# Knowledge Compilation for Incremental and Checkable Stochastic Boolean Satisfiability

**Che Cheng**[1], **Yun-Rong Luo**[2] and **Jie-Hong R. Jiang**[1,3]
[1]Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan
[2]University of Michigan, Ann Arbor MI, USA
[3]Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
r11943097@ntu.edu.tw, yunrong@umich.edu, jhjiang@ntu.edu.tw

## Abstract

Knowledge compilation has proven effective in (weighted) model counting, uniquely supporting incrementality and checkability. For incrementality, compiling an input formula once suffices to answer multiple queries, thus reducing the total solving effort. For checkability, the compiled formula is amenable to producing machine-checkable proofs for verification, thus strengthening the solver's reliability. In this work, we extend knowledge compilation from model counting to stochastic Boolean satisfiability (SSAT) solving by generalizing the dec-DNNF representation to accommodate the SSAT quantifier structure and integrate it into `SharpSSAT`, a state-of-the-art SSAT solver. We further study proof generation from the compiled representation and extend `CPOG`, a certified model-counting toolchain, to generate proofs for certifying the results of `SharpSSAT`. Experimental results show the benefits of the proposed knowledge compilation approach for SSAT in sharing computation efforts for multiple queries and producing checkable dec-DNNF logs with negligible overhead.

## 1 Introduction

Stochastic Boolean satisfiability (SSAT) is a formalism that generalizes quantified Boolean formula (QBF) by replacing the universal quantifier with a randomized one. Its ability to capture decision-making under uncertainty makes it suitable for succinct encoding of a variety of problems, e.g., formal verification of probabilistic design [Lee and Jiang, 2018], planning under uncertainty [Salmon and Poupart, 2020], statistical inference [Hsieh and Jiang, 2022], and fairness analysis of machine learning models [Ghosh *et al.*, 2021]. Efficient solving of SSAT has also been an actively studied topic [Majercik and Boots, 2005; Salmon and Poupart, 2020; Chen *et al.*, 2021; Wang *et al.*, 2022; Fan and Jiang, 2023].

Another direction that has been long studied for propositional reasoning tasks is knowledge compilation. Through compiling a propositional theory into a suitable target language, many intractable tasks can be done in polynomial time [Darwiche and Marquis, 2002]. The earliest and most well-known target language is the binary decision diagram (BDD), which is widely applied in circuit synthesis and verification [Clarke *et al.*, 2018].

Despite its support for a wide range of queries and transformations, BDD construction often suffers from memory explosion and fails for large problem instances. Hence, the quest for succinct formats is an important pursuit. Huang and Darwiche [2005] observed that the trace of an exhaustive DPLL-style search with different constraints imposed corresponds to different target languages. Among them, the decision decomposable negation normal form (dec-DNNF) is the most succinct. Recent advances in model counting have led to the development of several dec-DNNF compilers [Muise *et al.*, 2012; Lagniez and Marquis, 2017]. Recent work also studied certified knowledge compilation [Capelli, 2019; Capelli *et al.*, 2021; Bryant *et al.*, 2023], and developed various approaches to verify whether the compiled result is logically equivalent to the compiler's given input. Certified knowledge compilation is crucial in verifying the results of model counters.

Previous studies concerning knowledge compilation and quantified formulas focus mainly on the theoretical bounds for QBF evaluation. In [Coste-Marquis *et al.*, 2005], the tractability of QBF solving on a variety of target languages, including BDD and dec-DNNF, was studied. In [Fargier and Marquis, 2006; Capelli and Mengel, 2019], new target languages were proposed to achieve quantifier elimination. In [Lai *et al.*, 2017], BDDs were generalized to provide more flexibility with canonicity being maintained. Regarding SSAT, Lee *et al.* [2018] exploited BDDs for weighted model counting in solving the exist-random quantified subclass of SSAT. Ojala [2022] evaluated SSAT by compiling the matrix formula into a BDD or sentential decision diagram (SDD) with compatible ordering using off-the-shelf compilers. However, its performance is inferior to the state-of-the-art SSAT solvers.

In this paper, we propose a new target language of knowledge compilation, based on the trace of the state-of-the-art SSAT solver `SharpSSAT` [Fan and Jiang, 2023], named *levelized dec-DNNF*, that supports incremental and checkable evaluations on an SSAT formula. It allows more relaxed constraints on the ordering compared to similar ones proposed in previous methods [Fargier and Marquis, 2006; Lai *et al.*, 2017], thus allowing more flexible and efficient compilation. In addition, we formalize the pruning techniques implemented in `SharpSSAT` into our representation to allow partial exploration, resulting in both faster compilation and more succinct representation.

In addition, to certify the correctness of SSAT solving, we develop a proof framework `cert-SSAT` based on `CPOG` [Bryant *et al.*, 2023], which generates and checks clausal proof for logical equivalence between the input and output of a dec-DNNF compiler. To accommodate partial exploration from the pruning techniques, two dec-DNNF graphs need to be compiled, which are then passed to `cert-SSAT` for detailed clausal proof generation and validation. Moreover, we relax the logical equivalence checking in `CPOG` into two logical implication checks to separately certify lower and upper bounds of SSAT satisfying probability.

Experiments on various benchmark instances show the performance advantage of compilation over repeated solving. When certification is of concern, little computational overhead is required for `SharpSSAT` to generate the additional dec-DNNF logs for further proof checking. `SharpSSAT` is able to generate dec-DNNF proofs for all solvable instances, and `cert-SSAT` successfully turns $80.5\,\%$ of them into clausal proofs and verifies the clausal proofs within a reasonable time. For the remaining $19.5\,\%$, `cert-SSAT` can neither prove or disprove their correctness.

In fact, our proof framework helps reveal and fix previously undetected bugs in `SharpSSAT`. It is the first work to formally verify the results of an SSAT solver, in contrast to [Fan and Jiang, 2023] with only (lower bound) witnessing ability. Our method enhances the reliability and trustworthiness of `SharpSSAT` results.

The rest of this paper is organized as follows. Section 2 defines notations and introduces essential backgrounds. Section 3 proposes a new knowledge compilation format for SSAT and discusses its supported queries. Section 4 presents the proof generation and validation framework for `SharpSSAT`. The experimental evaluation follows in Section 5. Finally, Section 6 concludes this paper.

## 2 Preliminaries

In this section, we define the notations used throughout the paper, and briefly review the foundations of SSAT and knowledge compilation.

Boolean values TRUE and FALSE are denoted by $\top$ and $\bot$, respectively. Boolean connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$ are interpreted in their conventional semantics. A *literal* $l$ is a Boolean variable $v$ or its negation $\neg v$ (also written as $\bar{v}$ for simplicity), and we write $\mathrm{var}(l) = v$ to denote the corresponding Boolean variable. A clause $C$ is a disjunction of literals. A Boolean formula $\phi$ is in conjunctive normal form (CNF) if it is a conjunction of clauses. We also view a clause $C$ as a set of literals and a CNF formula $\phi$ as a set of clauses whenever appropriate. The *substitution* of variable $v$ with formula $\psi$ in formula $\phi$ is denoted by $\phi[\psi/v]$. We denote the set of variables occurring in a formula $\phi$ by $\mathrm{vars}(\phi)$.

Let $V$ be a set of Boolean variables. An *assignment* $\alpha$ on $V$ is a mapping from $V$ to $\mathbb{B} = \{\top, \bot\}$, and we denote the set of all possible assignments on $V$ by $\mathcal{A}(V)$. The *projection* of formula $\phi$ over variables $V$ onto an assignment $\alpha$ on $V' \subseteq V$ is denoted by $\phi[\alpha]$, which is obtained by substituting each occurrence of each $v \in V'$ with the Boolean value $\alpha(v)$.

### 2.1 Knowledge Compilation

The idea of knowledge compilation is to first compile a propositional theory (or equivalently, a Boolean function) into a target language in an offline phase. Then, queries on the target language are performed during the online phase. A comparison of supported queries and succinctness of different target languages can be found in [Darwiche and Marquis, 2002].

Among the target languages, we are particularly interested in dec-DNNF, which is a special negation normal form (NNF), to be defined as follows.

**Definition 1** (Negation normal form). *A Boolean formula $\phi$ is in the* negation normal form *(NNF) if it consists of only the connectives $\wedge$, $\vee$, and $\neg$, and the negation connective $\neg$ only occurs immediately before a variable.*

We define the notions of decomposability and decision for a formula as follows.

**Definition 2** (Decomposability). *A conjunction $\bigwedge_i \phi_i$ of subformulas $\phi_i$'s is* decomposable *if $\mathrm{vars}(\phi_i) \cap \mathrm{vars}(\phi_j) = \varnothing$ for all $i \neq j$. A Boolean formula $\phi$ in NNF is in the* decomposable negation normal form *(DNNF) if every conjunction in $\phi$ is decomposable.*

**Definition 3** (Decision). *A disjunction $\phi_1 \vee \phi_2$ of two subformulas $\phi_1$ and $\phi_2$ is called a* decision *with* decision variable $v$ *if $\phi_1 = v \wedge \psi_1$ and $\phi_2 = \neg v \wedge \psi_2$.*

Finally, we define dec-DNNF.

**Definition 4** (Dec-DNNF). Dec-DNNF *is a language consisting of Boolean formulas in NNF, where each conjunction is decomposable, and each disjunction is a decision.*

In the sequel, we shall view a dec-DNNF formula $\phi$ as a directed acyclic graph (DAG) with a unique root node labeled with $\phi$. Each leaf node is labeled with either a literal or a Boolean constant, and each internal node is either an *AND node*, whose $i^{\text{th}}$ child is labeled with the conjoined subformula $\phi_i$ as in Definition 2, or a *decision node* $N$ associated with decision variable $v$, denoted $\mathrm{dvar}(N) = v$, with exactly two children labeled with $\psi_1$ and $\psi_2$ as in Definition 3. Note that each subgraph induced by a node $N$ and its descendants is a dec-DNNF graph representing the subformula $\phi^N$ that $N$ is labeled with. We sometimes use the terms dec-DNNF formula and dec-DNNF graph interchangeably.

### 2.2 Stochastic Boolean Satisfiability

Stochastic Boolean satisfiability (SSAT) is initially proposed in [Papadimitriou, 1985] as a *game against nature*.

**Definition 5** (SSAT Syntax). *An SSAT formula $\Phi = \mathcal{Q}.\,\phi$ over variables $V$ is of the form*

$$Q_1 v_1, Q_2 v_2, \ldots, Q_n v_n.\, \phi\,, \tag{1}$$

*where $\mathcal{Q} = Q_1 v_1, Q_2 v_2, \ldots, Q_n v_n.$ is the* quantifier prefix *(*prefix *for short), and $\phi$ is the* matrix*. Each $Q_i \in \{\exists, ⅃^{p_i}\}$, and $\phi$ is a quantifier-free Boolean formula over $V$ in CNF. The symbol $⅃^{p}$ denotes a* randomized quantifier*, where $⅃^{p}\, r$ states that variable $r$ evaluates to $\top$ with probability $p$ and $\bot$ with probability $1-p$. We denote the quantifier of variable $v$ as $Q(v)$, i.e., $Q(v_i) = Q_i$. If $Q(v) = \exists$ (resp. $⅃$), we say that $v$ is an* existential *(resp.* randomized*) variable.*

The *quantification level* (to count quantifier alternations) of variable $v_i$, denoted by $\mathrm{lvl}(v_i)$, is recursively defined as

- $\mathrm{lvl}(v_1) = 1$, and
- $\mathrm{lvl}(v_i) = \begin{cases} \mathrm{lvl}(v_{i-1}) & \text{if } Q_i = Q_{i-1} \\ \mathrm{lvl}(v_{i-1}) + 1 & \text{otherwise} \end{cases}$ for $i > 1$.

The satisfying probability of an SSAT formula $\Phi$ is defined recursively by

- $\Pr[\top] = 1, \Pr[\bot] = 0$,
- $\Pr[\exists v.\Phi'] = \max(\Pr[\Phi'[\bot/v]], \Pr[\Phi'[\top/v]])$, and
- $\Pr[\text{Ⅎ}^p\, v.\Phi'] = (1-p) \cdot \Pr[\Phi'[\bot/v]] + p \cdot \Pr[\Phi'[\top/v]]$,

where $\Phi'$ is the SSAT formula obtained by removing the outermost quantifier of $\Phi$, i.e., $\Phi = Qv.\Phi'$.

Given an SSAT formula $\Phi$ and some $\theta \in [0, 1]$, deciding whether $\Pr[\Phi] \geq \theta$ is PSPACE-complete [Papadimitriou, 1985]. In addition, Theorem 1 shows that the satisfying probability of an SSAT formula can be computed by the satisfying probability of its disjoint components. A similar property is exhibited in model counting.

**Theorem 1** ([Salmon and Poupart, 2020]). *Given an SSAT formula $\Phi = \mathcal{Q}.\,\phi$, if $\phi = \bigwedge_{i=1}^{k} \phi_i$ is decomposable (Definition 2), then $\Pr[\Phi] = \prod_{i=1}^{k} \Pr[\mathcal{Q}.\,\phi_i]$.*

### 2.3 Clausal Proof for Knowledge Compilation

CPOG [Bryant *et al.*, 2023] can be used to certify the results of a knowledge compiler by giving it a CNF formula $\phi$ and its compiled dec-DNNF formula $G$ as input. Given $\phi$ and $G$, CPOG checks whether $\phi$ is logically equivalent to the Boolean function represented by $G$. It is a clausal proof system based on *extended resolution* [Tseitin, 1983]. A CPOG proof is a sequence of extension, clause addition, and clause deletion steps. An extension step introduces one extension variable $v$ that either encodes a conjunction or disjunction of literals whose variables, called the *defining variables* of $v$, can be the previously introduced extension variables or original variables in the input formula. CPOG generates CNF encodings to describe the logical relation between extension variables and their respective defining variables [Tseitin, 1983]. A clause can be added to or deleted from the proof if the clause is an *implication redundancy* with respect to the set of other clauses in the proof. That is, if $\psi \rightarrow C$ holds for a clause $C$ and a CNF formula $\psi$, then $C$ can be added to $\psi$ to form $\psi \cup \{C\}$ or deleted from $\psi \cup \{C\}$ to form $\psi$.

CPOG consists of a proof generator `cpog-gen` and a proof checker `cpog-check`, and works as follows. `cpog-gen` first encodes $G$ into a CNF formula $\psi_G$ by extension, and then generates a clausal proof for $\phi \rightarrow \psi_G$ by clause addition and a proof for $\psi_G \rightarrow \phi$ by clause deletion. Given a CPOG proof log $P$, `cpog-check` checks whether each extension, clause addition, and deletion step in $P$ is sound. Upon the completion of proof checking, `cpog-check` claims $\phi \equiv \psi_G$ if and only if 1) there is exactly one final unit clause $\{r\}$ added via implication redundancy, where $r$ is the extended literal representing the root of $G$, and 2) all input clauses of $\phi$ have been deleted.

## 3 Knowledge Compilation for SSAT

We develop the idea of knowledge compilation for SSAT tightly around the DPLL-style solver `SharpSSAT` [Fan and Jiang, 2023]. `SharpSSAT` is a state-of-the-art solver that utilizes component analysis, which is responsible for compiling formulas with the decomposability property introduced in Section 2.1 [Huang and Darwiche, 2005]. Inspired by the design of `SharpSSAT`, we propose adding an ordering constraint on dec-DNNF and allowing partial exploration to incorporate the prefix structure of SSAT into the representation.

### 3.1 Levelized Dec-DNNF

As observed in [Huang and Darwiche, 2005], the trace of an exhaustive DPLL-style search with component analysis corresponds to a dec-DNNF formula. Also, in model counting, due to its exhaustive nature, the effort of compilation only has a slight overhead compared to direct solving, which makes it especially compelling. As observed in [Salmon and Poupart, 2020], decomposability holds for SSAT, hence the idea of SSAT evaluation with dec-DNNF seems plausible.

However, we recall the observation that deciding a QBF is PSPACE-hard for arbitrary BDD [Coste-Marquis *et al.*, 2005, Proposition 1] and in P for BDD with compatible ordering [Coste-Marquis *et al.*, 2005, Proposition 2]. As QBF evaluation can be reduced to SSAT evaluation and dec-DNNF is a superset of BDD, answering the satisfying probability of an SSAT formula given an arbitrary dec-DNNF formula of its matrix is also PSPACE-hard. Therefore, we know that some ordering constraints must be imposed upon the dec-DNNF formula to achieve SSAT evaluation in polynomial time with respect to the formula size.

We thus propose *levelized dec-DNNF* to capture such representation. To simplify our discussion, in the sequel we view a leaf node $N$ associated with literal $l$ as a decision node with $\mathrm{dvar}(N) = \mathrm{var}(l)$, and $\top$ and $\bot$ as the two children, thus only considering constant nodes as leaf nodes. Note that this modification increases the graph size by a constant factor, for having two additional edges per literal node and two additional constant nodes in total.

**Definition 6** (Levelization). *Given an SSAT formula $\Phi = \mathcal{Q}.\,\phi$ in the form of Eq.* (1) *and a dec-DNNF graph $G$,*[1] *we say that $G$ is* levelized *with respect to $\Phi$ if for each pair of decision nodes $N_1$ and $N_2$, if $N_2$ is reachable from $N_1$, then* $\mathrm{lvl}(\mathrm{dvar}(N_1)) \leq \mathrm{lvl}(\mathrm{dvar}(N_2))$.

The condition ensures that for all paths from the root to the leaves in the dec-DNNF formula $G$, the ordering of the decision nodes is consistent with the prefix of $\Phi$. In addition, we remark that by the decomposability of the conjunctions in $N_1$, $\mathrm{dvar}(N_1)$ cannot appear in the formulas represented by the descendants of $N_1$. In particular, we have $\mathrm{dvar}(N_1) \neq \mathrm{dvar}(N_2)$. This corresponds to the well-known *read-once property* [Darwiche and Marquis, 2002].

The idea of levelized dec-DNNF closely resembles that of O-DDG [Fargier and Marquis, 2006] and BDD[$\wedge$] [Lai

[1]Note that we do not require the function represented by $G$ to be equivalent to $\phi$, as this will not be the case when pruning techniques are considered in Section 3.3.

**Algorithm 1** SSAT Evaluation with dec-DNNF

1: **procedure** EVALSSAT($\Phi, G$)
2: $\triangleright$ *$\Phi = \mathcal{Q}.\,\phi$ is an SSAT formula, and $G$ is a levelized dec-DNNF graph equivalent to $\phi$ with $m$ nodes*
3: Let $P$ be a vector of size $m$ indexed by the nodes of $G$
4: Initialize each cell of $P$ to $null$
5: EVAL-NODE($\Phi, G, G.root$)
6: **return** $P[G.root]$
7: **procedure** EVAL-NODE($\Phi, G, N$)
8: $\triangleright$ *$N$ is a node in $G$*
9: **if** $P[N] = null$ **then** $\triangleright$ *Each node is computed once*
10: **if** $\phi^N = \bot$ **then**
11: $P[N] \leftarrow 0$
12: **else if** $\phi^N = \top$ **then**
13: $P[N] \leftarrow 1$
14: **else if** $N$ is an AND node **then**
15: $\triangleright$ *Let $N_1, \ldots, N_k$ be the children of $N$*
16: $p \leftarrow 1$
17: **for each** child $N_i$ of $N$ **do**
18: EVAL-NODE($\Phi, G, N_i$)
19: $p \leftarrow p \cdot P[N_i]$
20: $P[N] \leftarrow p$
21: **else**
22: $\triangleright$ *$N$ is a decision node with decision variable $v$ and children $N_1, N_2$*
23: EVAL-NODE($\Phi, G, N_1$)
24: EVAL-NODE($\Phi, G, N_2$)
25: **if** $Q(v) = \text{Я}^p$ **then**
26: $P[N] \leftarrow p \cdot P[N_1] + (1-p) \cdot P[N_2]$
27: **else** $\triangleright$ $Q(v) = \exists$
28: $P[N] \leftarrow \max(P[N_1], P[N_2])$

*et al.*, 2017]. However, we relax the ordering constraint to respect only the quantifier level instead of enforcing a total order on the variables. By doing so, different branches of a dec-DNNF formula can have different variable orders, allowing more flexible ordering in the compilation. This levelization also reflects the internal process of SSAT solvers, such as `SharpSSAT` [Fan and Jiang, 2023], where an ordering heuristics is invoked when choosing each branching variable, instead of having a predetermined branching order.

### 3.2 SSAT Evaluation with Levelized Dec-DNNF

We start the subsection by showing that the satisfying probability of an SSAT formula $\Phi$ with its matrix in the levelized dec-DNNF can be computed efficiently via Algorithm 1, as formally stated in Theorem 2.

**Theorem 2.** *Given an SSAT formula $\Phi = \mathcal{Q}.\,\phi$ in the form of Eq.* (1) *and $\phi$ represented in the levelized dec-DNNF graph $G$, Algorithm 1 computes the satisfying probability of $\Phi$ in time $O(|G|)$.*

*Proof.* We first prove the correctness of the algorithm by induction on the structure of $G$.

For the base case, let $G$ consist of a single node $N$, which must be a constant node. Its satisfying probability is 1 if the node is associated with $\top$, and 0 if associated with $\bot$, as shown on Lines 10 to 13.

Suppose the node $N$ is an AND node with $k$ children $N_1, \ldots, N_k$. By Theorem 1, we have $\Pr\left[\mathcal{Q}.\,\phi^N\right] = \prod_{i=1}^{m} \Pr\left[\mathcal{Q}.\,\phi^{N_i}\right]$, as done on Lines 14 to 20.

Finally, suppose the root node $N$ is a decision node with decision variable $v$ and children $N_1$ and $N_2$. By the read-once property, we have $v \notin \mathrm{vars}(\phi^{N_1}) \cup \mathrm{vars}(\phi^{N_2})$, and since $\phi^N = (v \wedge \phi^{N_1}) \vee (\neg v \wedge \phi^{N_2})$, we have $\phi^{N_1} = \phi^N[\top/v]$ and $\phi^{N_2} = \phi^N[\bot/v]$. Next, by the levelized assumption, $v$ has the smallest level among all variables in $\mathrm{vars}(\phi_N)$. The satisfying probability can thus be computed as done on Lines 21 to 28 according to the recursive definition of satisfying probability of SSAT introduced in Section 2.2.

It follows that the satisfying probability of the SSAT formula is correctly computed. In addition, since the probability of each node is computed only once using memoization and each edge is visited once, the procedure runs in time $O(|G|)$. $\square$

Just as dec-DNNF supports weighted model counting of different weights and under different partial assignments, levelized dec-DNNF can do the same for SSAT formulas. We first define the two types of queries of concern:

**Definition 7** (Reweighting and cofactoring)**.** *Given SSAT formulas $\Phi_1 = \mathcal{Q}_1.\,\phi_1$ and $\Phi_2 = \mathcal{Q}_2.\,\phi_2$, we say that $\Phi_2$ is a* reweighting *of $\Phi_1$ if $\mathcal{Q}_1$ and $\mathcal{Q}_2$ differ only in the probabilities of the randomized variables and $\phi_1 = \phi_2$. We say that $\Phi_2$ is a* cofactoring *of $\Phi_1$ if $\mathcal{Q}_1 = \mathcal{Q}_2$ and $\phi_2 = \phi_1[\alpha]$ for some assignment $\alpha \in \mathcal{A}(V')$, where $V' \subseteq \mathrm{vars}(\phi_1)$.*

The result is formally stated in the following theorem:

**Theorem 3.** *Given an SSAT formula $\Phi = \mathcal{Q}.\,\phi$, for $\phi$ represented in the levelized dec-DNNF graph $G$, the satisfying probability of a reweighting or a cofactoring of $\Phi$ can be computed in time $O(|G|)$.*

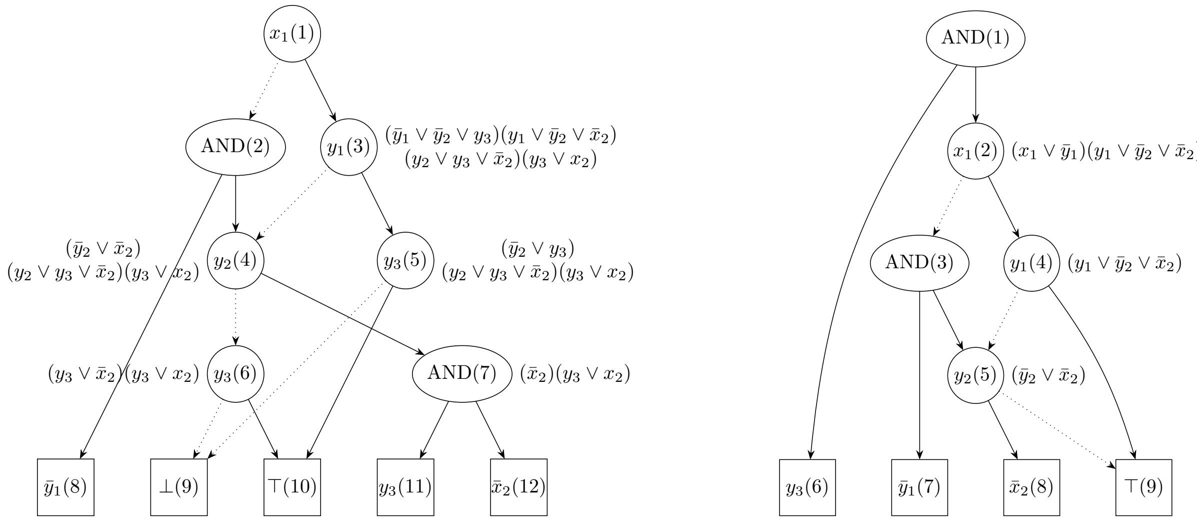
*Proof.* Let $\Phi' = \mathcal{Q}'.\,\phi$ be a reweighting of $\Phi$. Since the proof for Theorem 2 does not make any assumption on the probability of the randomized variables, $\Phi'$ can be computed by EVALSSAT($\Phi, G$) directly.

Let $\Phi'' = \mathcal{Q}.\,\phi[\alpha]$ be a cofactoring of $\Phi$ for some assignment $\alpha \in \mathcal{A}(V')$. We only need to modify Lines 26 and 28 to assign $P[N]$ to $P[N_1]$ (resp. $P[N_2]$) if $v \in V'$ and $\alpha(v) = \top$ (resp. $\bot$). $\square$

### 3.3 Pruning Techniques

In the previous subsection, we demonstrate how to perform SSAT-related queries with dec-DNNF by imposing an ordering constraint on the representation. However, in addition to the ordering constraint, SSAT also differs from model counting in its optimization nature brought by existential quantification.

`SharpSSAT` [Fan and Jiang, 2023] utilizes this property and implements some pruning techniques, including *pure literal detection* and *existential early return*, to enhance its performance. The techniques exploit the fact that if $Q(v) = \exists$ and the implication $\phi[\top/v] \rightarrow \phi[\bot/v]$ (resp. $\phi[\bot/v] \rightarrow \phi[\top/v]$) holds, then $\Pr[\mathcal{Q}.\,\phi]$ equals $\Pr[\mathcal{Q}.\,\phi[\bot/v]]$ (resp. $\Pr[\mathcal{Q}.\,\phi[\top/v]]$). It follows that the solver only needs to explore the branch $\phi[\bot/v]$ (resp. $\phi[\top/v]$) when $\phi[\top/v] \rightarrow$




(a) Without pure literal detection. (b) With pure literal detection.

Figure 1: Levelized dec-DNNF graphs for Example 1.

$\phi[\bot/v]$ (resp. $\phi[\bot/v] \to \phi[\top/v]$) holds, as the unexplored branch $\phi[\top/v]$ (resp. $\phi[\bot/v]$) cannot have a higher satisfying probability.

We incorporate the idea of pruning in our representation by replacing each unexplored node with the constant node $\bot$. By doing so, the compiled dec-DNNF graph no longer represents the matrix of the formula $\phi$ but rather represents a formula $\phi_l$ such that $\phi_l \to \phi$. We now revisit Theorems 2 and 3 on the pruned representation.

**Theorem 4.** *Given an SSAT formula $\Phi = \mathcal{Q}.\,\phi$ and a levelized dec-DNNF graph $G_l$ compiled from $\phi$ with pruning enabled, the satisfying probability of $\Phi$ and a reweighting $\Phi'$ of $\Phi$ can be computed correctly by Algorithm 1 using $G_l$, while that of a cofactoring $\Phi''$ of $\Phi$ cannot.*

*Proof.* For $\Phi$ and $\Phi'$, since the unexplored branch must have a smaller satisfying probability and the decision variable is existential, Algorithm 1 still computes the correct satisfying probability, as the unexplored branch is never chosen on Line 28.

For the cofactoring case, let $\Phi = \exists v_1, \text{Ⅎ}^{0.5} v_2.\,(v_1 \vee v_2)$ and consider the assignment $\alpha : v_1 \mapsto \bot$. Since $Q(v_1) = \exists$ and $v_1$ is a pure literal, the pruned representation will essentially represent the formula $v_1$. Hence, Algorithm 1 will return $\Pr[\Phi[\alpha]] = 0$, while it is in fact $0.5$. $\square$

We remark that replacing each unexplored node with the node of the explored branch will result in a dec-DNNF graph $G_u$, which represents a formula $\phi_u$ such that $\phi \to \phi_u$. Notice that replacing $G_l$ with $G_u$ does not affect the correctness of Theorem 4. This observation paves the way for checkable SSAT solving, which will be detailed in Section 4.

By introducing the pruning techniques into the compilation, we effectively utilize the maximizing nature of SSAT to compile a dec-DNNF graph of smaller size that still faithfully captures the essence of the SSAT formula. In addition, we note that the overhead in detecting pure literals is on traversing the clauses, which occurs purely during the compilation process. Hence, this overhead is amortized by the improvement it brings in the multiple queries performed on the compiled graphs.

We conclude this section with an example.

**Example 1.** *Consider the SSAT formula*

$$\begin{aligned}\Phi \;=\; & \text{Ⅎ}^{0.4} x_1, \exists y_1, \exists y_2, \exists y_3, \text{Ⅎ}^{0.6} x_2. \\ & (x_1 \vee \bar{y}_1)(\bar{y}_1 \vee \bar{y}_2 \vee y_3)(y_1 \vee \bar{y}_2 \vee \bar{x}_2) \\ & (y_2 \vee y_3 \vee \bar{x}_2)(y_3 \vee x_2)\,.\end{aligned}$$

*The compiled levelized dec-DNNF is shown in Fig. 1a. It is easily verified that the read-once and levelization properties are satisfied.*

*We now demonstrate how to compute $\Pr[\mathcal{Q}.\,\phi[y_1]]$ by calculating the probability $p_i$ of each node $N_i$, labeled with "$(i)$" in the figure, of the compiled levelized dec-DNNF graph:*

- $p_{10} = 1$ *and* $p_9 = 0$ *by the base cases,*
- $p_{12} = 0.4 \cdot 1 + 0.6 \cdot 0 = 0.4$, $p_{11} = p_6 = p_5 = \max(0, 1) = 1$,
- $p_8 = 0$ *since* $y_1 = \top$ *is assumed,*
- $p_7 = p_{11} \cdot p_{12} = 0.4$,
- $p_4 = \max(p_6, p_7) = 1$, $p_3 = p_5 = 1$, $p_2 = p_4 \cdot p_8 = 0$, *and* $p_1 = 0.6 \cdot p_2 + 0.4 \cdot p_3 = 0.4$ *following the same set of rules.*

*We hence conclude that* $\Pr[\mathcal{Q}.\,\phi[y_1]] = 0.4$.

*Since $y_3$ is a pure literal in $\Phi$, we can simplify the compilation by exploring the branch $y_3 = \top$ only. The resulting dec-DNNF graph is shown in Fig. 1b. It can be seen that the technique simplifies the representation when compared to that shown in Fig. 1a.*

## 4 SSAT Validation with Levelized Dec-DNNF

In order to extend `CPOG` to `cert-SSAT` for `SharpSSAT` certification, two main modifications to `CPOG` are needed.

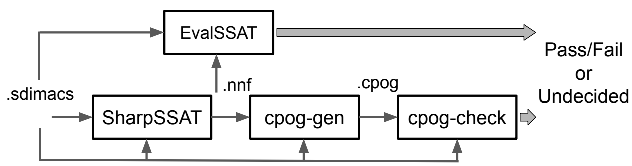



Figure 2: Computation flow of `cert-SSAT`.

First, `cert-SSAT` has to additionally validate levelization of dec-DNNF. Second, more importantly, `cert-SSAT` has to support checking the partial exploration introduced by the pruning techniques. Checking the former is relatively straightforward. To achieve the latter, instead of simply checking logical equivalence between the input CNF and compiled dec-DNNF formulas as mentioned in Section 2.3, `cert-SSAT` requires `SharpSSAT` to compile two dec-DNNF graphs $G_u$ and $G_l$ and check Eq. (2) and Eq. (3) by Theorem 5 as follows.

**Theorem 5.** *Given an SSAT formula* $\Phi = \mathcal{Q}.\,\phi$ *and two levelized dec-DNNF graphs* $G_l$ *and* $G_u$ *respecting the quantifier level of* $\Phi$, *then* $\Pr[\Phi] = p$ *if the following holds:*

$$(G_l \rightarrow \phi) \wedge (\phi \rightarrow G_u), \text{ and} \tag{2}$$

$$\text{EVALSSAT}(\Phi, G_l) = \text{EVALSSAT}(\Phi, G_u) = p. \tag{3}$$

*Proof.* The proof follows from the same reasoning for pruning techniques in Section 3.3. □

`SharpSSAT` requires a minor modification to compile $G_l$ and $G_u$ in addition to computing satisfying probability. Whenever pruning occurs, `SharpSSAT` marks the pruned branch so that the unexplored branch of the decision node can later be connected to the constant node $\perp$ or the explored branch to form the nodes $N_l \in G_l$ and $N_u \in G_u$, as explained in Section 3.3.

The computation flow of `cert-SSAT` is shown in Fig. 2. Given an input SSAT instance $\Phi$ in the `sdimacs` format, the modified `SharpSSAT` under the *certificate generation* option first generates two dec-DNNF graphs $G_l$ and $G_u$ in the `nnf` format in addition to computing $\Pr[\Phi]$. Next, `EvalSSAT` is applied to check whether Eq. (3) and the levelization constraints hold for $G_l$ and $G_u$. Then, the proof generator `cpog-gen` takes $\Phi$, $G_l$, and $G_u$ as input, generates CNF formulas $\psi_{G_l}$ of $G_l$ and $\psi_{G_u}$ of $G_u$, and emits a clausal proof that establishes Eq. (2) of the `cpog` format [Bryant *et al.*, 2023]. Finally, the proof checker `cpog-check` checks whether $\psi_{G_l}$ (resp. $\psi_{G_u}$) is generated from $G_l$ (resp. $G_u$) by the extension process and verifies the correctness of the clausal proof for Eq. (2) by the standard *reverse unit propagation* (RUP) checking [Goldberg and Novikov, 2003; Gelder, 2008].

Let the outputs of `SharpSSAT` and `cpog-gen` be referred to as *`nnf`-proof* and *`cpog`-proof*, respectively. Suppose `SharpSSAT` reports $\Pr[\Phi] = p$ for an SSAT instance $\Phi = \mathcal{Q}.\phi$. The `cpog`-proof for $(\psi_{G_l} \rightarrow \phi)$ and that for $(\phi \rightarrow \psi_{G_u})$ can certify the lower and upper bounds of $\Pr[\Phi]$, respectively. Since it is computationally easier to verify the first implication,[2] even if `cert-SSAT` fails to verify the second implication and hence fails to conclusively prove that $\Pr[\Phi] = p$, chances are that `cert-SSAT` could at least provide a lower-bound proof. In particular, when $p = 1$, it suffices to prove that $\Pr[\Phi] \geq 1$, and `cert-SSAT` only need to prove the lower-bound case.

## 5 Experimental Results

We equipped the state-of-the-art SSAT solver `SharpSSAT` [Fan and Jiang, 2023] with a knowledge compilation capability,[3] implemented `EvalSSAT` of Algorithm 1 in the C++ language, and modified `CPOG` for the proposed SSAT validation toolchain `cert-SSAT`.[4] For incremental SSAT evaluation using `EvalSSAT`, an option (`-d`) was implemented in `SharpSSAT` to compile the levelized dec-DNNF graph $G_l$. For proof checking using the `cert-SSAT` flow, another option (`-l`) was implemented in `SharpSSAT` to compile both $G_l$ and $G_u$, and `EvalSSAT` was also modified to check the ordering of the levelized dec-DNNF graph.

We only enable the pruning techniques of Section 3.3 when the original option for pure literal detection (`-p`) is turned on. To study the effectiveness of incremental SSAT evaluation, we conducted experiments on reweighting and cofactoring queries. To study the effectiveness of SSAT proof checking, we applied the `cert-SSAT` computation on all the instances. The benchmark formulas are taken from [Chen *et al.*, 2021].[5]

### 5.1 Incremental SSAT Evaluation

The experiments were conducted on a Linux machine with an Intel Xeon Silver 4210 CPU processor at 2.2 GHz. For each SSAT formula, we randomly generated 10 instances for both query types and compared the performance of solving incrementally with knowledge compilation to that of solving from scratch for each instance. We refer to the three solving methods as `KC-p`, `KC-np`, and `baseline`, which corresponds to knowledge compilation with and without pruning and solving from scratch each time, respectively.[6] Because `SharpSSAT` is a state-of-the-art solver, and, to the best of our knowledge, no SSAT solvers support incremental solving, we did not compare the performance with other SSAT solvers, such as `ClauSSat` [Chen *et al.*, 2021] and `ElimSSAT` [Wang *et al.*, 2022], whose ability on SSAT solving has already been compared against `SharpSSAT` in [Fan and Jiang, 2023].

To fairly judge the performance of the two approaches, we set a time limit of 2000 sec for each SSAT formula as follows: For the settings `KC-p` and `KC-np`, we set a time limit of 1000 sec for compilation and 100 sec for each query. If the compilation failed within the time limit, we view the queries as failed as well. For the setting `baseline`, we set a time limit of 200 sec for each query. The overall performance is then

[2] The first implication proof can be generated in one topological traversal of the dec-DNNF, while proving the second implication involves calling an SAT solver.

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

[4] Available at https://github.com/NTU-ALComLab/cert-SSAT.

[5] Available at https://github.com/NTU-ALComLab/ClauSSat.

[6] For the `baseline` setting, the pruning techniques are enabled.

| | | Reweighting | | | | | | Cofactoring | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | `baseline` | | `KC-np` | | `KC-p` | | `baseline` | | `KC-np` | |
| Family | # | #S | PAR2 | #S | PAR2 | #S | PAR2 | #S | PAR2 | #S | PAR2 |
| ToiletA | 770 | **600** | **974.48** | 570 | 1080.65 | 570 | 1078.55 | **621** | **852.60** | 570 | 1080.61 |
| tiger | 50 | 50 | 9.63 | 50 | **2.93** | 50 | 3.03 | 50 | 9.66 | 50 | **2.90** |
| MaxCount | 250 | 110 | 2313.60 | 60 | 3043.53 | 110 | **2256.42** | **98** | **2450.17** | 60 | 3043.49 |
| sand-castle | 250 | 210 | 743.05 | **230** | 423.49 | **230** | **421.19** | 220 | 583.31 | **230** | **423.67** |
| k_ph_p | 40 | **40** | 38.52 | 20 | 2000.16 | **40** | **10.37** | **39** | **121.59** | 20 | 2000.15 |
| MPEC | 80 | 40 | 2153.34 | 40 | 2061.45 | 40 | **2059.91** | **47** | **1748.68** | 40 | 2060.45 |
| gttt_3x3 | 90 | 90 | 12.98 | 90 | 3.94 | 90 | **3.88** | 90 | 9.76 | 90 | **3.93** |
| arbiter | 100 | 0 | 4000.00 | 0 | 4000.00 | 0 | 4000.00 | **5** | **3800.12** | 0 | 4000.00 |
| depots | 90 | 0 | 4000.00 | 0 | 4000.00 | 0 | 4000.00 | **1** | **3955.56** | 0 | 4000.00 |
| Tree | 140 | 140 | 0.05 | 140 | **0.01** | 140 | **0.01** | 140 | 0.05 | 140 | **0.01** |
| k_branch_n | 100 | **100** | 61.08 | 20 | 3323.22 | **100** | **7.37** | **81** | **824.77** | 20 | 3323.06 |
| PEC | 80 | 40 | 2146.18 | 40 | **2040.21** | 40 | 2040.96 | **50** | **1607.91** | 40 | 2039.86 |
| tlc | 130 | 130 | 0.15 | 130 | 0.06 | 130 | **0.05** | 130 | 0.14 | 130 | **0.09** |
| Adder | 60 | **50** | 669.67 | 40 | 1382.20 | **50** | **667.44** | **50** | **669.60** | 40 | 1383.85 |
| pipesnotankage | 90 | 0 | 4000.00 | 0 | 4000.00 | 0 | 4000.00 | **9** | **3600.06** | 0 | 4000.00 |
| RobotsD2 | 100 | **41** | 2440.63 | 30 | 2803.77 | 40 | **2418.09** | **46** | **2263.88** | 30 | 2803.76 |
| conformant | 240 | 20 | **3671.57** | 20 | 3674.85 | 20 | 3674.86 | **22** | **3652.55** | 20 | 3674.80 |
| stracomp | 600 | 251 | 2363.32 | 80 | 3467.92 | **260** | **2280.44** | **354** | **1669.51** | 80 | 3467.92 |
| Counter | 80 | 70 | 578.93 | 75 | 338.91 | **80** | **284.00** | 78 | **264.96** | **80** | 275.68 |
| ev-pr-4x4-log | 70 | 70 | **0.72** | 70 | 15.41 | 70 | 15.43 | 70 | **8.07** | 70 | 15.27 |
| Connect2 | 160 | 160 | 1.38 | 160 | 42.32 | 160 | **0.29** | 160 | **1.33** | 160 | 40.26 |
| Overall | 3570 | 2212 | 157.36 | 1865 | 193.86 | **2220** | **154.22** | **2361** | **140.03** | 1870 | 193.71 |

Note: The column "#" denotes the total number of instances in each family, the column "#S" denotes the number of successfully computed queries, and the column "PAR2" denotes the average PAR2 score per instance. For each task, the best result out of all settings is marked in bold. The row "Overall" summarizes the total number of instances and the average performance per instance across all families.

Table 1: Experimental results of incremental solving.

judged based on the number of queries completed as well as the penalized average runtime (PAR2) score, which penalizes unfinished instances with two times the time limit.

The first experiment considers the reweighting query. The queries are generated by replacing the probability of each randomized variable with a value chosen uniformly at random from $[0, 1]$ independently. Since enabling pruning is sound for this query (by Theorem 4), we compare all three settings for this task. The second experiment considers the cofactoring query. The queries are generated by sampling a variable and randomly assigning it to $\top$ or $\bot$.

Table 1 shows the result of both experiments. We note that the setting `KC-p` significantly outperforms `KC-np` in the reweighting task, and outperforms `baseline` in all but three families based on the PAR2 score. This result suggests that the pruning techniques are indeed useful and that the task can indeed benefit from the proposed compilation-based incremental solving. On the other hand, `KC-np` fails to beat the performance of `baseline` in both tasks since the number of successfully compiled instances is a lot fewer in many families. Fan and Jiang [2023] reported that the pure literal detection technique has a significant impact on the performance of `SharpSSAT`. Since all ten queries are considered failed when the compilation failed, the benefit of incremental solving is overshadowed by the timeout penalty.

To truly understand the power of incremental solving, we further break down the compilation and query time for the successfully compiled formulas and compare them against the baseline setting. The results are shown in Table 2.[7] We can see that each query only takes around $9.3\,\%$ of the compilation time for the reweighting task and around $5.8\,\%$ for the cofactoring task. On average, it takes $1.53$ and $3.62$ queries for the compilation-based method to outperform the `baseline` for the reweighting and cofactoring task, respectively. This result shows that whenever compilation is successful, it only takes very few queries to compensate for the additional effort.

The reason that the cofactoring task requires more queries is two-fold: First, as discussed in Theorem 3, pruning techniques are not applicable to the task. Therefore, `baseline` has the advantage of utilizing pruning techniques to speed up. Second, the cofactoring query introduces additional unit clauses in the SSAT formula in the `baseline` setting, and may thus be easier to solve.

To summarize, the reweighting task can be readily sped up by the `KC-p` setting, and while the cofactoring task may seem less favorable from Table 1, the breakdown analysis in Table 2 demonstrates its potential.

## 5.2 SSAT Proof Generation and Validation

We ran `cert-SSAT` on a Linux machine with a 12th Gen Intel Core i9-12900 processor, and set a time limit of 1000 sec

[7] We skip the setting `KC-np` for the reweighting task, as it is inferior to `KC-p`.

| | Reweighting | | | | | Cofactoring | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | `baseline` | `KC-p` | | | | `baseline` | `KC-np` | | |
| Family | #C | Total | Compile | Query | Total | #C | Total | Compile | Query | Total |
| ToiletA | 57 | 297.76 | 33.48 | 6.12 | **39.59** | 57 | 153.18 | 34.60 | 7.05 | **41.65** |
| tiger | 5 | 9.63 | 1.67 | 1.36 | **3.03** | 5 | 9.66 | 1.64 | 1.26 | **2.90** |
| MaxCount | 11 | 73.60 | 8.22 | 8.20 | **16.42** | 6 | 4.00 | 1.21 | 2.27 | **3.49** |
| sand-castle | 23 | 489.50 | 54.73 | 46.46 | **101.19** | 23 | 352.04 | 57.04 | 46.64 | **103.67** |
| k_ph_p | 4 | 38.52 | 4.56 | 5.81 | **10.37** | 2 | **0.03** | 0.06 | 0.09 | 0.15 |
| MPEC | 4 | 153.34 | 21.59 | 38.32 | **59.91** | 4 | 74.84 | 20.82 | 39.64 | **60.45** |
| gttt_3x3 | 9 | 12.98 | 1.93 | 1.96 | **3.88** | 9 | 9.76 | 1.93 | 2.00 | **3.93** |
| Tree | 14 | 0.05 | 0.01 | 0.00 | **0.01** | 14 | 0.05 | 0.01 | 0.01 | **0.01** |
| k_branch_n | 10 | 61.08 | 6.14 | 1.23 | **7.37** | 2 | **0.08** | 74.13 | 48.93 | 123.06 |
| PEC | 4 | 146.18 | 17.10 | 23.86 | **40.96** | 4 | 63.30 | 16.46 | 23.40 | **39.86** |
| tlc | 13 | 0.15 | 0.01 | 0.04 | **0.05** | 13 | 0.14 | 0.03 | 0.06 | **0.09** |
| Adder | 5 | 3.00 | 0.37 | 0.41 | **0.77** | 4 | **0.31** | 34.02 | 16.50 | 50.52 |
| RobotsD2 | 4 | 72.46 | 10.65 | 7.44 | **18.09** | 3 | **2.29** | 2.09 | 1.67 | 3.76 |
| conformant | 2 | **4.90** | 5.30 | 2.89 | 8.20 | 2 | 13.69 | 5.23 | 2.90 | **8.13** |
| stracomp | 27 | 148.67 | 20.49 | 48.85 | **69.34** | 8 | **0.01** | 0.51 | 0.74 | 1.26 |
| Counter | 8 | 437.33 | 131.21 | 152.79 | **284.00** | 8 | **251.11** | 124.54 | 151.14 | 275.68 |
| ev-pr-4x4-log | 7 | **0.72** | 14.28 | 1.15 | 15.43 | 7 | **8.07** | 14.11 | 1.16 | 15.27 |
| Connect2 | 16 | 1.38 | 0.22 | 0.07 | **0.29** | 16 | **1.33** | 39.01 | 1.25 | 40.26 |
| Overall | 223 | 240.54 | 32.22 | 30.06 | **62.28** | 187 | 130.38 | 38.97 | 22.74 | **61.70** |

Note: The column "#C" denotes the number of successfully compiled instances. The columns "Compile," "Query,", and "Total" denote the time spent per instance on the compilation, performing the ten queries, and the total time, respectively. For each task, the best result out of all settings is marked in bold. The row "Overall" summarizes the total number of formulas and the average performance per formula across all families.

Table 2: Breakdown results of compilation and query time.

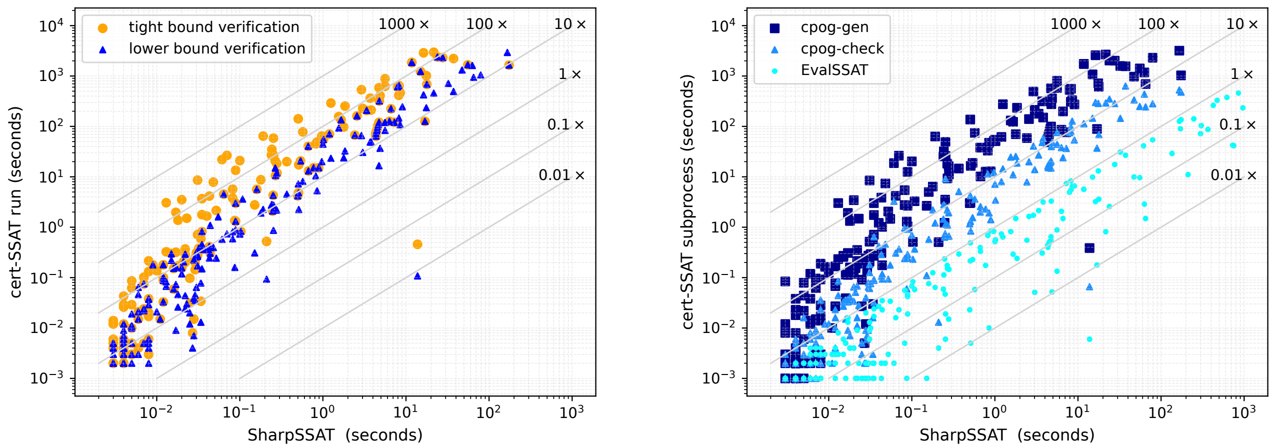
for `SharpSSAT` and `EvalSSAT` and a time limit of 2500 sec for `cpog-gen` and `cpog-check`. We excluded the unsolvable instances by `SharpSSAT` reported in previous work [Fan and Jiang, 2023], yielding 18 families of benchmark with 236 instances in total for the experiment.

The experimental results are summarized as follows. `SharpSSAT` successfully generated `nnf`-proofs for all 236 instances, and all of them passed the check of `EvalSSAT`. For 205 instances (86.9%), `cert-SSAT` successfully proved the easy implication ($\psi_{G_l} \rightarrow \phi$) and hence verified their lower bounds. Among these instances, the other implication ($\phi \rightarrow \psi_{G_u}$) was proved for 190 instances (80.5 %); hence their satisfying probabilities were proved to be tight. For the 190 proven instances, the ratios between the combined time for running `EvalSSAT`, `cpog-gen`, and `cpog-check` versus the time for `SharpSSAT` had a harmonic mean of 6.6. As for proof sizes, the ratios between the number of clauses in `cpog`-proofs and that in the original `sdimacs` files had a harmonic mean of 3.21.

We analyzed the 46 (236 − 190) instances that `cert-SSAT` cannot fully prove or disprove the solving results. Among them, 17 instances suffered from time out; `cpog-gen` unexpectedly aborted on 13 cases. `cpog-gen` failed to generate `cpog`-proofs for all 13 instances in the `tlc` family because the matrix of these instances is unsatisfiable, which `cpog-gen` cannot handle. As an alternative, we can pass their matrices directly into SAT solvers and apply standard proof logging and checking tools [Heule *et al.*, 2013; Wetzler *et al.*, 2014] to verify the results. `cpog-check` detected three cases, `Tiger-25`, `c432_re`, and `cnt03e` violating decomposability. We conjectured that this was caused by integrating both component caching and clause learning in `SharpSSAT`. After we disabled clause learning in `SharpSSAT`, `Tiger-25` passed the overall check of `cert-SSAT`, while `SharpSSAT` timed out on solving the other two instances. We leave fixing these issues for future work.

We provide detailed runtime and proof size analyses of `cert-SSAT` in Fig. 3 and Fig. 4, respectively. Fig. 3a shows the relative runtime between a `cert-SSAT` run (i.e., the combined runtime for `EvalSSAT`, `cpog-gen`, and `cpog-check`) and `SharpSSAT`. We report both the runtime of complete (proving both implications) and partial (proving only the lower bound) runs. The complete (resp. partial) runs are marked in orange (resp. blue) dots (resp. triangles). Fig. 3b shows relative runtime between each `cert-SSAT` subprocess and `SharpSSAT`. Dark-blue squares, sky-blue triangles, and cyan dots represent the results of `cpog-gen`, `cpog-check`, and `EvalSSAT`, respectively. We can see that generating and verifying certificates tend to take much longer when the formula is harder to compile. In addition, the partial verification is often substantially faster than the complete verification. This fact is helpful when certifying the lower bound suffices.

Fig. 4 compares the relative proof size between `cpog`-proofs and `nnf`-proofs. The proof sizes are measured in terms of the number of clauses. We calculate the clause number of `nnf`-proofs by the number of clauses in their corresponding




(a) cert-SSAT run vs. SharpSSAT.

(b) cert-SSAT subprocesses vs. SharpSSAT.

Figure 3: Runtime Analysis.

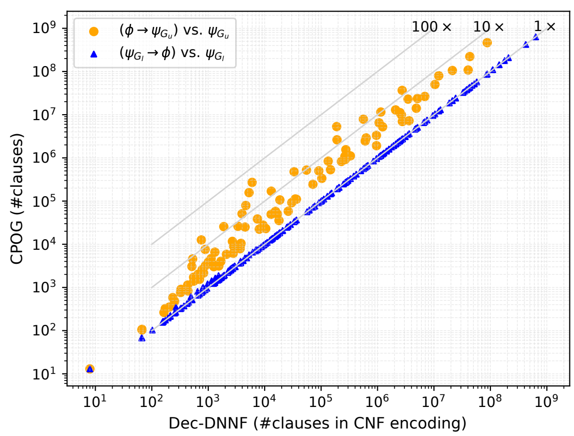



Figure 4: Proof Size Analysis.

CNF encodings. It can be observed that cpog-proofs for the easy implication (marked in blue triangles) are almost linear in size with respect to their corresponding nnf-proofs, while cpog-proofs for the difficult implication (marked in orange dots) are typically larger than their corresponding nnf-proofs.

With cert-SSAT, we were able to discover a tricky bug of SharpSSAT in pure literal detection, which mistakenly negated the phase of pure literals. The bug was hard to find because it did not affect the reported satisfying probability but resulted in wrongly extracted strategies. This use case of cert-SSAT suggests its usefulness in validation solving results and enhancing solver reliability.

## 6 Conclusions and Future Work

We proposed a dec-DNNF-based knowledge compilation technique for SSAT. The technique was implemented based on the state-of-the-art SSAT solver SharpSSAT [Fan and Jiang, 2023], and the toolchain CPOG was extended to generate checkable proofs from the compiled results. Empirical results demonstrated its effectiveness for incremental solving of a wide range of benchmarks, and that checkable dec-DNNF logs can be generated with little computational overhead. Essentially, the cert-SSAT enhanced the trustworthiness of SharpSSAT results.

For future work, we remark that our implementation may serve as a general-purpose level-ordered dec-DNNF compiler, analogous to the dec-DNNF compilers based on #SAT solvers. It would be interesting to explore more applications of the compiler. In addition, the current supported incremental solving is restricted to reweighting and cofactoring queries. It might be worthwhile to explore other powerful techniques, such as clause-learning-based incremental QBF solving [Lonsing and Egly, 2014] and dynamic model counting [Li *et al.*, 2006], to allow addition and deletion of literals and clauses. For cert-SSAT, we plan to make the entire toolchain formally verified with theorem provers. We also plan to extend it to support proof generation and validation for SSAT preprocessor and DSSAT solver [Cheng and Jiang, 2023].

## Acknowledgements

This work was supported in part by the National Science and Technology Council of Taiwan under Grant NSTC 111-2923-E-002-013-MY3, 112-2119-M-002-017, and 113-2119-M-002-024.

## Contribution Statement

C.C. and Y.R.L. contributed equally.

## References

[Bryant *et al.*, 2023] Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 271, pages 6:1–6:20, 2023.

[Capelli and Mengel, 2019] Florent Capelli and Stefan Mengel. Tractable QBF by knowledge compilation. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Science*, volume 126, pages 18:1–18:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[Capelli *et al.*, 2021] Florent Capelli, Jean-Marie Lagniez, and Pierre Marquis. Certifying top-down decision-DNNF compilers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6244–6253, 2021.

[Capelli, 2019] Florent Capelli. Knowledge compilation languages as proof systems. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 90–99, 2019.

[Chen *et al.*, 2021] 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.

[Cheng and Jiang, 2023] Che Cheng and Jie-Hong R. Jiang. Lifting (D)QBF preprocessing and solving techniques to (D)SSAT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3906–3914. AAAI Press, 2023.

[Clarke *et al.*, 2018] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of Model Checking*. Springer International Publishing, 2018.

[Coste-Marquis *et al.*, 2005] Sylvie Coste-Marquis, Daniel Le Berre, Florian Letombe, and Pierre Marquis. Propositional fragments for knowledge compilation and quantified Boolean formulae. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, pages 288–293. AAAI Press, 2005.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17(1):229–264, 2002.

[Fan and Jiang, 2023] Yu-Wei Fan and Jie-Hong R. Jiang. SharpSSAT: A witness-generating stochastic Boolean satisfiability solver. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 3949–3958, 2023.

[Fargier and Marquis, 2006] Hélène Fargier and Pierre Marquis. On the use of partially ordered decision graphs for knowledge compilation and quantified Boolean formulae. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, pages 42–47. AAAI Press, 2006.

[Gelder, 2008] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, 2008.

[Ghosh *et al.*, 2021] Bishwamittra Ghosh, Debabrota Basu, and Kuldeep S. Meel. Justicia: A stochastic SAT approach to formally verify fairness. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(9):7554–7563, 2021.

[Goldberg and Novikov, 2003] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe*, volume 1, pages 886–891, 2003.

[Heule *et al.*, 2013] Marijn J.H. Heule, Warren A. Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*, pages 181–188, 2013.

[Hsieh and Jiang, 2022] Cheng-Han Hsieh and Jie-Hong R. Jiang. Encoding probabilistic graphical models into stochastic Boolean satisfiability. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1834–1842. IJCAI Organization, 2022.

[Huang and Darwiche, 2005] Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 156–162. Morgan Kaufmann Publishers Inc., 2005.

[Lagniez and Marquis, 2017] Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 667–673. AAAI Press, 2017.

[Lai *et al.*, 2017] Yong Lai, Dayou Liu, and Minghao Yin. New canonical representations by augmenting OBDDs with conjunctive decomposition. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 5010–5014. AAAI Press, 2017.

[Lee and Jiang, 2018] Nian-Ze Lee and Jie-Hong R. Jiang. Towards formal evaluation and verification of probabilistic design. *IEEE Transactions on Computers*, 67(8):1202–1216, 2018.

[Lee *et al.*, 2018] 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. AAAI Press, 2018.

[Li *et al.*, 2006] Wei Li, Peter van Beek, and Pascal Poupart. Performing incremental Bayesian inference by dynamic model counting. In *Proceedings of the National Conference on Artificial Intelligence*, volume 2, pages 1173–1179. AAAI Press, 2006.

[Lonsing and Egly, 2014] Florian Lonsing and Uwe Egly. Incremental QBF solving. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 514–530. Springer International Publishing, 2014.

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

[Muise *et al.*, 2012] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF compilation with SharpSAT. In *Proceedings of the Canadian Conference on Advances in Artificial Intelligence*, pages 356–361. Springer-Verlag, 2012.

[Ojala, 2022] Oskari Ojala. Evaluation of SSAT by reduction to decision diagrams. Master's thesis, Aalto University, 2022.

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

[Salmon and Poupart, 2020] Ricardo Salmon and Pascal Poupart. On the relationship between satisfiability and Markov decision processes. In *Proceedings of the Uncertainty in Artificial Intelligence Conference*, volume 115, pages 1105–1115. PMLR, 2020.

[Tseitin, 1983] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. *Automation of Reasoning*, pages 466–483, 1983.

[Wang *et al.*, 2022] Hao-Ren Wang, Kuan-Hua Tu, Jie-Hong Roland Jiang, and Christoph Scholl. Quantifier elimination in stochastic Boolean satisfiability. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 23:1–23:17, 2022.

[Wetzler *et al.*, 2014] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429, 2014.