A Top-Down Compiler for Sentential Decision Diagrams

Umut Oztok and **Adnan Darwiche**
Computer Science Department
University of California, Los Angeles
{umut,darwiche}@cs.ucla.edu

Abstract

The sentential decision diagram (SDD) has been recently proposed as a new tractable representation of Boolean functions that generalizes the influential ordered binary decision diagram (OBDD). Empirically, compiling CNFs into SDDs has yielded significant improvements in both time and space over compiling them into OBDDs, using a *bottom-up* compilation approach. In this work, we present a *top-down* CNF to SDD compiler that is based on techniques from the SAT literature. We compare the presented compiler empirically to the state-of-the-art, bottom-up SDD compiler, showing orders-of-magnitude improvements in compilation time.

1 Introduction

The area of *knowledge compilation* has a long tradition in AI (see, e.g., Marquis [1995], Selman and Kautz [1996], Cadoli and Donini [1997]). Since Darwiche and Marquis [2002], this area has settled on three major research directions: (1) identifying new tractable representations that are characterized by their succinctness and polytime support for certain queries and transformations; (2) developing efficient knowledge compilers; and (3) using those representations and compilers in various applications, such as diagnosis [Barrett, 2005; Elliott and Williams, 2006], planning [Barrett, 2004; Palacios *et al.*, 2005; Huang, 2006], and probabilistic inference [Chavira and Darwiche, 2008]. For a recent survey on knowledge compilation, see Darwiche [2014].

This work focuses on developing efficient compilers. In particular, our emphasis is on the compilation of the sentential decision diagram (SDD) [Darwiche, 2011] that was recently discovered as a tractable representation of Boolean functions. SDDs are a strict superset of ordered binary decision diagrams (OBDDs) [Bryant, 1986], which are one of the most popular, tractable representations of Boolean functions. Despite their generality, SDDs still maintain some key properties behind the success of OBDDs in practice. This includes canonicity, which leads to unique representations of Boolean functions. It also includes the support of an efficient `Apply` operation that combines SDDs using Boolean operators.[1] SDDs also come with tighter size upper bounds than OBDDs [Darwiche, 2011; Oztok and Darwiche, 2014; Razgon, 2014b]. Moreover, SDDs have been used in different applications, such as probabilistic planning [Herrmann and de Barros, 2013], probabilistic logic programs [Vlasselaer *et al.*, 2014; 2015], probabilistic inference [Choi *et al.*, 2013; Renkens *et al.*, 2014], verification of multi-agent systems [Lomuscio and Paquet, 2015], and tractable learning [Kisa *et al.*, 2014; Choi *et al.*, 2015].

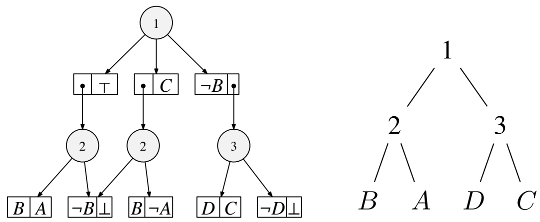
Almost all of these applications are based on the *bottom-up* SDD compiler developed by Choi and Darwiche [2013a], which was also used to compile CNFs into SDDs [Choi and Darwiche, 2013b]. This compiler constructs SDDs by compiling small pieces of a knowledge base (KB) (e.g., clauses of a CNF). It then combines these compilations using the `Apply` 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 and Darwiche, 2004]. Thus, it has a potential to further improve the results on CNF to SDD compilations. Motivated by this, we study the compilation of CNFs into SDDs by a top-down approach.

This paper is based on the following contributions. We first identify a subset of SDDs, called *Decision-SDDs*, which facilitates the top-down compilation of SDDs. We then introduce a top-down algorithm for compiling CNFs into Decision-SDDs, which is harnessed with techniques used in modern SAT solvers, and a new component caching scheme. We finally present empirical results, showing orders-of-magnitude improvement in compilation time, compared to the state-of-the-art, bottom-up SDD compiler.

This paper is organized as follows. Section 2 provides technical background. Section 3 introduces the new representa-

[1] The `Apply` operation combines two SDDs using any Boolean operator, and has its origins in the OBDD literature [Bryant, 1986].



(a) An SDD (b) A vtree

Figure 1: An SDD and a vtree for $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.

tion Decision-SDD. Section 4 provides a formal framework for the compiler, which is then presented in Section 5. Experimental results are given in Section 6 and related work in Section 7. Due to space limitations, proofs of theorems are delegated to the full version of the paper.[2]

2 Technical Background

Upper case letters (e.g., X) denote variables and bold upper case letters (e.g., $\mathbf{X}$) denote sets of variables. A *literal* is a variable or its negation. A *Boolean function* $f(\mathbf{Z})$ maps each instantiation $\mathbf{z}$ of variables $\mathbf{Z}$ to true ($\top$) or false ($\bot$).

CNF: A *conjunctive normal form* (CNF) is a set of clauses, where each clause is a disjunction of literals. 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. Given two CNFs Δ and Γ, we will write $\Delta \models \Gamma$ to mean that Δ entails Γ.

SDD: A Boolean function $f(\mathbf{X}, \mathbf{Y})$, with disjoint sets of variables $\mathbf{X}$ and $\mathbf{Y}$, can always be decomposed into

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

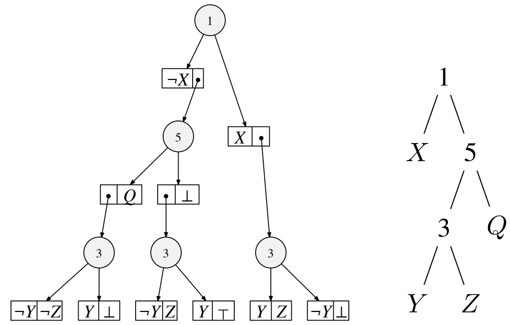
such that $p_i \neq \bot$ for all i; $p_i \wedge p_j = \bot$ for $i \neq j$; and $\bigvee_i p_i = \top$. A decomposition satisfying the above properties is known as an $(\mathbf{X}, \mathbf{Y})$-partition [Darwiche, 2011]. Moreover, each p_i is called a *prime,* each s_i is called a *sub*, and the $(\mathbf{X}, \mathbf{Y})$-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*, which is a full binary tree whose leaves are labeled with variables; see Figure 1(b). Consider now the vtree in Figure1(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. Its left subtree contains variables $\mathbf{X} = \{A, B\}$ and its 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 1(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 $\boxed{p|s}$. 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 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 1(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 1(a), SDD nodes are labeled with vtree nodes they are normalized for.



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

Figure 2: A Decision-SDD and its corresponding vtree.

3 Decision-SDDs

We will next define the language of *Decision-SDDs*, which is a strict subset of SDDs and a strict superset of OBDDs. Our new top-down compiler will construct Decision-SDDs.

To define Decision-SDDs, 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(b) are Shannon nodes, with X and Y as their Shannon variables. Vtree node 5 is a decomposition node. An SDD node that is normalized for a Shannon (decomposition) vtree node is called a *Shannon (decomposition) decision node*. A Shannon decision node has the form $\{(X, \alpha), (\neg X, \beta)\}$, where X is a Shannon variable.

Definition 1 (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* $s_1 = \neg s_2$.

[2]Available at http://reasoning.cs.ucla.edu.

Figure 2 shows a Decision-SDD and a corresponding vtree for the CNF $\{Y \vee \neg Z, \neg X \vee Z, X \vee \neg Y, X \vee Q\}$. The language of Decision-SDDs is complete: every Boolean function can be represented by a Decision-SDD using an appropriate vtree.

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.

If an SDD is based on a general vtree, it may or may not be a Decision-SDD. However, the following class of vtrees, identified by Oztok and Darwiche [2014], guarantees a Decision-SDD.

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 said to be a decision vtree for Δ iff every clause in Δ is compatible with only Shannon vtree nodes.*[3]

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

Proposition 1 *Let v be a decision vtree for CNF Δ. An SDD for Δ that respects vtree v must be a Decision-SDD.*

As such, the input to our compiler will be a CNF and a corresponding decision vtree, and the result will be a Decision-SDD for the CNF. Note that one can always construct a decision vtree for any CNF [Oztok and Darwiche, 2014].

When every internal vtree node is a Shannon vtree node (i.e., the vtree is right-linear), the Decision-SDD corresponds to an OBDD. A quasipolynomial separation between SDDs and OBDDs was given by Razgon [2014b]. As it turns out, the SDDs used to show this separation are actually Decision-SDDs. We now complement this result by showing that Decision-SDDs can be simulated by OBDDs with at most a quasipolynomial increase in size (it is currently unknown if this holds for general SDDs).

Theorem 1 *Every Decision-SDD with n variables and size N has an equivalent OBDD with size $\leq N^{1+\log n}$.*

The above result is based on [Razgon, 2014a], which simulates decomposable AND-OBDDs with OBDDs.[4]

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.[5] 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 1. We finally note that there is no known separation result between Decision-SDDs and SDDs.

[3] Without loss of generality, Δ has no empty or unit clauses.

[4] A decomposable AND-OBDD can be turned into a Decision-SDD in polytime, but it is not clear whether the converse is true.

[5] A vtree dissects a variable order if the order is generated by a left-right traversal of the vtree.

Algorithm 1: $\texttt{SAT}(\Delta)$

Input: Δ : a CNF
Output: $\top$ if Δ is satisfiable; $\bot$ otherwise
1 $\Gamma \leftarrow \{\}$ `// learned clauses`
2 $D \leftarrow \langle\rangle$ `// decision sequence`
3 **while** true **do**
4   **if** *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
5   **if** $D = \langle\rangle$ **then return** $\bot$
6     $\alpha \leftarrow$ asserting clause for (Δ, Γ, D)
7     $m \leftarrow$ the assertion level of α
8     $D \leftarrow$ the first m decisions of D
9     $\Gamma \leftarrow \Gamma \cup \{\alpha\}$ `// learning clause α`
10   **else**
11   **if** *ℓ is a literal where neither ℓ nor $\neg\ell$ are implied by unit resolution from $\Delta \wedge \Gamma \wedge D$* **then** $D \leftarrow D; \ell$
12   **else return** $\top$

We will next provide a top-down algorithm for compiling CNFs into Decision-SDDs, which is based on state-of-the-art techniques from SAT solving. Our intention is to provide a formal description of the algorithm, which is precise and detailed enough to be reproducible by the community. We will start by providing a formal description of our framework in Section 4, and then present our algorithm in Section 5.

4 A Formal Framework for the Compiler

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 SDD compiler (we employ first-UIP asserting clauses as implemented by `RSat` [Pipatsrisawat and Darwiche, 2007]).

As a first step towards introducing our compiler, we present in Algorithm 1 a modern SAT solver that is based on unit resolution and clause learning. This algorithm 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 erased until we reach the decision level corresponding to the *assertion level* of clause α, at which point α is added to Γ.[6] 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 12). In the first case, the input CNF must be unsatisfiable. In the second case, the CNF is

[6] The assertion level is computed when the clause is learned. It corresponds to the lowest decision level at which unit resolution is guaranteed to derive a new literal using the learned clause.

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 3: Macros for some SAT-solver primitives.

Algorithm 2: $\#\mathtt{SAT}(\pi, S)$

Input: π : a variable order, S : a SAT state (Δ, Γ, D, I)
Output: Model count of $\Delta \wedge D$ over variables in π, or a clause
1 **if** *there is no variable in* π **then return** 1
2 $X \leftarrow$ first variable in π
3 **if** X *or* $\neg X$ *belongs to* I **then return** $\#\mathtt{SAT}(\pi\backslash\{X\}, S)$
4 $h \leftarrow decide\_literal(X, S)$
5 **if** h *is success* **then** $h \leftarrow \#\mathtt{SAT}(\pi\backslash\{X\}, S)$
6 $undo\_decide\_literal(X, S)$
7 **if** h *is a learned clause* **then**
8   **if** $at\_assertion\_level(h, S)$ **then**
9     $h \leftarrow assert\_clause(h, S)$
10   **if** h *is success* **then return** $\#\mathtt{SAT}(\pi, S)$
11   **else return** h
12   **else return** h
13 $l \leftarrow decide\_literal(\neg X, S)$
14 **if** l *is success* **then** $l \leftarrow \#\mathtt{SAT}(\pi\backslash\{X\}, S)$
15 $undo\_decide\_literal(\neg X, S)$
16 **if** l *is a learned clause* **then**
17   **if** $at\_assertion\_level(l, S)$ **then**
18     $l \leftarrow assert\_clause(l, S)$
19   **if** l *is success* **then return** $\#\mathtt{SAT}(\pi, S)$
20   **else return** l
21   **else return** l
22 **return** $h + l$

satisfiable with D as a satisfying assignment.

Algorithm 1 is iterative. Our SDD compiler, however, will be recursive. To further prepare for this recursive algorithm, we will take two extra steps. The first step is to abstract the primitives used in SAT solvers (Figure 3), viewing them as operations on what we shall call a SAT state.

Definition 3 *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. 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.*[7]

Here, Δ is the input CNF, Γ is the set of learned clauses, D is the decision sequence, and I are the literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$. Hence, $\Delta \models \Gamma$ and $D \subseteq I$.

The second step towards presenting our compilation algorithm is a recursive algorithm for counting the models of a CNF, which utilizes the above abstractions (i.e., the SAT state and its associated primitives in Figure 3). To simplify the presentation, we will assume a variable order π of the CNF. If X is the first variable in order π, then one recursively counts the models of $\Delta \wedge X$, recursively counts the models of $\Delta \wedge \neg X$, and then add these results to obtain the model count of Δ. This is given in Algorithm 2, which is called initially with the SAT state $(\Delta, \{\}, \langle\rangle, \{\})$ to count the models of Δ. 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 SDD compilation algorithm — but in contrast to SAT solvers which only consider one phase of the variable. Moreover, Algorithm 2 employs the primitives of Figure 3 in the same way that our SDD compiler will employ them later.

The following is a key observation about Algorithm 2 (and the SDD compilation 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 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.[8]

5 A Top-Down SDD Compiler

We are now ready to present our SDD compilation algorithm, whose overall structure is similar to Algorithm 2, but with a few exceptions. First, the SDD compilation algorithm is driven by a vtree instead of a variable order. Second, it uses the vtree structure to identify disconnected CNF components and compiles these components independently. Third, it employs a component caching scheme to avoid compiling the same component multiple times.

This is given in Algorithm 3, which is called initially with the SAT state $S = (\Delta, \{\}, \langle\rangle, \{\})$ and a decision vtree v for Δ, to compile an SDD for CNF Δ.[9] When the algorithm is applied to a Shannon vtree node, its behavior is similar to Algorithm 2 (Lines 15–44). 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

[7] Without loss of generality, Δ has no empty or unit clauses.

[8] 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 $\Delta \models \Gamma$).

[9] Algorithm 3 assumes that certain negations are freely available (e.g., $\neg p$ on Line 14). One can easily modify the algorithm so it returns both an SDD and its negation, making all such negations freely available. We skip this refinement here for clarity of exposition, but it can be found in the longer version of the paper.

Algorithm 3: $\texttt{c2s}(v, S)$

$unique(\alpha)$ 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 or a clause

1 **if** *v is a leaf node* **then**
2     $X \leftarrow$ variable of v
3   **if** *X or $\neg X$ belongs to I* **then** **return** *the literal of X that belongs to I*
4   **else return** $\top$
5 **else if** *v is a decomposition vtree node* **then**
6     $p \leftarrow \texttt{c2s}(v^l, S)$
7   **if** *p is a learned clause* **then**
8         $clean\_cache(v^l)$
9   **return** p
10     $s \leftarrow \texttt{c2s}(v^r, S)$
11   **if** *s is a learned clause* **then**
12         $clean\_cache(v)$
13   **return** s
14   **return** $unique(\{(p, s), (\neg p, \bot)\})$
15 **else**
16     $key \leftarrow Key(v, S)$
17   **if** $cache(key) \neq nil$ **then return** $cache(key)$
18     $X \leftarrow$ Shannon variable of v
19   **if** *either X or $\neg X$ belongs to I* **then**
20         $p \leftarrow$ the literal of X that belongs to I
21         $s \leftarrow \texttt{c2s}(v^r, S)$
22   **if** *s is a learned clause* **then return** s
23   **return** $unique(\{(p, s), (\neg p, \bot)\})$
24     $s_1 \leftarrow decide\_literal(X, S)$
25   **if** *s_1 is success* **then** $s_1 \leftarrow \texttt{c2s}(v^r, S)$
26     $undo\_decide\_literal(X, S)$
27   **if** *s_1 is a learned clause* **then**
28   **if** $at\_assertion\_level(s_1, S)$ **then**
29             $s_1 \leftarrow assert\_clause(s_1, S)$
30   **if** *s_1 is success* **then return** $\texttt{c2s}(v, S)$
31   **else return** s_1
32   **else return** s_1
33     $s_2 \leftarrow decide\_literal(\neg X, S)$
34   **if** *s_2 is success* **then** $s_2 \leftarrow \texttt{c2s}(v^r, S)$
35     $undo\_decide\_literal(\neg X, S)$
36   **if** *s_2 is a learned clause* **then**
37   **if** $at\_assertion\_level(s_2, S)$ **then**
38             $s_2 \leftarrow assert\_clause(s_2, S)$
39   **if** *s_2 is success* **then return** $\texttt{c2s}(v, S)$
40   **else return** s_2
41   **else return** s_2
42     $\alpha \leftarrow unique(\{(X, s_1), (\neg X, s_2)\})$
43     $cache(key) \leftarrow \alpha$
44   **return** α

the left child v^l and another with the right child v^r (since v is a decision vtree for Δ). In this case, the algorithm compiles each component independently and combines the results.

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:

- $CNF(v)$: The clauses of Δ mentioning only variables inside the vtree rooted at 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).

We start with the following invariant of Algorithm 3.

Theorem 2 *Every call* $\texttt{c2s}(v, S)$ *with* $S = (\Delta, \Gamma, D, I)$ *satisfies* $\Delta \models \Gamma$ *and* $Vars(D) \subseteq ContextV(v) \subseteq Vars(I)$.

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 4 *The component of vtree node v and state* $S = (.,.,., I)$ *is* $CNF(v, S) = CNF(v) \wedge ContextC(v)|\gamma$, *where γ are the literals of $ContextV(v)$ appearing in I.*

Hence, the component $CNF(v, S)$ will only mention variables in vtree v. Moreover, 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 43).

Theorem 3 *A call* $\texttt{c2s}(v, S)$ *with a satisfiable state S will return either an SDD for component $CNF(v, S)$ or a learned clause. Moreover, if v is the root vtree node, then a learned clause will not be returned.*

Theorem 4 *A call* $\texttt{c2s}(v, S)$ *with an unsatisfiable state S will return a learned clause, or one of its ancestral calls* $\texttt{c2s}(v', S')$ *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* $\texttt{c2s}(v, (\Delta, \{\}, \langle\rangle, \{\}))$ *returns an SDD for 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 43). This requires an explanation of the difference in behavior between satisfiable and unsatisfiable states (based on Theorem 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 $\Delta_{\mathbf{Y}}$). In this case, one would prefer to compile $\Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ instead of $\Delta_{\mathbf{X}}$ as the former can make unit resolution more complete, leading to a more efficient compilation. In fact, this is exactly what Algorithm 3 does, as it includes the learned clauses Γ in unit resolution when compiling 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, compiling $\Delta_{\mathbf{X}} \wedge I_{\mathbf{X}}$ will yield an SDD that implies $\Delta_{\mathbf{X}}$ 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 compiled incorrectly, the conjunction remains a valid result. Without component caching, the incorrect compilation will be discarded. However, with component caching, one also needs to ensure that incorrect compilations are not cached (as observed by Sang *et al.* [2004]).

By Theorem 4, if we reach Line 8 or 12, then state S may be unsatisfiable and we can no longer trust the results cached below v. Hence, $clean\_cache(v)$ on Line 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 43.

Definition 5 *A function $Key(v, S)$ is called a* <u>*component key*</u> *iff $Key(v, S) = Key(v, S')$ implies that components $CNF(v, S)$ and $CNF(v, S')$ are equivalent.*

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 5 *Consider a vtree node v and a corresponding 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 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.*

6 Experimental Results

We now present an empirical evaluation of the new top-down compiler. In our experiments, we used two sets of benchmarks. First, we used some CNFs from the `iscas85`, `iscas89`, and `LGSynth89` suites, which correspond to 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 and Darwiche, 2013a]. 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 1.

The first experiment compares the top-down compiler against the bottom-up SDD compiler. Here, we first generate a decision vtree[10] 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.[11] 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 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 our 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.

7 Related Work

Our algorithm for compiling CNFs into SDDs is based on a similar algorithm, introduced recently [Oztok and Darwiche, 2014]. The latter algorithm was proposed to improve a size upper bound on SDDs. However, it did not identify Decision-SDDs, nor did it suggest a practical implementation. The current work makes the previously introduced algorithm practical by adding powerful techniques from the SAT literature and defining a practical caching scheme, resulting in an efficient compiler that advances the state-of-the-art.

Combining clause learning and component caching was already used in the context of knowledge compilation [Darwiche, 2004] and model counting [Sang *et al.*, 2004]. Yet, neither of these works described the corresponding algorithms and their properties at the level of detail and precision that we did here. A key difference between the presented top-down compiler and the one introduced in Darwiche [2004], called `c2d`, is that we compile CNFs into SDDs, while `c2d` compiles CNFs into d-DNNFs. These two languages differ in their succinctness and tractability (SDDs are a strict subset of d-DNNFs, and are less succinct but more tractable). For example, SDDs can be negated in linear time. Hence, the CNF-to-SDD compiler we introduced can easily be used as a DNF-to-SDD compiler. For that, we first negate the DNF into a CNF by flipping the literals and treating each term as a clause. After compiling the resulting CNF into an SDD, we can negate the resulting SDD efficiently, which would be-

[10] We obtained decision vtrees as in Oztok and Darwiche [2014].

[11] 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 1: 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.

come the SDD for the given DNF. Since no efficient negation algorithm is known for d-DNNFs, one cannot use `c2d` when the original knowledge base is represented in DNF. We note that we did not evaluate our compiler for compiling DNFs into SDDs, so we do not know how practical it would be. Still, it can be immediately used to compile DNFs, which has not been discussed before in the context of top-down compilation. Another top-down compiler, called `eadt`, was presented recently [Koriche *et al.*, 2013], which compiles CNFs into a tractable language that makes use of decision trees with `xor` nodes. A detailed comparison of bottom-up and top-down compilation has been made before in the context of compiling CNFs into OBDDs [Huang and Darwiche, 2004]. Our work can be seen as making a similar comparison for compiling CNFs into SDDs.

8 Conclusion

We identified a subset of SDDs, called Decision-SDDs, and introduced a top-down algorithm for compiling CNFs into Decision-SDDs that is based on techniques from the SAT literature. We provided a formal description of the new algorithm with the hope that it would facilitate the development of efficient compilers by the community. Our empirical evaluation showed that the presented top-down compiler can yield significant improvements in compilation time against the state-of-the-art bottom-up SDD compiler, assuming that the input is a CNF.

Acknowledgments

This work has been partially supported by ONR grant #N00014-12-1-0423 and NSF grant #IIS-1118122.

References

[Barrett, 2004] Anthony Barrett. From Hybrid Systems to Universal Plans Via Domain Compilation. In *KR*, pages 654–661, 2004.

[Barrett, 2005] Anthony Barrett. Model Compilation for Real-Time Planning and Diagnosis with Feedback. In *IJCAI*, pages 1195–1200, 2005.

[Bryant, 1986] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[Cadoli and Donini, 1997] Marco Cadoli and Francesco M. Donini. A Survey on Knowledge Compilation. *AI Commun.*, 10(3,4):137–150, 1997.

[Chavira and Darwiche, 2008] Mark Chavira and Adnan Darwiche. On Probabilistic Inference by Weighted Model Counting. *Artif. Intell.*, 172(6-7):772–799, 2008.

[Choi and Darwiche, 2013a] Arthur Choi and Adnan Darwiche. `http://reasoning.cs.ucla.edu/sdd`, 2013.

[Choi and Darwiche, 2013b] Arthur Choi and Adnan Darwiche. Dynamic Minimization of Sentential Decision Diagrams. In *AAAI*, 2013.

[Choi *et al.*, 2013] Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling Probabilistic Graphical Models using Sentential Decision Diagrams. In *ECSQARU*, pages 121–132, 2013.

[Choi *et al.*, 2015] Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. Tractable Learning for Structured Probability Spaces: A Case Study in Learning Preference Distributions. In *IJCAI*, 2015. To appear.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *JAIR*, 17:229–264, 2002.

[Darwiche, 2004] Adnan Darwiche. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *ECAI*, pages 328–332, 2004.

[Darwiche, 2011] Adnan Darwiche. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *IJCAI*, pages 819–826, 2011.

[Darwiche, 2014] Adnan Darwiche. Tractable Knowledge Representation Formalisms. In Lucas Bordeaux, Youssef Hamadi, and Pushmeet Kohli, editors, *Tractability*, pages 141–172. Cambridge University Press, 2014.

[Elliott and Williams, 2006] Paul Elliott and Brian C. Williams. DNNF-based Belief State Estimation. In *AAAI*, pages 36–41, 2006.

[Herrmann and de Barros, 2013] Ricardo G. Herrmann and Leliane N. de Barros. Algebraic Sentential Decision Diagrams in Symbolic Probabilistic Planning. In *BRACIS*, 2013.

[Huang and Darwiche, 2004] Jinbo Huang and Adnan Darwiche. Using DPLL for Efficient OBDD Construction. In *SAT*, 2004.

[Huang, 2006] Jinbo Huang. Combining Knowledge Compilation and Search for Conformant Probabilistic Planning. In *ICAPS*, pages 253–262, 2006.

[Kisa *et al.*, 2014] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic Sentential Decision Diagrams. In *KR*, 2014.

[Koriche *et al.*, 2013] Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. Knowledge Compilation for Model Counting: Affine Decision Trees. In *IJCAI*, 2013.

[Lomuscio and Paquet, 2015] Alessio Lomuscio and Hugo Paquet. Verification of Multi-Agent Systems via SDD-based Model Checking (Extended Abstract). In *AAMAS15*, 2015.

[Marquis, 1995] Pierre Marquis. Knowledge Compilation Using Theory Prime Implicates. In *IJCAI*, pages 837–845, 1995.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535, 2001.

[Oztok and Darwiche, 2014] Umut Oztok and Adnan Darwiche. On Compiling CNF into Decision-DNNF. In *CP*, 2014.

[Palacios *et al.*, 2005] Héctor Palacios, Blai Bonet, Adnan Darwiche, and Hector Geffner. Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations. In *ICAPS*, pages 141–150, 2005.

[Pipatsrisawat and Darwiche, 2007] Knot Pipatsrisawat and Adnan Darwiche. RSat 2.0: SAT Solver Description. Technical Report D–153, UCLA, 2007.

[Razgon, 2014a] Igor Razgon. Personal communication, 2014.

[Razgon, 2014b] Igor Razgon. On OBDDs for CNFs of Bounded Treewidth. *ArXiv e-prints*, 2014.

[Renkens *et al.*, 2014] Joris Renkens, Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Explanation-Based Approximate Weighted Model Counting for Probabilistic Logics. In *AAAI*, pages 2490–2496, 2014.

[Sang *et al.*, 2004] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT*, 2004.

[Selman and Kautz, 1996] Bart Selman and Henry A. Kautz. Knowledge Compilation and Theory Approximation. *J. ACM*, 43(2):193–224, 1996.

[Vlasselaer *et al.*, 2014] Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. Compiling probabilistic logic programs into Sentential Decision Diagrams. In *PLP*, 2014.

[Vlasselaer *et al.*, 2015] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. Anytime Inference in Probabilistic Logic Programs with Tp-compilation. In *IJCAI*, 2015. To appear.

[Xue *et al.*, 2012] Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing Decisions on Sentences in Decision Diagrams. In *AAAI*, pages 842–849, 2012.