Solving Stochastic Boolean Satisfiability under Random-Exist Quantification

Nian-Ze Lee[†], Yen-Shi Wang[‡], and Jie-Hong R. Jiang[†,‡]
[†]Graduate Institute of Electronics Engineering, [‡]Department of Electrical Engineering
National Taiwan University, Taipei, Taiwan
{d04943019, b03901116, jhjiang}@ntu.edu.tw

Abstract

Stochastic Boolean Satisfiability (SSAT) is a powerful formalism to represent computational problems with uncertainty, such as belief network inference and propositional probabilistic planning. Solving SSAT formulas lies in the PSPACE-complete complexity class same as solving Quantified Boolean Formulas (QBFs). While many endeavors have been made to enhance QBF solving in recent years, SSAT has drawn relatively less attention. This paper focuses on random-exist quantified SSAT formulas, and proposes an algorithm combining modern satisfiability (SAT) techniques and model counting to improve computational efficiency. Unlike prior exact SSAT algorithms, the proposed method can be easily modified to solve approximate SSAT by deriving upper and lower bounds of satisfying probability. Experimental results show that our method outperforms the state-of-the-art algorithm on random k-CNF and AI-related formulas in both runtime and memory usage, and has effective application to approximate SSAT on VLSI circuit benchmarks.

1 Introduction

First formulated in [Papadimitriou, 1985], *stochastic Boolean satisfiability* (SSAT) was described as a game against nature. Probabilistic factors are introduced into the world of Boolean logic through the creation of the *randomized quantifier*. A Boolean variable randomly quantified with probability p has a probability p of evaluating to TRUE. Exploiting randomized quantifiers, SSAT is capable of modeling a variety of computational problems inherent with uncertainty [Hnich *et al.*, 2011], such as propositional probabilistic planning, belief network inference, and trust management [Majercik, 2009]. Recently, it has also been applied to the formal verification of probabilistic circuits [Lee and Jiang, 2014]. From computational complexity point of view, SSAT lies in the PSPACE-complete complexity class, same as quantified Boolean satisfiability (QSAT). Therefore, developing good algorithms for SSAT not only brings practical benefits for real-world applications, but also is of theoretical interest.

Among the prior efforts made to approach SSAT, most of them are based on Davis-Putnam-Logemann-Loveland (DPLL) search [Davis *et al.*, 1962]. MAXPLAN [Majercik and Littman, 1998] improves DPLL-based search by considering pure literal, unit propagation, and subproblem memorization. ZANDER [Majercik and Littman, 2003] incorporates several threshold pruning heuristics to reduce the search space. DC-SSAT [Majercik and Boots, 2005] divides the SSAT formulas into many subproblems and conquer them separately. The solution of the original formula is then constructed on top of the solutions of subproblems. *Approximate SSAT* solving, which derives upper or lower bounds of satisfying probability of an SSAT formula instead of solving it exactly, was investigated in [Majercik, 2007]. Resolution for SSAT has also been addressed in [Teige and Fränzle, 2010].

Model counting algorithms are under active development in recent years. In addition to *exact model counting* [Sang *et al.*, 2004; 2005a], *approximate model counting* [Gomes *et al.*, 2006; 2007; Chakraborty *et al.*, 2016] has been addressed to improve scalability by relaxing exactness. In this work, exact or approximate model counting is exploited as an ingredient for SSAT solving. On the other hand, *projected model counting* [Aziz *et al.*, 2015], which computes the numbers of satisfying assignments projected on a subset of original variables, is subsumed by random-exist quantified SSAT. Therefore, our proposed method can be applied to solve projected model counting.

In contrast to the prior DPLL-based SSAT algorithms with pruning heuristics and subproblem memorization, our goal is to take advantage of modern SAT solvers [Eén and Sörensson, 2003a; 2003b] and model counters as a black-box engine for solving SSAT. To the best of our knowledge, this work is the first attempt to directly use a SAT solver as a plug-in engine for SSAT solving. In this paper, we focus on the random-exist quantified SSAT formula of the form $\Phi = \exists X \exists Y.\phi$, which is the counterpart of the forall-exist quantified QBF formula. The random-exist quantified SSAT has applications in belief network inference [Cooper, 1990; Bacchus *et al.*, 2003], probabilistic planning, and probabilistic circuit verification [Lee and Jiang, 2014]. In addition to SAT solving, we incorporate weighted model counting, which has been widely used in probabilistic inference [Sang *et al.*, 2005b; Chavira and Darwiche, 2008], in the proposed algorithm to tackle randomized quantifiers. The proposed al-

gorithm uses the SAT solver and model counter in a *stand-alone* manner, leaving the internal structures of these solvers intact. Due to the stand-alone usage of these solvers, the proposed algorithm may directly benefit from the advancement of the solvers without any modification. Moreover, unlike previous DPLL-based methods, the proposed algorithm can be easily modified to solve *approximate SSAT* by deriving upper and lower bounds of satisfying probability.

To evaluate the proposed algorithm, a prototype program was implemented and experimented with random k-CNF formulas, *strategic companies* [Cadoli *et al.*, 1997] formulas arising from the artificial intelligence (AI) domain, and benchmarks from analysis of very-large-scale integration (VLSI) circuits. Our method outperforms the state-of-the-art algorithm on random k-CNF and strategic companies formulas in both runtime and memory consumption, and is capable of solving approximate SSAT on circuit benchmarks while the state-of-the-art method fails to compute the exact satisfying probability.

2 Preliminaries

Let $\mathbb{B} = \{\top, \bot\}$, where $\top$ and $\bot$ denote logic TRUE and FALSE. A *literal* is a Boolean variable or its negation. A *clause* (resp. *cube*) is a disjunction (resp. conjunction) of literals. A *conjunctive normal form* (CNF) formula is a conjunction of clauses. A Boolean formula ϕ over variables $X = \{x_1, \ldots, x_n\}$ induces a Boolean function mapping from $\mathbb{B}^n$ to $\mathbb{B}$. The set of Boolean variables appearing in a Boolean formula ϕ is denoted as $\texttt{vars}(\phi)$. An *assignment* τ on variables $X \subseteq \texttt{vars}(\phi)$ of formula ϕ is a mapping from X to $\mathbb{B}$. An assignment τ is called a *complete assignment* on $\texttt{vars}(\phi)$ if $X = \texttt{vars}(\phi)$; otherwise, it is called a *partial assignment* on $\texttt{vars}(\phi)$. The formula of ϕ induced under the assignment τ on $X \subseteq \texttt{vars}(\phi)$ is obtained by substituting every appearance of $x \in X$ in ϕ by $\tau(x)$, and is denoted as $\phi|_\tau$. A complete assignment τ is called a *satisfying* (resp. an *unsatisfying*) *(complete) assignment* for ϕ if $\phi|_\tau = \top$ (resp. $\phi|_\tau = \bot$). Similarly, a partial assignment τ^+ on $X \subset \texttt{vars}(\phi)$ for ϕ is called a *satisfying* (resp. an *unsatisfying*) *(partial) assignment* on $\texttt{vars}(\phi)$ if for some (resp. every) assignment μ on $\texttt{vars}(\phi) \setminus X$, ϕ valuates to $\top$ (resp. $\bot$) under the complete assignment obtained by combining τ and μ. In the sequel, we alternatively represent an assignment τ for ϕ as a cube. A cube is called a *minterm* when it corresponds to a complete assignment with respect to a specified set of variables. A Boolean formula ϕ is called *satisfiable* if there exists a satisfying complete assignment for ϕ. We write $\texttt{SAT}(\phi) = \top$ to denote ϕ is satisfiable. A satisfying assignment of ϕ is also called a *model* of ϕ, denoted by $\phi.\mathrm{model}$. On the other hand, if ϕ has no satisfying assignment, it is unsatisfiable and written as $\texttt{SAT}(\phi) = \bot$.

2.1 Generalization of SAT or UNSAT Assignments

Consider a CNF formula $\phi(X, Y)$, where X and Y are two disjoint sets of Boolean variables. Given an assignment τ on X, if $\phi(X, Y)|_\tau$ is satisfiable (resp. unsatisfiable), τ is called a SAT (resp. an UNSAT) minterm of ϕ on X. The generalization process of a SAT or an UNSAT minterm τ aims at expanding it to a cube τ^+, while maintaining the satisfiability of $\phi(X, Y)|_{\tau^+}$ the same as $\phi(X, Y)|_\tau$.

Example 1 *Consider formula* $\phi(x_1, x_2, y_1, y_2) = x_1 \wedge (\neg x_2 \vee y_1 \vee y_2)$. *The assignment* $\tau = x_1x_2$ *on* X, *i.e.,* $\tau(x_1) = \top, \tau(x_2) = \top$, *is a satisfying assignment, or a SAT minterm, of* ϕ *on* X *as* $\phi|_\tau$ *is satisfiable by assignment* $\mu = y_1y_2$. *On the other hand, the partial assignment* $\tau^+ = \neg x_1$, *i.e.,* $\tau^+(x_1) = \bot$, *is an unsatisfying partial assignment, or an UNSAT cube, of* ϕ *as* $\phi|_{\tau^+}$ *is unsatisfiable.*

Minimum Satisfying Assignment

For a CNF formula $\phi(X, Y)$, let τ be a SAT minterm on X and let μ be a satisfying assignment for $\phi(X, Y)|_\tau$ on Y. To generalize τ into a cube, one can find a subset of literals from τ and μ that are able to satisfy all clauses in ϕ while the number of literals taken from τ is as few as possible. If some literals in τ are irrelevant to the satisfiability, they can be dropped from τ, thus expanding τ to a SAT cube τ^+. If in the SAT cube τ^+, the number of literals taking from τ is minimized, τ^+ is called the *minimum satisfying assignment*. The process of finding the minimum satisfying assignment is also known as finding the *minimum hitting set*.

Minimum Conflicting Assignment

Given an UNSAT minterm τ of a CNF formula ϕ, modern SAT solvers, such as `Minisat` [Eén and Sörensson, 2003a; 2003b], are able to analyze the reason of unsatisfiability, which is represented as a conjunction of literals in τ causing the conflict. If some literals in τ are irrelevant to the conflict, they are dropped from τ, thus expanding τ to an UNSAT cube τ^+. If the number of literals in the UNSAT cube τ^+ is minimized, τ^+ is called the *minimum conflicting assignment*. The process of finding the minimum conflicting assignment is also known as finding the *minimum UNSAT core*.

2.2 Model Counting

Given a CNF formula ϕ, the *model counting* problem finds the number of satisfying assignments of ϕ. In its weighted version, a weight function ω maps each Boolean variable $x \in \texttt{vars}(\phi)$ to a *weight* $\omega(x) \in [0, 1]$, which represents $\Pr[x = \top]$. The weight of a negative literal $\neg x$ is defined to be $1 - \omega(x)$. The weight of an assignment equals the product of the weights of its individual literals. The weight of a Boolean formula equals the summation of weights of its satisfying assignments. There are two categories of model counting algorithms: *Exact model counting* computes the precise number of satisfying assignments of a formula; *approximate model counting* computes the upper or lower bounds of the number of satisfying assignments of a formula with some confidence level.

2.3 Stochastic Boolean Satisfiability

A *stochastic Boolean satisfiability* (SSAT) formula is of the form

$$\Phi = Q_1x_1 \ldots Q_nx_n.\phi,$$

where $Q_i \in \{\exists, \text{ᗺ}^{p_i}\}$ and ϕ is a quantifier-free Boolean formula. In addition to the existential quantifier $\exists$, the randomized quantifier ᗺ^{p_i} on x_i assigns a probability $p_i \in [0, 1]$ for x_i to be true. The quantifier part is called the *prefix*, and the

Boolean formula is called the *matrix*. Given an SSAT formula Φ, let v be the outermost variable in the prefix. The satisfying probability of Φ can be computed by the following rules.

a) $\Pr[\top] = 1$,

b) $\Pr[\bot] = 0$,

c) $\Pr[\Phi] = \max\{\Pr[\Phi|_{\neg v}], \Pr[\Phi|_v]\}$, if v is existentially quantified,

d) $\Pr[\Phi] = (1-p)\Pr[\Phi|_{\neg v}] + p\Pr[\Phi|_v]$, if v is randomly quantified by $\exists^p$,

where $\Phi|_{\neg v}$ and $\Phi|_v$ denote the SSAT formulas derived by eliminating the outermost quantifier of v by substituting the value of v in the matrix with $\bot$ and $\top$, respectively. In this paper, we aims at solving the *random-exist quantified SSAT formula* of the form $\Phi = \exists X \exists Y.\phi(X,Y)$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ are two disjoint sets of Boolean variables.

3 Solving Random-Exist SSAT

Consider a random-exist quantified SSAT formula $\Phi = \exists X \exists Y.\phi(X,Y)$. The satisfying probability of Φ equals the summation of weight of all SAT minterms on X, or, equivalently, 1 minus the summation of weight of all UNSAT minterms on X. To identify an assignment τ on X as a SAT or an UNSAT minterm, it suffices to check whether $\phi(X,Y)|_\tau$ is satisfiable or not. A naive solution to computing the satisfying probability of Φ exhaustively examines all assignments on X, classifies them as SAT or UNSAT minterms, and aggregates the weight of collected minterms.

The above naive solution can be improved by exploiting the minterm-generalization techniques discussed in Section 2.1. For instance, in Example 1, $\tau = x_1x_2\neg x_3$ is a SAT minterm for $\phi(x_1,x_2,x_3) = x_1 \wedge (x_2 \vee x_3)$. Observe that $\phi(x_1,x_2,x_3)$ is satisfiable under the partial assignment $\tau^+ = x_1x_2$. In other words, the SAT minterm τ can be generalized into the SAT cube τ^+, which contains two minterms. Through the generalization analysis, more than one minterm can be collected in a single SAT solving run, enhancing the efficiency of the enumeration of all possible assignments on X. As will be shown in Section 4, the minterm-generalization techniques are essential to the efficiency of the proposed method. However, the weight of each collected cubes cannot be summed up directly due to the potential non-disjointness between the generalized cubes. This difficulty can be overcome by applying weighted model counting to evaluate the weight of the collected cubes.

The above idea gives rise to the proposed algorithm, formalized in Figure 1, to evaluate $\Phi = \exists X \exists Y.\phi(X,Y)$. The proposed algorithm works as follows. First assume the runtime limit `TO` to be infinity, while the effect of imposing a runtime limit on *SolveRESSAT* will be explained in Section 3.4. Two SAT solvers are used in *SolveRESSAT*. In addition to the SAT solver holding the matrix CNF $\phi(X,Y)$, the other SAT solver $\psi(X)$, called the selector in the sequel, is initialized with any empty set of clauses. The selector $\psi(X)$ is in charge of selecting an assignment τ on X. After τ is chosen, the matrix solver $\phi(X,Y)$ will check whether or not $\phi(X,Y)|_\tau$ is satisfiable. Depending on the satisfiability of $\phi(X,Y)|_\tau$, the minterm τ is generalized into a cube by the subroutine *MinimalSatisfying* or *MinimalConflicting*. Instead of finding the *minimum* satisfying or conflicting assignment, which is computationally expansive, we resort to finding a *minimal* satisfying or conflicting assignment, i.e., an assignment having no literals removable without affecting the (un)satisfiability, to leverage the efficiency of UNSAT core computation for effective generalization. After τ is generalized to τ^+ and enlisted in $C_\bot$ or $C_\top$, the negation of τ^+, which is a blocking clause, will be conjoined with ψ to prune the assignments contained by τ^+. The above process is repeated until ψ becomes unsatisfiable, which signifies the Boolean space spanned by X has been exhaustively searched. The subroutine *ComputeWeight* is then invoked to evaluate the weight of the collected cubes. The subroutines *MinimalConflicting*, *MinimalSatisfying*, and *ComputeWeight* are detailed below.

SolveRESSAT
input: $\Phi = \exists X \exists Y.\phi(X,Y)$ and a runtime limit `TO`
output: Upper and lower bounds (P_U, P_L) of satisfying prob.
begin
01 $\psi(X) := \top$;
02 $C_\top := \emptyset$;
03 $C_\bot := \emptyset$;
04 **while** $\texttt{SAT}(\psi) = \top \wedge runtime < \texttt{TO}$
05 $\tau := \psi$.model;
06 **if** $\texttt{SAT}(\phi|_\tau) = \top$
07 $\tau^+ :=$ *MinimalSatisfying*(ϕ, τ);
08 $C_\top := C_\top \cup \{\tau^+\}$;
09 **else** $//\texttt{SAT}(\phi|_\tau) = \bot$
10 $\tau^+ :=$ *MinimalConflicting*(ϕ, τ);
11 $C_\bot := C_\bot \cup \{\tau^+\}$;
12 $\psi := \psi \wedge \neg\tau^+$;
13 **return** (1−*ComputeWeight*($C_\bot$), *ComputeWeight*($C_\top$));
end

Figure 1: Algorithm: Solving random-exist quantified SSAT.

3.1 Minimal Satisfying Assignment

Given a SAT minterm τ on X for $\phi(X,Y)$, let μ be the satisfying assignment on Y for $\phi(X,Y)$. The subroutine *MinimalSatisfying* generalizes τ to τ^+ via the following steps.

a) Remove every clause C in $\phi(X,Y)|_\tau$ that contains some true literal in μ.

b) For each literal l in τ, drop l and examine whether the rest of clauses remain satisfied by scanning these clauses and checking if each of them still contains some true literal. If the rest of clauses are all satisfied, discard l; otherwise, put l in τ^+.

After the above steps, the SAT minterm τ is generalized into a minimal satisfying assignment τ^+.

3.2 Minimal Conflicting Assignment

Let τ be an UNSAT minterm on X for $\phi(X,Y)$. The analysis of unsatisfiability can be done with a modern SAT solver (e.g., using `analyzeFinal()` in `Minisat`) to find a conjunction of literals from τ responsible for the conflict. However, in general this conjunction of literals might not be minimal, meaning that some of the literals can be dropped. The

subroutine *MinimalConflicting* takes the conjunction of literals responsible for the conflict computed by a SAT solver and makes it minimal as follows. For each literal l in the conjunction, drop l and examine whether $\phi(X, Y)$ remains unsatisfiable by invoking a SAT call. If it is unsatisfiable, discard l; otherwise, put l in τ^+. After the above steps, the UNSAT minterm τ is generalized into a minimal conflicting assignment τ^+.

3.3 Weight Computation

The subroutine *ComputeWeight* aggregates the weight of collected cubes by invoking a weighted model counter. Because the weighted model counter takes CNF formulas as input, *ComputeWeight* first negates each collected cube to turn it into a clause, and conjoins the resulting clauses into a CNF formula. As the CNF formula is the negation of the disjunction of the cubes, the weight of the cubes equals one minus the weight of the CNF formula, which is computed by a weighted model counter.

3.4 Modification for Approximate SSAT

The proposed algorithm can be easily modified to solve *approximate SSAT*, where upper and lower bounds of the satisfying probability of an SSAT formula are computed. Suppose *SolveRESSAT* is forced to terminate before the selector ψ becomes unsatisfiable. The weights of the collected SAT and UNSAT cubes are still valid and can be aggregated by *ComputeWeight*, and the resulted weights reflect the lower and upper bounds of the satisfying probability, respectively. The early termination can be triggered by imposing a runtime limit for *SolveRESSAT*. Compared to previous DPLL-based methods that branch on a single variable, the proposed algorithm considers all randomly quantified variables together and exploits the concept of SAT and UNSAT cubes over the Boolean space spanned by randomly quantified variables, making the intermediate collected SAT and UNSAT cubes convey useful information about the upper and lower bounds of the exact satisfying probability. As will be seen in our experiments over formulas arising from VLSI circuit analysis, the proposed algorithm is able to find tight bounds even with a short time limit. Compared to the DPLL-based state-of-the-art methods, which cannot be easily modified for approximate SSAT, the proposed method enjoys the flexibility of solving SSAT approximately or exactly, depending on the imposed runtime constraint.

We note that the proposed algorithm is more efficient in memory consumption than previous DPLL-based algorithms. Prior DPLL-based algorithms mostly apply subproblem memorization to avoid repeated computation on the same subproblem. However, without special treatment, such memorization may result in rapid growth in memory usage. On the other hand, in the proposed algorithm, the numbers of collected cubes are greatly reduced by the aforementioned minterm generalization, which gives rise to the memory efficiency. In our empirical evaluation, the proposed algorithm consumed two orders of magnitude less memory than the state-of-the-art DPLL-based algorithm.

The following example illustrates how *SolveRESSAT* solves a random-exist quantified SSAT formula.

Example 2 *Consider the random-exist quantified SSAT formula* $\Phi = \text{ꓤ}^{0.5} r_1 r_2 r_3 \exists e_1 e_2 e_3 . \phi$*, with* ϕ *consisting of clauses*

$C_1 : (r_1 \vee r_2 \vee e_1)$
$C_2 : (r_1 \vee \neg r_3 \vee e_2)$
$C_3 : (r_2 \vee \neg r_3 \vee \neg e_1 \vee \neg e_2)$
$C_4 : (r_3 \vee e_3)$
$C_5 : (r_3 \vee \neg e_3)$

At the beginning, the selector $\psi(r_1, r_2, r_3)$ *is initialized to an empty set of clauses, and the sets* $C_\top$ *and* $C_\bot$ *for collecting SAT and UNSAT cubes are empty. Suppose* ψ *first selects the assignment* $\tau_0 = \neg r_1 \neg r_2 \neg r_3$*. Since* $\phi|_{\tau_0}$ *is unsatisfiable (due to the conflict between* C_4 *and* C_5*), the subroutine MinimalConflicting returns* $\tau_0^+ = \neg r_3$*, which is the minimal conflicting assignment responsible for this conflict. Note that this minimal conflicting assignment* τ_0^+ *reflects an upper bound of* 0.5 *for the satisfying probability of* Φ*. The selector* ψ *is then strengthened through conjunction with the negation of* τ_0^+ *to block the searched subspace. Next, suppose* $\tau_1 = \neg r_1 \neg r_2 r_3$ *is selected. Under* τ_1*, formula* $\phi|_{\tau_1}$ *is unsatisfiable due to the conflict among clauses* C_1*,* C_2*, and* C_3*, and the minimal conflicting assignment* τ_1^+ *equals* τ_1*. After conjoining* ψ *with* $\neg\tau_1^+$*, suppose* $\tau_2 = \neg r_1 r_2 r_3$ *is chosen. Formula* $\phi|_{\tau_2}$ *is satisfiable through the assignment* $\mu_0 = \neg e_1 e_2 \neg e_3$*. The subroutine MinimalSatisfying is invoked to generalize* τ_2 *to* $\tau_2^+ = r_2 r_3$*, which reflects a lower bound of* 0.25 *for the satisfying probability of* Φ*. Similarly, the negation of* τ_2^+ *is conjoined with* ψ*. Next, let the assignment chosen by* ψ *be* $\tau_3 = r_1 \neg r_2 r_3$*. Since* $\phi|_{\tau_3}$ *is satisfiable through the assignment* $\mu_1 = \neg e_1 \neg e_2 \neg e_3$*, assignment* τ_3 *is generalized to* $\tau_3^+ = r_1 r_3$ *by MinimalSatisfying. After conjoined with* $\neg\tau_3^+$*, formula* ψ *becomes unsatisfiable, which indicates the Boolean space over* $\{r_1, r_2, r_3\}$ *has been explored exhaustively. At the end, we have* $C_\bot = \{\tau_0^+, \tau_1^+\} = \{\neg r_3, \neg r_1 \neg r_2 r_3\}$ *and* $C_\top = \{\tau_2^+, \tau_3^+\} = \{r_2 r_3, r_1 r_3\}$*. The subroutine ComputeWeight is finally invoked and returns* 0.375 *as the satisfying probability of* Φ*.*

4 Experimental Results

Our solver, named `reSSAT`, was implemented in the `C++` language. The experiments were conducted on a Linux machine with Intel Xeon 2.1 GHz CPU and 126 GB RAM. We compared the proposed approach with the state-of-the-art DPLL-based SSAT solver `DC-SSAT`. The SAT solver `Minisat-2.2` and model counter `Cachet` were used in our program as underlying computational engines. (We tried approximate model counter `ApproxMC` [Chakraborty *et al.*, 2013] and observed that `ApproxMC` outperformed `Cachet` mainly on difficult instances while `Cachet` tended to perform better on other instances. As our benchmarks favored `Cachet`, it was used in our experiments.) Three sets of formulas, including random k-CNF, strategic companies, and probabilistic equivalence checking formulas, were used to evaluate our solver against `DC-SSAT`, abbreviated as `Dc` in the sequel. A basic version of `reSSAT` without minterm generalization is denoted as `reSSAT-b`. A runtime limit of 1000 seconds was imposed on each formula. No memory limit was

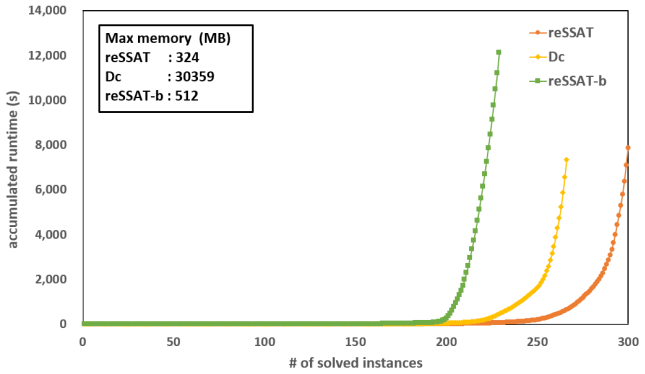

Figure 2: Results of selected random SSAT formulas.

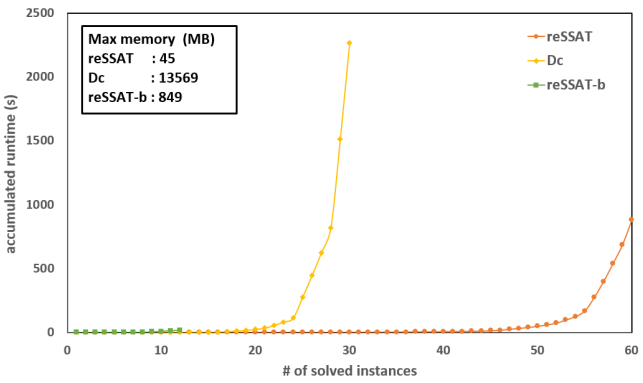

Figure 3: Results of strategic companies SSAT formulas.

imposed, but peak memory usage during execution was examined.

4.1 Random k-CNF Formulas

The random k-CNF formulas are generated using the `cnfgen` command [Lauria, 2012]. Let k be the number of literals in a clause, n be the number of variables, and m be the number of clauses. A collection of 700 CNF formulas is generated with the following parameter settings. Let k range from 3 to 9, n equal 10, 20, 30, 40, 50, and the clauses-to-variables ratio $\frac{m}{n}$ range from $k-1$ to $k+2$. For each combination of parameters, five formulas were generated. To convert the generated CNF formulas into random-exist quantified SSAT formulas, half of the variables in a CNF formula are randomly quantified with probability 0.5, and the rest of the variables are existentially quantified. (Note that a random-exist SSAT formula with randomized quantifiers of arbitrary probabilities can be converted to one with only randomized quantifiers of probability 0.5 through rewriting using auxiliary variables [Lee and Jiang, 2014].) Out of the 700 generated formulas, we selected 300 formulas with their satisfying probabilities evenly distributed in $[0, 1]$ for evaluation.

The results are shown in Figure 2. Without minterm generalization, `reSSAT-b` only solved 229 formulas within the runtime limit of 1000 seconds, while `Dc` solved 266 formulas. On the other hand, `reSSAT` solved all 300 formulas, showing that the minterm-generalization technique is essential to the efficiency of `reSSAT`. Specifically, the number of collected cubes with minterm generalization is on average 8 times smaller than that without the generalization, and the runtime is reduced by an order of magnitude. Compared to `Dc`, `reSSAT` achieved an average runtime improvement by 5 times, and a maximum peak memory reduction by 2 orders of magnitude.

4.2 Planning Formulas

Many planning problems can be formulated in terms of forall-exist quantified QBFs, i.e., QBFs of the form $\Phi = \forall X \exists Y . \phi(X, Y)$. By changing universal quantifiers to randomized ones, random-exist SSAT formulas can be obtained. In essence, under the game interpretation of QBFs, the satisfying probability of these SSAT formulas corresponds to the likelihood for the existential player to win the QBF game when the universal player plays her/his moves at random. We take the *strategic companies* problem, which was defined in [Cadoli *et al.*, 1997], as an example to evaluate the performance of SSAT algorithms on planning applications.

The *strategic companies problem* can be briefly described as follows. Suppose a businessman owns n companies that produce m kinds of products. A company is *strategic* if it is in a minimal set of companies that together produce all kinds of products. The information of whether a company is strategic is valuable to the businessman. Suppose the businessman encounters a crisis and considers to sell out some companies, but hopes to keep producing every kind of products. As a result, he may prefer to sell out a non-strategic company. The problem is more complicated by taking the *controlling relations* into account. If a company is *controlled* by other companies, it means that the company can be sold out only if some of its controlling companies is also sold out. The problem to decide whether a company is strategic or not can be encoded as forall-exist quantified QBFs [Faber and Ricca, 2005; Leone *et al.*, 2006].

We modify the QBFs of the strategic companies problem, taken from `QBFLIB` [Giunchiglia *et al.*, 2005], to their SSAT variants by changing the universal quantifiers in the original QBFs to randomized ones with probabilities 0.5. The satisfying probability reflects the likelihood for a company to be strategic. The QBFs we experimented with describe the strategic companies problems under the following setting of parameters: n equals $5, 10, 15, \ldots, 75$, $m = 3n$, and the number of controlling relations equals $4, 9, 14, 19$.

The results of the 60 formulas are plotted in Figure 3. While `reSSAT-b` only solved 12 formulas, `Dc` solved 30 formulas. Enabling the minterm-generalization technique makes `reSSAT` solve all 60 formulas. The results show that the technique worked well not only on random formulas, but also on structured instances from AI applications. Compared to `Dc`, `reSSAT` achieved an average runtime improvement by 2 orders of magnitude, and a maximum peak memory reduction by 2 orders of magnitude as well.

4.3 Probabilistic Equivalence Checking Formulas

Another SSAT application, formulated in [Lee and Jiang, 2014], is formal verification of *probabilistic circuits*. As probabilistic errors are becoming more common in advanced

Table 1: Results of solving SSAT formulas from probabilistic equivalence checking.

	formula statistics				reSSAT (TO=60s)			reSSAT (TO=1000s)			Dc (TO=1000s)	
circuit	#Var	#Cla	#Ran	Ans.	runtime (s)	UB	LB	runtime (s)	UB	LB	runtime (s)	Prob.
c432	330	879	37	1.03E-02	TO	1.07E-02	4.30E-05	TO	1.05E-02	8.50E-05	TO	TO
c499	217	522	42	1.56E-13	0.32	1.56E-13	1.56E-13	0.13	1.56E-13	1.56E-13	0.00	1.56E-13
c880	451	1167	62	4.18E-02	TO	9.78E-02	3.00E-06	TO	8.18E-02	3.00E-06	TO	TO
c1355	771	2181	44	6.41E-02	TO	3.20E-01	0.00E+00	TO	3.08E-01	0.00E+00	TO	TO
c1908	270	705	35	7.38E-04	TO	8.83E-04	4.00E-05	TO	7.38E-04	7.90E-05	210.86	7.38E-04
c3540	321	807	52	1.71E-03	TO	1.17E-02	5.03E-04	TO	1.17E-02	1.61E-03	217.42	1.71E-03
c5315	918	2190	188	4.64E-01	TO	6.28E-01	0.00E+00	TO	6.28E-01	0.00E+00	TO	TO
c7552	648	1308	212	2.34E-01	TO	2.35E-01	7.32E-03	TO	2.35E-01	7.32E-03	TO	TO
maximum memory usage (MB)							164			718		42466

nanometer technology, the *probabilistic equivalence checking* (PEC) problem asks to compute the probability for a probabilistic circuit to produce different outputs from its faultless specification. PEC was encoded into a random-exist quantified SSAT formula in [Lee and Jiang, 2014].

The SSAT formulas for probabilistic equivalence checking (PEC) are created as follows. The circuit with probabilistic errors is generated by assigning some gates in the original faultless circuit to be erroneous randomly. Two parameters are specified to control the generation of probabilistic circuits. The erroneous rate ϵ controls the probability of the happening of an error at a logic gate, and the defective rate δ controls the ratio of the number of erroneous gates to the total number of gates in the whole circuit. In our experiments, we use circuits from ISCAS benchmark suits and set $\epsilon = 0.125$, $\delta = 0.01$.

Table 1 shows the experimental results on PEC formulas. Columns 2, 3, and 4 show the numbers of variables, clauses, and randomly quantified variables in the formulas. Column 5 shows the exact satisfying probability of the formulas, computed by the signal probability method based on binary decision diagram (BDD) [Lee and Jiang, 2014]. Since reSSAT has the capability of solving approximate SSAT, we set a relatively short timeout of 60 seconds to evaluate its ability to derive upper and lower bounds for satisfying probability under a strict time constraint. Columns 6, 7, and 8 (resp. Columns 9, 10, and 11) show the runtime, upper bounds, and lower bounds of reSSAT with the timeout set to 60 (resp. 1000) seconds. Columns 12 and 13 show the runtime and satisfying probability of Dc with a 1000-second timeout.

While both methods failed to compute the exact answers on most of the cases, reSSAT derived tight bounds on some of the benchmarks. For example, for formulas c432 and c7552, the upper bounds derived by reSSAT with a 60-second timeout are very close to the exact satisfying probabilities, while Dc could not finish the computation on these two benchmarks even after 1000 seconds. On the other hand, notice that the derived bounds were not refined much after 60 seconds. The reason for the bounds not getting tight might lie in that in early search period, large cubes might have been collected and blocked huge search space. In other words, even if the runtime limit is increased, the additional small cubes collected do not contribute to a large fraction of the estimated probability. However, such a phenomenon does not hinder the application of reSSAT when the upper or lower bound of satisfying probability converges rapidly. The inexact estimation can be valuable when the exact computation is unavailable, such as c432 and c7552 in the experiment.

The above empirical results on the three benchmark families suggested that

- reSSAT outperforms Dc in terms of both runtime and memory consumption on random and strategic companies formulas, and
- reSSAT gives reasonable bounds on satisfying probability within 60 seconds, while Dc timed out on some of the PEC formulas.

5 Conclusions

In this paper, we focused on solving random-exist quantified SSAT formulas. In contrast to the previous DPLL-based algorithms, we proposed a novel algorithm using SAT solver and weighted model counter as underlying engines to improve computational efficiency. Leveraging the great success of modern SAT solving techniques, the proposed algorithm outperforms the state-of-the-art method in the experiment on random k-CNF and strategic companies formulas. Moreover, unlike previous exact SSAT methods, the proposed algorithm can be easily modified to solve approximate SSAT by deriving upper and lower bounds of satisfying probability. We demonstrated the applicability of our SSAT solver to VLSI circuit analysis. While the state-of-the-art solver fails to compute the exact satisfying probability, the proposed method succeeded in finding bounds of the formulas. In several cases, the derived bounds are very close to, or even match the exact satisfying probability. This approximation flexibility of our method can be helpful when SSAT is applied to real-world applications. For future work, we intend to extend the proposed algorithm to arbitrary quantified SSAT formulas.

Acknowledgements

The authors thank Stephen Majercik for kindly providing the SSAT solver DC−SSAT, and anonymous reviewers for valuable suggestions. This work was supported in part by the Ministry of Science and Technology of Taiwan under grants 104-2628-E-002-013-MY3, 105-2221-E-002-196-MY3, 105-2923-E-002-016-MY3, and 106-2912-E-002-002-MY3.

References

[Aziz *et al.*, 2015] R. Aziz, G. Chu, C. Muise, and P. Stuckey. #∃SAT: Projected model counting. In *Proc. SAT*, pages 121–137, 2015.

[Bacchus *et al.*, 2003] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proc. FOCS*, pages 340–351, 2003.

[Cadoli *et al.*, 1997] M. Cadoli, T. Eiter, and G. Gottlob. Default logic as a query language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, 1997.

[Chakraborty *et al.*, 2013] S. Chakraborty, K. Meel, and M. Vardi. A scalable approximate model counter. In *Proc. CP*, pages 200–216, 2013.

[Chakraborty *et al.*, 2016] S. Chakraborty, K. Meel, and M. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proc. IJCAI*, 2016.

[Chavira and Darwiche, 2008] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

[Cooper, 1990] G. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42(2-3):393–405, 1990.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Eén and Sörensson, 2003a] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT*, pages 502–518, 2003.

[Eén and Sörensson, 2003b] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[Faber and Ricca, 2005] W. Faber and F. Ricca. Solving hard ASP programs efficiently. In *Proc. LPNMR*, pages 240–252, 2005.

[Giunchiglia *et al.*, 2005] E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2005. `http://www.qbflib.org`.

[Gomes *et al.*, 2006] C. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *Proc. AAAI*, pages 54–61, 2006.

[Gomes *et al.*, 2007] C. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. From sampling to model counting. In *Proc. IJCAI*, pages 2293–2299, 2007.

[Hnich *et al.*, 2011] B. Hnich, R. Rossi, A. Tarim, and S. Prestwich. A survey on CP-AI-OR hybrids for decision making under uncertainty. *Hybrid Optimization*, pages 227–270, 2011.

[Lauria, 2012] M. Lauria. CNFgen: Combinatorial benchmarks for SAT solvers, 2012. `http://massimolauria.github.io/cnfgen/`.

[Lee and Jiang, 2014] N.-Z. Lee and J.-H. Jiang. Towards formal evaluation and verification of probabilistic design. In *Proc. ICCAD*, pages 340–347, 2014.

[Leone *et al.*, 2006] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[Majercik and Boots, 2005] S. Majercik and B. Boots. DC-SSAT: A divide-and-conquer approach to solving stochastic satisfiability problems efficiently. In *Proc. AAAI*, pages 416–422, 2005.

[Majercik and Littman, 1998] S. Majercik and M. Littman. Maxplan: A new approach to probabilistic planning. In *Proc. AIPS*, pages 86–93, 1998.

[Majercik and Littman, 2003] S. Majercik and M. Littman. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence*, 147(1-2):119–162, 2003.

[Majercik, 2007] S. Majercik. APPSSAT: Approximate probabilistic planning using stochastic satisfiability. *International Journal of Approximate Reasoning*, 45(2):402–419, 2007.

[Majercik, 2009] S. Majercik. Stochastic Boolean satisfiability. *Handbook of Satisfiability*, 185:887–925, 2009.

[Papadimitriou, 1985] C. Papadimitriou. Games against nature. *Journal of Computer and System Sciences*, 31(2):288–301, 1985.

[Sang *et al.*, 2004] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. SAT*, pages 20–28, 2004.

[Sang *et al.*, 2005a] T. Sang, P. Beame, and H. Kautz. Heuristics for fast exact model counting. In *Proc. SAT*, pages 226–240, 2005.

[Sang *et al.*, 2005b] T. Sang, P. Beame, and H. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. AAAI*, pages 475–481, 2005.

[Teige and Fränzle, 2010] T. Teige and M. Fränzle. Resolution for stochastic Boolean satisfiability. In *Proc. LPAR*, pages 625–639, 2010.