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 \textit{Apply} function). While such a polytime \textit{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 \textit{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 \textit{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 languages 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 Rooslenas 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 \textit{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 \textit{Apply} function, which combines two SDDs using Boolean operators.\footnote{\textit{Apply} originated in the OBDD literature (Bryant 1986).} 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 (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., X). A literal is a variable or its negation. A Boolean function f(X) maps each instantiation x of variables X into ⊤ (true) or ⊥ (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, X and Y. We can always decompose the function f as

\[ f = \left( p_1(X) \land s_1(Y) \right) \lor \cdots \lor \left( p_n(X) \land s_n(Y) \right), \]

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

\[ (A \land B \land \top) \lor (\neg A \land B \land C) \lor (\neg B \land C \land D). \]  

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

\[
\{(A \land B, \top), (\neg A \land B, C), (A \land \neg B, D \land C), (\neg A \land \neg B, D \land C)\}. \]  

(2)

An uncompressed (X,Y)-partition can be compressed by merging all elements \((p_1,s), \ldots, (p_n,s)\) that share the same sub into one element \((p_1 \lor \cdots \lor p_n, s)\). Compressing (2) combines the two last elements into \([A \land \neg B] \lor [\neg A \land \neg B], D \land C\) = \((\neg B, D \land C)\), 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 (X,Y)-partitions. To build an SDD, we need to determine which X and 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 appearing in the left subtree (X) and those appearing in the right subtree (Y). This implies an (X,Y)-partition β of the
Two Forms of Canonicity

Even though compressed (X,Y)-partitions are unique for a fixed X and Y, we need one of two additional properties for a compressed SDD to be unique (i.e., canonical) given a vtree:

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

- **Trimming**: The SDD contains no (X,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 (X,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., clause 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 \(f = (A \land B) \lor (B \land C) \lor (C \land D)\), 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 \(A \land B\). The process is repeated to obtain SDDs for the terms \(B \land C\) and \(C \land D\). 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., \(\land\), \(\lor\), xor), and returns the SDD for \(\alpha \circ \beta\) (Darwiche 2011). 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 quantification, can be supported in polytime on canonical SDDs. Moreover, according to common wisdom, a neg-
Theorem 1. There exists a class of Boolean functions $m$ that may blow up the size of an SDD. We now show that compression, given a fixed vtree, the size of an SDD is the sum of sizes attained by its decision nodes. The size of a decision node is the number of its elements, and the sub-vtrees over variables $X_i$. 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 $f_m$ that respects the root vtree node, that is, an $(X, Y, Z)$-partition.

![Diagram of a partition tree]

The proof is constructive, identifying a class of functions $f_m$ with the given properties. The functions $f_m(X, Y, Z) = \bigvee_{i=1}^{m} (X_{i-1} \land Y_i \land Z_i)$ have $2m+1$ variables. Of these, $Z$ is non-essential. Consider a vtree $T_m$ of the form

Consider

$$
\begin{align*}
&\{ (Y_1 \land X_1, T), \\
&(-Y_1 \land Y_2 \land X_2, T), \\
&\quad \ldots, \\
&(-Y_1 \land \cdots \land \neg Y_{m-1} \land Y_m \land X_m, T), \\
&(Y_1 \land \neg X_1, \bot), \\
&(-Y_1 \land Y_2 \land \neg X_2, \bot), \\
&\quad \ldots, \\
&(-Y_1 \land \cdots \land \neg Y_{m-1} \land Y_m \land \neg X_m, \bot), \\
&(-Y_1 \land \cdots \land \neg Y_m, \bot) \\
\end{align*}
$$

which is equivalently written as

$$
\bigcup_{i=1}^{m} \left\{ \begin{array}{l}
(Y_1 \land X_1, T), \\
(-Y_1 \land Y_2 \land X_2, T), \\
\quad \ldots, \\
(-Y_1 \land \cdots \land \neg Y_{m-1} \land Y_m \land X_m, T), \\
(Y_1 \land \neg X_1, \bot), \\
(-Y_1 \land Y_2 \land \neg X_2, \bot), \\
\quad \ldots, \\
(-Y_1 \land \cdots \land \neg Y_{m-1} \land Y_m \land \neg X_m, \bot), \\
(-Y_1 \land \cdots \land \neg Y_m, \bot)
\end{array} \right\}.
$$

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 sub already respect the leaf vtree node labeled with variable $Z$.

In a second step, each prime above is written as a compressed $(X, Y)$-partition that respects the left child of the vtree root. Prime $X_{i-1} \land Y_i \land X_i$, leading to

$$
\left( X_{i-1} \land \begin{array}{l}
Y_i, \\
\neg Y_i, \\
\neg X_i, \bot
\end{array} \right),
$$

prime $X_{i-1} \land Y_i \land \neg X_i$, becomes

$$
\left( X_{i-1} \land \begin{array}{l}
Y_i, \\
\neg Y_i, \\
\neg X_i, \bot
\end{array} \right),
$$

and prime $X_{i-1} \land Y_i$, becomes

$$
\left( X_{i-1} \land \begin{array}{l}
Y_i, \\
\neg Y_i, \\
\neg X_i, \bot
\end{array} \right).
$$

The sizes of these partitions are bounded by 2.

Finally, we need to represent the above primes as SDDs over variables $X$ and the subs as SDDs over variables $Y$. 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 $X$ and $Y$. For example, we can choose a right-linear vtree over variables $X$, and similarly for variables $Y$, leading to an OBDD representation of each prime and sub, with a size linear in $m$ for each OBDD. The full SDD for function $f_m$ 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
Our proof of Theorem 1 critically depends on the ability to split the variables into arbitrary sets \( X \) and \( Y \).

In the Appendix, 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\(^3\) 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.\(^4\) In our first experiment, we compiled CNFs from the LGSynth89 benchmarks into the following (all trimmed):\(^5\)

- Compressed SDDs respecting an arbitrary vtree. Dynamic

---

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

4Dynamic vtree search requires compressed SDDs as canonicity reduces the search space over SDDs into one over vtrees.

5For a comparison with OBDD, see Choi and Darwiche (2013).
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. 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 \textit{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 \textit{Apply} function. Consider in particular Line 4 of Algorithm 1. The test $\text{Cache}(\alpha, \beta, \phi) \neq \text{nil}$ checks whether SDDs $\alpha$ and $\beta$ have been previously combined using the Boolean operator $\phi$. Without canonicity, it is possible that we would have combined some $\alpha'$ and $\beta'$ using $\phi$, 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 \textit{Apply} to recompute the same result again. Worse, the SDD returned by $\text{Apply}(\alpha, \beta, \phi)$ may be distinct from the SDD returned by $\text{Apply}(\alpha', \beta', \phi)$, 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 $\text{Apply}(\alpha, \beta, \phi)$ is distinct from the one returned by $\text{Apply}(\alpha', \beta', \phi)$.

Two observations are due here. First, this redundancy is still under control when calling $\text{Apply}$ only once: $\text{Apply}$ runs in $O(nm)$ time, where $n$ and $m$ are the sizes of input SDDs. However, this redundancy becomes problematic when calling $\text{Apply}$ multiple times (as in bottom-up com-
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.

**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 Oztop, 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**


Complexity of Transformations

We now prove Theorem 3, stating that the results in Table 2 hold. We will first show the results for uncompressed SDDs, and then prove the results for compressed SDDs.

For uncompressed SDDs, Darwiche (2011) showed support for $\land BC$, $\lor BC$, and $\neg C$ (see Algorithm 1). We show support for uncompressed CD next.

**Theorem 4.** We can condition an uncompressed SDD on a literal $\ell$ in polynomial time by replacing $\ell$ by $T$ and $\neg \ell$ by $\bot$. When removing all elements whose prime is equivalent to $\bot$, the resulting sentence is an uncompressed SDD.

**Proof.** It is clear that the procedure transforms $\alpha$ into a sentence that is logically equivalent to $\alpha | \ell$: the first step directly follows the definition of conditioning, and the second step maintains logical equivalence. We need to show next that the result is syntactically an SDD, by showing that the primes in its partitions are consistent, exhaustive, and mutually exclusive. The second step enforces consistency of the primes. Moreover, if the primes are exhaustive, that is, $p_1 \lor \cdots \lor p_n \equiv T$, then $p_1 | (\lor \cdots \lor p_n) | (\lor \ell) \equiv (p_1 | (\lor \cdots \lor p_n)) | (\lor T) \equiv T$, and the result of conditioning is also exhaustive. Finally, when $p_1$ and $p_2$ are mutually exclusive, that is, $p_1 \land p_2 \equiv \bot$, then $p_1 | (\lor p_2) | (\lor \ell) \equiv (p_1 | (\lor p_2)) | (\lor \bot) \equiv \bot | (\lor \ell) \equiv \bot$, and the conditioned primes are also mutually exclusive. $\square$

Support for SFO follows from the support for CD and $\lor BC$. The negative results for FO, $\land C$ and $\lor C$ follow from identical OBDD results in Darwiche and Marquis (2002), and the fact that OBDDs are a special case of SDDs.

For compressed SDDs, the negative FO, $\land C$ and $\lor C$ results also follow from OBDD results. It is also clear from Algorithm 1 that negating a compressed SDD $\alpha$ by computing $\text{Apply}(\alpha, T, \lor C)$ does not cause any subs to become equivalent. Therefore, negating a compressed SDD leads to a compressed result, and compressed SDDs support $\neg C$. The remaining results in Table 2, on CD, SFO, $\land BC$ and $\lor BC$ are discussed next.

**Theorem 5.** There is a class of Boolean functions $f(X_1, \ldots, X_n)$ and vtrees $T_n$ for which the compressed SDD has size $O(n)$, yet the compressed SDD for the function $f(X_1, \ldots, X_n)$ has size $\Omega(2^n)$ for some literal $\ell$.

**Proof.** Consider the function

$$f_n(X, Y, Z, W) = \bigwedge_{i=1}^{n-1} \bigvee_{j=1}^{n-2} \neg Y_j \land Y_i \land \left( (X_i \land (W \lor Z_i)) \lor (\neg X_i \land Z_i) \right)$$

and the vtree depicted in Figure 5a.

The root of the reduce SDD for $f_n$ is an $(XY, ZW)$-partition that respects vtree node 1, consisting of elements

$$\bigcup_{i=1}^{n} \left\{ \left( \bigwedge_{j=1}^{i-1} \neg Y_j \land Y_i \land X_i, \ W \lor Z_i \right) \right\},
\left\{ \left( \bigwedge_{j=1}^{i-1} \neg Y_j \land Y_i \land \neg X_i, \ Z_i \right) \right\},$$

with $Y_1 = \bot$. The size of this partition is linear in $n$. It has the same primes as the uncompressed SDD for $f_n$ in the proof of Theorem 1, only now the partition is compressed, as all subs are distinct.

The primes of this partition can be represented as compressed $(X, Y)$-partitions, exactly as in the second step for Theorem 1. The remaining primes and subs (over $X$, over $Y$, over $Z$, and over $W \cup \{W\}$) are all simple conjunctions or disjunctions of literals that have a linear compressed SDD representation for any vtree.

We now have obtained a polysize SDD. However, when we condition this SDD on the literal $W$, all $n$ subs of the form $W \lor Z_i$ become equivalent to $T$. Their elements need to be compressed into the single element $(V_{i=1}^{n-1} \neg Y_j \land Y_i \land X_i, T)$. Its prime is again the function $f_n(X, Y)$, which has no polysize SDD wrt vtree node 2 with exponential size. $\square$

**Theorem 6.** There is a class of Boolean functions $f(X_1, \ldots, X_n)$ and vtrees $T_n$ for which the compressed SDD has size $O(n)$, yet the compressed SDD for the Boolean function $f(X_1, \ldots, X_n) \land \ell$ has size $\Omega(2^n)$ for some literal $\ell$.

**Proof.** Consider again the compressed SDD for $f_n$ that was constructed in the proof of Theorem 5 for the vtree in Figure 5a. Conjoining this SDD with the SDD for literal $W$ makes the $n$ subs of the form $W \lor Z_i$ equivalent to $W \land (W \lor Z_i) = W$. Compressing these creates the element $(V_{i=1}^{n-1} \neg Y_j \land Y_i \land X_i, W)$, whose prime is again $f_n(X, Y)$, which has no polysize SDD for vtree node 2. $\square$

This already proves that $\text{Apply}$ is worst-case exponential when performing conjunctions on compressed SDDs. Given that compressed SDDs support polysize negation, this result generalizes to any binary Boolean operator $\circ$ that is functionally complete together with negation. Wernick (1942). Support for these operators would allow us to do polysize conjunction by combining $\circ$ and negation. One such operator is disjunction, which is therefore also is worst-case exponential.

Suppose now that we can perform singleton forgetting in polysize, which is defined as $\exists L. \alpha = (\alpha[L] \lor (\alpha[\neg L])$. Then given any two compressed SDDs $\beta$ and $\gamma$ respecting the same vtree $T$, we can obtain $\beta \lor \gamma$ in polysize as follows. Add a new variable $L$ to vtree $T$, as depicted in Figure 5b.
The compressed SDD $\alpha$ for the function $(L \land \beta) \lor (\neg L \land \gamma)$ has the root partition $\{(L, \beta), (\neg L, \gamma)\}$. Forgetting $L$ from $\alpha$ results in the compressed SDD for $\beta \lor \gamma$. Hence, if single forgetting can be done in polytime, then bounded disjunction can also be done in polytime. Since the latter is impossible, the former is also impossible.

**Bounded Vtrees**

A bounded vtree is one for which the number of variables in any left subtree is bounded. This includes right-linear vtrees which give rise to OBDDs, since each left subtree contains a single variable in this case. We now have the following.

**Theorem 7.** The time and space complexity of Algorithm 1, with compression, is in $O(nm)$, where $n$ and $m$ are the sizes of its inputs, assuming that the input SDDs are compressed and respect a bounded vtree.

The compression step of Algorithm 1 identifies elements $(p_i, s)$ and $(p_j, s)$ that share sub $s$, and merges these elements into the element $(p_i \lor p_j, s)$ by calling `Apply` recursively to disjoin primes $p_i$ and $p_j$. Since the vtree is bounded, primes $p_i$ and $p_j$ must be over a bounded number of variables. Hence, the complexity of compression is bounded, leading `Apply` to have the same complexity with or without compression.

For example, in right-linear vtrees (i.e., OBDDs), primes are literals over a single variable. Hence, all decision nodes are of the form $\{(X, \alpha), (\neg X, \beta)\}$. On these, compression occurs when $\alpha = \beta$, resulting in the partition $\{(X \lor \neg X, \alpha)\} = \{(\top, \alpha)\}$, which trimming replaces by $\alpha$. This corresponds to the OBDD reduction rule that eliminates decision nodes with isomorphic children (Bryant 1986).

Xue, Choi, and Darwiche (2012) showed a class of Boolean functions whose OBDDs have exponential size with respect to certain orders (right-linear vtrees), but which have SDDs of linear size when the vtrees are not right-linear (but have the same left-to-right variable order). The used vtrees, however, were not bounded. It would be interesting to see if a similar result can be obtained for bounded vtrees.