Journal of Artificial Intelligence Research 62 (2018) 1-32 Submitted 11/16; published 05/18

An Exhaustive DPLL Algorithm for Model Counting

Umut Oztok UMUT@CS.UCLA.EDU
Adnan Darwiche DARWICHE@CS.UCLA.EDU
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095

Abstract

State-of-the-art model counters are based on exhaustive DPLL algorithms, and have been successfully used in probabilistic reasoning, one of the key problems in AI. In this article, we present a new exhaustive DPLL algorithm with a formal semantics, a proof of correctness, and a modular design. The modular design is based on the separation of the core model counting algorithm from SAT solving techniques. We also show that the *trace* of our algorithm belongs to the language of Sentential Decision Diagrams (SDDs), which is a subset of Decision-DNNFs, the trace of existing state-of-the-art model counters. Still, our experimental analysis shows comparable results against state-of-the-art model counters. Furthermore, we obtain the first *top-down* SDD compiler, and show orders-of-magnitude improvements in SDD construction time against the existing *bottom-up* SDD compiler.

1. Introduction

Model counting is the problem of determining the number of satisfying assignments of a propositional formula. Being a #P-complete problem (Valiant, 1979), model counting is central to many AI problems such as probabilistic reasoning (Roth, 1996; Darwiche, 2002b), and state-of-the-art model counters have been successfully used for doing probabilistic inference (Chavira & Darwiche, 2005; Sang et al., 2005; Chavira et al., 2006; Chavira & Darwiche, 2008; Fierens et al., 2015). Up-to-date, state-of-the-art model counters have been based on *exhaustive DPLL* algorithm (Birnbaum & Lozinskii, 1999), which counts the models of a Boolean formula by searching the space of truth assignments until identifying all the satisfying ones. Those counters are divided into two main categories: ones that save the trace of the search performed by exhaustive DPLL algorithm (Huang & Darwiche, 2007), and ones not saving the trace.[1] In the latter case, one immediately obtains the model count, whereas the former case constructs a graph structure over which the model count can be obtained easily and efficiently. Furthermore, these model counters generally augment exhaustive DPLL with some other effective techniques, such as component analysis (Bayardo & Pehoushek, 2000) and component caching (Majercik & Littman, 1998). The latter technique is used to avoid counting the models of the same components multiple times. The former technique is used to identify disconnected components and count their models independently to improve efficiency. Component analysis has been done using two methods: either a *static* one identifying the components a priori search (Darwiche, 2002a, 2004), or a *dynamic* one identifying the components during the search (Bayardo & Pehoushek, 2000; Sang et al., 2004). For instance, CACHET (Sang et al., 2004) performs dynamic component

1. See Section 4 for a detailed discussion of the trace of an exhaustive search algorithm.

analysis and does not save the trace, whereas c2d (Darwiche, 2004) performs static component analysis and saves the trace. Another model counter sharpSAT (Thurley, 2006) performs dynamic component analysis and does not save the trace. Yet, DSharp (Muise et al., 2012) is obtained by simply saving the trace of sharpSAT. For all these model counters, the trace of the search (regardless of saved or not) is in a propositional language, called Decision-DNNF (Huang & Darwiche, 2007), which is tractable for model counting.

In this article, we introduce yet another exhaustive DPLL algorithm. Our algorithm uses static component analysis, which is performed by using a new structure, called *decision vtree* (Oztok & Darwiche, 2014). Because of this new structure, our algorithm comes with a new trace in the language of Sentential Decision Diagrams (SDDs) (Darwiche, 2011) that is a subset of Decision-DNNF. As such, it can be seen as performing more work compared to other model counters. Despite this fact, we empirically show that its performance is comparable against state-of-the-art model counters. Further, by saving the trace of our algorithm, we obtain the first *top-down* SDD compiler, which shows orders-of-magnitude improvements in compilation times against the state-of-the-art SDD compiler (Choi & Darwiche, 2013b).

Our algorithm has a modular design that enables easy integration with SAT solving techniques. This is due to a new interface that separates the core model counting algorithm from SAT solving technology, allowing one to plugin different SAT solvers without changing the counting algorithm. Beyond significantly enhancing the clarity of our source code,[2] this modularity is critical for the formal semantics of the counting algorithm and its proof of correctness, both of which are lacking in the previous model counters. Our hope is that this will facilitate the development of model counters, particularly, their extensibility and easier integration with new SAT solvers. For instance, clause-learning is a crucial technique in the success of modern SAT solvers. Indeed, there exists various clause-learning schemes implemented by SAT solvers, leading to significant performance gains depending on the class of CNFs used (see, e.g., Pipatsrisawat & Darwiche, 2008a). We believe that similar gains could be obtained for model counting by using our new framework.

This paper is organized as follows. Section 2 introduces some background and reviews the core algorithm behind modern SAT solvers. It also introduces the new formal framework that our model counting algorithm will be based on. Section 3 describes the new model counting algorithm in detail, together with its proof of correctness and experimental evaluation. Section 4 reviews the close connection between exhaustive DPLL algorithm and knowledge compilation by showing that the traces of exhaustive DPLL algorithms correspond to certain knowledge compilation languages. It also shows how our model counting algorithm can be used in the context of knowledge compilation (i.e., in the compilation of SDDs), along with an empirical evaluation of the algorithm as a knowledge compiler. After the related work in Section 5, we conclude the paper. The appendix contains the proofs.

2. SAT Solving by CDCL

In this section, we will review the CDCL algorithm that forms the basis of modern SAT solvers, and will introduce a new formal framework that will be used in our model counter. We start by defining some technical preliminaries.

2. The source code of our system, miniC2D, is available at `http://reasoning.cs.ucla.edu/minic2d`.

Algorithm 1: SAT(Δ)

Input: Δ : a CNF
Output: $\top$ if Δ is satisfiable; $\bot$ otherwise

```
1  Γ ← {} // learned clauses
2  D ← ⟨⟩ // decision sequence
3  while true do
4      if unit resolution detects a contradiction in Δ ∧ Γ ∧ D then
5          if D = ⟨⟩ then return ⊥
6          α ← asserting clause for (Δ, Γ, D)
7          m ← the assertion level of α
8          D ← the first m decisions of D
9          Γ ← Γ ∪ {α} // learning clause α
10     else
11         if ℓ is a literal where neither ℓ nor ¬ℓ are implied by unit resolution from Δ ∧ Γ ∧ D then
12             D ← D; ℓ // add a new decision to D
13         else return ⊤
```

Upper-case letters (e.g., X) denote variables and lower-case letters (e.g., x) denote their instantiations. That is, x is a *literal* denoting X or $\neg X$. Bold upper-case letters (e.g., $\mathbf{X}$) denote sets of variables and bold lower-case letters (e.g., $\mathbf{x}$) denote their instantiations. A *Boolean function* $f(\mathbf{Z})$ maps each instantiation $\mathbf{z}$ of variables $\mathbf{Z}$ to true ($\top$) or false ($\bot$). A *conjunctive normal form* (CNF) is a set of clauses, where each clause is a disjunction of literals. For instance, the CNF $\{X \vee \neg Y \vee \neg Z,\ Y \vee Z,\ \neg X\}$ represents the Boolean function $(X \vee \neg Y \vee \neg Z) \wedge (Y \vee Z) \wedge \neg X$. Conditioning a CNF Δ on a literal ℓ, denoted $\Delta|\ell$, amounts to removing literal $\neg\ell$ from all clauses and then dropping all clauses that contain literal ℓ. For two CNFs Δ and Γ, we write $\Delta \models \Gamma$ to mean that Δ entails Γ. For a CNF Δ, we write $\Delta \vdash I$ to mean that I is the set of literals derived from Δ using unit resolution.

Modern SAT solvers utilize two powerful and complementary techniques: *unit resolution* and *clause learning*. Unit resolution is an efficient, but incomplete, inference rule which identifies some of the literals implied by a CNF. Clause learning is a process which identifies clauses that are implied by a CNF, then adds them to the CNF so as to empower unit resolution (i.e., allows it to derive more literals). These clauses, also called *asserting clauses,* are learned when unit resolution detects a contradiction in the given CNF. We will neither justify asserting clauses, nor delve into the details of computing them, since these clauses have been well justified and extensively studied in the SAT literature (see, e.g., Moskewicz et al., 2001). We will, however, employ asserting clauses in our model counter (we employ first-UIP asserting clauses as implemented in RSat, Pipatsrisawat & Darwiche, 2007).

We now present in Algorithm 1 a modern SAT solver that is based on unit resolution and clause learning (a.k.a, a *conflict-driven clause learning* (CDCL) solver). This algorithm takes as input a CNF Δ. It maintains a set of clauses Γ (for learned clauses) and a decision sequence D (for literal assignments), both of which are initially empty. Given Δ, Γ, and D, the solver repeatedly performs the following process. A literal ℓ is chosen and added to the decision sequence D (we say that ℓ has been decided at *level* $|D|$). After deciding the literal ℓ, unit resolution is performed on $\Delta \wedge \Gamma \wedge D$. If no contradiction is found, another literal is decided. Otherwise, an asserting clause α is identified. A number of decisions are then

Macro : *decide_literal*$(\ell, S = (\Delta, \Gamma, D, I))$
$D \leftarrow D; \ell$ `// add a new decision to` D
if *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
  return *an asserting clause for* (Δ, Γ, D)
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
return *success*

Macro : *undo_decide_literal*$(\ell, S = (\Delta, \Gamma, D, I))$
erase the last decision ℓ from D
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$

Macro : *at_assertion_level*$(\alpha, S = (\Delta, \Gamma, D, I))$
$m \leftarrow$ assertion level of α
if *there are* m *literals in* D **then return** true
else return false

Macro : *assert_clause*$(\alpha, S = (\Delta, \Gamma, D, I))$
$\Gamma \leftarrow \Gamma \cup \{\alpha\}$ `// add learned clause to` Γ
if *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
  return *an asserting clause for* (Δ, Γ, D)
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
return *success*

Figure 1: Macros for some SAT-solver primitives.

erased until we reach the decision level corresponding to the *assertion level* of clause α, at which point α is added to Γ.[3] The solver terminates under one of two conditions: either a contradiction is found under an empty decision sequence D (Line 5), or all literals are successfully decided (Line 13). In the first case, the input CNF must be unsatisfiable. In the second case, the CNF is satisfiable with D as a (partial) satisfying assignment.[4]

As we discussed earlier, one of our goals is to obtain a model counter that can benefit from the advances in SAT solving in a modular manner. This requires an interface that separates SAT solving techniques from the core model counting algorithm. This way, by only changing the implementation of the interface (e.g., using a different SAT solver), one can immediately improve the performance of the model counter. For that, we will abstract the primitives used in SAT solvers (Figure 1), viewing them as operations on what we shall call a SAT state.

Definition 1 (SAT State)**.** *A SAT state is a tuple* $S = (\Delta, \Gamma, D, I)$ *where* Δ *and* Γ *are sets of clauses,* D *is a sequence of literals, and* I *is a set of literals, such that* $\Delta \models \Gamma$ *and* $\Delta \wedge \Gamma \wedge D \vdash I$*. The number of literals in* D *is called the decision level of* S*. Moreover,* S *is said to be satisfiable iff* $\Delta \wedge D$ *is satisfiable.*[5]

Here, Δ is the input CNF, Γ is the set of learned clauses, and D is the decision sequence.

3. The assertion level is computed when the clause is learned. It equals the lowest decision level at which unit resolution is guaranteed to derive a new literal using the learned clause (Moskewicz et al., 2001).
4. Note that D can be partial as the algorithm will stop instantiating variables once all variables are implied. When D is partial, the variables not appearing in D can take any value assignment.
5. Without loss of generality, Δ has no empty or unit clauses. If Δ has an empty clause, one can immediately detect that it is unsatisfiable, leading to model count of 0. If Δ has unit clauses, Δ is equivalent to $I \wedge \Delta|I$ where $\Delta \vdash I$, leading to the model count of Δ being equal to the model count of $\Delta|I$. So, counting the models of $\Delta|I$ (which has no unit clauses) is enough to obtain the model count of Δ.

We now provide a small description of the primitives in Figure 1:

- *decide_literal*$(\ell, S = (\Delta, \Gamma, D, I))$ takes as input a literal ℓ and a SAT state S. It then adds literal ℓ to the decision sequence D and runs unit resolution. If unit resolution detects a conflict in the SAT state, it then constructs an asserting clause and returns it. Otherwise, it simply updates the set of implied literals I.
- *undo_decide_literal*$(\ell, S = (\Delta, \Gamma, D, I))$ takes as input a literal ℓ and a SAT state S. It simply removes literal ℓ from the decision sequence D and updates the set of implied literals I.
- *at_assertion_level*$(\alpha, S = (\Delta, \Gamma, D, I))$ takes as input a clause α and a SAT state S. It decides if the assertion level of clause α is the same as the number of literals in the decision sequence D (i.e., whether the decision level is the same as the assertion level of clause α).
- *assert_clause*$(\alpha, S = (\Delta, \Gamma, D, I))$ takes as input a clause α and a SAT state $S = (\Delta, \Gamma, D, I))$. It then adds α to the set of learned clauses Γ and runs unit resolution. If unit resolution detects a conflict in the SAT state, it then constructs an asserting clause and returns it. Otherwise, it simply updates the set of implied literals I.

To further prepare for our model counting algorithm, we now present in Algorithm 2 a CDCL solver that makes use of the new SAT interface. Note that this algorithm is recursive, just like our model counting algorithm will be.

Algorithm 2: SAT(S)

Input: S : a SAT state (Δ, Γ, D, I)
Output: *success* or $\bot$

```
1  find a literal ℓ where neither ℓ nor ¬ℓ are implied by unit resolution from Δ ∧ Γ ∧ D
2  if there is no such literal ℓ then return success
3  ret ← decide_literal(ℓ, S)
4  if ret is a success then ret ← SAT(S)
5  undo_decide_literal(ℓ, S)
6  if ret is a learned clause then
7      if at_assertion_level(ret, S) then
8          ret ← assert_clause(ret, S)
9          if ret is a success then return SAT(S)
10         else return ret
11     else return ret
12 else return success
```

3. Model Counting by Exhaustive DPLL

In this section, we will introduce a new model counting algorithm based on exhaustive DPLL, together with its proof of correctness and experimental analysis.

Our algorithm will take as input a CNF and will return its model count. It will be recursive and will utilize the SAT state and its associated primitives in Figure 1. Before

Algorithm 3: $\#\texttt{SAT}(\pi, S)$

Input: π : a variable order, S : a SAT state (Δ, Γ, D, I)
Output: Model count of $\Delta \wedge D$, or a clause

```
1  if there is no variable in π then return 1
2  X ← first variable in π
3  if X or ¬X belongs to I then return #SAT(π\{X}, S)
4  h ← decide_literal(X, S)
5  if h is success then h ← #SAT(π\{X}, S)
6  undo_decide_literal(X, S)
7  if h is a learned clause then
8      if at_assertion_level(h, S) then
9          h ← assert_clause(h, S)
10         if h is success then return #SAT(π, S)
11         else return h
12     else return h
13 l ← decide_literal(¬X, S)
14 if l is success then l ← #SAT(π\{X}, S)
15 undo_decide_literal(¬X, S)
16 if l is a learned clause then
17     if at_assertion_level(l, S) then
18         l ← assert_clause(l, S)
19         if l is success then return #SAT(π, S)
20         else return l
21     else return l
22 return h + l
```

introducing the full algorithm, we will start with a simpler version for presentation purposes. This is given in Algorithm 3, which is initially called with the SAT state $(\Delta, \{\}, \langle\rangle, \{\})$, along with an order π of the variables of Δ. In a recursive call with a SAT state $(\Delta, ., D, .)$, the algorithm will attempt to count the models of $\Delta \wedge D$. For that, we first pick a variable X from order π (Line 2). If a literal ℓ of X is already implied in the current SAT state, then we simply count the models of $(\Delta \wedge D)|\ell$ recursively (Line 3) as it will be the same as the model count of $\Delta \wedge D$. Otherwise, we try to count the models of $\Delta \wedge D \wedge X$ recursively (Lines 4–5). If the result is a model count (otherwise, see the below key observation), we try to count the models of $\Delta \wedge D \wedge \neg X$ recursively (Lines 13–14). If this second phase is also successful, we then obtain the model count of $\Delta \wedge D$ (Line 22). What makes this algorithm additionally useful for our presentation purposes is that it is *exhaustive* in nature. That is, when considering variable X, it must process both its phases, X and $\neg X$. This is similar to our model counting algorithm — but in contrast to SAT solvers which only consider one phase of the variable. Moreover, Algorithm 3 employs the primitives of Figure 1 in the same way that our model counter will employ them later.

The following is a key observation about Algorithm 3 (and our model counting algorithm). When a recursive call returns a learned clause, instead of a model count, this only means that while counting the models of the CNF $\Delta \wedge D$ targeted by the call, unit resolution has discovered an opportunity to learn a clause (and learned one). Hence, we must backtrack to the assertion level, add the clause, and then try again (Lines 10 and 19). In particular, returning a learned clause does not necessarily mean that the CNF targeted by

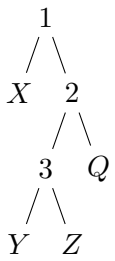


Figure 2: A decision vtree for the CNF $\{Y \vee \neg Z,\ \neg X \vee Z,\ X \vee \neg Y,\ X \vee Q\}$.

the recursive call is unsatisfiable. The only exception is the root call, for which the return of a learned clause implies an unsatisfiable CNF (and, hence, a zero model count) since the learned clause must be empty in this case.[6]

Our algorithm will augment Algorithm 3 with two additional techniques. First, we will perform component analysis to identify disconnected CNF components and count their models independently. For that, we will use a new control structure to guide the process of decomposing a CNF into disconnected components. Second, we will employ a component caching scheme to avoid counting the models of the same CNF component multiple times. We next describe the new control structure.

3.1 Component Analysis by Decision Vtrees

A *vtree* (Pipatsrisawat & Darwiche, 2008b) is a full binary tree whose leaves are labeled with variables; see Figure 2. The control structure we will use to guide our algorithm will be a special type of vtree, called *decision vtree* (Oztok & Darwiche, 2014), defined specifically for CNFs. To define decision vtrees, we first need to distinguish between internal vtree nodes as follows. An internal vtree node is a *Shannon* node if its left child is a leaf, otherwise it is a *decomposition* node. The variable labeling the left child of a Shannon node is called the *Shannon variable* of the node. Vtree nodes 1 and 3 in Figure 2 are Shannon nodes, with X and Y as their Shannon variables, respectively. Vtree node 2 is a decomposition node. The left (resp., right) child of an internal vtree node v will be denoted by v^l (resp., v^r).

Definition 2 (Decision Vtree)**.** *A clause is compatible with an internal vtree node v iff the clause mentions some variables inside v^l and some variables inside v^r. A vtree for CNF Δ is a decision vtree for Δ iff every clause in Δ is compatible with only Shannon vtree nodes.*

Figure 2 depicts a decision vtree for the CNF $\{Y \vee \neg Z,\ \neg X \vee Z,\ X \vee \neg Y,\ X \vee Q\}$.

A decision vtree for a CNF can be thought of as an auxiliary structure that allows one to perform static component analysis on the CNF, due to the following proposition.

Proposition 1. *Consider a CNF and a corresponding decision vtree. Take a path from the vtree root to a decomposition vtree node v. Then, on this path, as long as one conditions the CNF at every Shannon vtree node on the corresponding Shannon variable, the conditioned*

6. When the decision sequence D is empty, and unit resolution detects a contradiction in $\Delta \wedge \Gamma$, the only learned clause is the empty clause, which implies that Δ is unsatisfiable (since $\Delta \models \Gamma$).

CNF associated with the vtree node v must be decomposed into two sub-components; one is over the variables of the left child v^l and the other is over the variables of the right child v^r.

To see why Proposition 1 is correct, recall that each clause in the CNF can be compatible with only Shannon vtree nodes. Hence, if the conditioned CNF does not decompose into two sub-components, then it would imply that there is a clause that is compatible with a decomposition vtree node, which contradicts with the vtree being a decision one. As such, the input to our model counter will be a CNF and a corresponding decision vtree. Note that we can always construct a decision vtree for any CNF (Oztok & Darwiche, 2014).[7]

3.2 A New Model Counting Algorithm

We are now ready to present the final version of our model counting algorithm. This is given in Algorithm 4, which is called initially with the SAT state $S = (\Delta, \{\}, \langle\rangle, \{\})$ and a decision vtree v for Δ. When the algorithm is applied to a Shannon vtree node, its behavior is similar to Algorithm 3 (Lines 15–43). That is, it basically uses the Shannon variable X and considers its two phases, X and $\neg X$. However, when applied to a decomposition vtree node v (Lines 5–14), one is guaranteed that the CNF associated with v is decomposed into two components, one associated with the left child v^l and another with the right child v^r (due to Proposition 1). In this case, the algorithm counts the models for each component independently and combines the results.

3.3 Soundness of the Algorithm

We will next show the soundness of the algorithm, which requires some additional definitions. Let Δ be the input CNF. Each vtree node v is then associated with the following:

- $Vars(v)$: The variables inside v.
- $CNF(v)$: The clauses of Δ mentioning only variables inside v (*clauses of* v).
- $ContextC(v)$: The clauses of Δ mentioning some variables inside v and some outside v (*context clauses* of v).
- $ContextV(v)$: The Shannon variables of all vtree nodes that are ancestors of v (*context variables of* v). $ContextC(v)$ will only mention variables in $Vars(v) \cup ContextV(v)$.
- $ContextL(v, I)$: The literals of $ContextV(v)$ appearing in a given set of literals I.[8]

We start with the following invariant of Algorithm 4.

Theorem 1. *Consider a call* #SAT(v, S) *with* $S = (., ., D, I)$. *Then,* $D \subseteq ContextL(v, I)$ *and* $ContextL(v, I)$ *contains exactly one literal for each variable of* $ContextV(v)$.

7. One can construct a decision vtree based on hypergraph partitioning or variable ordering (such as min-fill order). The former would yield more balanced vtrees, whereas the latter would yield vtrees with lower treewidth. In our experiments, we chose the latter to construct decision vtrees.
8. Note that $ContextL(v, I)$ is a set of literals and $ContextV(v)$ is a set of variables, and hence $ContextL(v, I)$ is not the same as $ContextV(v) \cap I$.

Algorithm 4: $\#\mathtt{SAT}(v, S)$

Input: v : a vtree node, S : a SAT state (Δ, Γ, D, I)
Output: A model count or a clause

```
1  if v is a leaf node then
2      X ← variable of v
3      if X or ¬X belongs to I then return 1
4      else return 2
5  else if v is a decomposition vtree node then
6      left ← #SAT(v^l, S)
7      if left is a learned clause then
8          clean_cache(v^l)
9          return left
10     right ← #SAT(v^r, S)
11     if right is a learned clause then
12         clean_cache(v)
13         return right
14     return left × right
15 else
16     key ← Key(v, S)
17     if cache(key) ≠ nil then return cache(key)
18     X ← Shannon variable of v
19     if either X or ¬X belongs to I then
20         right ← #SAT(v^r, S)
21         if right is a learned clause then return right
22         return right
23     h ← decide_literal(X, S)
24     if h is success then h ← #SAT(v^r, S)
25     undo_decide_literal(X, S)
26     if h is a learned clause then
27         if at_assertion_level(h, S) then
28             h ← assert_clause(h, S)
29             if h is success then return #SAT(v, S)
30             else return h
31         else return h
32     l ← decide_literal(¬X, S)
33     if l is success then l ← #SAT(v^r, S)
34     undo_decide_literal(¬X, S)
35     if l is a learned clause then
36         if at_assertion_level(l, S) then
37             l ← assert_clause(l, S)
38             if l is success then return #SAT(v, S)
39             else return l
40         else return l
41     sum ← h + l
42     cache(key) ← sum
43     return sum
```

Hence, when calling vtree node v, all its context variables must be either decided or implied. We can now define the CNF component associated with a vtree node v at state S.

Definition 3. *The component of vtree node v and state $S = (.,.,.,I)$ is $CNF(v,S) = CNF(v) \wedge ContextC(v)|\gamma$, where $\gamma = ContextL(v,I)$.*

Hence, component $CNF(v,S)$ will only mention variables in vtree v, upon a call $\#\texttt{SAT}(v,S)$. Further, the root component ($CNF(v,S)$ with v being the root vtree node) is equal to Δ.

Following is the soundness result assuming no component caching (i.e., while omitting Lines 8, 12, 16, 17 and 42). We break the result into two cases as follows.

Theorem 2. *A call $\#\texttt{SAT}(v,S)$ with a satisfiable state S will return either the model count of component $CNF(v,S)$ or a learned clause. Moreover, if v is the root vtree node, then it will return the model count of $CNF(v,S)$.*

Theorem 3. *A call $\#\texttt{SAT}(v,S)$ with an unsatisfiable state S will return a learned clause, or one of its ancestral calls $\#\texttt{SAT}(v',.)$ will return a learned clause, where v' is a decomposition vtree node.*

We now have our soundness result (without caching).

Corollary 1. *If v is the root vtree node, then call $\#\texttt{SAT}(v,(\Delta,\{\},\langle\rangle,\{\}))$ returns the model count of Δ if Δ is satisfiable, and returns an empty clause if Δ is unsatisfiable.*

We are now ready to discuss the soundness of our caching scheme (Lines 8, 12, 16, 17 and 42). This requires an explanation of the difference in behavior between satisfiable and unsatisfiable states (based on Thm. 1 of Sang et al., 2004). Consider the component CNFs $\Delta_{\mathbf{X}}$ and $\Delta_{\mathbf{Y}}$ over disjoint variables $\mathbf{X}$ and $\mathbf{Y}$, and let Γ be another CNF such that $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}} \models \Gamma$ (think of Γ as some learned clauses). Suppose that $I_{\mathbf{X}}$ is the set of literals over variables $\mathbf{X}$ implied by unit resolution from $\Delta_{\mathbf{X}} \wedge \Gamma$. One would expect that $\Delta_{\mathbf{X}} \equiv \Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ (and similarly for $\Delta_{\mathbf{Y}}$). Moreover, one would prefer to count the models of $\Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ instead of $\Delta_{\mathbf{X}}$ as the former can make unit resolution more complete, leading to a more efficient counting algorithm. In fact, this is exactly what Algorithm 4 does, as it includes the learned clauses Γ in unit resolution when counting a component. However, $\Delta_{\mathbf{X}} \equiv \Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ is not guaranteed to hold unless $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}}$ is satisfiable. When this is not the case, counting the models of $\Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ will only yield a lower bound on the model count of $\Delta_{\mathbf{X}}$ but is not necessarily equivalent to it. However, this is not problematic for our algorithm, for the following reason. If $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}}$ is unsatisfiable, then either $\Delta_{\mathbf{X}}$ or $\Delta_{\mathbf{Y}}$ is unsatisfiable and, hence, either $\Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ or $\Delta_{\mathbf{Y}} \wedge I_{\mathbf{Y}}$ will be unsatisfiable, and their conjunction will be unsatisfiable. Hence, even though one of the components was counted incorrectly, the conjunction remains a valid result. Without component caching, the incorrect model counts will be discarded. However, with component caching, one also needs to ensure that incorrect model counts are not cached (as observed by Sang et al., 2004).[9]

9. Another possible solution to this is to ensure that $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}}$ is satisfiable before counting the models. This can be achieved by two separate SAT calls on $\Delta_{\mathbf{X}}|I_{\mathbf{X}} \wedge I_{\mathbf{X}}$ and $\Delta_{\mathbf{Y}}|I_{\mathbf{Y}} \wedge I_{\mathbf{Y}}$ (which will appear in the current SAT state). If either one of the components is unsatisfiable, then $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}}$ will be unsatisfiable, so that we can backtrack without doing any counting and caching. On the other hand, if both components are satisfiable, then $\Delta_{\mathbf{X}} \wedge \Delta_{\mathbf{Y}}$ will be satisfiable, and hence we can safely do model counting and caching. One caveat here is that when both components are satisfiable, the SAT calls will be redundant and will incur overhead. So, one needs to perform comprehensive experiments to compare this method with our current solution which is to discard (possibly) wrong cache results.

By Theorem 3, if we reach Line 8 or Line 12, then state S may be unsatisfiable and we can no longer trust the results cached below v. Hence, *clean_cache*(v) on Lines 8 and 12 removes all cache entries that are indexed by $Key(v', .)$, where v' is a descendant of v. We now discuss Lines 16, 17 and 42, which describe the way we cache results.

Definition 4. *Let v be a vtree node, and let $S = (., ., ., I)$ and $S' = (., ., ., I')$ corresponding SAT states. Let $\mathbf{x} \subseteq I$ (resp., $\mathbf{x}' \subseteq I'$) be the instantiation of variables appearing in both v and $ContextC(v)$. A function $Key(v, S)$ is called a component key iff $Key(v, S) = Key(v, S')$ implies that components $CNF(v, S) \wedge \mathbf{x}$ and $CNF(v, S') \wedge \mathbf{x}'$ are equivalent.*[10]

Hence, as long as Line 16 uses a component key according to this definition, then caching is sound. The following theorem describes the component key we used in our algorithm.

Theorem 4. *Consider a vtree node v and a corresponding SAT state $S = (., ., ., I)$. Define $Key(v, S)$ as the following bit vector: (1) each clause δ in $ContextC(v)$ is mapped into one bit that captures whether $I \models \delta$, and (2) each variable X that appears in both vtree v and $ContextC(v)$ is mapped into two bits that capture whether $X \in I$, $\neg X \in I$, or neither. Then function $Key(v, S)$ is a component key.*

3.4 Weighted Model Counting

In many applications (e.g., probabilistic inference), a generalized version of model counting is required, called *weighted model counting* (e.g., Sang et al., 2005; Chavira & Darwiche, 2008), where the *weighted* model count of a CNF is defined as the sum of the weights of its models — as opposed to only the number of models. The weight of a model is defined as the multiplication of the weights of the literals the model has. So, if each literal has weight 1, then the problem reduces to model counting as we have discussed so far.

It is worth to mention here that our model counting algorithm can be trivially adapted to do weighted model counting. All we need is to apply the changes in Table 1.

Line	Modification
3	**return** $weight(X)$ or $weight(\neg X)$ whichever X or $\neg X$ belongs to I
4	**return** $weight(X) + weight(\neg X)$
22	**return** $weight(X) \times right$
41	$sum \leftarrow (weight(X) \times h) + (weight(\neg X) \times l)$

Table 1: Changes to adapt Algorithm 4 to do weighted model counting. $weight(\ell)$ returns the weight of a literal ℓ.

3.5 Experiments

We will now present an empirical evaluation of our model counter miniC2D, which is a C implementation of Algorithm 4. In our experiments we used 1392 CNFs available at `http://www.cril.univ-artois.fr/PMC/pmc.html`, which come from different applications such as planning and product configuration. We compared our model counter against

10. This definition corrects Definition 5 in the work of Oztok and Darwiche (2015), which misses the sets $\mathbf{x}$ and $\mathbf{x}'$.

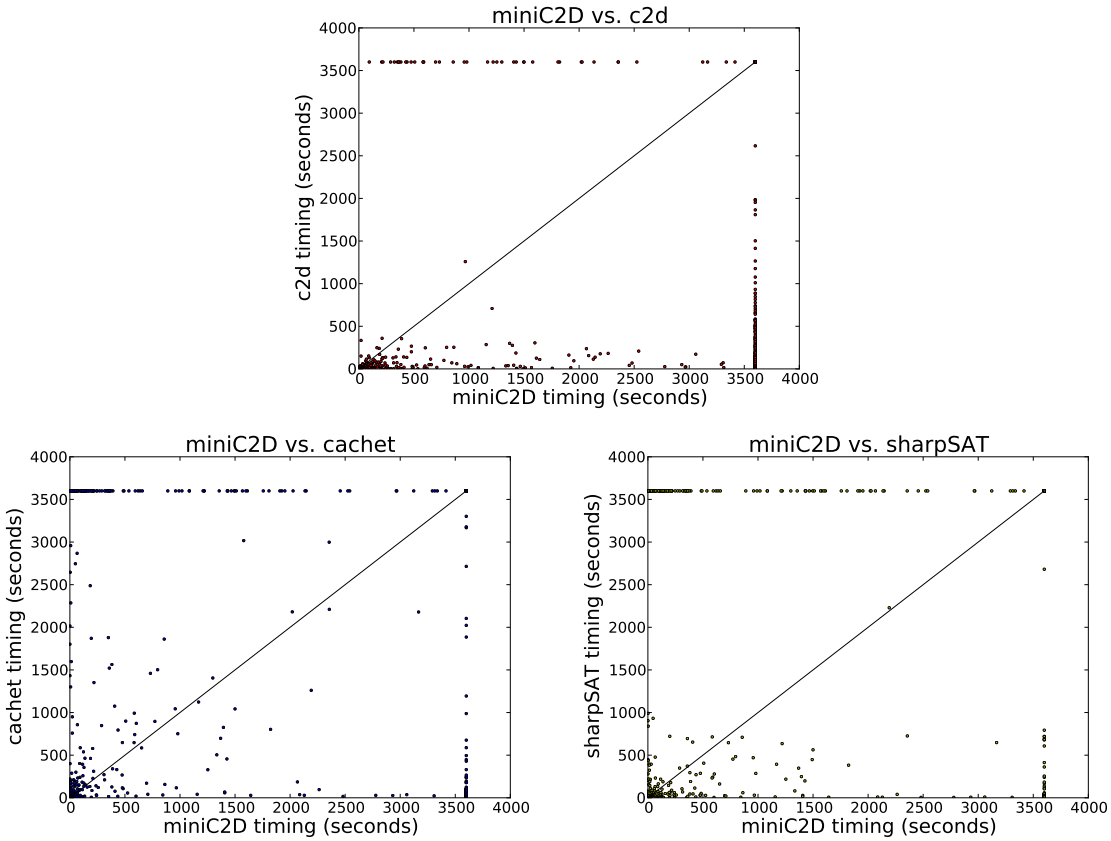


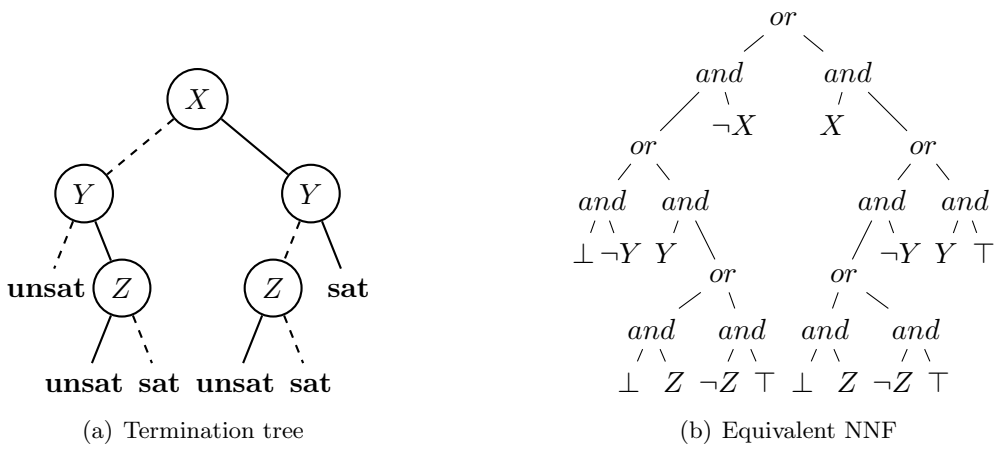
Figure 3: Comparison of miniC2D against state-of-the-art model counters.

three different systems: cachet[11], c2d[12], sharpSAT[13]. All experiments were performed on a 2.6GHz Intel Xeon X5650 CPU under 1 hour of time limit and with access to 8GB RAM. Figure 3 presents the results using scattered plots. For each system mentioned above, there is a plot that compares the corresponding system with miniC2D. In each plot, the points represent timings for counting the models of a specific CNF using miniC2D and another model counter. The points above the straight line denote instances where miniC2D performs better. Here are a few observations. The model counts of the instances were the same across all the systems. miniC2D was able to compute model counts of 1042 CNF instances, whereas c2d, cachet, and sharpSAT were able to compute model counts of 1178, 1027, and 1040 CNF instances, respectively. Further, Table 2 shows a comparison of the mentioned systems, by showing the number of CNF instances solved by one system but not by another one. Finally, the average counting times were as follows: miniC2D took 164.27 seconds, c2d took 62.69 seconds, cachet took 116.49 seconds, and sharpSAT took 39.45 seconds. Finally, there were 12 instances solved by only miniC2D.

11. Available at `http://www.cs.rochester.edu/users/faculty/kautz/Cachet`.
12. Available at `http://reasoning.cs.ucla.edu/c2d`.
13. Available at `https://sites.google.com/site/marcthurley/sharpsat`.

	miniC2D	c2d	cachet	sharpSAT
miniC2D	X	42	147	133
c2d	178	X	201	181
cachet	132	50	X	12
sharpSAT	131	43	25	X

Table 2: Comparison among model counters. An entry (x, y) shows the number of instances solved by the system x but not by the system y.



(a) Termination tree

(b) Equivalent NNF

Figure 4: The trace of an exhaustive DPLL.

Accordingly, despite being slower on average, our new model counter was able to solve more instances than cachet and sharpSAT, only surpassed by c2d. This is indeed a surprising result as the trace of our algorithm belongs to a strict subset of Decision-DNNFs that is not as succinct. We will next show that the trace of our algorithm belongs to a special class of Sentential Decision Diagrams.

4. Knowledge Compilation by Exhaustive DPLL

In this section, we will first review the close relationship between exhaustive DPLL (and its variants) and knowledge compilation (Huang & Darwiche, 2007). We will then show how our model counting algorithm can be used in the context of knowledge compilation. In particular, we will describe a top-down SDD compiler that is based off of our algorithm.

Exhaustive DPLL counts the models of a Boolean formula by searching the space of truth assignments until identifying all the satisfying ones. In particular, given a Boolean formula Δ, it chooses a variable X of Δ, and then considers two cases recursively, which correspond to $\Delta|X$ and $\Delta|\neg X$. It then obtains the model count of Δ by adding up the model counts of $\Delta|X$ and $\Delta|\neg X$. This procedure is similar to what we already described in Algorithm 3, which is additionally augmented with powerful techniques from SAT literature. In fact, exhaustive DPLL (and its variants) can be seen as constructing a tree. For example, the tree in Figure4(a) shows all the paths that are traversed during an exhaustive DPLL

on a Boolean formula. Each circled node represents a variable on which two decisions are performed: the variable is either set to false (dashed edge) or set to true (solid edge). This way, paths from the root to leaf nodes represent (partial) variable assignments. Each leaf node then represents the result of the search when the variable assignment on the path from the root to the leaf is applied on the Boolean formula, with the label **unsat** being unsatisfiable and the label **sat** being satisfiable. This tree is called the *trace* of the search performed by an exhaustive DPLL (Huang & Darwiche, 2007). Here one can think of each circled node of the tree as an *or* node, by utilizing the following:

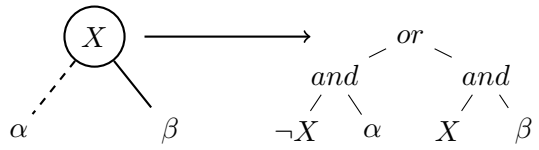


Figure 4(b) shows the tree obtained from Figure 4(a) using the above conversion, and also replacing each **sat** with $\top$, and each **unsat** with $\bot$. From this structure, by replacing the same nodes with unique nodes, one would obtain a (rooted) DAG. In its most general form, this DAG is known as *Negation Normal Form* (NNF) (Darwiche & Marquis, 2002): a rooted DAG that has *and* nodes (representing conjunctions) and *or* nodes (representing disjunctions) as internal nodes, and literals or constants as leaves. Indeed, as Huang and Darwiche (2007) showed, the traces of exhaustive DPLL correspond to FBDDs, and they become Decision-DNNFs when exhaustive DPLL is augmented with component analysis.[14]

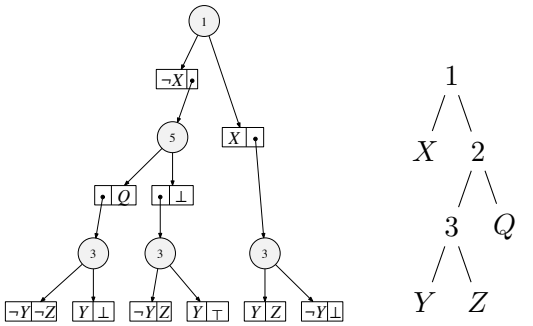
This close connection between exhaustive DPLL and knowledge compilation has two implications. First, it allows one to translate lower bounds on NNFs immediately into lower bounds on the complexity of model counters. Second, it allows one to use model counters as knowledge compilers, by simply bookkeeping the trace of the search. We will next show how we can use our model counter in the context of knowledge compilation – leading to orders of magnitude faster compilations.

4.1 Trace of Algorithm 4

Since Algorithm 4 is an exhaustive DPLL augmented with component analysis, we already know that its trace is a Decision-DNNF. However, because we employ decision vtrees in the algorithm, its trace is structured. As we will show later, the trace will be a special type of Sentential Decision Diagram (SDD) (Darwiche, 2011), called *Decision-SDD* (Oztok & Darwiche, 2015), which is introduced next.[15]

To define Decision-SDDs, we first need to distinguish between decision nodes. An SDD node that is normalized for a Shannon (resp., decomposition) vtree node is called

14. An NNF node is called a *decision* node if it is $\top$, $\bot$, or an *or* node having the form $(X \wedge \alpha) \vee (\neg X \wedge \beta)$ where X is a variable, and α and β are NNF nodes. A *Free Binary Decision Diagram* (FBDD) is an NNF in which every node is a *decision* node (Blum et al., 1980). A *Decision-DNNF* is an NNF which consists of decision nodes and *and* nodes having the form $\alpha \wedge \beta$ where the variables of α and β are disjoint (Huang & Darwiche, 2007) – making FBDDs a strict subset of Decision-DNNFs.
15. The definition of SDD and related concepts are presented in Appendix A. Here, we directly introduce Decision-SDDs with some insights on them. Some other interesting properties of Decision-SDDs are discussed in Appendix B.



(a) A Decision-SDD (b) A vtree

Figure 5: A Decision-SDD and its corresponding vtree.

a *Shannon (resp., decomposition) decision node.* A Shannon decision node has the form $\{(X, \alpha), (\neg X, \beta)\}$, where X is a Shannon variable.

Definition 5 (Decision-SDD)**.** *A Decision-SDD is an SDD in which each decomposition decision node has the form* $\{(p, s_1), (\neg p, s_2)\}$ *where* $s_1 = \top$*,* $s_1 = \bot$*, or* $s_1 = \neg s_2$*.*

Figure 5 shows a Decision-SDD and a corresponding vtree for the CNF $\{Y \vee \neg Z, \neg X \vee Z, X \vee \neg Y, X \vee Q\}$. For further insights into Decision-SDDs, note that a decomposition decision node must have the form $\{(f, g), (\neg f, \bot)\}$, $\{(f, \top), (\neg f, g)\}$, or $\{(f, \neg g), (\neg f, g)\}$. Moreover, these forms represent the Boolean functions $f \wedge g$, $f \vee g$, and $f \oplus g$, respectively, where f and g are over disjoint sets of variables. We also note that Decision-DNNFs differ from Decision-SDDs in two ways. First, Decision-DNNFs have no structure (that is, they are not guided by a vtree). Second, other than decision nodes, Decision-DNNFs only support nodes of the form $f \wedge g$ where f and g are over disjoint sets of variables, whereas Decision-SDDs also support $f \vee g$ and $f \oplus g$. Moreover, Decision-SDDs cannot polynomially simulate Decision-DNNFs. That is, there exists a Boolean function which has a polynomial size Decision-DNNF, yet each Decision-SDD representation of it has exponential size.[16]

We will next show that the trace of Algorithm 4 belongs to the language of Decision-SDDs. For that, we will present an algorithm that simulates Algorithm 4 by doing bookkeeping of its trace. This is given in Algorithm 5, whose overall structure is similar to Algorithm 4, except that it constructs an NNF instead of counting the models.[17]

Theorem 5. *If v is the root vtree node, then call* `c2s`$(v, (\Delta, \{\}, \langle\rangle, \{\}))$ *returns a Decision-SDD equivalent to Δ if Δ is satisfiable, and returns an empty clause if Δ is unsatisfiable.*

4.2 Top-Down SDD Compiler

SDDs have been used in different applications, such as probabilistic planning (Herrmann & de Barros, 2013), probabilistic logic programs (Vlasselaer et al., 2015), probabilistic

16. The mentioned function was used to show that SDDs cannot polynomially simulate Decision-DNNFs (Beame & Liew, 2015). As each Decision-SDD is an SDD, the same result applies between Decision-SDDs and Decision-DNNFs.
17. To construct an SDD, Algorithm 5 needs to have certain negations are freely available (e.g., $\neg p$ on Line 14). Hence, its recursive calls return both an SDD and its negation, making all such negations freely available.

Algorithm 5: $\texttt{c2s}(v, S)$

unique(α) removes an element from α if its prime is $\bot$. It then returns s if $\alpha = \{(p_1, s), (p_2, s)\}$ or $\alpha = \{(\top, s)\}$; returns p_1 if $\alpha = \{(p_1, \top), (p_2, \bot)\}$; else returns the unique SDD node with elements α.

Input: v : a vtree node, S : a SAT state (Δ, Γ, D, I)
Output: A Decision-SDD and its negation or a clause

```
1  if v is a leaf node then
2      X ← variable of v
3      if X or ¬X belongs to I then return the literal of X that belongs to I
4      else return ⊤, ⊥
5  else if v is a decomposition vtree node then
6      p, ¬p ← c2s(v^l, S)
7      if p is a learned clause then
8          clean_cache(v^l)
9          return p, p
10     s, ¬s ← c2s(v^r, S)
11     if s is a learned clause then
12         clean_cache(v)
13         return s, s
14     α  ← unique({(p, s), (¬p, ⊥)})
15     ¬α ← unique({(p, ¬s), (¬p, ⊤)})
16     return α, ¬α
17 else
18     key ← Key(v, S)
19     if cache(key) ≠ nil then return cache(key)
20     X ← Shannon variable of v
21     if either X or ¬X belongs to I then
22         p, ¬p ← the literal of X that belongs to I and its negation
23         s, ¬s ← c2s(v^r, S)
24         if s is a learned clause then return s, s
25         α  ← unique({(p, s), (¬p, ⊥)})
26         ¬α ← unique({(p, ¬s), (¬p, ⊤)})
27         return α, ¬α
28     s1 ← decide_literal(X, S)
29     if s1 is success then s1, ¬s1 ← c2s(v^r, S)
30     undo_decide_literal(X, S)
31     if s1 is a learned clause then
32         if at_assertion_level(s1, S) then
33             s1 ← assert_clause(s1, S)
34             if s1 is success then return c2s(v, S)
35             else return s1, s1
36         else return s1, s1
37     s2 ← decide_literal(¬X, S)
38     if s2 is success then s2, ¬s2 ← c2s(v^r, S)
39     undo_decide_literal(¬X, S)
40     if s2 is a learned clause then
41         if at_assertion_level(s2, S) then
42             s2 ← assert_clause(s2, S)
43             if s2 is success then return c2s(v, S)
44             else return s2, s2
45         else return s2, s2
46     α  ← unique({(X, s1), (¬X, s2)})
47     ¬α ← unique({(X, ¬s1), (¬X, ¬s2)})
48     cache(key) ← α, ¬α
49     return α, ¬α
```

inference (Choi et al., 2013), verification of multi-agent systems (Lomuscio & Paquet, 2015), and tractable learning (Kisa et al., 2014; Choi et al., 2015).

Almost all of these applications are based on the *bottom-up* SDD compiler developed by Choi and Darwiche (2013a), which was also used to compile CNFs into SDDs (Choi & Darwiche, 2013b). This compiler constructs SDDs by first compiling small pieces of a knowledge base (KB) (e.g., clauses of a CNF). It then combines these compilations using the `Apply`[18] operation to build a compilation for the full KB.

An alternative to bottom-up compilation is *top-down* compilation. This approach starts the compilation process with a full KB. It then recursively compiles the fragments of the KB that are obtained through conditioning. The resulting compilations are then combined to obtain the compilation of the full KB. All existing top-down compilers assume CNFs as input, while bottom-up compilers can work on any input due to the `Apply` operation. Yet, compared to bottom-up compilation, top-down compilation has been previously shown to yield significant improvements in compilation time and space when compiling CNFs into OBDDs (Huang & Darwiche, 2004). Thus, it has a potential to further improve the results on CNF to SDD compilations. Indeed, Algorithm 5 is a top-down algorithm that constructs Decision-SDDs. So, we will next show an empirical analysis of Algorithm 5 to see to what extent it would be helpful in the compilation of SDDs.

In our experiments, we used two sets of benchmarks. First, we used some CNFs from the `iscas85`, `iscas89`, and `LGSynth89` suites, which consist of sequential and combinatorial circuits used in the CAD community. We also used some CNFs available at `http://www.cril.univ-artois.fr/PMC/pmc.html`, which correspond to different applications such as planning and product configuration. We compiled those CNFs into SDDs and Decision-SDDs. To compile SDDs, we used the SDD package (Choi & Darwiche, 2013a).[19] All experiments were performed on a 2.6GHz Intel Xeon E5-2670 CPU under 1 hour of time limit and with access to 50GB RAM. We next explain our results shown in Table 3.

The first experiment compares the top-down compiler against the bottom-up SDD compiler. Here, we first generate a decision vtree[20] for the input CNF, and then compile the CNF into an SDD using (1) the bottom-up compiler without dynamic minimization (denoted BU), (2) the bottom-up compiler with dynamic minimization (denoted BU+), and (3) the top-down compiler (denoted TD), using the same vtree.[21] Note that BU+ uses a minimization method, which dynamically searches for better vtrees during the compilation process, leading to general SDDs, whereas both BU and TD do not modify the input decision vtree, hence generating Decision-SDDs with the same sizes. We report the corresponding compilation times and sizes in Columns 2–4 and 6–7, respectively. The top-down Decision-SDD compiler was consistently faster than the bottom-up SDD compiler, regardless of the use of dynamic minimization. In fact, in Column 5 we report the speed-ups obtained by

18. The `Apply` operation combines two SDDs using any Boolean operator, and has its origins in the OBDD literature (Bryant, 1986).
19. The SDD package was already shown to perform significantly better than a state-of-the-art OBDD compiler (Choi & Darwiche, 2013b). Moreover, the comparison of c2d and miniC2D in Section 3.5 can also be seen as comparing miniC2D against a state-of-the-art Decision-DNNF compiler, since c2d compiles into this form before it counts models.
20. We obtained decision vtrees as described by Oztok and Darwiche (2014).
21. Choi and Darwiche (2013b) used balanced vtrees constructed from the natural variable order, and manual minimization. We chose to use decision vtrees as they performed better than balanced vtrees.

	Without post-processing							With post-processing	
	Compilation time				SDD size			Compilation time	SDD size
CNF	BU	TD	BU+	Speed-up	TD	BU+	Ratio	TD+	TD+
c1355	3423.95	189.0	1292.87	6.84	71,642,606	2,430,882	**0.03**	—	—
c432	1.59	0.14	5.62	**11.36**	66,004	13,660	0.21	1.95	14,388
c499	1360.05	31.48	—	**43.20**	29,791,654	—	—	1800.14	3,356,190
c880	3372.87	896.47	—	3.76	214,504,174	—	—	—	—
s1196	763.39	1.86	709.93	**381.68**	2,381,672	245,549	**0.10**	131.53	97,641
s1238	1039.39	2.19	2114.01	**474.61**	1,539,440	139,475	**0.09**	74.42	76,690
s1423	1860.56	5.67	354.62	**62.54**	11,363,370	454,711	**0.04**	588.23	782,464
s1488	564.25	0.57	206.41	**362.12**	457,420	111,671	0.24	19.47	88,671
s1494	2672.46	0.59	1035.91	**1755.78**	465,092	98,812	0.21	21.33	91,690
s510	49.02	0.09	55.38	**544.67**	19,732	10,192	0.52	0.68	7,411
s641	3.84	0.28	4.54	**13.71**	257,322	13,910	**0.05**	5.36	14,623
s713	4.08	0.36	5.91	**11.33**	230,886	13,809	**0.06**	5.22	12,079
s832	80.94	0.33	28.45	**86.21**	501,098	30,841	**0.06**	11.23	28,773
s838	0.71	0.1	4.82	7.10	46,490	9,853	0.21	1.79	13,540
s953	—	1.92	—	—	2,772,894	—	—	90.06	161,056
9symml	6.15	0.08	5.29	**66.12**	59,616	15,572	0.26	1.57	14,453
alu2	1164.19	0.13	91.12	**700.92**	114,194	26,866	0.24	2.88	13,093
alu4	—	0.71	—	—	2,147,052	—	—	172.81	87,562
apex6	—	235.06	—	—	156,430,304	—	—	—	—
frg1	165.61	0.46	22.64	**49.22**	1,551,328	76,632	**0.05**	183.92	123,890
frg2	1876.64	49.76	690.63	**13.88**	21,820,292	235,761	**0.01**	2613.82	1,624,002
term1	517.52	25.36	454.08	**17.91**	5,545,908	249,372	**0.04**	468.92	818,343
ttt2	20.79	0.69	6.54	9.48	468,884	15,328	**0.03**	10.00	18,706
vda	—	0.14	—	—	126,152	—	—	11.21	29,266
x4	21.22	0.36	12.04	**33.44**	252,530	23,920	**0.09**	9.16	27,102
2bitcomp_5	16.29	0.35	119.82	**46.54**	337,642	19,289	**0.06**	9.06	58,043
2bitmax_6	—	45.22	—	—	153,512,364	—	—	—	—
4blocksb	30.99	168.53	16.85	0.10	1,634	1,989	1.22	168.63	1,530
C163_FW	2457.58	10.55	—	**232.95**	3,909,336	—	—	153.49	84,773
C171_FR	140.77	0.7	92.17	**131.67**	743,212	53,484	**0.07**	69.96	72,415
C210_FVF	1265.00	9.01	—	**140.40**	7,052,986	—	—	426.93	165,582
C211_FS	7.80	0.17	3.93	**23.12**	111,004	8,590	**0.08**	3.00	9,243
C215_FC	—	16.45	—	—	11,625,728	—	—	1294.15	431,589
C230_FR	—	32.69	3320.03	**101.56**	38,975,404	571,611	**0.01**	2869.13	763,845
C638_FKA	497.18	5.21	50.35	9.66	1,106,488	17,930	**0.02**	61.95	25,669
ais10	—	2.6	1464.48	**563.26**	61,950	13,940	0.23	4.35	11,997
bw_large.a	62.77	0.01	17.81	**1781.00**	1,512	1,642	1.09	0.16	1,290
bw_large.b	3246.49	0.17	961.77	**5657.47**	5,552	4,309	0.78	0.63	3,698
cnt06.shuffled	2.03	0.04	27.74	**50.75**	3,004	2,874	0.96	0.10	2,994
huge	83.01	0.05	23.79	**475.80**	1,512	1,654	1.09	0.20	1,290
log-1	41.02	0.23	21.39	**93.00**	69,358	6,650	**0.10**	1.99	7,622
log-2	—	8.85	—	—	11,249,348	—	—	—	—
log-3	—	4.76	—	—	440,868	—	—	185.88	24,418
par16-1-c	224.79	1.22	116.10	**95.16**	1,220	1,204	0.99	1.23	1,214
par16-2-c	356.94	1.26	—	**283.29**	1,362	—	—	1.32	1,242
par16-2	1098.42	1.28	1048.58	**819.20**	3,938	3,938	1.00	1.36	3,922
par16-3	666.46	4.46	713.34	**149.43**	3,960	3,960	1.00	4.54	3,934
par16-5-c	516.75	0.87	—	**593.97**	1,330	—	—	0.93	1,226
par16-5	864.91	4.38	1722.34	**197.47**	3,960	4,000	1.01	4.46	3,934
prob004-log-a	—	181.13	—	—	212,553,140	—	—	—	—
qg1-07	—	0.36	—	—	4,576	—	—	0.79	2,485
qg2-07	—	0.39	—	—	8,072	—	—	1.38	3,992
qg3-08	—	0.15	—	—	18,310	—	—	2.69	6,674
qg6-09	—	0.12	—	—	6,458	—	—	1.63	4,592
qg7-09	—	0.1	—	—	6,712	—	—	1.31	4,004
ra	269.96	4.77	—	**56.60**	619,146	—	—	116.14	342,034
ssa7552-038	4.71	0.14	9.38	**33.64**	44,902	18,786	0.42	1.57	19,147
tire-2	6.98	0.18	5.58	**31.00**	75,472	4,013	**0.05**	1.27	4,487
tire-3	42.13	0.23	26.67	**115.96**	73,914	7,599	**0.10**	1.85	13,038
tire-4	593.53	0.28	98.75	**352.68**	164,996	17,129	**0.10**	5.07	8,395
uf250-026	—	1667.7	—	—	8,880	—	—	1667.91	1,013

Table 3: Bottom-up and top-down SDD compilations over `iscas85`, `iscas89`, `LGSynth89`, and some sampled benchmarks. BU refers to bottom-up compilation without dynamic minimization and BU+ with dynamic minimization. TD refers to top-down compilation, and TD+ with a single minimization step applied at the end.

using the top-down compiler against the best result of the bottom-up compiler (i.e., either BU or BU+, whichever was faster). There are 40 cases (out of 61) where we observe at least an order-of-magnitude improvement in time. Also, there are 15 cases where top-down compilation succeeded and both bottom-up compilations failed. However, the situation is different for the sizes, when the bottom-up SDD compiler employs dynamic minimization.

In almost all of those cases, BU+ constructed smaller representations. As reported in Column 8, which shows the relative sizes of SDDs generated by TD and BU+, there are 21 cases where BU+ produced an order-of-magnitude smaller SDDs. This is not a surprising result though, given that BU+ produces general SDDs and our top-down compiler produces Decision-SDDs, and that SDDs are a strict superset of Decision-SDDs.

Since Decision-SDDs are a subset of SDDs, any minimization algorithm designed for SDDs can also be applied to Decision-SDDs. In this case, however, the results may not be necessarily Decision-SDDs, but general SDDs. In our second experiment, we applied the minimization method provided by the bottom-up SDD compiler to the compiled Decision-SDDs (as a post-processing step). We then added the top-down compilation times to the post-processing minimization times and reported those in Column 9, with the resulting SDD sizes in Column 10. As is clear, the post-processing minimization step significantly reduces the sizes of SDDs generated by our top-down compiler. In fact, the sizes are almost equal to the sizes generated by BU+ (Column 7). The top-down compiler gets slower due to the cost of the post-processing minimization step, but its total time still dominates the bottom-up compiler. Indeed, it can still be an order-of-magnitude faster than the bottom-up compiler (18 cases). This shows that one can also use Decision-SDDs as a representation that facilitates the compilation of CNFs into general SDDs.

5. Related Work

The works by Sang et al. (2004), Darwiche (2004), Thurley (2006), and Muise et al. (2012) are closest to ours, yielding the systems cachet, c2d, sharpSAT, and DSharp.

There are two main differences between cachet and our model counter miniC2D. We use a static component analysis whereas cachet performs a dynamic one. Moreover, the trace of cachet is a Decision-DNNF, whereas ours is a Decision-SDD. Finally, miniC2D can be used as a knowledge compiler as we implemented a version of it that saves its trace.

The main differences between sharpSAT and miniC2D are the same ones as with cachet. DSharp is simply a knowledge compiler that is obtained by saving the trace of sharpSAT, which is a Decision-DNNF.

There are two main differences between c2d and our model counter miniC2D. To perform static component analysis we use decision vtrees, whereas c2d uses a different structure called a *dtree*. Even though one can obtain vtrees from dtrees (Oztok & Darwiche, 2014), the use of vtrees provides a simpler algorithm. Furthermore, c2d is a CNF to Decision-DNNF compiler, whereas the trace of miniC2D belongs to Decision-SDDs.

The executive summary of the comparison is shown in Table 4. We note here that since Decision-SDDs are more structured than Decision-DNNFs, our model counter can be thought of as performing more work. Moreover, thanks to its modular design, our software package can easily be integrated with different SAT solvers. Finally, we remark that the works by Sang et al. (2004), Darwiche (2004), Thurley (2006), and Muise et al. (2012) do not provide a proof of correctness of their methods, as we did in this paper.

System	Counter	Compiler	Component Analysis	Trace	SAT solver
cachet	✓	X	Dynamic	Decision-DNNF	zChaff
c2d	✓	✓	Static	Decision-DNNF	RSat
sharpSAT	✓	X	Dynamic	Decision-DNNF	–
DSharp	✓	✓	Dynamic	Decision-DNNF	–
miniC2D	✓	✓	Static	Decision-SDD	RSat

Table 4: Comparison of related work.

6. Conclusion

In this article, we introduced a new exhaustive DPLL algorithm with a formal semantics, a proof of correctness, and a modular design. The modular design was based on the separation of the core model counting algorithm from SAT solving techniques, which allows one to easily integrate techniques from SAT solvers. We hope that this will facilitate the development of model counters. Moreover, we showed that the trace of our algorithm belongs to a special class of SDDs, which is a subset of Decision-DNNFs, the trace of existing state-of-the-art model counters. Still, our extensive experimental evaluation showed comparable results against state-of-the-art model counters. Finally, we obtained the first *top-down* SDD compiler by saving the traces of our model counting algorithm, and showed orders-of-magnitude improvements in compilation times against the state-of-the-art *bottom-up* SDD compiler. Our system, miniC2D, is publicly available at `http://reasoning.cs.ucla.edu/minic2d`. The distribution provides the source codes for Algorithm 4 and Algorithm 5.

Appendix A. Sentential Decision Diagrams

A Boolean function $f(\mathbf{X}, \mathbf{Y})$, where $\mathbf{X}$ and $\mathbf{Y}$ are disjoint, can always be decomposed into

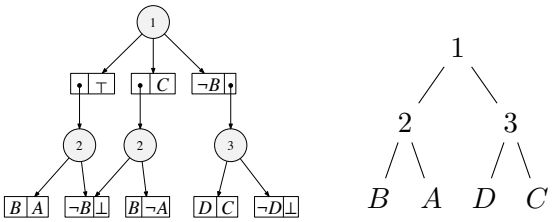
$$f(\mathbf{X}, \mathbf{Y}) = \big(p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})\big) \vee \ldots \vee \big(p_n(\mathbf{X}) \wedge s_n(\mathbf{Y})\big),$$

such that $p_i \neq \bot$ for all i; $p_i \wedge p_j = \bot$ for $i \neq j$; and $\bigvee_i p_i = \top$. A decomposition satisfying the above properties is known as an $(\mathbf{X}, \mathbf{Y})$-partition (Darwiche, 2011). Moreover, each p_i is called a *prime,* each s_i is called a *sub*, and the $(\mathbf{X}, \mathbf{Y})$-partition is said to be *compressed* when its subs are distinct, i.e., $s_i \neq s_j$ for $i \neq j$.

SDDs result from the recursive decomposition of a Boolean function using $(\mathbf{X}, \mathbf{Y})$-partitions. To determine the $\mathbf{X}/\mathbf{Y}$ variables of each partition, we use a vtree. Consider now the vtree in Figure 6(b), and also the Boolean function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$. Node $v = 1$ is the vtree root, whose left subtree contains variables $\mathbf{X} = \{A, B\}$ and right subtree contains $\mathbf{Y} = \{C, D\}$. Decomposing function f at node $v = 1$ amounts to generating an $(\mathbf{X}, \mathbf{Y})$-partition:

$$\{(\underbrace{A \wedge B}_{\text{prime}}, \underbrace{\top}_{\text{sub}}), (\underbrace{\neg A \wedge B}_{\text{prime}}, \underbrace{C}_{\text{sub}}), (\underbrace{\neg B}_{\text{prime}}, \underbrace{D \wedge C}_{\text{sub}})\}.$$

This partition is represented by the root node of Figure 6(a). This node, which is a circle, represents a *decision node* with three branches, where each branch is called an *element*. Each element is depicted by a paired box $\boxed{p \mid s}$. The left box corresponds to a prime p and the right box corresponds to its sub s. A prime p or sub s are either a constant, literal, or pointer to a decision node. In this case, the three primes are decomposed recursively, but



(a) An SDD (b) A vtree

Figure 6: An SDD and a vtree for $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.

using the vtree rooted at $v = 2$. Similarly, the subs are decomposed recursively, using the vtree rooted at $v = 3$. This decomposition process moves down one level in the vtree with each recursion, terminating at leaf vtree nodes.

SDDs constructed as above are said to *respect* the used vtree. These SDDs may contain trivial decision nodes which correspond to $(\mathbf{X}, \mathbf{Y})$-partitions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$. When these decision nodes are removed (by directing their parents to α), the resulting SDD is called *trimmed*. Moreover, an SDD is called *compressed* when each of its partitions is compressed. Compressed and trimmed SDDs are canonical for a given vtree (Darwiche, 2011). Here, we restrict our attention to compressed and trimmed SDDs. Figure 6(a) depicts a compressed and trimmed SDD for the above example. Finally, an SDD node representing an $(\mathbf{X}, \mathbf{Y})$-partition is *normalized* for the vtree node v with variables $\mathbf{X}$ in its left subtree v^l and variables $\mathbf{Y}$ in its right subtree v^r. In Figure 6(a), SDD nodes are labeled with vtree nodes they are normalized for.

Appendix B. Decision-SDDs

We will now provide some interesting properties of Decision-SDDs, and discuss their relationship to OBDDs and SDDs. We first note that Decision-SDDs are a strict subset of SDDs. In particular, if an SDD is based on a general vtree, it may or may not be a Decision-SDD. However, a decision vtree guarantees a Decision-SDD.

Proposition 2. *Let v be a decision vtree for CNF Δ. An SDD equivalent to Δ that respects vtree v must be a Decision-SDD.*

Proof. Due to Theorem 5, Algorithm 5 constructs a Decision-SDD equivalent to Δ that respects vtree v. As SDDs are canonical for a given vtree, we are done. □

A vtree is called *right-linear* when every internal vtree node is a Shannon vtree node. Consider an SDD that respects a right-linear vtree. In this case, the SDD corresponds to an OBDD. Further, note that each OBDD is a Decision-SDD. Hence, Decision-SDDs are a superset of OBDDs. Because of this, the language of Decision-SDDs is complete: every Boolean function can be represented by a Decision-SDD using an appropriate vtree.

In terms of succinctness, a quasipolynomial separation between SDDs and OBDDs was given by Razgon (2014).[22] As it turns out, the SDDs used in this separation were indeed

22. A quasipolynomial grows slightly faster than a polynomial, but not exponentially fast.

Decision-SDDs. Further, the following complements this result by showing that Decision-SDDs can be simulated by OBDDs with at most a quasipolynomial increase in size.

Theorem 6 (Oztok & Darwiche, 2015)**.** *Each Decision-SDD over n variables and of size N has an equivalent OBDD of size $\leq N^{1+\log n}$.*

Furthermore, Xue et al. (2012) have identified a class of Boolean functions f_i, with corresponding variable orders π_i, such that the OBDDs based on orders π_i have exponential size, yet the SDDs based on vtrees that dissect orders π_i have linear size.[23] Interestingly, the SDDs used in this result turn out to be Decision-SDDs as well. Hence, a variable order that blows up an OBDD can sometimes be dissected to obtain a vtree that leads to a compact Decision-SDD. This reveals the practical significance of Decision-SDDs despite the quasipolynomial simulation of Theorem 6. We finally note that SDDs have been recently shown to be exponentially separated from OBDDs (Bova, 2016). Due to this result and Theorem 6, SDDs are also exponentially separated from Decision-SDDs.

Appendix C. Soundness of the Model Counting Algorithm

We will now present the proofs of the theorems that were used in the soundness of Algorithm 4 (i.e., Theorems 1–3). We start by listing some assumptions/observations that will be used in the rest of the paper. First, S will denote a state (Δ, Γ, D, I), where Δ and Γ are sets of clauses, D is a sequence of literals, and I is a set of literals.[24] Second, each (recursive) call $\#\texttt{SAT}(v, S)$ of Algorithm 4 will take two inputs v and S such that v is a vtree node belonging to a decision vtree for Δ and S is a SAT state.[25] The latter implies that $\Delta \models \Gamma$ and $\Delta \wedge \Gamma \wedge D \vdash I$. This holds due to the following three facts: (1) initial call is made with $(\Delta, \{\}, \langle\rangle, \{\})$, which is a SAT state as Δ has no unit or empty clause is no unit or empty clause in Δ; (2) any clause added to Γ is a learned clause, which must be implied by Δ; and (3) literals I is always adjusted before making a call (see SAT primitives in Figure 1). Finally, a SAT state S is said to be *callable* iff unit resolution does not detect a contradiction in $\Delta \wedge \Gamma \wedge D$. Indeed, S will be callable for each call $\#\texttt{SAT}(v, S)$. This is true due to the following two facts: (1) initial SAT state $(\Delta, \{\}, \langle\rangle, \{\})$ is callable as Δ has no unit or empty clause; and (2) whenever a new SAT state, which is not callable, is constructed during a call, Algorithm 4 backtracks until the contradiction is resolved. So, it will initiate a call only on callable SAT states. We now prove Theorem 1.

Theorem 1. *Consider a call $\#\texttt{SAT}(v, S)$ with $S = (., ., D, I)$. Then, $D \subseteq ContextL(v, I)$ and $ContextL(v, I)$ contains exactly one literal for each variable of $ContextV(v)$.*

Proof. Let $\gamma = ContextL(v, I)$. We first show $D \subseteq \gamma$. For that, we show $D \subseteq I$ and $Vars(D) \subseteq ContextV(v)$. The former is due to $\Delta \wedge \Gamma \wedge D \vdash I$. For the latter, note that when Algorithm 4 decides on a literal (i.e., Line 23 and Line 32), it undoes its decision after completing a recursive call on the next line (i.e., Line 25 and Line 34). Thus, when call $\#\texttt{SAT}(v, S)$ is made, literals of D must come from recursive calls that are not completed. Indeed, these calls can only be made on the ancestors of v. So, each literal of D must be

23. A vtree dissects a variable order if the order is generated by a left-right traversal of the vtree.
24. We will sometimes abuse notation to use D as a set of literals.
25. All calls considered in the proofs are assumed to be legal (i.e., can be generated by executing Algorithm 4).

a literal of some context variable of v. That is, $Vars(D) \subseteq ContextV(v)$. We next show γ contains exactly one literal for each context variable of v. First, we show γ cannot contain two literals of any variable. Assume otherwise. Then, since $\gamma \subseteq I$ and $\Delta \wedge \Gamma \wedge D \vdash I$, unit resolution detects a contradiction in $\Delta \wedge \Gamma \wedge D$, which is a contradiction. We now show γ contains a literal for each context variable X of v. Assume otherwise. Let p be the Shannon node whose Shannon variable is X. Then, variable X is not implied during the corresponding ancestral call to p. As p is a Shannon node, Algorithm 4 will not recurse on p's right child before ensuring X is implied (see Lines 20, 24, and 33). That is, call $\#\mathtt{SAT}(v, S)$ cannot happen, which is a contradiction. So, a literal of X appears in I. Hence, I contains a literal for each context variable of v, and so does γ. □

We remark that Algorithm 4 is recursive. As such, its execution can be viewed as constructing a tree whose nodes are labeled with recursive calls $\#\mathtt{SAT}(.,.)$, and whose edges are from a recursive call R_1 to another R_2 if R_2 is called within R_1. As some of upcoming proofs will be based on this tree, which we denote by T, we next present two useful observations.

Proposition 3. *Consider an internal node* $\boxed{\#\mathtt{SAT}(v, S)}$ *on* T*. Then,* v *is either a decomposition node or a Shannon node.*

Proof. As $\boxed{\#\mathtt{SAT}(v, S)}$ is an internal node, it must have a child. Then, v cannot be a leaf vtree node, as no recursive call is made on Lines 1–4. So, the proposition follows. □

Proposition 4. *Consider a leaf node* $\boxed{\#\mathtt{SAT}(v, S)}$ *on* T*. Then, either* v *is a leaf vtree node or* v *is a Shannon node and call* $\#\mathtt{SAT}(v, S)$ *returns a clause on Line 30 or Line 31.*

Proof. Node $\boxed{\#\mathtt{SAT}(v, S)}$ can be a leaf node iff no recursive call happens during call $\#\mathtt{SAT}(v, S)$. Due to Line 6, v cannot be a decomposition node. So, v is either a leaf vtree node or a Shannon node. If v is a Shannon node, then the call can return on one of the following lines: 21, 22, 29, 30, 31, 38, 39, 40, and 43. It is not hard to see that no recursive call happens only when call $\#\mathtt{SAT}(v, S)$ returns a clause on Line 30 or Line 31. □

To prove Theorem 2 and Theorem 3, we will next present some lemmas.

Lemma 1. *Consider a call* $\#\mathtt{SAT}(v, S)$*. Let* $S' = (\Delta, \Gamma', D', I')$ *be a callable SAT state that appears during call* $\#\mathtt{SAT}(v, S)$*. Then, we have* $ContextL(v, I) = ContextL(v, I')$*.*

Proof. Note that $\Delta \wedge \Gamma \wedge D \vdash I$ and $\Delta \wedge \Gamma' \wedge D' \vdash I'$. We first show $\Gamma \subseteq \Gamma'$ and $D \subseteq D'$, which implies that $I \subseteq I'$. The former holds as Algorithm 4 never erases learned clauses and Γ is obtained before Γ'. The latter holds as call $\#\mathtt{SAT}(v, S)$ does not undo any decision made before its initiation. Hence, $I \subseteq I'$. Let $\gamma = ContextL(v, I)$. Since $\gamma \subseteq I$, $\gamma \subseteq I'$. Also, by Theorem 1, γ contains exactly one literal for each context variable of v. So, I' cannot contain any other literal than γ for context variables of v. Otherwise, unit resolution detects contradiction in $\Delta \wedge \Gamma' \wedge D'$, which violates S' being callable. So, $\gamma = ContextL(v, I')$. □

Corollary 2. *Consider a call* $\#\mathtt{SAT}(v, S)$*. Let* S' *be a callable SAT state that appears during call* $\#\mathtt{SAT}(v, S)$*. Then, we have* $CNF(v, S) = CNF(v, S')$*.*

Lemma 2. *Let* S *be a SAT state and* v *be a decomposition node of a decision vtree for* Δ*. Then, we have* $CNF(v, S) = CNF(v^l, S) \wedge CNF(v^r, S)$*.*

Proof. Since the input vtree is a decision vtree, there is no clause compatible with v. Thus, we have $CNF(v) = CNF(v^l) \wedge CNF(v^r)$, $ContextC(v) = ContextC(v^l) \wedge ContextC(v^r)$ and $ContextV(v^l) = ContextV(v^r)$, implying $CNF(v, S) = CNF(v^l, S) \wedge CNF(v^r, S)$. □

Lemma 3. *Consider a call* $\#\texttt{SAT}(v, S)$ *on a decomposition node* v*. Let* S' *be a callable state that appears during call* $\#\texttt{SAT}(v, S)$*. Then,* $CNF(v', S) = CNF(v', S')$ *for* v*'s child* v'*.*

Proof. $CNF(v, S) = CNF(v, S')$ by Corollary 2. Further, due to Lemma 2, $CNF(v, S) = CNF(v^l, S) \wedge CNF(v^r, S)$ and $CNF(v, S') = CNF(v^l, S') \wedge CNF(v^r, S')$. This implies $CNF(v^l, S) = CNF(v^l, S')$ and $CNF(v^r, S) = CNF(v^r, S')$, and thus the lemma holds. □

Lemma 4. *Consider a call* $\#\texttt{SAT}(v, S)$*. Let* $v_1, \ldots, v_n$ *be the decomposition nodes on the path from the vtree root to* v *(excluding* v*) and* v'_i *the child of* v_i *not appearing on the path. Then,* $\Delta|\gamma \equiv \bigwedge_{i=1}^{n} CNF(v'_i, S) \wedge CNF(v, S)$*, where* $\gamma = ContextL(v, I)$*.*

Proof. Note that $\Delta \equiv \big(\bigwedge_{i=1}^{n} CNF(v'_i) \wedge ContextC(v'_i)\big) \wedge CNF(v) \wedge ContextC(v) \wedge \Sigma$, where Σ is a set of clauses that only mention context variables of v. So, the following holds:

$$
\begin{aligned}
\Delta|\gamma &\equiv \Big(\bigwedge_{i=1}^{n} CNF(v'_i)|\gamma \wedge ContextC(v'_i)|\gamma\Big) \wedge CNF(v)|\gamma \wedge ContextC(v)|\gamma \wedge \Sigma|\gamma \\
&\equiv \Big(\bigwedge_{i=1}^{n} CNF(v'_i)|\gamma \wedge ContextC(v'_i)|\gamma\Big) \wedge CNF(v)|\gamma \wedge ContextC(v)|\gamma \qquad (1) \\
&\equiv \Big(\bigwedge_{i=1}^{n} CNF(v'_i) \wedge ContextC(v'_i)|\gamma\Big) \wedge CNF(v) \wedge ContextC(v)|\gamma \qquad (2) \\
&\equiv \bigwedge_{i=1}^{n} CNF(v'_i, S) \wedge CNF(v, S). \qquad (3)
\end{aligned}
$$

We now explain why Equations (1)–(3) hold. Equation (1) holds as $\Sigma|\gamma \equiv \top$. To see this, note that γ contains exactly one literal for each context variable of v (see Theorem 1), so that $\Sigma|\gamma \equiv \bot$ or $\Sigma|\gamma \equiv \top$. If $\Sigma|\gamma \equiv \bot$, then unit resolution detects a contradiction in $\Delta \wedge \Gamma \wedge D$ (since $\Delta \wedge \Gamma \wedge D \vdash I$ and $\gamma \subseteq I$). So, $\Sigma|\gamma \equiv \top$. Equation (2) holds as $CNF(v'_i)$ and $CNF(v)$ does not mention any context variable of v. Equation (3) holds as $CNF(v'_i, S) \equiv CNF(v'_i) \wedge ContextC(v'_i)|\gamma$ (since $ContextV(v'_i) \subseteq ContextV(v)$). □

Lemma 5. *Consider a call* $\#\texttt{SAT}(v, S)$*. Then,* $\Delta \wedge D \equiv \Delta \wedge \gamma$ *where* $\gamma = ContextL(v, I)$*.*

Proof. Note that $\Delta \wedge \Gamma \wedge D \vdash I$. Then, since $\gamma \subseteq I$, $\Delta \wedge \Gamma \wedge D \models \gamma$. Further, since $\Delta \models \Gamma$, $\Delta \wedge \Gamma \wedge D \equiv \Delta \wedge D$. So, $\Delta \wedge D \models \gamma$. By Theorem 1, $D \subseteq \gamma$. So, $\Delta \wedge D \equiv \Delta \wedge \gamma$. □

Lemma 6. *Consider a call* $\#\texttt{SAT}(v, S)$*. Then,* S *is satisfiable iff* $\Delta|\gamma$ *is satisfiable where* $\gamma = ContextL(v, I)$*.*

Proof. By Lemma 5, $\Delta \wedge D \equiv \Delta \wedge \gamma$. Also, as γ is a set of literals, $\Delta \wedge \gamma \equiv \Delta|\gamma \wedge \gamma$, and so $\Delta \wedge D \equiv \Delta|\gamma \wedge \gamma$. Thus, $\Delta \wedge D$ is satisfiable iff $\Delta|\gamma$ is satisfiable. So, the lemma holds. □

Lemma 7. *Consider a call* $\#\texttt{SAT}(v, S)$ *on a Shannon node* v *with Shannon variable* X*. Let* $S' = (\Delta, \Gamma', D', I')$ *be a callable SAT state that appears during call* $\#\texttt{SAT}(v, S)$*. If a literal* ℓ *of* X *appears in* I'*, then* $CNF(v^r, S') \equiv CNF(v, S)|\ell$*.*

Proof. Assume a literal ℓ of X appears in I'. We note that $CNF(v) \wedge ContextC(v) \equiv CNF(v^l) \wedge CNF(v^r) \wedge ContextC(v^r) \wedge \Sigma$, where $\Sigma = ContextC(v^l) \setminus ContextC(v^r)$. Also, $CNF(v^l) \equiv \top$ as v is a Shannon node and there is no unit or empty clause. Then, the following holds, where $\gamma = ContextL(v, I)$:

$$\begin{aligned} CNF(v,S)|\ell \equiv \big(CNF(v) \wedge ContextC(v)\big)|\gamma\ell &\equiv CNF(v^r) \wedge \big(ContextC(v^r) \wedge \Sigma\big)|\gamma\ell \\ &\equiv CNF(v^r) \wedge ContextC(v^r)|\gamma\ell \quad (4) \\ &\equiv CNF(v^r, S'). \quad (5) \end{aligned}$$

We now explain why Equations (4)–(5) hold. Note that $ContextV(v^r) = ContextV(v) \cup \{X\}$ and $\gamma = ContextL(v, I')$ (see Lemma 1). Then, $ContextL(v^r, I') = \gamma\ell$, and so Equation (5) holds. Also, Equation (4) holds since $\Sigma|\gamma\ell \equiv \top$. To see this, note that Σ is defined over context variables of v^r and $\gamma\ell$ contains exactly one literal for each context variable of v^r, which implies that $\Sigma|\gamma\ell \equiv \bot$ or $\Sigma|\gamma\ell \equiv \top$. If $\Sigma|\gamma\ell \equiv \bot$, then unit resolution must detect a contradiction in $\Delta \wedge \Gamma' \wedge D'$ (since $\Delta \wedge \Gamma' \wedge D' \vdash I'$ and $\gamma\ell \subseteq I'$). However, this contradicts with S' being callable. Hence, $\Sigma|\gamma\ell \equiv \top$. □

Lemma 8. *Consider a call $\#\texttt{SAT}(v, S)$ with a satisfiable state S. If a literal ℓ of some variable inside v appears in I, then $CNF(v, S) \equiv \ell \wedge CNF(v, S)|\ell$.*

Proof. Assume a literal ℓ of some variable X inside v appears in I. Note that $\Delta \wedge \Gamma \wedge D \vdash I$. So, $\Delta \wedge \Gamma \wedge D \models \ell$. Further, since $\Delta \models \Gamma$, $\Delta \wedge \Gamma \wedge D \equiv \Delta \wedge D$. So, $\Delta \wedge D \models \ell$. Then, due to Lemma 5, $\Delta \wedge \gamma \models \ell$ where $\gamma = ContextL(v, I)$. Since γ cannot contain ℓ, $\Delta|\gamma \models \ell$. By Lemma 4, $\Delta|\gamma \equiv \Sigma \wedge CNF(v, S)$ where Σ and $CNF(v, S)$ are decomposable CNFs. Here, $CNF(v, S)$ mentions X but Σ does not. Then, given that $\Delta|\gamma$ is satisfiable (see Lemma 6), we have $CNF(v, S) \models \ell$, which implies that $CNF(v, S) \equiv \ell \wedge CNF(v, S)|\ell$. □

Lemma 9. *Consider a call $\#\texttt{SAT}(v, S)$ with a satisfiable state S. Let ℓ be a literal of some variable inside v. If $\Delta \wedge D \wedge \ell$ is unsatisfiable, then $CNF(v, S)|\ell$ is unsatisfiable.*

Proof. Assume $\Delta \wedge D \wedge \ell$ is unsatisfiable. Since S is satisfiable, $\Delta \wedge D$ is satisfiable. So, $(\Delta \wedge D)|\ell$ must be unsatisfiable. Then, due to Lemma 5, $(\Delta \wedge \gamma)|\ell$ is unsatisfiable where $\gamma = ContextL(v, I)$. Since γ cannot contain ℓ, $\Delta|\gamma\ell$ is unsatisfiable. Due to Lemma 4, $\Delta|\gamma \equiv \Sigma \wedge CNF(v, S)$ where Σ and $CNF(v, S)$ are decomposable. As $\Delta|\gamma$ is satisfiable (see Lemma 6), Σ is satisfiable. Since Σ does not mention any variable inside v, $\Sigma|\ell \equiv \Sigma$. So, $\Delta|\gamma\ell \equiv \Sigma \wedge CNF(v, S)|\ell$, and hence $CNF(v, S)|\ell$ is unsatisfiable. □

Lemma 10. *Consider a call $\#\texttt{SAT}(v, S)$ on a leaf node v labeled with variable X. Then, $CNF(v, S)$ is equivalent to one of the following: $X, \neg X$, or $\top$.*

Proof. Note that $CNF(v, S) = CNF(v) \wedge ContextC(v)|\gamma$ where $\gamma = ContextL(v, I)$. So, by Theorem 1, $CNF(v, S)$ must be equivalent to one of $X, \neg X, \top$, or $\bot$. We show it cannot be equivalent to $\bot$. Assume otherwise. Note that Δ has neither an empty clause nor a unit clause. So, $CNF(v, S) \equiv ContextC(v)|\gamma$. Thus, $ContextC(v)$ must include two clauses β_1 and β_2 such that $\beta_1|\gamma = X$ and $\beta_2|\gamma = \neg X$. Note that $\Delta \wedge \Gamma \wedge D \vdash I$. Then, since $\gamma \subseteq I$, $\beta_1|\gamma = X$, and $\beta_2|\gamma = \neg X$, unit resolution must detect a contradiction in $\Delta \wedge \Gamma \wedge D$. As this is a contradiction, $CNF(v, S)$ cannot be equivalent to $\bot$. □

Lemma 11. *Consider a call* $\#\texttt{SAT}(v, S)$*. If* $CNF(v, S)$ *is unsatisfiable, then call* $\#\texttt{SAT}(v, S)$ *will return a (learned) clause.*

Proof. Assume $CNF(v, S)$ is unsatisfiable. We use strong induction on the height of node $\boxed{\#\texttt{SAT}(v, S)}$ on T to show that call $\#\texttt{SAT}(v, S)$ returns a clause.

Basis: Consider a leaf node $\boxed{\#\texttt{SAT}(v, S)}$ (i.e., at height 0). Due to Lemma 10, v cannot be a leaf vtree node. Then, by Proposition 4, call $\#\texttt{SAT}(v, S)$ must return a clause.

Inductive step: As an induction hypothesis (IH), assume that the statement holds for the calls at height less than k where $k \geq 1$. Consider an internal node $\boxed{\#\texttt{SAT}(v, S)}$ (i.e., at height k). By Proposition 3, v is either a decomposition node or a Shannon node.

Assume v is a decomposition node. Then, $CNF(v, S) = CNF(v^l, S) \wedge CNF(v^r, S)$ due to Lemma 2. Since $CNF(v^l, S)$ and $CNF(v^r, S)$ are decomposable, one of them must be unsatisfiable. Let S^l (resp., S^r) be the SAT state before the call on Line 6 (resp., Line 10). Due to Lemma 3, $CNF(v^l, S^l) = CNF(v^l, S)$ and $CNF(v^r, S^r) = CNF(v^r, S)$. Then, by IH, either Line 6 or Line 10 must construct a clause (whichever component CNF is unsatisfiable), and hence call $\#\texttt{SAT}(v, S)$ returns a clause on either Line 9 or Line 13.

Assume v is a Shannon node with Shannon variable X. Since $CNF(v, S)$ is unsatisfiable, $CNF(v, S)|\ell$ is unsatisfiable for any literal ℓ. Suppose a literal ℓ of X belongs to I. Then, the call on Line 20 is made with SAT state $S' = S$. By Lemma 7, $CNF(v^r, S') \equiv CNF(v, S)|\ell$, and so is unsatisfiable. Thus, by IH, Line 20 returns a clause, so does call $\#\texttt{SAT}(v, S)$. Suppose no literal ℓ of X belongs to I. We first show that the condition on Line 26 must be satisfied. This can happen iff Line 23 or Line 24 constructs a clause. Note that if Line 23 does not construct a clause, then the call on Line 24 is made with a SAT state $S' = (., ., ., I')$ where $X \in I'$. By Lemma 7, $CNF(v^r, S') \equiv CNF(v, S)|X$, and so is unsatisfiable. So, by IH, Line 24 returns a clause. That is, the condition on Line 26 must be satisfied. Hence, call $\#\texttt{SAT}(v, S)$ must return on either Line 29, 30 or 31. As Line 30 and Line 31 both return a clause, it remains to show Line 29 returns a clause. Let S' be the state before the call on Line 29. By Corollary 2, $CNF(v, S) = CNF(v, S')$. So, by IH, Line 29 returns a clause. □

Lemma 12. *A call* $\#\texttt{SAT}(v, S)$ *with a satisfiable state* S *will return either the model count of component* $CNF(v, S)$ *or a (learned) clause.*

Proof. We use strong induction on the height of node $\boxed{\#\texttt{SAT}(v, S)}$ on T to show that call $\#\texttt{SAT}(v, S)$ returns either the model count of $CNF(v, S)$ or a clause.

Basis: Consider a leaf node $\boxed{\#\texttt{SAT}(v, S)}$ (i.e., at height 0). By Proposition 4, either v is a leaf vtree node or the call returns a clause. Assume v is a leaf vtree node labeled with variable X. Then, by Lemma 10, $CNF(v, S)$ is equivalent to one of $X, \neg X$, or $\top$. So, we simply identify $CNF(v, S)$ on Lines 1–4, and return its model count.

Inductive step: As an induction hypothesis (IH), assume that the statement holds for the calls at height less than k where $k \geq 1$. Consider an internal node $\boxed{\#\texttt{SAT}(v, S)}$ (i.e., at height k). By Proposition 3, v is either a decomposition node or a Shannon node.

Assume v is a decomposition node. So, call $\#\texttt{SAT}(v, S)$ can return on one of the following lines: 9, 13, or 14. Line 9 and Line 13 return clauses. So, assume the call returns on Line 14. That is, Line 6 and Line 10 do not return clauses. Let S^l (resp., S^r) be the SAT state before the call on Line 6 (resp., Line 10). Since S is satisfiable, both S^l and S^r are satisfiable (note that decision sequence D stays the same). Then, by IH, Line 6 and Line 10 must return

the model counts of $CNF(v^l, S^l)$ and $CNF(v^r, S^r)$, respectively. Further, due to Lemma 3, $CNF(v^l, S^l) = CNF(v^l, S)$ and $CNF(v^r, S^r) = CNF(v^r, S)$. As the model count of $CNF(v, S)$ is equal to the product of the model counts of $CNF(v^l, S)$ and $CNF(v^r, S)$ (due to Lemma 2), Line 14 returns the model count of $CNF(v, S)$.

Assume v is a Shannon node with Shannon variable X. So, call $\#\texttt{SAT}(v, S)$ can return on one of the following lines: 21, 22, 29, 30, 31, 38, 39, 40 or 43. Lines 21, 30, 31, 39, 40 return clauses. So, we study the remaining lines in the following:

[**22**] Here, a literal ℓ of X belongs to I. Then, by Lemma 7, $CNF(v^r, S) \equiv CNF(v, S)|\ell$. So, by IH, Line 20 returns the model count of $CNF(v, S)|\ell$. Since $CNF(v, S) \equiv \ell \wedge CNF(v, S)|\ell$ (see Lemma 8), Line 22 returns the model count of $CNF(v, S)$.

[**29, 38**] Let $S' = (., ., D', .)$ be the SAT state before the call on Line 29. It is easy to see that $D' = D$. Then, S' is satisfiable (as S is satisfiable). Also, by Corollary 2, $CNF(v, S') = CNF(v, S)$. So, by IH, Line 29 returns either the model count of $CNF(v, S)$ or a clause. We can use the same argument for Line 38.

[**43**] To reach this line, calls on Line 24 and Line 33 should not construct clauses. Note that the call on Line 24 should be made with a SAT state $S' = (\Delta, ., DX, .)$. We now show S' is satisfiable. Assume otherwise. That is, $\Delta \wedge D \wedge X$ is unsatisfiable. Then, by Lemma 9, $CNF(v, S)|X$ is unsatisfiable. Note that $CNF(v^r, S') \equiv CNF(v, S)|X$ by Lemma 7. That is, $CNF(v^r, S')$ is unsatisfiable. Then, by Lemma 11, Line 24 returns a clause, which is a contradiction. So, S' is satisfiable. Then, by IH, Line 24 returns the model count of $CNF(v^r, S')$, which is equivalent to $CNF(v, S)|X$. Similarly, we can show Line 33 returns the model count of $CNF(v, S)|\neg X$. So, Line 43 returns the model count of $CNF(v, S)$. □

Lemma 13. *A call $\#\texttt{SAT}(v, S)$ with a satisfiable state S cannot return on neither Line 30 nor Line 39.*

Proof. Assume call $\#\texttt{SAT}(v, S)$ returns on Line 30. Then, Line 28 must construct a clause. Let $S' = (\Delta, \Gamma', D', .)$ be the SAT state before the call on Line 28. So, unit resolution must detect a contradiction in $\Delta \wedge \Gamma' \wedge D'$. As $\Delta \models \Gamma'$, $\Delta \wedge \Gamma' \wedge D' \equiv \Delta \wedge D'$. So, $\Delta \wedge D'$ is unsatisfiable. Yet, it is easy to see that $D' = D$. That is, $\Delta \wedge D$ is unsatisfiable, which contradicts with S being satisfiable. Thus, call $\#\texttt{SAT}(v, S)$ cannot return on Line 30. Similarly, we can show that it cannot return on Line 39. □

Lemma 14. *Consider a call $\#\texttt{SAT}(v, S)$ with a satisfiable state S. If call $\#\texttt{SAT}(v, S)$ returns on either Line 31 or Line 40, then $D \neq \emptyset$.*

Proof. Assume call $\#\texttt{SAT}(v, S)$ returns on Line 31. So, the condition on Line 27 must fail. That is, the current decision level is strictly greater than the assertion level of the learned clause, and hence $D \neq \emptyset$. Similarly, we can show $D \neq \emptyset$ if the call returns on Line 40. □

Lemma 15. *Consider a call $\#\texttt{SAT}(v, S)$ with a satisfiable state S. If call $\#\texttt{SAT}(v, S)$ returns a (learned) clause, then $D \neq \emptyset$.*

Proof. Assume call $\#\texttt{SAT}(v, S)$ returns a clause. We use strong induction on the height of node $\boxed{\#\texttt{SAT}(v, S)}$ on T to show that $D \neq \emptyset$.

Basis: Consider a leaf node $\boxed{\#\texttt{SAT}(v, S)}$ (i.e., at height 0). By Proposition 4, either v is a leaf vtree node or the call returns a clause on Line 30 or Line 31. As call $\#\texttt{SAT}(v, S)$

returns a clause, v cannot be a leaf node (see Lines 1–4). Also, by Lemma 13, the call cannot return on Line 30. So, by Lemma 14, $D \neq \emptyset$.

Inductive step: As an induction hypothesis (IH), assume that the statement holds for the calls at height less than k where $k \geq 1$. Consider an internal node $\boxed{\#\texttt{SAT}(v, S)}$ (i.e., at height k). By Proposition 3, v is either a decomposition node or a Shannon node.

Assume v is a decomposition node. As call $\#\texttt{SAT}(v, S)$ returns a clause, it must return on either Line 9 or Line 13. Suppose it returns on Line 9. So, Line 6 must construct a clause. So, due to IH, $D \neq \emptyset$. Similarly, we can show $D \neq \emptyset$ if the call returns on Line 13.

Assume v is a Shannon node. As call $\#\texttt{SAT}(v, S)$ returns a clause, it must return on one of the following lines: 21, 29, 30, 31, 38, 39 or 40. By Lemma 13, the call cannot return on Line 30 or Line 39. If the call returns on either Line 31 or Line 40, then $D \neq \emptyset$ by Lemma 14. For the remaining lines (21, 29, 38), using IH, one can easily see that $D \neq \emptyset$. □

We are now ready to prove Theorem 2 and Theorem 3.

Theorem 2. *A call $\#\texttt{SAT}(v, S)$ with a satisfiable state S will return either the model count of component $CNF(v, S)$ or a learned clause. Moreover, if v is the root vtree node, then it will return the model count of $CNF(v, S)$.*

Proof. Due to Lemma 12 and Lemma 15 (note that $D = \emptyset$ in the initial SAT state). □

Theorem 3. *A call $\#\texttt{SAT}(v, S)$ with an unsatisfiable state S will return a learned clause, or one of its ancestral calls $\#\texttt{SAT}(v', .)$ will return a learned clause, where v' is a decomposition vtree node.*

Proof. Let $v_1, \ldots, v_n$ be the decomposition vtree nodes on the path from the vtree root to v (excluding v) and v'_i the child of v_i not appearing on the path. By Lemma 4 and Lemma 6, $\bigwedge_{i=1}^{n} CNF(v'_i, S) \wedge CNF(v, S)$ is unsatisfiable. So, one of the (decomposable) components $CNF(v'_i, S)$ and $CNF(v, S)$ is unsatisfiable. Assume $CNF(v, S)$ is unsatisfiable. Then, due to Lemma 11, call $\#\texttt{SAT}(v, S)$ returns a clause. Assume one of $CNF(v'_i, S)$ is unsatisfiable. Then, by Lemma 2, $CNF(v_i, S)$ is unsatisfiable. Consider the ancestral call $\#\texttt{SAT}(v_i, S_i)$ of $\#\texttt{SAT}(v, S)$. By Corollary 2, $CNF(v_i, S_i) = CNF(v_i, S)$, and hence $CNF(v_i, S_i)$ is unsatisfiable. Thus, by Lemma 11, call $\#\texttt{SAT}(v_i, S_i)$ returns a clause. □

Appendix D. Computing Cache Key

Theorem 4. *Consider a vtree node v and a corresponding SAT state $S = (., ., ., I)$. Define $Key(v, S)$ as the following bit vector: (1) each clause δ in $ContextC(v)$ is mapped into one bit that captures whether $I \models \delta$, and (2) each variable X that appears in both vtree v and $ContextC(v)$ is mapped into two bits that capture whether $X \in I$, $\neg X \in I$, or neither. Then function $Key(v, S)$ is a component key.*

Proof. Let $S' = (., ., ., I')$ be a callable SAT state such that $Key(v, S) = Key(v, S')$. We show $CNF(v, S) \wedge \mathbf{x} \equiv CNF(v, S') \wedge \mathbf{x}'$, where $\mathbf{x}$ (resp., $\mathbf{x}'$) is a term of variables appearing in both v and $ContextC(v)$ such that $\mathbf{x} \subseteq I$ (resp., $\mathbf{x}' \subseteq I'$). Due to (2) in the key definition, $\mathbf{x}$ and $\mathbf{x}'$ are the same. So, it is enough to show $CNF(v, S)|\mathbf{x} \equiv CNF(v, S')|\mathbf{x}$. Let $\gamma = ContextL(v, I)$ and $\gamma' = ContextL(v, I')$. By Theorem 1, γ and γ' must include exactly one literal for each variable in $ContextV(v)$. Then, due to (1) in the key definition, we have $ContextC(v)|\gamma\mathbf{x} = ContextC(v)|\gamma'\mathbf{x}$. So, $CNF(v, S)|\mathbf{x} \equiv CNF(v, S')|\mathbf{x}$. □

Appendix E. Trace of the Model Counting Algorithm

Theorem 5. *If v is the root vtree node, then call* `c2s`$(v, (\Delta, \{\}, \langle\rangle, \{\}))$ *returns a Decision-SDD equivalent to Δ if Δ is satisfiable, and returns an empty clause if Δ is unsatisfiable.*

Proof. The structure of Algorithm 5 is similar to Algorithm 4, except that it constructs an NNF instead of counting models. So, due to Corollary 1, we can conclude that Algorithm 5 returns an NNF equivalent to Δ if Δ is satisfiable, and returns an empty clause otherwise. To complete the proof, we show that the constructed NNF is a Decision-SDD. For that, note that Algorithm 5 is guided by a decision vtree. That is, it generates an NNF structured by a vtree. Further, it is clear from the pseudocode that generated NNF nodes are simply $(\mathbf{X}, \mathbf{Y})$-partitions appearing in Decision-SDDs. Thus, we generate a Decision-SDD. □

References

Bayardo, R. J., & Pehoushek, J. D. (2000). Counting Models Using Connected Components. In *Proceedings of the Seventeenth AAAI Conference on Artificial Intelligence*, pp. 157–162.

Beame, P., & Liew, V. (2015). New Limits for Knowledge Compilation and Applications to Exact Model Counting. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, pp. 131–140.

Birnbaum, E., & Lozinskii, E. L. (1999). The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research*, *10*(1), 457–477.

Blum, M., Chandra, A. K., & Wegman, M. N. (1980). Equivalence of Free Boolean Graphs can be Decided Probabilistically in Polynomial Time. *Information Processing Letters*, *10*(2), 80–82.

Bova, S. (2016). SDDs are Exponentially More Succinct than OBDDs. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 929–935.

Bryant, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, *35*(8), 677–691.

Chavira, M., & Darwiche, A. (2005). Compiling Bayesian Networks with Local Structure. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pp. 1306–1312.

Chavira, M., & Darwiche, A. (2008). On Probabilistic Inference by Weighted Model Counting. *Artifical Intelligence*, *172*(6-7), 772–799.

Chavira, M., Darwiche, A., & Jaeger, M. (2006). Compiling Relational Bayesian Networks for Exact Inference. *International Journal of Approximate Reasoning*, *42*(1-2), 4–20.

Choi, A., & Darwiche, A. (2013a) `http://reasoning.cs.ucla.edu/sdd`.

Choi, A., & Darwiche, A. (2013b). Dynamic Minimization of Sentential Decision Diagrams. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, pp. 187–194.

Choi, A., Den Broeck, G., & Darwiche, A. (2015). Tractable Learning for Structured Probability Spaces: A Case Study in Learning Preference Distributions. In *Proceedings*

of the Twenty-Fourth International Joint Conference on Artificial Intelligence, pp. 2861–2868.

Choi, A., Kisa, D., & Darwiche, A. (2013). Compiling Probabilistic Graphical Models using Sentential Decision Diagrams. In *Proceedings of the Twelfth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pp. 121–132.

Darwiche, A. (2002a). A Compiler for Deterministic, Decomposable Negation Normal Form. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pp. 627–634.

Darwiche, A. (2002b). A Logical Approach to Factoring Belief Networks. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 409–420.

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

Darwiche, A. (2011). SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pp. 819–826.

Darwiche, A., & Marquis, P. (2002). A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, *17*(1), 229–264.

Fierens, D., Den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., & De Raedt, L. (2015). Inference and Learning in Probabilistic Logic Programs using Weighted Boolean Formulas. *Theory and Practice of Logic Programming*, *15*(3), 358–401.

Herrmann, R. G., & de Barros, L. N. (2013). Algebraic Sentential Decision Diagrams in Symbolic Probabilistic Planning. In *Proceedings of the 2013 Brazilian Conference on Intelligent Systems*, pp. 175–181.

Huang, J., & Darwiche, A. (2004). Using DPLL for Efficient OBDD Construction. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, pp. 157–172.

Huang, J., & Darwiche, A. (2007). The Language of Search. *Journal of Artificial Intelligence Research*, *29*(1), 191–219.

Kisa, D., Den Broeck, G., Choi, A., & Darwiche, A. (2014). Probabilistic Sentential Decision Diagrams. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*.

Lomuscio, A., & Paquet, H. (2015). Verification of Multi-Agent Systems via SDD-based Model Checking (Extended Abstract). In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pp. 1713–1714.

Majercik, S. M., & Littman, M. L. (1998). Using Caching to Solve Larger Probabilistic Planning Problems. In *Proceedings of the Fifteenth AAAI Conference on Artificial Intelligence*, pp. 954–959.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, pp. 530–535.

Muise, C., Mcilraith, S. A., Beck, J. C., & Hsu, E. (2012). DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Proceedings of the Twenty-Fifth Canadian Conference on Artificial Intelligence*, pp. 356–361.

Oztok, U., & Darwiche, A. (2014). On Compiling CNF into Decision-DNNF. In *Proceedings of the Twentieth International Conference on Principles and Practice of Constraint Programming*, pp. 42–57.

Oztok, U., & Darwiche, A. (2015). A Top-Down Compiler for Sentential Decision Diagrams. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pp. 3141–3148.

Pipatsrisawat, K., & Darwiche, A. (2007). RSat 2.0: SAT Solver Description. Tech. rep. D–153, UCLA.

Pipatsrisawat, K., & Darwiche, A. (2008a). A New Clause Learning Scheme for Efficient Unsatisfiability Proofs. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pp. 1481–1484.

Pipatsrisawat, K., & Darwiche, A. (2008b). New Compilation Languages Based on Structured Decomposability. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pp. 517–522.

Razgon, I. (2014). On OBDDs for CNFs of bounded treewidth. *CoRR*, *abs/1308.3829*.

Roth, D. (1996). On the Hardness of Approximate Reasoning. *Artificial Intelligence*, *82*(1-2), 273–302.

Sang, T., Bacchus, F., Beame, P., Kautz, H. A., & Pitassi, T. (2004). Combining Component Caching and Clause Learning for Effective Model Counting. In *The Seventh International Conference on Theory and Applications of Satisfiability Testing*.

Sang, T., Beame, P., & Kautz, H. A. (2005). Performing Bayesian Inference by Weighted Model Counting. In *Proceedings of the Twentieth AAAI Conference on Artificial Intelligence*, pp. 475–482.

Thurley, M. (2006). sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, pp. 424–429.

Valiant, L. G. (1979). The Complexity of Computing the Permanent. *Theoretical Computer Science*, *8*(2), 189–201.

Vlasselaer, J., Den Broeck, G., Kimmig, A., Meert, W., & De Raedt, L. (2015). Anytime Inference in Probabilistic Logic Programs with Tp-compilation. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pp. 1852–1858.

Xue, Y., Choi, A., & Darwiche, A. (2012). Basing Decisions on Sentences in Decision Diagrams. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pp. 842–849.