an and-node n has exactly one child, meaning that $n.f$ itself is a connected component. Such and-node is redundant as its formula is identical to its child's, and thus can be omitted by connecting its parents to its child. Given an and-node n and its children $Children(n)$, the relationship of their corresponding formulas is

$$n.f = \bigwedge_{c \in Children(n)} c.f.$$

Also, the children of an and-node must be decision-nodes. With the above definition, a trace can characterize the solving steps of the `SharpSSAT` procedure for witness extraction.

Example 3. *Consider the SSAT formula in Example 2. The resulting trace can be constructed as shown in Fig. 1. The rectangular nodes are the constant sink nodes, the circular nodes are decision-nodes, and the elliptic nodes are and-nodes. We note that the multiple incoming edges of nodes* x_2 *and* y_3 *are due to caching. To support caching, we extend the hybrid-code mentioned in the previous section with an extra pointer pointing to the node representing the cached component. When there is a cache hit, the trace can be updated by adding an edge from the current node to the cached node.*

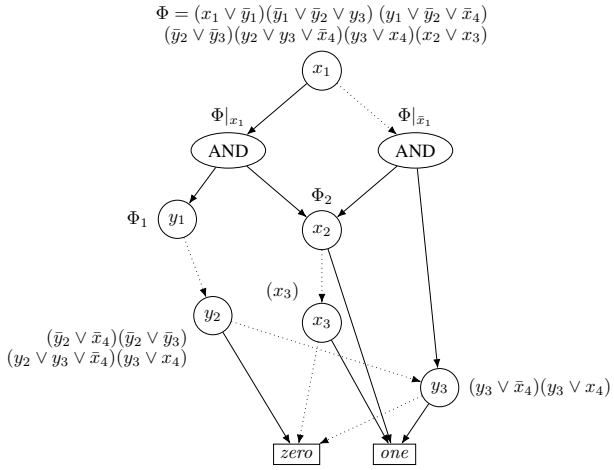

Figure 1: The trace in Example 3

Unfortunately, a naive implementation of the above-mentioned trace becomes impractical regarding memory usage when a solving procedure includes hundreds of thousands of decisions. In our implementation, we observe that any and-node in the trace can be eliminated by directly connecting its parent to its child and marking these children negative (positive) if this and-node is a negative (positive) child of its parent. The resulting trace consists of decision-nodes only. On the other hand, recall that the descendants of the current node, corresponding to the current solving steps, would be removed from the trace when the function $removePollutedCache$ is called. To reuse the memory occupied by those removed nodes, we maintain a reference count for each node and periodically free those nodes with zero reference count.

Witness Extraction from Trace

Upon the solving procedure is completed, we then extract a Skolem-function witness from the trace. The witness is constructed by traversing the trace once in topological order. The procedure is shown in Algorithm 2. We assumed that the Skolem function f_x for each existential variable x is globally accessible. For each node n, the formula $n.\tau$ is maintained to characterize the precondition (in terms of randomized variables) when this decision is made. When a node n is visited, its precondition is the disjunction over all the preconditions of $Parents(n)$ (Line 5). Suppose the decision variable x is existential. In that case, it will update the Skolem functions with λ by the function *updteSkolemFunction* shown in Algorithm 3, according to the decision branch ($n.maxBranch$) that leads to a greater satisfying probability and the implied

Algorithm 2: extractStrategyFromTrace

Input: a trace
Output: $\{f_x \mid x \in X_\exists\}$

for each node $n \in$ trace **do**
  $n.\tau \leftarrow \top$
for $x \in X_\exists$ **do**
  $f_x \leftarrow \bot$
for each node $n \in$ trace in topological order **do**
  $x \leftarrow n.v$
  $\lambda \leftarrow \bigvee_{p \in Parents(n)} p.\tau$
  if $x \in X_\exists$ **then**
    if $n.maxBranch = \top$ **then**
      $L \leftarrow$ exist. literals implied by x
      updateSkolemFunction$(L \cup \{x\}, \lambda)$
    else
      $L \leftarrow$ exist. literals implied by $\neg x$
      updateSkolemFunction$(L \cup \{\neg x\}, \lambda)$
    $n.\tau \leftarrow \lambda$
  else
    for $d \in Children(n)$ **do**
      if d is positive **then**
        $\lambda \leftarrow \lambda \wedge x$
        $L \leftarrow$ exist. literals implied by x
        updateSkolemFunction(L, λ)
      else
        $\lambda \leftarrow \lambda \wedge \neg x$
        $L \leftarrow$ exist. literals implied by $\neg x$
        updateSkolemFunction(L, λ)
      $d.\tau \leftarrow \lambda$
return $\{f_x \mid x \in X_\exists\}$

Algorithm 3: updateSkolemFunction

Input: a set of literals L, the precondition λ
Output: void

for $l \in L$ **do**
  $x \leftarrow var(l)$
  if l is positive **then**
    $f_x^+ \leftarrow f_x^+ \vee \lambda$
  else
    $f_x^- \leftarrow f_x^- \vee \lambda$
return

existential literals L. For the case when x is randomized, the Skolem functions are constructed similarly, except that the decision of x must be included in the precondition.

Proposition 1. *Algorithm 2 generates a set of Skolem functions as a witness to the satisfying probability p returned by* `SharpSSAT`.

The correctness of the proposition can be established by induction on the structure of the trace. We consider Proposition 1 on the associated formula of a node and process the nodes in reverse topological order. The induction starts from the base case, the decision nodes with two constant children. The induction hypothesis assumes that the proposition holds at the k-th visited node. Now we consider the $k+1$-th visited node n. Where $n.v$ is existential, the Skolem functions of its descendants remain unchanged, while the Skolem function of $n.v$ is $\top$ or $\bot$ depending on the decision that leads to a greater probability. Otherwise, the Skolem functions of the descendants of its positive child and negative child have to be strengthened by conjunction with $n.v$ or $\neg n.v$ depending on the decision branch. It can be checked that with this update, the resulting Skolem functions are the same as those generated by Algorithm 2. Finally, by induction, the proposition holds in the last processed node, which is the only source node in the trace.

5 ClauSSat Witness Generation

In addition to witnessing the probability returned by `SharpSSAT`, we develop the witness generation algorithm for `ClauSSat`. We first give some background about the clause selection mechanism of `ClauSsat` before elaborating the witness generation procedure as follows.

Clause Selection

The solver `ClauSSat` (Chen, Huang, and Jiang 2021) extends the *clause selection* technique of quantified Boolean formula (QBF) (Janota and Marques-Silva 2015) to SSAT. It introduces local and global selection variables for each clause to track the satisfaction of clauses in each quantification level. We follow the notation used in (Chen, Huang, and Jiang 2021). Given an SSAT formula of Eq. (1), where $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, the *subclause* of C_i with respect to some quantification level j, denoted by $C_i^{\bowtie j}$, is the set of literals $\{l \mid l \in C_i, qlev(l) \bowtie j\}$, where $\bowtie \in \{<, >, \leq, \geq, =\}$. In the sequel, the "=" in $C_i^{=j}$ is omitted as C_i^j for simplicity. The *local selection variable* and *global selection variable* can be defined as follows.

Definition 5.1 (Local and global selection variable)**.** The *local selection variable* t_i^j for clause C_i at quantification level *j* is defined and constrained by

$$t_i^j \equiv \bigwedge \neg C_i^j.$$

The *global selection variable* for clause C_i at quantification level *j* is defined and constrained by

$$s_i^j \equiv \bigwedge \neg C_i^{\leq j}.$$

A clause C_i is said to be *locally selected* (resp. *locally deselected*) at quantification level k if t_j^k is evaluated $\top$ (resp. $\bot$). Similarly, a clause is said to be *globally selected* (resp. *globally deselected*) at quantification level k if s_j^k is evaluated $\top$ (resp. $\bot$).

The global selection variables describe the selection status up to the current quantification level, i.e., which clauses are already satisfied and which are not. The local selection variables further characterize that the clauses are satisfied at which quantification levels. The reader is referred to (Chen, Huang, and Jiang 2021) for detailed information of clause selection.

Algorithm 4: solveSSAT-$\exists$ (Chen, Huang, and Jiang 2021)

Input: $\Phi = \exists X_j \cdots Q_n X_n.\phi$
Output: p : satisfying probability

1: $p \leftarrow 0$
2: $\tau \leftarrow \emptyset$
3: $C_L \leftarrow \emptyset$
4: **if** $j = n$ **then**
5: **if** SAT(ϕ) = $\top$ **then**
6: $p \leftarrow$ *WeightedModelCount*($\exists X_n.\phi$)
7: **else**
8: $V \leftarrow \emptyset$
9: $\psi_j(X_1, T_1) \leftarrow \bigwedge_{C_i \in \phi} (t_i^j \equiv \neg C_i^1)$
10: $\psi_{j+1}(X_2, T_2) \leftarrow \bigwedge_{C_i \in \phi} (t_i^{j+1} \equiv \neg C_i^1)$
11: **while** SAT(ψ_1 = $\top$) **do**
12: $\tau_j \leftarrow$ the found model of ψ_j for X_j
13: $(p, \tau_{j+1}) \leftarrow$ *SolveSSAT-*$\exists(\Phi|_{\tau_j})$
14: $\tau'_{j+1} \leftarrow$ *MaximalPruning*$(\psi_j|\tau_j, \psi_{j+1}, \tau_{j+1})$
15: $c_{T_j} \leftarrow$ *PruneSelection*$(\psi_j|_{\tau_j}, \psi_{j+1}|_{\tau'_{j+2}})$
16: *V.CollectProbabilitySelectionCubesPair*(p, c_{T_j})
17: $C_L \leftarrow \neg c_{T_j}$
18: $\psi_1 \leftarrow \psi_1 \wedge C_L$
19: **if** $p = 0$ **then**
20: *AddLearntClausesToPriorLevels*(C_L)
21: **else**
22: *ConstructWitness*(C_L, τ_{j+1}, j)
23: $p \leftarrow$ *ComputeProbabiity*(V)
24: **return** p

Algorithm 5: constructWitness

Input: C_L, $\tau_{\max}$, j
Output: $\{f_x \,|\, x \in X_{j+1}\}$

1: $\lambda \leftarrow \neg C_L$
2: **if** $j \geq 1$ **then**
3: $\lambda \leftarrow \lambda \wedge \bigwedge_{s_i^{j-1}=\top} s_i^{j-1} \wedge \bigwedge_{s_i^{j-1}=\bot} \neg s_i^{j-1}$
4: **for** $l \in \tau_{\max}$ **do**
5: $x \leftarrow var(l)$
6: **if** $sign(l)$ **then**
7: $f_x^- \leftarrow f_x^- \vee \lambda$
8: **else**
9: $f_x^+ \leftarrow f_x^+ \vee \lambda$
10: **return** $\{f_x \,|\, x \in X_{j+1}\}$

Witness Generation

Witness generation under `ClauSSat` is shown in Algorithm 5. Algorithm 5 takes three arguments returned: the learned clause C_L, the maximum assignment $\tau_{\max}$, and the quantification level j. Considering an induced SSAT formula $\exists X_j \exists X_{j+1} \cdots Q_n X_n.\phi$, Algorithm 5 is invoked by *SolveSSAT-*$\exists$ (Chen, Huang, and Jiang 2021) shown in Algorithm 4 (line 22) at quantification level j whenever the assignment $\tau_{\max}$ is returned by *SolveSSAT-*$\exists$ (Chen, Huang, and Jiang 2021). Under the clause selection framework, C_L consists of disjunctions of local selection literals. Several pruning techniques are dedicated to improve the quality of C_L since the performance of `ClauSSat` largely depends on the length of C_L. Here we assume that C_L is the final learned clause strengthened by those techniques. Algorithm 5 can be interpreted as follows. Given the global selection status S^{j-1} at level $j-1$, the maximum probability can be achieved by $\tau_{\max}$, which is an assignment over X_{j+1} under the assumption of the Boolean space blocked by C_L. By adding $S^{j-1} \wedge \neg C_L$ into the onset/offset of the existential variables according to $\tau_{\max}$, Algorithm 5 can correctly construct the Skolem functions for X_{j+1}.

Proposition 2. *`ClauSSat` with Algorithm 5 generates a set of Skolem functions as a witness of the obtained satisfying probability.*

The validity of Proposition 2 is explained as follows. To generate the Skolem functions, Algorithm 5 has to ensure that the Boolean space that the Skolem functions already defined on must be the same as that explored by `ClauSSat` during solving. For two-level SSAT formula with random-exist prefix, $\neg C_L$ exactly characterizes the newly explored Boolean space at the randomized quantification (Line 1). For general SSAT formula, the Boolean space has to be strengthened by conjunction with the assumptions (the selection status at the previous level, S^{j-1}) that lead to the current induced formula as shown in line 3. With the newly explored Boolean space λ, the onset or offset of the Skolem functions can be updated by conjunction with λ (Lnes 6 – 9) according to the phase of each literal appearing in $\tau_{\max}$.

6 Experimental Evaluation

We implemented the SSAT solver, named `SharpSSAT`[1] , in *C++* based on the model counter `SharpSAT` (Thurley 2006). The witness generation algorithms were integrated into `SharpSSAT` and `ClauSSat`. All experiments were conducted on a Linux machine with 2.2 GHz Intel Xeon CPU and 128 GB RAM. A time limit of 1000 seconds and a 64GB memory limit were imposed on solving an instance.

We compared our solver with the state-of-the-art SSAT solvers `DC-SSAT`, `ClauSSat`, `Prime`, and `ElimSSAT`. The benchmark set used in `ClauSSat`, with 20 families in total of 357 SSAT instances available at https://github.com/NTU-ALComLab/ClauSSat, was taken for evaluation. Among the 20 families (listed in Table 1), 8 of them are two-level quantified formulas (listed in the lower part of Table 1) and the other 12 are multi-level quantified instances (listed in the upper part of Table 1). The detailed descriptions and statistics of the benchmarks are omitted and can be found in (Chen, Huang, and Jiang 2021). Except for `SharpSSAT`, we turned on all the optimization options for all the other solvers. We let the default option of `SharpSSAT` contain component analysis, caching, pure-literal detection, and clause learning, and use `-p` and `-l` options to disable pure-literal detection and disable clause learning, respectively. The cactus plot of `SharpSSAT` and other solvers is shown in Fig. 2.

[1] Available at https://github.com/NTU-ALComLab/SharpSSAT

Family	total	SharpSSAT -pl	SharpSSAT -l	SharpSSAT default	Prime	DC-SSAT	ClauSSat	ElimSSAT
tlc	13	**13**	**13**	**13**	**13**	**13**	**13**	**13**
gttt_3x3	9	**9**	**9**	**9**	**9**	**9**	**9**	0
ev-pr-4x4	7	**7**	**7**	**7**	4	0	1	1
arbiter	10	0	0	0	0	0	0	0
Tree	14	**14**	**14**	**14**	**14**	**14**	**14**	**14**
Robot	10	3	4	5	4	3	**10**	5
Planning-CTE	18	0	0	0	0	0	0	**2**
Counter	8	6	**8**	**8**	6	4	4	**8**
Connect2	16	**16**	**16**	**16**	9	**16**	11	2
Adder	6	4	**5**	**5**	**5**	4	**5**	4
k_branch_n	10	**10**	**10**	**10**	3	2	1	2
k_ph_p	4	**4**	**4**	**4**	3	3	2	**4**
conformant	24	2	2	3	2	2	1	**6**
Tiger	5	**5**	**5**	**5**	4	**5**	2	**5**
ToiletA	77	59	63	64	59	42	47	**77**
sand-castle	25	23	23	23	20	**24**	11	14
MaxCount	25	10	**11**	**11**	9	4	1	10
MPEC	8	4	4	4	4	4	1	**8**
PEC	8	3	4	4	3	0	3	**7**
stracomp	60	16	16	28	14	28	**60**	44
all	357	208	218	**232**	186	177	196	226

Table 1: Performance comparison in the number of solved instances.

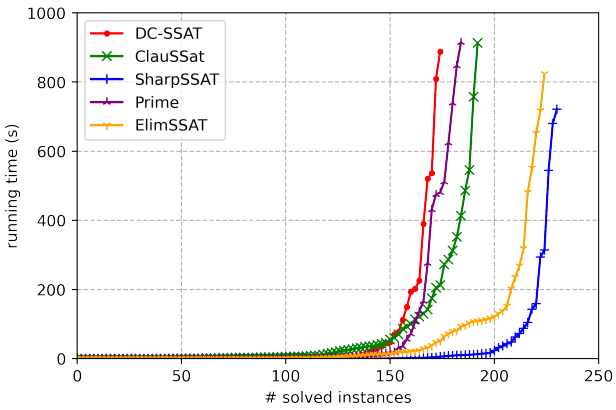

Figure 2: Number of solved instances within run-time limit.

Evaluation of Solver Performance

In the first experiment, we examine the number of solved instances in each family. The statistics are listed in Table 1. The largest number of solved instances in each row is marked in bold. By comparing `SharpSSAT -p` (solving 218 in total) with `SharpSSAT -pl` (solving 208), the effectiveness of clause learning is seen as it improves in solving several families. By comparing `SharpSSAT` (solving 232) with `SharpSSAT -p`, the effectiveness of pure-literal detection is especially significant in family `stracomp`, where there are several long clauses containing only one randomized literal and other positive-phase existential literals, and these clauses can be eliminated when the existential literals are detected as pure.

By comparing `SharpSSAT` with other DPLL-based solvers, `Prime` and `DC-SSAT`, it is seen that `SharpSSAT` solves the most instances in almost all families, except for the `sand-castle` family. The superiority of `SharpSSAT` to `Prime` is expected as `Prime`, which only applies component analysis and caching, neglects the implication power strengthened by clause learning. We also noticed that `Prime` selects the branch-literal blindly, i.e., it always chooses the first occurred literal in the component to branch. This strategy might be one of the reasons to blame for its sub-optimal performance as the design of the branching heuristic has a great impact on the performance of such DPLL-based solvers (Sang, Beame, and Kautz 2005). On the other hand, although `DC-SSAT` can deal with general SSAT formulas, it is originally designed for probabilistic planning problems. From the results, we see that `SharpSSAT` outperforms `DC-SSAT` in those families that are not planning problems. For the planning families, such as `stracomp`, `sand-castle`, and `Tiger`, `SharpSSAT` exhibits quite similar strength to `DC-SSAT`, suggesting that the decomposability of the planning problem exploited by `DC-SSAT` can also be detected through component analysis.

Next, we compare `SharpSSAT` with the abstraction-based solver `ClauSSat`. `SharpSSAT` can solve 75 cases that are not solvable by `ClauSSat` and `ClauSSat` can solve 38 cases that are not solvable by `SharpSSAT`. When inspecting the number of solved instances in each family, we noticed that `SharpSSAT` outperforms `ClauSSat` in most families, while `ClauSSat` performs especially well in the `Robot` and `stracomp` families, where both of them consist of two-level formulas with the random-exist prefix.

Configuration	# solved
Random	215
VSIDS	232
VSADS	232

Table 2: Comparison of branching heuristics in SharpSSAT.

It is observed that when a component consists of only existential literals, called an *existential component*, the SSAT problem of such components reduces to an SAT problem. However, the current implementation of `SharpSSAT` is not aware of such existential components, and it will insist on component analysis instead, which might be too expensive for an SAT problem. We believe that it would be helpful as our future work for `SharpSSAT` to more efficiently solve these random-exist formulas if the probability of existential components can be determined more effectively, e.g., by *conflict-driven-clause-learning* (CDCL).

Lastly, we compare `SharpSSAT` with `ElimSSAT`. Note that because `ElimSSAT` requires probability values representable in binary fractions, it rewrites SSAT formulas to approximate probability values with four-bit precision by default. Therefore, the comparison is not on an equal footing but for a reference. It can be seen that `SharpSSAT` performs slightly better than `ElimSSAT` in the total number of solved instances. When inspecting by family, we observed that `SharpSSAT` outperforms `ElimSSAT` in most multi-level quantified families and `ElimSSAT` solves more instances in the two-level quantified families. The results suggest that `SharpSSAT` and `ElimSSAT` exhibit different capabilities in solving multi-level and two-level quantified formulas.

Evaluation of SharpSSAT Branching Heuristics

Since `SharpSSAT` is a DPLL-based solver, the branching heuristics are critical to its performance. In this experiment, we empirically study the effectiveness of different branching heuristics on SSAT solving. The default branching heuristics of `SharpSSAT` is the *variable state aware decaying sum* (VSADS) (Sang, Beame, and Kautz 2005) that the score for each variable is calculated by

$$\text{score}(VSADS) = p \cdot \text{score}(VSIDS) + q \cdot \text{score}(DLCS),$$

where p and q are constants weighting the scores of VSIDS *variable state independent decaying sum* (VSIDS) (Mahajan, Fu, and Malik 2004) and *dynamic largest combined sum* (DLCS) (Sang, Beame, and Kautz 2005). The score functions of different heuristics can be found in (Sang, Beame, and Kautz 2005). We tested `SharpSSAT` with different configurations of branching strategies, namely *Random* ($p = 0$, $q = 0$), *VSIDS* ($p = 1$, $q = 0$), and *VSADS* ($p = 1$, $q = 1$). The results are reported in Table 2. From Table 2, we observe that the branching heuristics can indeed play a significant role in SSAT solving by comparing *Random* with *VSIDS*. On the other hand, we see no significant impact of the DLCS score on SSAT solving.

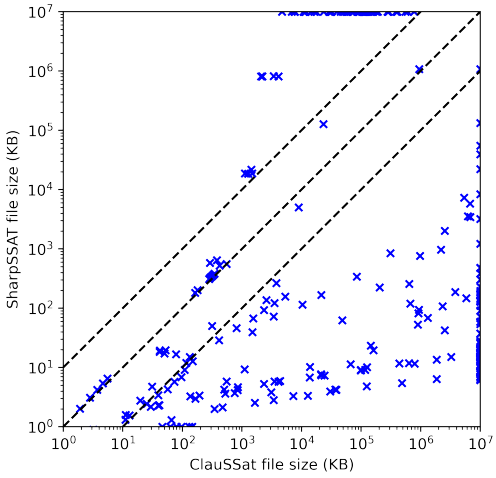

Figure 3: Witness size comparison in text file size.

Evaluation of Witness Generation

In this experiment, we evaluate the overhead of witness generation under `SharpSSAT` and `ClauSSAT` and the quality of the generated witnesses. For both solvers, the generated witness circuits are written in the BLIF format (Sentovich et al. 1992). Let `-k` denote the witness generation option. `SharpSSAT -k` solved all the instances solved by `SharpSSAT` while `ClauSSat -k` solved 10 less instances than `ClauSSat`. For memory usage, `SharpSSAT -k` typically consumed twice the memory used by `SharpSSAT` while `ClauSSat -k` remained the same as `ClauSSat`. We further verify each instance using `Cachet` (Sang et al. 2004) under a time limit of 300 seconds. For `SharpSSAT -k` (resp. `ClauSSat -k`), 3 (resp. 21) out of the solved instances cannot be verified within this time limit.

To evaluate the quality of the witness, we synthesized the generated witness into AIG using `ABC` (Brayton and Mishchenko 2010). The synthesized witness is further optimized with the `ABC` command `dc2`. The witness comparison in the BLIF file size and the synthesized AIG size are shown in Fig. 3 and Fig. 4, respectively. In Fig. 3 (resp. Fig. 4), the horizontal and vertical 10^7 (resp. 10^8) lines correspond to timeouts for `SharpSSAT` and `ClauSSat`, respectively.

The horizontal axis corresponds to the file size (in KB) or the AIG size (in AIG node count) of the witness generated by `ClauSSat -k`, and the vertical axis corresponds to those generated by `SharpSSAT -k`. As can be seen, in both figures, most of the spots lie in the bottom right half-plane, suggesting that `SharpSSAT-k` tends to produce smaller witnesses in most cases.

In summary, `SharpSSAT -k` incurs very little runtime overhead and tends to generate a witness with higher quality when compared to `ClauSSat -k`. On the other hand, the memory usage of `SharpSSAT -k` is higher than

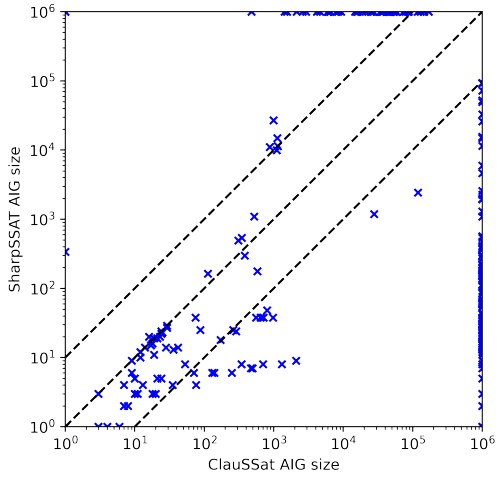

Figure 4: Witness size comparison in AIG node count.

`ClauSSat -k`. This is not surprising since the strategy of `ClauSSat -k` and `SharpSSAT -k` are pretty different. The witness construction of `ClauSSat -k` is bottom-up while `SharpSSAT-k` is more like a top-down approach. Therefore, `SharpSSAT -k` can easily prune irrelevant information, such as the smaller branch of existential variable, while `ClauSSat -k` can not. We observed that the redundancy in the witness generated by `ClauSSat -k` is the reason why the AIG synthesis step closes the gap between `SharpSSAt -k` and `ClauSSat -k` when comparing Fig. 3 with Fig. 4, as the synthesis step may exploit the redundancy and then produce a smaller witness. Such difference also leads to the trade-off between run-time and memory. In order to have global information of witness, `SharpSSAT` maintains a trace until the end of solving, which may significantly increase the memory usage. On the other hand, with the trace, `SharpSSAT` can efficiently construct the witness by traversing the trace once.

7 Conclusions and Future Work

In this work, we have implemented `SharpSSAT`, a new SSAT solver extending model counting techniques to SSAT solving. Also, we have equipped `SharpSSAT` and `ClauSSat` with witness generation capability. Experimental results have demonstrated the superiority of `SharpSSAT` compared to state-of-the-art SSAT solvers.

For future work, we plan to further improve `SharpSSAT` with existential component detection and explore techniques in *knowledge compilation* (Darwiche 2004) to benefit `SharpSSAT`.

Acknowledgements

The authors are grateful to Ricardo Salmon for his technical assistance about using `Prime`. This work was supported in part by the National Science and Technology Council of Taiwan under Grant 111-2221-E-002-182 and Grant 111-2923-E-002-013-MY3.

References

Brayton, R.; and Mishchenko, A. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, 24–40.

Chen, P.-W.; Huang, Y.-C.; and Jiang, J.-H. R. 2021. A Sharp Leap from Quantified Boolean Formula to Stochastic Boolean Satisfiability Solving. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 3697–3706.

Darwiche, A. 2004. New Advances in Compiling CNF to Decomposable Negation Normal Form. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, 328–332.

Freudenthal, E.; and Karamcheti, V. 2003. QTM: Trust Management with Quantified Stochastic Attributes. Technical Report TR 2003-848, CS Department, New York University.

Gent, I.; Giunchiglia, E.; Narizzano, M.; Rowley, A.; and Tacchella, A. 2003. Watched Data Structures for QBF Solvers. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 25–36.

Ghosh, B.; Basu, D.; and Meel, K. S. 2021. Justicia: A Stochastic SAT Approach to Formally Verify Fairness. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 7554–7563.

Janota, M.; and Marques-Silva, J. 2015. Solving QBF by Clause Selection. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 325–331.

Lee, N.-Z.; and Jiang, J.-H. R. 2018. Towards Formal Evaluation and Verification of Probabilistic Design. *IEEE Transactions on Computers (TCOMP)*, 67(8): 1202–1216.

Littman, M. L.; Majercik, S. M.; and Pitassi, T. 2001. Stochastic Boolean Satisfiability. *Journal of Automated Reasoning (JAR)*, 27(3): 251–296.

Mahajan, Y. S.; Fu, Z.; and Malik, S. 2004. Zchaff2004: An Efficient SAT Solver. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 360–375.

Majercik, S. M.; and Boots, B. 2005. DC-SSAT: A Divide-and-Conquer Approach to Solving Stochastic Satisfiability Problems Efficiently. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 416–422.

Marques-Silva, J. P.; and Sakallah, K. A. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers (TCOMP)*, 48(5): 506–521.

Salmon, R.; and Poupart, P. 2019. On the Relationship between Stochastic Satisfiability and Partially Observable Markov Decision Processes. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 1–407.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 20–28.

Sang, T.; Beame, P.; and Kautz, H. 2005. Heuristics for Fast Exact Model Counting. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 226–240.

Sentovich, E.; Singh, K.; Lavagno, L.; Moon, C.; Murgai, R.; Saldanha, A.; Savoj, H.; Stephan, P.; Brayton, R. K.; and Sangiovanni-Vincentelli, A. L. 1992. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley.

Thurley, M. 2006. sharpSAT–Counting Models with Advanced Component Caching and Implicit BCP. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 424–429.

Wang, H.-R.; Tu, K.-H.; Jiang, J.-H. R.; and Scholl, C. 2022. Quantifier Elimination in Stochastic Boolean Satisfiability. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 1–17.