# On the Role of Canonicity in Knowledge Compilation

**Guy Van den Broeck** and **Adnan Darwiche**
Computer Science Department
University of California, Los Angeles
{guyvdb,darwiche}@cs.ucla.edu

### Abstract

Knowledge compilation is a powerful reasoning paradigm with many applications across AI and computer science more broadly. We consider the problem of bottom-up compilation of knowledge bases, which is usually predicated on the existence of a polytime function for combining compilations using Boolean operators (usually called an `Apply` function). While such a polytime `Apply` function is known to exist for certain languages (e.g., OBDDs) and not exist for others (e.g., DNNFs), its existence for certain languages remains unknown. Among the latter is the recently introduced language of Sentential Decision Diagrams (SDDs): while a polytime `Apply` function exists for SDDs, it was unknown whether such a function exists for the important subset of compressed SDDs which are canonical. We resolve this open question in this paper and consider some of its theoretical and practical implications. Some of the findings we report question the common wisdom on the relationship between bottom-up compilation, language canonicity and the complexity of the `Apply` function.

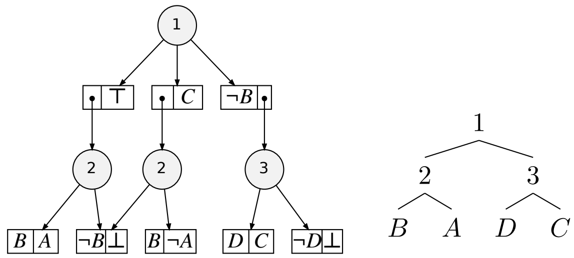
## Introduction

Knowledge compilation is an area of research that has a long tradition in AI; see Cadoli and Donini (1997). Initially, work in this area took the form of searching for tractable languages based on CNFs (e.g. Selman and Kautz; del Val; Marquis (1991; 1994; 1995)). However, the area took a different turn a decade ago with the publication of the "Knowledge Compilation Map" (Darwiche and Marquis 2002). Since then, the work on knowledge compilation became structured across three major dimensions; see Darwiche (2014) for a recent survey: (1) identifying new tractable languages and placing them on the map by characterizing their succinctness and the polytime operations they support; (2) building compilers that map propositional knowledge bases into tractable languages; and (3) using these languages in various applications, such as diagnosis (Elliott and Williams 2006; Huang and Darwiche 2005; Barrett 2005; Siddiqi and Huang 2007), planning (Palacios et al. 2005; Huang 2006), probabilistic reasoning (Chavira, Darwiche, and Jaeger 2006; Chavira and Darwiche 2008; Fierens et al. 2011), and statistical relational learning (Fierens et al. 2013). More recently, knowledge compilation has greatly influenced the area of probabilistic databases (Suciu et al. 2011; Jha and Suciu 2011; Rekatsinas, Deshpande, and Getoor 2012; Beame et al. 2013) and became also increasingly influential in first-order probabilistic inference (Van den Broeck et al. 2011; Van den Broeck 2011; Van den Broeck 2013). Another area of influence is in the learning of tractable probabilistic models (Lowd and Rooshenas 2013; Gens and Domingos 2013; Kisa et al. 2014a), as knowledge compilation has formed the basis of a number of recent approaches in this area of research (ICML hosted the First International Workshop on Learning Tractable Probabilistic Models (LTPM) in 2014).

One of the more recent introductions to the knowledge compilation map is the Sentential Decision Diagram (SDD) (Darwiche 2011). The SDD is a target language for knowledge compilation. That is, once a propositional knowledge base is compiled into an SDD, the SDD can be reused to answer multiple hard queries efficiently (e.g., clausal entailment or model counting). SDDs subsume Ordered Binary Decision Diagrams (OBDDs) (Bryant 1986) and come with tighter size bounds (Darwiche 2011; Razgon 2013; Oztok and Darwiche 2014), while still being equally powerful as far as their polytime support for classical queries (e.g., the ones in Darwiche and Marquis (2002)). Moreover, SDDs are a specialization of d-DNNFs (Darwiche 2001), which received much attention over the last decade. Even though SDDs are less succinct than d-DNNFs, they can be compiled *bottom-up,* just like OBDDs. For example, a clause can be compiled by disjoining the SDDs corresponding to its literals, and a CNF can be compiled by conjoining the SDDs corresponding to its clauses. This bottom-up compilation is implemented using the `Apply` function, which combines two SDDs using Boolean operators.[1] Bottom-up compilation makes SDDs attractive for several AI applications, in particular for reasoning in probabilistic graphical models (Choi, Kisa, and Darwiche 2013) and probabilistic programs, both exact (Vlasselaer et al. 2014) and approximate (Renkens et al. 2014), as well as tractable learning (Kisa et al. 2014a; 2014b). Bottom-up compilation can be critical when the knowledge base to be compiled is constructed incrementally



[1] `Apply` originated in the OBDD literature (Bryant 1986).




(a) An SDD (b) A vtree

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

(see the discussion in Pipatsrisawat and Darwiche (2008)).

## An Open Problem and its Implications

According to common wisdom, a language supports bottom-up compilation only if it supports a polytime `Apply` function. For example, OBDDs are known to support bottom-up compilation and have traditionally been compiled this way. In fact, the discovery of SDDs was mostly driven by the need for bottom-up compilation, which was preceded by the discovery of *structured decomposability* (Pipatsrisawat and Darwiche 2008): a property that enables some Boolean operations to be applied in polytime. SDDs satisfy this property and stronger ones, leading to a polytime `Apply` function (Darwiche 2011). It was unknown, however, whether this function existed for the important subset of *compressed* SDDs which are canonical. This has been an open question since SDDs were first introduced in (Darwiche 2011).

We resolve this open question in this paper, showing that such an `Apply` function does not exist in general. We also pursue some theoretical and practical implications of this result, on bottom-up compilation in particular. On the practical side, we reveal an empirical finding that seems quite surprising: bottom-up compilation with compressed SDDs is much more feasible practically than with uncompressed ones, even though the latter supports a polytime `Apply` function while the former does not. This finding questions common convictions on the relative importance of a polytime `Apply` in contrast to canonicity as desirable properties for a language that supports efficient bottom-up compilation. On the theoretical side, we show that some transformations (e.g., conditioning) can blow up the size of compressed SDDs, while they do not for uncompressed SDDs.

## Technical Background

We will use the following notation for propositional logic. Upper-case letters (e.g., $X$) denote propositional *variables* and bold letters represent sets of variables (e.g., $\mathbf{X}$). A *literal* is a variable or its negation. A *Boolean function* $f(\mathbf{X})$ maps each instantiation $\mathbf{x}$ of variables $\mathbf{X}$ into $\top$ (true) or $\bot$ (false).

**The SDD Representation** The SDD can be thought of as a "data structure" for representing Boolean functions since SDDs can be canonical and support a number of efficient operations for constructing and manipulating Boolean functions (Darwiche 2011; Xue, Choi, and Darwiche 2012; Choi and Darwiche 2013).

**Partitions** SDDs are based on a new type of Boolean function decomposition, called *partitions.* Consider a Boolean function $f$ and suppose that we split its variables into two disjoint sets, $\mathbf{X}$ and $\mathbf{Y}$. We can always decompose the function $f$ as

$$f = \Big[p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})\Big] \vee \cdots \vee \Big[p_n(\mathbf{X}) \wedge s_n(\mathbf{Y})\Big],$$

where we require that the sub-functions $p_i(\mathbf{X})$ are mutually exclusive, exhaustive, and consistent (non-false). This kind of decomposition is called an $(\mathbf{X}, \mathbf{Y})$-*partition*, and it always exists. The sub-functions $p_i(\mathbf{X})$ are called *primes* and the sub-functions $s_i(\mathbf{Y})$ are called *subs* (Darwiche 2011). For an example, consider the function: $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$. By splitting the function variables into $\mathbf{X} = \{A, B\}$ and $\mathbf{Y} = \{C, D\}$, we get the following decomposition:

$$(\underbrace{A \wedge B}_{\text{prime}} \wedge \underbrace{\top}_{\text{sub}}) \vee (\underbrace{\neg A \wedge B}_{\text{prime}} \wedge \underbrace{C}_{\text{sub}}) \vee (\underbrace{\neg B}_{\text{prime}} \wedge \underbrace{C \wedge D}_{\text{sub}}). \quad (1)$$

The primes are mutually exclusive, exhaustive and non-false. This decomposition is represented by a *decision SDD node*, which is depicted by a circle ◯ as in Figure 1. The above decomposition corresponds to the root decision node in this figure. The children of a decision SDD node are depicted by paired boxes $\boxed{p\,|\,s}$, called *elements.* The left box of an element corresponds to a prime $p$, while the right box corresponds to its sub $s$. In the graphical depiction of SDDs, a prime $p$ or sub $s$ are either a constant, literal or pointer to a decision SDD node. Constants and literals are called *terminal* SDD nodes.

**Compression** An $(\mathbf{X}, \mathbf{Y})$-partition is *compressed* when its subs $s_i(\mathbf{Y})$ are distinct. Without the compression property, a function can have many different $(\mathbf{X}, \mathbf{Y})$-partitions. However, for a function $f$ and a particular split of the function variables into $\mathbf{X}$ and $\mathbf{Y}$, there exists a unique *compressed* $(\mathbf{X}, \mathbf{Y})$-partition of function $f$. The $(AB, CD)$-partition in (1) is compressed. Its function has another $(AB, CD)$-partition, which is not compressed:

$$\begin{aligned}&\{(A \wedge B, \top), (\neg A \wedge B, C),\\ &\ (A \wedge \neg B, D \wedge C), (\neg A \wedge \neg B, D \wedge C)\}. \end{aligned} \quad (2)$$

An uncompressed $(\mathbf{X}$,$\mathbf{Y})$-partition can be compressed by merging all elements $(p_1, s), \ldots, (p_n, s)$ that share the same sub into one element $(p_1 \vee \cdots \vee p_n, s)$. Compressing (2) combines the two last elements into $([A \wedge \neg B] \vee [\neg A \wedge \neg B], D \wedge C) = (\neg B, D \wedge C)$, resulting in (1). This is the unique compressed $(AB, CD)$-partition of $f$. A *compressed SDD* is one which contains only compressed partitions.

**Vtree** An SDD can be defined using a sequence of recursive $(\mathbf{X}, \mathbf{Y})$-partitions. To build an SDD, we need to determine which $\mathbf{X}$ and $\mathbf{Y}$ are used in every partition in the SDD. This process is governed by a vtree: a full, binary tree, whose leaves are labeled with the function variables; see Figures 1b and 2. The root $v$ of the vtree partitions variables into those

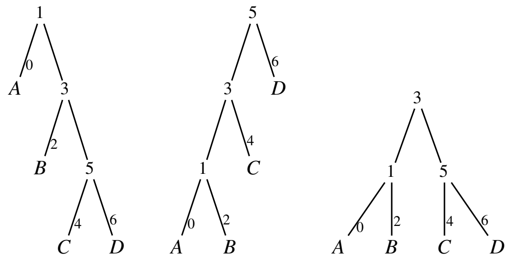


Figure 2: Different vtrees over the variables $A, B, C$, and $D$. The vtree on the left is right-linear.

| Query | Description | OBDD | SDD | d-DNNF |
|---|---|---|---|---|
| **CO** | consistency | $\surd$ | $\surd$ | $\surd$ |
| **VA** | validity | $\surd$ | $\surd$ | $\surd$ |
| **CE** | clausal entailment | $\surd$ | $\surd$ | $\surd$ |
| **IM** | implicant check | $\surd$ | $\surd$ | $\surd$ |
| **EQ** | equivalence check | $\surd$ | $\surd$ | ? |
| **CT** | model counting | $\surd$ | $\surd$ | $\surd$ |
| **SE** | sentential entailment | $\surd$ | $\surd$ | $\circ$ |
| **ME** | model enumeration | $\surd$ | $\surd$ | $\surd$ |

Table 1: Analysis of supported queries, following Darwiche and Marquis (2002). $\surd$ means that a polytime algorithm exists for the corresponding language/query, while $\circ$ means that no such algorithm exists unless $P = NP$.

appearing in the left subtree ($\mathbf{X}$) and those appearing in the right subtree ($\mathbf{Y}$). This implies an $(\mathbf{X}, \mathbf{Y})$-partition $\beta$ of the Boolean function, leading to the root SDD node (we say in this case that partition $\beta$ is *normalized* for vtree node $v$). The primes and subs of this partition are turned into SDDs, recursively, using vtree nodes from the left and right subtrees. The process continues until we reach variables or constants (i.e., terminal SDD nodes). The vtree used to construct an SDD can have a dramatic impact on the SDD, sometimes leading to an exponential difference in the SDD size.

**Two Forms of Canonicity** Even though compressed ($\mathbf{X}$,$\mathbf{Y}$)-partitions are unique for a fixed $\mathbf{X}$ and $\mathbf{Y}$, we need one of two additional properties for a compressed SDD to be unique (i.e., canonical) given a vtree:

– *Normalization:* If an ($\mathbf{X}$,$\mathbf{Y}$)-partition $\beta$ is normalized for vtree node $v$, then the primes (subs) of $\beta$ must be normalized for the left (right) child of $v$—as opposed to a left (right) descendant of $v$.

– *Trimming:* The SDD contains no ($\mathbf{X}$,$\mathbf{Y}$)-partitions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$.

For a Boolean function, and a fixed vtree, there is a unique compressed, normalized SDD. There is also a unique compressed, trimmed SDD (Darwiche 2011). Thus, both representations are canonical, although trimmed SDDs tend to be smaller. One can trim an SDD by replacing ($\mathbf{X}$,$\mathbf{Y}$)-partitions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$ with $\alpha$. One can normalize an SDD by adding intermediate partitions of the same form. Since these translations are efficient, our theoretical results will apply to both canonical representations. In what follows, we will restrict our attention to compressed, trimmed SDDs and refer to them as *canonical* SDDs.

**SDDs and OBDDs** OBDDs correspond precisely to SDDs that are constructed using a special type of vtree, called a right-linear vtree (Darwiche 2011); see Figure 2. The left child of each inner node in these vtrees is a variable. With right-linear vtrees, compressed, trimmed SDDs correspond to reduced OBDDs, while compressed, normalized SDDs correspond to oblivious OBDDs (Xue, Choi, and Darwiche 2012) (reduced and oblivious OBDDs are also canonical). The size of an OBDD depends critically on the underlying variable order. Similarly, the size of an SDD depends critically on the vtree used (right-linear vtrees correspond to variable orders). Vtree search algorithms can sometimes find SDDs that are orders-of-magnitude more succinct than OBDDs found by searching for variable orders (Choi and Darwiche 2013). Such algorithms assume canonical SDDs, allowing one to search the space of SDDs by searching the space of vtrees instead.

**Queries** SDDs are a strict subset of deterministic, decomposable negation normal form (d-DNNF). They are actually a strict subset of structured d-DNNF and, hence, support the same polytime queries supported by structured d-DNNF (Pipatsrisawat and Darwiche 2008); see Table 1. We defer the reader to Darwiche and Marquis (2002) for a detailed description of the queries typically considered in knowledge compilation. This makes SDDs as powerful as OBDDs in terms of their support for certain queries (e.g., clausal entailment, model counting, and equivalence checking).

**Bottom-up Construction** SDDs are typically constructed in a bottom-up fashion. For example, to construct an SDD for the function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$, we first retrieve terminal SDDs for the literals $A$, $B$, $C$, and $D$. We then *conjoin* the terminal SDD for literal $A$ with the one for literal $B$, to obtain an SDD for the term $A \wedge B$. The process is repeated to obtain SDDs for the terms $B \wedge C$ and $C \wedge D$. The resulting SDDs are then *disjoined* to obtain an SDD for the whole function. These operations are not all efficient on structured d-DNNFs. However, SDDs satisfy stronger properties than structured d-DNNFs, allowing one, for example, to conjoin or disjoin two SDDs in polytime.

This bottom-up compilation is performed using the `Apply` function. Algorithm 1 outlines an `Apply` function that takes two SDDs $\alpha$ and $\beta$, and a binary Boolean operator $\circ$ (e.g., $\wedge$, $\vee$, xor), and returns the SDD for $\alpha \circ \beta$ (Darwiche 2011).[2] Line 13 optionally compresses each partition, in order to return a compressed SDD. Without compression, this algorithm has a time and space complexity of $O(nm)$, where $n$ and $m$ are the sizes of input SDDs. This comes at the expense of losing canonicity. Whether a polytime complexity can be attained under compression was an open question.

There are several implications of this question. For example, depending on the answer, one would know whether certain *transformations,* such as conditioning and existential

[2]This code assumes that the SDD is normalized. The `Apply` for trimmed SDDs is similar, although a bit more technically involved.

**Algorithm 1** $\texttt{Apply}(\alpha, \beta, \circ)$

```
if α and β are constants or literals then
    return α ∘ β                     // result is a constant or literal
else if Cache(α, β, ∘) ≠ nil then
    return Cache(α, β, ∘)   // has been computed before
else
    γ←{}
    for all elements (p_i, s_i) in α do
        for all elements (q_j, r_j) in β do
            p←Apply(p_i, q_j, ∧)
            if p is consistent then
                s←Apply(s_i, r_j, ∘)
                add element (p, s) to γ
    (optionally) γ ← Compress(γ)          // compression
                       // get unique decision node and return it
    return Cache(α, β, ∘)←UniqueD(γ)
```

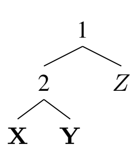
quantification, can be supported in polytime on canonical SDDs. Moreover, according to common wisdom, a negative answer may preclude bottom-up compilation from being feasible on canonical SDDs. We answer this question and explore its implications next.

## Complexity of Apply on Canonical SDDs

The size of a decision node is the number of its elements, and the size of an SDD is the sum of sizes attained by its decision nodes. We now show that compression, given a fixed vtree, may blow up the size of an SDD.

**Theorem 1.** *There exists a class of Boolean functions $f_m(X_1, \dots, X_m)$ and corresponding vtrees $T_m$ such that $f_m$ has an SDD of size $O(m^2)$ wrt vtree $T_m$, yet the canonical SDD of function $f_m$ wrt vtree $T_m$ has size $\Omega(2^m)$.*

The proof is constructive, identifying a class of functions $f_m$ with the given properties. The functions $f^a_m(\mathbf{X}, \mathbf{Y}, Z) = \bigvee_{i=1}^{m} \left( \bigwedge_{j=1}^{i-1} \neg Y_j \right) \wedge Y_i \wedge X_i$ have $2m+1$ variables. Of these, $Z$ is non-essential. Consider a vtree $T_m$ of the form




where the sub-vtrees over variables $\mathbf{X}$ and $\mathbf{Y}$ are arbitrary. We will now construct an uncompressed SDD for this function using vtree $T_m$ and whose size is $O(m^2)$. We will then show that the compressed SDD for this function and vtree has a size $\Omega(2^m)$.

The first step is to construct a partition of function $f^a_m$ that respects the root vtree node, that is, an ($\mathbf{XY}$,Z)-partition. Consider

$$\left\{ \begin{array}{l} (Y_1 \wedge X_1, \top), \\ (\neg Y_1 \wedge Y_2 \wedge X_2, \top), \\ \dots, \\ (\neg Y_1 \wedge \dots \wedge \neg Y_{m-1} \wedge Y_m \wedge X_m, \top), \\ (Y_1 \wedge \neg X_1, \bot), \\ (\neg Y_1 \wedge Y_2 \wedge \neg X_2, \bot), \\ \dots, \\ (\neg Y_1 \wedge \dots \wedge \neg Y_{m-1} \wedge Y_m \wedge \neg X_m, \bot), \\ (\neg Y_1 \wedge \dots \wedge \neg Y_m, \bot) \end{array} \right\},$$

which is equivalently written as

$$\bigcup_{i=1}^{m} \left\{ \left( \bigwedge_{j=1}^{i-1} \neg Y_j \wedge Y_i \wedge X_i, \top \right), \left( \bigwedge_{j=1}^{i-1} \neg Y_j \wedge Y_i \wedge \neg X_i, \bot \right) \right\} \cup \left\{ \left( \bigwedge_{j=1}^{m} \neg Y_j, \bot \right) \right\}.$$

The size of this partition is $2m + 1$, and hence linear in $m$. It is *uncompressed*, because there are $m$ elements that share sub $\top$ and $m+1$ elements that share sub $\bot$. The subs already respect the leaf vtree node labeled with variable $Z$.

In a second step, each prime above is written as a compressed ($\mathbf{X}$,$\mathbf{Y}$)-partition that respects the left child of the vtree root. Prime $\bigwedge_{j=1}^{i-1} \neg Y_j \wedge Y_i \wedge X_i$ becomes

$$\left\{ \left( X_i, \bigwedge_{j=1}^{i-1} \neg Y_j \wedge Y_i \right), (\neg X_i, \bot) \right\},$$

prime $\bigwedge_{j=1}^{i-1} \neg Y_j \wedge Y_i \wedge \neg X_i$ becomes

$$\left\{ \left( \neg X_i, \bigwedge_{j=1}^{i-1} \neg Y_j \wedge Y_i \right), ( X_i, \bot) \right\}$$

and prime $\bigwedge_{j=1}^{m} \neg Y_j$ becomes

$$\left\{ \left( \top, \bigwedge_{j=1}^{m} \neg Y_j \right) \right\}.$$

The sizes of these partitions are bounded by 2.

Finally, we need to represent the above primes as SDDs over variables $\mathbf{X}$ and the subs as SDDs over variables $\mathbf{Y}$. Since these primes and subs correspond to terms (i.e. conjunctions of literals), each has a compact SDD representation, independent of the chosen sub-vtree over variables $\mathbf{X}$ and $\mathbf{Y}$. For example, we can choose a right-linear vtree over variables $\mathbf{X}$, and similarly for variables $\mathbf{Y}$, leading to an OBDD representation of each prime and sub, with a size linear in $m$ for each OBDD. The full SDD for function $f^a_m$ will then have a size which is $O(m^2)$. Recall that this SDD is uncompressed as some of its decision nodes have elements with equal subs.

The compressed SDD for this function and vtree is unique. We now show that its size must be $\Omega(2^m)$. We first observe that the unique, compressed (**XY**,Z)-partition of function $f_m^a$ is

$$\left\{ \left( \bigvee_{i=1}^{m} \left( \bigwedge_{j=1}^{i-1} \neg Y_j \right) \wedge Y_i \wedge X_i, \top \right), \left( \left[ \bigvee_{i=1}^{m} \left( \bigwedge_{j=1}^{i-1} \neg Y_j \right) \wedge Y_i \wedge \neg X_i \right] \vee \left[ \bigwedge_{j=1}^{m} \neg Y_j \right], \bot \right) \right\}.$$

Its first prime is the function

$$f_m^b(\mathbf{X}, \mathbf{Y}) = \bigvee_{i=1}^{m} \left( \bigwedge_{j=1}^{i-1} \neg Y_j \right) \wedge Y_i \wedge X_i,$$

which we need to represent as an $(\mathbf{X},\mathbf{Y})$-partition to respect left child of the vtree root. However, Xue, Choi, and Darwiche (2012) proved the following.

**Lemma 2.** *The compressed* $(\mathbf{X},\mathbf{Y})$*-partition of* $f_m^b(\mathbf{X}, \mathbf{Y})$ *has* $2^m$ *elements.*

This becomes clear when looking at the function $f_m^b$ after instantiating the $\mathbf{X}$-variables. Each distinct $\mathbf{x}$ results in a unique subfunction $f_m^b(\mathbf{x}, \mathbf{Y})$, and all states $\mathbf{x}$ are mutually exclusive and exhaustive. Therefore,

$$\{(\mathbf{x}, f_m^b(\mathbf{x}, \mathbf{Y})) \mid \mathbf{x} \text{ instantiates } \mathbf{X}\}$$

is the unique, compressed $(\mathbf{X},\mathbf{Y})$-partition of function $f_m^b(\mathbf{X}, \mathbf{Y})$, and it has $2^m$ elements. Hence, the compressed SDD must have size $\Omega(2^m)$.

Theorem 1 has a number of implications, which are summarized in Table 2; see also Darwiche and Marquis (2002).

**Theorem 3.** *The results in Table 2 hold.*

First, combining two canonical SDDs (e.g., using the conjoin or disjoin operator) may lead to a canonical SDD whose size is exponential in the size of inputs. Hence, if we activate compression in Algorithm 1, the algorithm may take exponential time in the worst-case. Second, conditioning a canonical SDD on a literal may exponentially increase its size (assuming the result is also canonical). Third, forgetting a variable (i.e., existentially quantifying it) from a canonical SDD may exponentially increase its size (again, assuming that the result is also canonical). The proof of this theorem is in the full version of this paper.[3]

Note that these theorems consider the same vtree for both the compressed and uncompressed SDD. They do not pertain to the complexity of compression and `Apply` when the vtree is allowed to change. In practice, dynamic vtree search is performed in between conditioning and `Apply`, but not during (vtree search itself calls `Apply`). Therefore, the setting where the vtree does not change is more accurate to describe the practical complexity of these operations.

These results may seem discouraging. However, we argue next that, in practice, working with canonical SDDs is actually favorable despite the lack of polytime guarantees on these transformations.

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

| **Notation** | **Transformation** | SDD | Canonical SDD |
|---|---|---|---|
| **CD** | conditioning | $\surd$ | • |
| **FO** | forgetting | • | • |
| **SFO** | singleton forgetting | $\surd$ | • |
| $\wedge$**C** | conjunction | • | • |
| $\wedge$**BC** | bounded conjunction | $\surd$ | • |
| $\vee$**C** | disjunction | • | • |
| $\vee$**BC** | bounded disjunction | $\surd$ | • |
| $\neg$**C** | negation | $\surd$ | $\surd$ |

Table 2: Analysis of supported transformations, following Darwiche and Marquis (2002). $\surd$ means "satisfies"; • means "does not satisfy". Satisfaction means the existence of a polytime algorithm that implements the transformation.

Our proof of Theorem 1 critically depends on the ability of a vtree to split the variables into arbitrary sets $\mathbf{X}$ and $\mathbf{Y}$. In the full paper, we define a class of *bounded vtrees* where such splits are not possible. Moreover, we show that the subset of SDDs for such vtrees do support polytime `Apply` even under compression. Right-linear vtrees, which induce an OBDD, are a special case.

## Canonicity or a Polytime `Apply`?

One has two options when working with SDDs. The first option is to work with uncompressed SDDs, which are not canonical, but are supported by a polytime `Apply` function. The second option is to work with compressed SDDs, which are canonical but lose the advantage of a polytime `Apply` function. The classical reason for seeking canonicity is that it leads to a very efficient equivalence test, which takes constant time (both compressed and uncompressed SDDs support a polytime equivalence test, but the one known for uncompressed SDDs is not a constant time test). The classical reason for seeking a polytime `Apply` function is to enable bottom-up compilation, that is, compiling a knowledge base (e.g., CNF or DNF) into an SDD by repeated application of the `Apply` function to components of the knowledge base (e.g., clauses or terms). If our goal is efficient bottom-up compilation, one may expect that uncompressed SDDs provide a better alternative. However, our next empirical results suggest otherwise. Our goal in this section is to shed some light on this phenomena through some empirical evidence and then an explanation.

We used the SDD package provided by the Automated Reasoning Group at UCLA[4] in our experiments. The package works with compressed SDDs, but can be adjusted to work with uncompressed SDDs as long as dynamic vtree search is not invoked.[5] In our first experiment, we compiled CNFs from the LGSynth89 benchmarks into the following (all trimmed):[6]

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

[5] Dynamic vtree search requires compressed SDDs as canonicity reduces the search space over SDDs into one over vtrees.

[6] For a comparison with OBDD, see Choi and Darwiche (2013).

| Name | Variables | Clauses | SDD Size | | | Compilation Time | | |
|---|---|---|---|---|---|---|---|---|
| | | | Compressed SDDs+s | Compressed SDDs | Uncompressed SDDs | Compressed SDDs+s | Compressed SDDs | Uncompressed SDDs |
| C17 | 17 | 30 | 99 | 171 | 286 | 0.00 | 0.00 | 0.00 |
| majority | 14 | 35 | 123 | 193 | 384 | 0.00 | 0.00 | 0.00 |
| b1 | 21 | 50 | 166 | 250 | 514 | 0.00 | 0.00 | 0.00 |
| cm152a | 20 | 49 | 149 | 3,139 | 18,400 | 0.01 | 0.01 | 0.02 |
| cm82a | 25 | 62 | 225 | 363 | 683 | 0.01 | 0.00 | 0.00 |
| cm151a | 44 | 100 | 614 | 1,319 | 24,360 | 0.04 | 0.00 | 0.04 |
| cm42a | 48 | 110 | 394 | 823 | 276,437 | 0.03 | 0.00 | 0.10 |
| cm138a | 50 | 114 | 463 | 890 | 9,201,336 | 0.02 | 0.01 | 109.05 |
| decod | 41 | 122 | 471 | 810 | 1,212,302 | 0.04 | 0.01 | 1.40 |
| tcon | 65 | 136 | 596 | 1,327 | 618,947 | 0.05 | 0.00 | 0.33 |
| parity | 61 | 135 | 549 | 978 | 2,793 | 0.02 | 0.00 | 0.00 |
| cmb | 62 | 147 | 980 | 2,311 | 81,980 | 0.12 | 0.02 | 0.06 |
| cm163a | 68 | 157 | 886 | 1,793 | 21,202 | 0.06 | 0.00 | 0.02 |
| pcle | 66 | 156 | 785 | 1,366 | n/a | 0.07 | 0.01 | n/a |
| x2 | 62 | 166 | 785 | 1,757 | 12,150,626 | 0.08 | 0.02 | 19.87 |
| cm85a | 77 | 176 | 1,015 | 2,098 | 19,657 | 0.08 | 0.01 | 0.03 |
| cm162a | 73 | 173 | 907 | 2,050 | 153,228 | 0.08 | 0.01 | 0.16 |
| cm150a | 84 | 202 | 1,603 | 5,805 | 17,265,164 | 0.16 | 0.06 | 60.37 |
| pcler8 | 98 | 220 | 1,518 | 4,335 | 15,532,667 | 0.18 | 0.05 | 33.32 |
| cu | 94 | 235 | 1,466 | 5,789 | n/a | 0.19 | 0.10 | n/a |
| pm1 | 105 | 245 | 1,810 | 3,699 | n/a | 0.27 | 0.05 | n/a |
| mux | 73 | 240 | 1,825 | 6,517 | n/a | 0.19 | 0.09 | n/a |
| cc | 115 | 265 | 1,451 | 6,938 | n/a | 0.22 | 0.04 | n/a |
| unreg | 149 | 336 | 3,056 | 668,531 | n/a | 0.66 | 263.06 | n/a |
| ldd | 145 | 414 | 1,610 | 2,349 | n/a | 0.23 | 0.10 | n/a |
| count | 185 | 425 | 4,168 | 51,639 | n/a | 1.05 | 0.24 | n/a |
| comp | 197 | 475 | 2,212 | 4,500 | 205,105 | 0.24 | 0.01 | 0.22 |
| f51m | 108 | 511 | 3,290 | 6,049 | n/a | 0.52 | 0.32 | n/a |
| my_adder | 212 | 612 | 2,793 | 4,408 | 35,754 | 0.24 | 0.02 | 0.04 |
| cht | 205 | 650 | 4,832 | 13,311 | n/a | 1.24 | 0.36 | n/a |

Table 3: LGSynth89 benchmarks: SDD sizes and compilation times. Compressed SDDs+s refers to compressed SDDs with dynamic vtree search.

– Compressed SDDs respecting an arbitrary vtree. Dynamic vtree search is used to minimize the size of the SDD during compilation, starting from a balanced vtree.

– Compressed SDDs respecting a fixed balanced vtree.

– Uncompressed SDDs respecting a fixed balanced vtree.

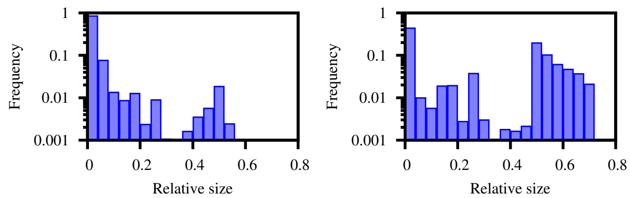
Table 3 shows the corresponding sizes and compilation times. According to these results, uncompressed SDDs end up several orders of magnitude larger than the compressed ones, with or without dynamic vtree search. For the harder problems, this translates to orders-of-magnitude increase in compilation times. Often, we cannot even compile the input without reduction (due to running out of 4GB of memory), even on relatively easy benchmarks. For the easiest benchmarks, dynamic vtree search is slower due to the overhead, but yields smaller compilations. The benefit of vtree search shows only in harder problems (e.g., "unreg").

Next, we consider the harder set of ISCAS89 benchmarks. Of the 17 ISCAS89 benchmarks that compile with compressed SDDs, only one (s27) could be compiled with uncompressed SDDs (others run out of memory). That benchmark has a compressed SDD+s size of 108, a compressed SDD size of 315, and an uncompressed SDD size of 4,551.

These experiments clearly show the advantage of compressed SDDs over uncompressed ones, even though the latter supports a polytime `Apply` function while the former does not. This begs an explanation and we provide one next that we back up by additional experimental results.

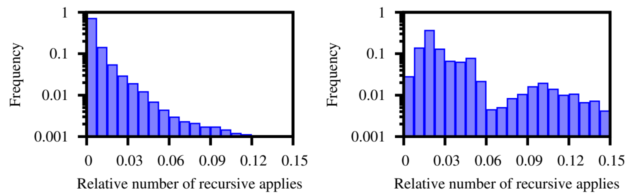
The benefit of compressed SDDs is canonicity, which plays a critical role in the performance of the `Apply` function. Consider in particular Line 4 of Algorithm 1. The test $\texttt{Cache}(\alpha, \beta, \circ) \neq \texttt{nil}$ checks whether SDDs $\alpha$ and $\beta$ have been previously combined using the Boolean operator $\circ$. Without canonicity, it is possible that we would have combined some $\alpha'$ and $\beta'$ using $\circ$, where SDD $\alpha'$ is equivalent to, but distinct from SDD $\alpha$ (and similarly for $\beta'$ and $\beta$). In this case, the cache test would fail, causing `Apply` to recompute the same result again. Worse, the SDD returned by $\texttt{Apply}(\alpha, \beta, \circ)$ may be distinct from the SDD returned by $\texttt{Apply}(\alpha', \beta', \circ)$, even though the two SDDs are equivalent. This redundancy also happens when $\alpha$ is not equivalent to $\alpha'$ (and similarly for $\beta$ and $\beta'$), $\alpha \circ \beta$ is equivalent to $\alpha' \circ \beta'$, but the result returned by $\texttt{Apply}(\alpha, \beta, \circ)$ is distinct from the one returned by $\texttt{Apply}(\alpha', \beta', \circ)$.

Two observations are due here. First, this redundancy is still under control when calling `Apply` only once: `Apply` runs in $O(nm)$ time, where $n$ and $m$ are the sizes of input SDDs. However, this redundancy becomes problematic




(a) Compressed SDDs (b) Uncompressed SDDs

Figure 3: Relative SDD size.




(a) Compressed SDDs (b) Uncompressed SDDs

Figure 4: Relative number of recursive `Apply` calls.

when calling `Apply` multiple times (as in bottom-up compilation), in which case quadratic performance is no longer as attractive. For example, if we use `Apply` to combine $k$ SDDs of size $n$ each, all we can say is that the output will be of size $O(n^k)$. The second observation is that the previous redundancy will not occur when working with compressed SDDs due to canonicity: Two SDDs are equivalent iff they are represented by the same structure in memory.[7]

This analysis points to the following conclusion: While `Apply` has a quadratic complexity on uncompressed SDDs, it may have a worse average complexity than `Apply` on compressed SDDs. Our next experiment is indeed directed towards this hypothesis.

For all benchmarks in Table 3 that can be compiled without vtree search, we intercept all non-trivial calls to `Apply` (when $|\alpha| \cdot |\beta| > 500$) and report the size of the output $|\alpha \circ \beta|$ divided by $|\alpha| \cdot |\beta|$. For uncompressed SDDs, we know that $|\alpha \circ \beta| = O(|\alpha| \cdot |\beta|)$ and that these ratios are therefore bounded above by some constant. For compressed SDDs, however, Theorem 3 states that there exists no constant bound.

Figure 3 shows the distribution of these ratios for the two methods (note the log scale). The number of function calls is 67,809 for compressed SDDs, vs. 1,626,591 for uncompressed ones. The average ratio is 0.027 for compressed, vs. 0.101 for uncompressed. Contrasting the theoretical bounds, compressed `Apply` incurs much smaller blowups than uncompressed `Apply`. This is most clear for ratios in the range $[0.48, 0.56]$, covering 30% of the uncompressed, but only 2% of the compressed calls.

The results are similar when looking at runtime for individual `Apply` calls, which we measure by the number of recursive `Apply` calls $r$. Figure 4 reports these, again relative to $|\alpha| \cdot |\beta|$. The ratio $r/(|\alpha| \cdot |\beta|)$ is on average 0.013 for compressed SDDs, vs. 0.034 for uncompressed ones. These results corroborate our earlier analysis, suggesting that canonicity is quite important for the performance of bottom-up compilers as they make repeated calls to the `Apply` function. In fact, this can be more important than a polytime `Apply`, perhaps contrary to common wisdom which seems to emphasize the importance of polytime `Apply` in effective bottom-up compilation (e.g., Pipatsrisawat and Darwiche (2008)).

[7]This is due to the technique of *unique nodes* from OBDDs; see `UniqueD` in Algorithm 1.

## Conclusions

We have shown that the `Apply` function on compressed SDDs can take exponential time in the worst case, resolving a question that has been open since SDDs were first introduced. We have also pursued some of the theoretical and practical implications of this result. On the theoretical side, we showed that it implies an exponential complexity for various transformations, such as conditioning and existential quantification. On the practical side, we argued empirically that working with compressed SDDs remains favorable, despite the polytime complexity of the `Apply` function on uncompressed SDDs. The canonicity of compressed SDDs, we argued, is more valuable for bottom-up compilation than a polytime `Apply` due to its role in facilitating caching and dynamic vtree search. Our findings appear contrary to some of the common wisdom on the relationship between bottom-up compilation, canonicity and the complexity of the `Apply` function.

## Acknowledgments

We thank Arthur Choi, Doga Kisa, Umut Oztok, and Jessa Bekker for helpful suggestions. This work was supported by ONR grant #N00014-12-1-0423, NSF grants #IIS-1118122 and #IIS-0916161, and the Research Foundation-Flanders (FWO-Vlaanderen). GVdB is also at KU Leuven, Belgium.

## References

Barrett, A. 2005. Model compilation for real-time planning and diagnosis with feedback. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 1195–1200.

Beame, P.; Li, J.; Roy, S.; and Suciu, D. 2013. Lower bounds for exact model counting and applications in probabilistic databases. In *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI)*, 52–61.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35:677–691.

Cadoli, M., and Donini, F. M. 1997. A survey on knowledge compilation. *AI Communications* 10:137–150.

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

Chavira, M.; Darwiche, A.; and Jaeger, M. 2006. Compiling relational bayesian networks for exact inference. *International Journal of Approximate Reasoning* 42(1):4–20.

Choi, A., and Darwiche, A. 2013. Dynamic minimization of sentential decision diagrams. In *Proceedings of AAAI*.

Choi, A.; Kisa, D.; and Darwiche, A. 2013. Compiling probabilistic graphical models using sentential decision diagrams. In *Proceedings of ECSQARU*.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *JAIR* 17:229–264.

Darwiche, A. 2001. On the tractability of counting theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics* 11(1-2):11–34.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of IJCAI*, 819–826.

Darwiche, A. 2014. Tractable knowledge representation formalisms. In Lucas Bordeaux, Youssef Hamadi, P. K., ed., *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press. chapter 5.

del Val, A. 1994. Tractable databases: How to make propositional unit resolution complete through compilation. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 551–561. Morgan Kaufmann Publishers, Inc., San Mateo, California.

Elliott, P., and Williams, B. 2006. DNNF-based belief state estimation. In *Proceedings of AAAI*.

Fierens, D.; Van den Broeck, G.; Thon, I.; Gutmann, B.; and Raedt, L. D. 2011. Inference in probabilistic logic programs using weighted CNF's. In *Proceedings of UAI*, 211–220.

Fierens, D.; Van den Broeck, G.; Renkens, J.; Shterionov, D.; Gutmann, B.; Thon, I.; Janssens, G.; and De Raedt, L. 2013. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*.

Gens, R., and Domingos, P. 2013. Learning the structure of sum-product networks. In *ICML*, 873–880.

Huang, J., and Darwiche, A. 2005. On compiling system models for faster and more scalable diagnosis. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, 300–306.

Huang, J. 2006. Combining knowledge compilation and search for conformant probabilistic planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-06)*, 253262.

Jha, A., and Suciu, D. 2011. Knowledge compilation meets database theory: Compiling queries to decision diagrams. In *Proceedings of the 14th International Conference on Database Theory (ICDT)*, 162–173.

Kisa, D.; Van den Broeck, G.; Choi, A.; and Darwiche, A. 2014a. Probabilistic sentential decision diagrams. In *Proceedings of KR*.

Kisa, D.; Van den Broeck, G.; Choi, A.; and Darwiche, A. 2014b. Probabilistic sentential decision diagrams: Learning with massive logical constraints. In *ICML Workshop on Learning Tractable Probabilistic Models (LTPM), Beijing, China, June 2014*.

Lowd, D., and Rooshenas, A. 2013. Learning Markov networks with arithmetic circuits. In *AISTATS*, 406–414.

Marquis, P. 1995. Knowledge compilation using theory prime implicates. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 837–843. Morgan Kaufmann Publishers, Inc., San Mateo, California.

Oztok, U., and Darwiche, A. 2014. On compiling cnf into decision-dnnf. In *Proceedings of CP*.

Palacios, H.; Bonet, B.; Darwiche, A.; and Geffner, H. 2005. Pruning conformant plans by counting models on compiled d-DNNF representations. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling*, 141–150.

Pipatsrisawat, K., and Darwiche, A. 2008. New compilation languages based on structured decomposability. In *Proceedings of AAAI*, 517–522.

Razgon, I. 2013. On OBDDs for CNFs of bounded treewidth. *CoRR* abs/1308.3829.

Rekatsinas, T.; Deshpande, A.; and Getoor, L. 2012. Local structure and determinism in probabilistic databases. In *ACM SIGMOD Conference*.

Renkens, J.; Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2014. Explanation-based approximate weighted model counting for probabilistic logics. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*.

Selman, B., and Kautz, H. 1991. Knowledge compilation using horn approximation. In *Proceedings of AAAI*. AAAI.

Siddiqi, S., and Huang, J. 2007. Hierarchical diagnosis of multiple faults. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*.

Suciu, D.; Olteanu, D.; Ré, C.; and Koch, C. 2011. Probabilistic databases. *Synthesis Lectures on Data Management* 3(2):1–180.

Van den Broeck, G.; Taghipour, N.; Meert, W.; Davis, J.; and De Raedt, L. 2011. Lifted Probabilistic Inference by First-Order Knowledge Compilation. In *Proceedings of IJCAI*, 2178–2185.

Van den Broeck, G. 2011. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In *Advances in Neural Information Processing Systems 24 (NIPS),*, 1386–1394.

Van den Broeck, G. 2013. *Lifted Inference and Learning in Statistical Relational Models*. Ph.D. Dissertation, KU Leuven.

Vlasselaer, J.; Renkens, J.; Van den Broeck, G.; and De Raedt, L. 2014. Compiling probabilistic logic programs into sentential decision diagrams. In *Workshop on Probabilistic Logic Programming (PLP)*.

Xue, Y.; Choi, A.; and Darwiche, A. 2012. Basing decisions on sentences in decision diagrams. In *Proceedings of the 26th Conference on Artificial Intelligence (AAAI)*, 842–849.