# SDD: A New Canonical Representation of Propositional Knowledge Bases

**Adnan Darwiche**
Computer Science Department
UCLA
darwiche@cs.ucla.edu

## Abstract

We identify a new representation of propositional knowledge bases, the *Sentential Decision Diagram (SDD)*, which is interesting for a number of reasons. First, it is canonical in the presence of additional properties that resemble reduction rules of OBDDs. Second, SDDs can be combined using any Boolean operator in polytime. Third, CNFs with $n$ variables and treewidth $w$ have canonical SDDs of size $O(n2^w)$, which is tighter than the bound on OBDDs based on pathwidth. Finally, every OBDD is an SDD. Hence, working with the latter does not preclude the former.

## 1 Introduction

Many areas of computer science have shown a great interest in tractable and canonical representations of propositional knowledge bases (aka, Boolean functions). The Ordered Binary Decision Diagram (OBDD) is one example representation that received much attention and proved quite influential in a variety of areas [Bryant, 1986]. Reduced OBDDs are canonical and can be combined using any Boolean operator in polytime, making them an indispensable tool in many research areas such as diagnosis, verification, system design, and planning.

Within AI, the study of tractable representations has become more systematic since [Darwiche and Marquis, 2002], which showed that many known representations of propositional knowledge bases are subsets of Negation Normal Form (NNF), and correspond to imposing specific properties on NNF. The most fundamental of these properties are *decomposability* and *determinism,* which lead to d-DNNF representations that proved influential in probabilistic reasoning applications (e.g., [Chavira and Darwiche, 2008]). Interestingly enough, OBDDs satisfy these two properties, but they also satisfy additional properties, making them a strict subset of d-DNNFs.

A fundamental concept highlighted by [Darwiche and Marquis, 2002] is the tension between the succinctness and tractability of a given representation. Here, succinctness refers to the size of a knowledge base once it is *compiled* into the representation, while tractability refers to the set of operations that have polytime implementations on the given representation. As one would expect, the more tractable a representation is, the less succinct it tends to be. Hence, in principle, one should first identify the operations needed by a certain application, and then use the most succinct representation that provides polytime support for those operations.

The practice can be a bit different, however, for two reasons. First, some operations may not be strictly needed for certain applications, yet they can significantly facilitate the development of software systems for those applications. For example, the ability to combine compilations efficiently using Boolean operators, such as conjoin, disjoin and negate, has proven to be quite valuable in system development. Moreover, the canonicity of a representation can be a critical factor when adopting a representation as it facilitates the search for *optimal compilations.* For example, the canonicity of OBDDs allows one to reduce the process of searching for an optimal compilation (OBDD) into one of identifying an optimal variable order (a reduced OBDD is completely determined by the used variable order). Hence, in practice, one may adopt a less succinct representation than one can afford, motivated by canonicity and the ability to perform Boolean combinations in polytime. This compromise is quite characteristic of many applications in which OBDDs are used (despite the availability of more succinct representations).

This paper is motivated by these practical considerations, where the goal is to identify a tractable representation that lies between d-DNNF and OBDD in terms of generality, yet maintains the properties that make OBDDs quite attractive in practical system development. In particular, we propose a new representation of propositional knowledge bases, called SDD, which results from imposing two properties on NNF that have been recently introduced: structured decomposability and strong determinism. These properties are stronger than decomposability and determinism which characterize d-DNNF, making SDDs a strict subset of d-DNNFs. On the other hand, these properties are weaker than the ones satisfied by OBDDs, making SDDs a strict superset of OBDDs. Despite their generality, SDDs maintain the key properties of OBDDs, including canonicity and a polytime support for Boolean combinations. They also come with a tighter bound on their size in terms of treewidth.

The SDD is inspired by two recent discoveries. The first is *structured decomposability* [Pipatsrisawat and Darwiche, 2008], which is based on the notion of *vtrees* that generalize variable orders. The second is *strongly deterministic decompositions* [Pipatsrisawat and Darwiche, 2010a], which generalize the Shannon decomposition on which OBDDs are based. Combining vtrees and this class of decompositions leads to the new representation. The foundations of SDDs are presented in Section 2; their syntax and semantics in Section 3; their canonicity in Section 4; their Boolean combination in Section 5; their relation to OBDDs in Section 6; and their upper bound based on treewidth in Section 7. Preliminary empirical results and a discussion of related work appear in Section 8. We will provide proofs or proof sketches for many of our results in this paper, leaving some to the full technical report.

## 2 Strongly deterministic decompositions

We start with some technical and notational preliminaries. Upper case letters (e.g., $X$) will be used to denote variables and lower case letters to denote their instantiations (e.g., $x$). Bold upper case letters (e.g., $\mathbf{X}$) will be used to denote sets of variables and bold lower case letters to denote their instantiations (e.g., $\mathbf{x}$).

A *Boolean function* $f$ over variables $\mathbf{Z}$ maps each instantiation $\mathbf{z}$ to 0 or 1. The *conditioning* of $f$ on instantiation $\mathbf{x}$, written $f|\mathbf{x}$, is a *subfunction* that results from setting variables $\mathbf{X}$ to their values in $\mathbf{x}$. A function $f$ *essentially depends* on variable $X$ iff $f|X \neq f|\neg X$. We write $f(\mathbf{Z})$ to mean that $f$ can only essentially depend on variables in $\mathbf{Z}$. A *trivial* function maps all its inputs to 0 (denoted *false*) or maps them all to 1 (denoted *true*).

Consider a Boolean function $f(\mathbf{X}, \mathbf{Y})$ with non-overlapping variables $\mathbf{X}$ and $\mathbf{Y}$. If $f = (p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})) \vee \ldots \vee (p_n(\mathbf{X}) \wedge s_n(\mathbf{Y}))$, then $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ is called an $(\mathbf{X}, \mathbf{Y})$-decomposition of function $f$ as it allows one to express $f$ in terms of functions on $\mathbf{X}$ and on $\mathbf{Y}$ only [Pipatsrisawat and Darwiche, 2010a]. If $p_i \wedge p_j = \mathit{false}$ for $i \neq j$, the decomposition is said to be *strongly deterministic* on $\mathbf{X}$ [Pipatsrisawat and Darwiche, 2010a]. In this case, we call each ordered pair $(p_i, s_i)$ an *element* of the decomposition, each $p_i$ a *prime* and each $s_i$ a *sub*. The decomposition size is the number of its elements.

SDDs utilize a more structured decomposition type.

**Definition 1** *Let $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$ be an $(\mathbf{X}, \mathbf{Y})$-decomposition of function $f$ that is strongly deterministic on $\mathbf{X}$. Then $\alpha$ is called an $\underline{\mathbf{X}\text{-partition}}$ of $f$ iff its primes form a partition (each prime is consistent, every pair of distinct primes are mutually exclusive, and the disjunction of all primes is valid). Moreover, $\alpha$ is $\underline{\text{compressed}}$ iff its subs are distinct ($s_i \neq s_j$ for $i \neq j$).*

Consider decompositions $\{(A, B), (\neg A, \mathit{false})\}$ and $\{(A, B)\}$ of $f = A \wedge B$. The first is an $A$-partition. The second is not. Decompositions $\{(\mathit{true}, B)\}$ and $\{(A, B), (\neg A, B)\}$ are both $A$-partitions of $f = B$. The first is compressed. The second is not since its subs are not distinct. A decomposition is compressed by repeated replacement of elements $(p, s)$ and $(q, s)$ with $(p \vee q, s)$.

Following are useful observations about $\mathbf{X}$-partitions. First, *false* can never be prime by definition. Second, if *true* is prime, then it is the only prime. Third, primes determine subs. Hence, if two $\mathbf{X}$-partitions of a function $f$ are distinct, their primes must form different partitions.

Ordered Binary Decision Diagrams (OBDDs) are based on the Shannon decomposition of a function $f$, $\{(X, f|X), (\neg X, f|\neg X)\}$, which is an $X$-partition of $f$. Here, decisions are binary since they are based on the value of literal primes ($X$ or $\neg X$). On the other hand, *Sentential Decision Diagrams (SDDs)* are based on $\mathbf{X}$-partitions, where $\mathbf{X}$ is a set of variables instead of being a single variable. As a result, decisions are not binary as they are based on the value of sentential primes.

The following property of partitioned decompositions is responsible for many properties of SDDs.

**Theorem 2** *Let $\circ$ be a Boolean operator and let $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ and $\{(q_1, r_1), \ldots, (q_m, r_m)\}$ be $\mathbf{X}$-partitions of $f(\mathbf{X}, \mathbf{Y})$ and $g(\mathbf{X}, \mathbf{Y})$. Then $\{(p_i \wedge q_j, s_i \circ r_j) \mid p_i \wedge q_j \neq \mathit{false}\}$ is an $\mathbf{X}$-partition of $f \circ g$.*

**Proof** Since $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ are partitions, then $p_i \wedge q_j$ is also a partition for $i = 1, \ldots, n$, $j = 1, \ldots, m$ and $p_i \wedge p_j \neq \mathit{false}$. Hence, the given decomposition is an $\mathbf{X}$-partition of some function. Consider now an instantiation $\mathbf{xy}$ of variables $\mathbf{XY}$. There must exist a unique $i$ and a unique $j$ such that $\mathbf{x} \models p_i$ and $\mathbf{y} \models q_j$. Moreover, $f(\mathbf{xy}) = s_i(\mathbf{y})$, $g(\mathbf{xy}) = r_j(\mathbf{y})$ and, hence, $[f \circ g](\mathbf{xy}) = s_i(\mathbf{y}) \circ r_j(\mathbf{y})$. Evaluating the given decomposition at instantiation $\mathbf{xy}$ also gives $s_i(\mathbf{y}) \circ r_j(\mathbf{y})$. $\square$

According to Theorem 2, the $\mathbf{X}$-partition of $f \circ g$ has size $O(nm)$, where $n$ and $m$ are the sizes of $\mathbf{X}$-partitions for $f$ and $g$. This is the basis for a future combination operation on SDDs that has a similar complexity.

Let $f = A \vee B$, $g = A \wedge B$ and consider the corresponding $A$-partitions $\{(A, \mathit{true}), (\neg A, B)\}$ and $\{(A, B), (\neg A, \mathit{false})\}$. The following is an $A$-partition for $f \oplus g$: $\{(A, \mathit{true} \oplus B), (\neg A, B \oplus \mathit{false})\}$ which equals $\{(A, \neg B), (\neg A, B)\}$.

Let $f = A \vee C$ , $g = B \vee C$ and consider the corresponding $\{A, B\}$-partitions $\{(A, \mathit{true}), (\neg A, C)\}$ and $\{(B, \mathit{true}), (\neg B, C)\}$. Theorem 2 gives $\{(A \wedge$

$B, true), (A \wedge \neg B, true), (\neg A \wedge B, true), (\neg A \wedge \neg B, C)\}$ as an $\{A, B\}$-partition for $f \vee g$. This is not compressed since its subs are not distinct. We compress it by simply disjoining the primes of equal subs, leading to $\{(A \vee B, true), (\neg A \wedge \neg B, C)\}$.

The canonicity of SDDs is due to the following result.

**Theorem 3** *A function $f(\mathbf{X}, \mathbf{Y})$ has exactly one compressed $\mathbf{X}$-partition.*

**Proof** Let $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be all instantiations of variables $\mathbf{X}$. Then $\{(\mathbf{x}_1, f|\mathbf{x}_1), \ldots, (\mathbf{x}_k, f|\mathbf{x}_k)\}$ is an $\mathbf{X}$-partition of function $f$. Let $s_1, \ldots, s_n$ be the distinct subfunctions in $f|\mathbf{x}_1, \ldots, f|\mathbf{x}_k$. For each $s_i$, define $p_i = \bigvee_{f|\mathbf{x}_j = s_i} \mathbf{x}_j$. Then $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$ is a compressed $\mathbf{X}$-partition of $f$. Suppose that $\beta = \{(q_1, r_1), \ldots, (q_m, r_m)\}$ is another compressed $\mathbf{X}$-partition of $f$. Then $\alpha$ and $\beta$ must have different partitions. Moreover, there must exist a prime $p_i$ of $\alpha$ that overlaps with two different primes $q_j$ and $q_k$ of $\beta$. That is, $\mathbf{x} \models p_i, q_j$ and $\mathbf{x}' \models p_i, q_k$ for some instantiations $\mathbf{x} \neq \mathbf{x}'$. We have $f|\mathbf{x} = \alpha|\mathbf{x} = s_i = r_j = \beta|\mathbf{x}$ and $f|\mathbf{x}' = \alpha|\mathbf{x}' = s_i = r_k = \beta|\mathbf{x}'$. Hence, $r_j = r_k$. This is impossible as $\beta$ is compressed. $\Box$

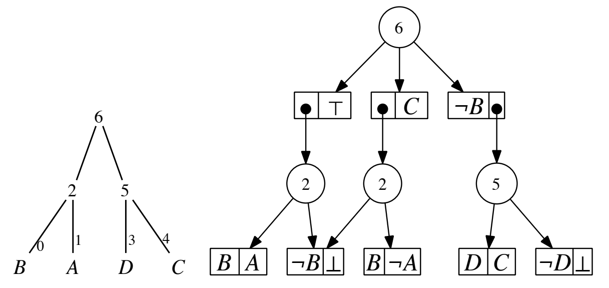
Let $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$ be an $\mathbf{X}$-partition for function $f$. Then $\beta = \{(p_1, \neg s_1), \ldots, (p_n, \neg s_n)\}$ is an $\mathbf{X}$-partition for its negation $\neg f$. This follows from Theorem 2 while noticing that $\neg f = f \oplus true$. Moreover, if $\alpha$ is compressed, then $\beta$ must be compressed as well. Consider function $f = A \vee B$ and $A$-partition $\{(A, true), (\neg A, B)\}$. Then $\{(A, false), (\neg A, \neg B)\}$ is an $A$-partition of function $\neg f = \neg A \wedge \neg B$.

Following is the second key notion underlying SDDs.

**Definition 4** *A vtree for variables $\mathbf{X}$ is a full binary tree whose leaves are in one-to-one correspondence with the variables in $\mathbf{X}$.*

Figure 1(a) depicts a vtree for variables $A, B, C$ and $D$. As is customary, we will often not distinguish between node $v$ and the subtree rooted at $v$, referring to $v$ as both a node and a subtree. Moreover, $v^l$ and $v^r$ denote the left and right children of node $v$.

The vtree was originally introduced in [Pipatsrisawat and Darwiche, 2008], but without making a distinction between left and right children of a node. In this paper, the distinction is quite important for the following reason. We will use a vtree to recursively decompose a Boolean function $f$, starting at the root of a vtree. Consider node $v = 6$ in Figure 1(a) which is root. The *left* subtree contains variables $\mathbf{X} = \{A, B\}$ and the *right* subtree contains $\mathbf{Y} = \{C, D\}$. Decomposing function $f$ at this node amounts to generating an $\mathbf{X}$-partition of function $f$. The result is quite different from generating a $\mathbf{Y}$-partition of the function. The SDD representation we shall present next is based on a recursive application of this decomposition technique. In particular, if $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ is the decomposition of function



(a) vtree (b) Graphical depiction of an SDD

Figure 1: Function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.

$f$ at node $v = 6$, then each prime $p_i$ will be further decomposed at node $v^l = 2$ and each sub $s_i$ will be further decomposed at node $v^r = 5$. The process continues until we have constants or literals.

The formal definition of SDDs is given next, after one more comment about vtrees. Each vtree induces a total variable order that is obtained from a left-right traversal of the vtree. The vtree in Figure 1(a) induces the total variable order $\langle B, A, D, C\rangle$. We will have more to say about these total orders when we discuss the relationship between SDDs and OBDDs.

## 3 The syntax and semantics of SDDs

We will use $\langle . \rangle$ to denote a mapping from SDDs into Boolean functions. This is needed for semantics.

**Definition 5** *$\alpha$ is an SDD that respects vtree $v$ iff:*

— *$\alpha = \bot$ or $\alpha = \top$.*
*Semantics: $\langle \bot \rangle = false$ and $\langle \top \rangle = true$.*

— *$\alpha = X$ or $\alpha = \neg X$ and $v$ is a leaf with variable $X$.*
*Semantics: $\langle X \rangle = X$ and $\langle \neg X \rangle = \neg X$.*

— *$\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$, $v$ is internal, $p_1, \ldots, p_n$ are SDDs that respect subtrees of $v^l$, $s_1, \ldots, s_n$ are SDDs that respect subtrees of $v^r$, and $\langle p_1 \rangle, \ldots, \langle p_n \rangle$ is a partition.*
*Semantics: $\langle \alpha \rangle = \bigvee_{i=1}^{n} \langle p_i \rangle \wedge \langle s_i \rangle$.*

*The size of SDD $\alpha$, denoted $|\alpha|$, is obtained by summing the sizes of all its decompositions.*

A constant or literal SDD is called *terminal.* Otherwise, it is called a *decomposition.* An SDD may respect multiple vtree nodes. Consider the vtree in Figure 1(a). The SDD $\{(\top, C)\}$ respects nodes $v = 5$ and $v = 6$. However, if an SDD respects node $v$, it will only mention variables in subtree $v$.

For SDDs $\alpha$ and $\beta$, we will write $\alpha = \beta$ to mean that they are syntactically equal. We will write $\alpha \equiv \beta$ to mean that they represent the same Boolean function: $\langle \alpha \rangle = \langle \beta \rangle$. It is possible to have $\alpha \equiv \beta$ and $\alpha \neq \beta$: $\alpha = \{(X, \gamma), (\neg X, \gamma)\}$ and $\beta = \{(\top, \gamma)\}$ is one such example. However, if $\alpha = \beta$, then $\alpha \equiv \beta$ by definition.

We will later provide further conditions on SDDs which guarantee $\alpha = \beta$ iff $\alpha \equiv \beta$ (i.e., canonicity).

SDDs will be notated graphically as in Figure 1(b), according to the following conventions. A decomposition is represented by a circle with outgoing edges pointing to its elements (numbers in circles will be explained later). An element is represented by a paired box $\boxed{p \mid s}$, where the left box represents the prime and the right box represents the sub. A box will either contain a terminal SDD or point to a decomposition SDD. The top level decomposition in Figure 1(b) has three elements with primes representing $A \wedge B$, $\neg A \wedge B$, $\neg B$ and corresponding subs representing $true$, $C$, and $C \wedge D$. Our graphical depiction of SDDs, which also corresponds to how they are represented in computer memory, will ensure that distinct nodes correspond to syntactically distinct SDDs (i.e., $\neq$). This is easily accomplished using the *unique-node* technique from the OBDD literature; see, e.g., [Meinel and Theobald, 1998].

If one adjusts for notation, by replacing circle-nodes with or-nodes, and paired-boxes with and-nodes, one obtains a structured DNNF [Pipatsrisawat and Darwiche, 2008] that satisfies strong determinism [Pipatsrisawat and Darwiche, 2010a]. The SDD has more structure, however, since its decompositions are not only strongly deterministic, but also partitioned (see Definition 1).

## 4 Canonicity of SDDs

We now address the canonicity of SDDs. We will first state two key definitions and assert some lemmas about them (without proof). We will then use these lemmas in proving our main canonicity theorem.

**Definition 6** *A Boolean function $f$ essentially depends on vtree node $v$ if $f$ is not trivial and if $v$ is a deepest node that includes all variables that $f$ essentially depends on.*

By definition, a trivial function does not essentially depend on any vtree node.

**Lemma 7** *A non-trivial function essentially depends on exactly one vtree node.*

**Definition 8** *An SDD is compressed iff for all decompositions $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ in the SDD, $s_i \not\equiv s_j$ when $i \neq j$. It is trimmed iff it does not have decompositions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$.*

An SDD is trimmed by traversing it bottom up, replacing decompositions $\{(\top, \alpha)\}$ and $\{(\alpha, \top), (\neg\alpha, \bot)\}$ with $\alpha$.

**Lemma 9** *Let $\alpha$ be a compressed and trimmed SDD. If $\alpha \equiv false$, then $\alpha = \bot$. If $\alpha \equiv true$, then $\alpha = \top$. Otherwise, there is a unique vtree node $v$ that SDD $\alpha$ respects, which is the unique node that function $\langle\alpha\rangle$ essentially depends on.*

Hence, two equivalent SDDs that are compressed and trimmed respect the same, unique vtree node (assuming the SDDs are not equal to $\bot$ or $\top$). This also implies

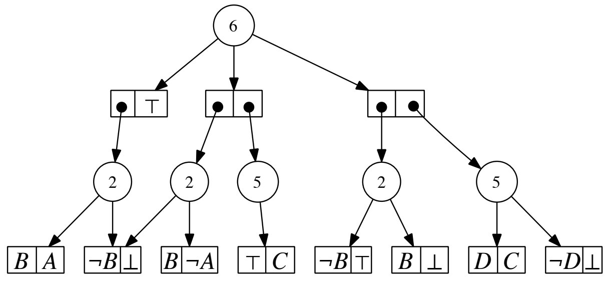


Figure 2: An SDD for $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$, which is normalized for $v = 6$ in Figure 1(a).

that SDDs which are not equal to $\bot$ or $\top$ can be grouped into equivalence classes, depending on the unique vtree node they respect. The SDD in Figure 1(b) is compressed and trimmed. Each of its decomposition nodes is labeled with the unique vtree node it respects.

**Theorem 10** *Let $\alpha$ and $\beta$ be compressed and trimmed SDDs respecting nodes in the same vtree. Then $\alpha \equiv \beta$ iff $\alpha = \beta$.*

**Proof** If $\alpha = \beta$, then $\alpha \equiv \beta$ by definition. Suppose $\alpha \equiv \beta$ and let $f = \langle\alpha\rangle = \langle\beta\rangle$. We will next show that $\alpha = \beta$, while utilizing Lemmas 7 and 9. If $f = false$, then $\alpha = \beta = \bot$. If $f = true$, then $\alpha = \beta = \top$. Suppose now that $f$ is not trivial and essentially depends on vtree node $v$, which must be unique. SDDs $\alpha$ and $\beta$ must both respect this unique node $v$. The proof continues by induction on node $v$.

Suppose node $v$ is a leaf. Only terminal SDDs respect leaf nodes, yet $\alpha$ and $\beta$ cannot be $\bot$ or $\top$ by assumption. Hence, $\alpha$ and $\beta$ must be literals. Since $\alpha \equiv \beta$, they must be equal literals, $\alpha = \beta$. Suppose now that node $v$ is internal and that the theorem holds for SDDs that respect descendants of $v$. Let $\mathbf{X}$ be the variables in subtree $v^l$, $\mathbf{Y}$ be the variables in subtree $v^r$, $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$ and $\beta = \{(q_1, r_1), \ldots, (q_m, r_m)\}$. By definition of an SDD, primes $p_i$ and $q_i$ respect nodes in subtree $v^l$ and can only mention variables in $\mathbf{X}$. Similarly, subs $s_i$ and $r_i$ respect nodes in subtree $v^r$ and can only mention variables in $\mathbf{Y}$. Hence, $\{(\langle p_1\rangle, \langle s_1\rangle), \ldots, (\langle p_n\rangle, \langle s_n\rangle)\}$ and $\{(\langle q_1\rangle, \langle r_1\rangle), \ldots, (\langle q_n\rangle, \langle r_n\rangle)\}$ are both $\mathbf{X}$-partitions of function $f$. They are also compressed by assumption. By Theorem 3, the decompositions must be equal. That is, $n = m$ and there is a one-to-one $\equiv$-correspondence between their primes and between their subs. By the induction hypothesis, there is a one-to-one $=$-correspondence between their primes and their subs. This implies $\alpha = \beta$. $\square$

## 5 The `Apply` operation for SDDs

If one looks carefully into the proof of canonicity, one finds that it depends on two key properties beyond com-

**Algorithm 1** $\texttt{Apply}(\alpha, \beta, \circ)$ : $\alpha$ and $\beta$ are SDDs normalized for the same vtree node and $\circ$ is a Boolean operator.

$\texttt{Cache}(.,.,.) = \texttt{nil}$ initially. $\texttt{Expand}(\gamma)$ returns $\{(\top,\top)\}$ if $\gamma = \top$; $\{(\top,\bot)\}$ if $\gamma = \bot$; else $\gamma$. $\texttt{UniqueD}(\gamma)$ returns $\top$ if $\gamma = \{(\top,\top)\}$; $\bot$ if $\gamma = \{(\top,\bot)\}$; else the unique SDD with elements $\gamma$.

```
1:  if $\alpha$ and $\beta$ are constants or literals then
2:      return $\alpha \circ \beta$ {must be a constant or literal}
3:  else if Cache($\alpha, \beta, \circ$) $\neq$ nil then
4:      return Cache($\alpha, \beta, \circ$)
5:  else
6:      $\gamma \leftarrow \{\}$
7:      for all elements $(p_i, s_i)$ in Expand($\alpha$) do
8:          for all elements $(q_j, r_j)$ in Expand($\beta$) do
9:              $p \leftarrow$ Apply($p_i, q_j, \wedge$)
10:             if $p$ is consistent then
11:                 $s \leftarrow$ Apply($s_i, r_j, \circ$)
12:                 add element $(p, s)$ to $\gamma$
13:             end if
14:         end for
15:     end for
16: end if
17: return Cache($\alpha, \beta, \circ$) $\leftarrow$ UniqueD($\gamma$)
```

pression. First, that trivial SDDs $\bot$ and $\top$ are the only ones that represent trivial functions. Second, that non-trivial SDDs respect unique vtree nodes, which depend on the functions they represent. Although trimming gives us both of these properties, one can also attain them using alternative conditions. For example, the first property can be ensured using only *light trimming:* excluding decompositions of the form $\{(\top,\top)\}$ or $\{(\top,\bot)\}$ by replacing them with $\top$ and $\bot$. The second property can be ensured by *normalization.* If $\alpha$ is a decomposition that respects node $v$, normalization requires that its primes respect $v^l$ and its subs respect $v^r$ (instead of respecting subtrees in $v^l$ and $v^r$; see Definition 5). Figure 2 depicts a normalized SDD of the one in Figure 1(b). Note how $C$ was replaced by $\{(\top, C)\}$ and $\neg B$ by $\{(\neg B, \top), (B, \bot)\}$ in the normalized SDD.

A Boolean function has a unique SDD that is compressed, lightly trimmed and normalized for a given vtree node. Even though normalization and light trimming lead to SDDs that are redundant compared to trimmed SDDs, the difference in size is only linear in the number of vtree nodes. Moreover, working with normalized SDDs can be very convenient as we shall see next.

Given SDDs $\alpha$ and $\beta$ that are lightly trimmed and normalized for the same vtree node, and given any Boolean operator $\circ$, Algorithm 1 provides pseudocode for the function $\texttt{Apply}(\alpha, \beta, \circ)$, which returns an SDD with the same properties for the function $\langle\alpha\rangle \circ \langle\beta\rangle$ and in $O(|\alpha||\beta|)$ time. The correctness of Algorithm 1 follows from Theorem 2 and a simple induction argument. Its time complexity follows using an argument similar to the one showing the complexity of $\texttt{Apply}$ for OBDDs.[1]

The simplicity of Algorithm 1 is due to normalization, which guarantees that the operands of every recursive call are normalized for the same vtree node if neither is trivial. The same function for trimmed SDDs will have to consider four cases depending on the relationship between the vtree nodes respected by its operands (i.e., equal, left descendant, right descendant, and neither).

Given $\texttt{Apply}$, two SDDs can be conjoined or disjoined in polytime. This also implies that an SDD can be negated in polytime by doing an exclusive-or with $\top$. This is similar to the $\texttt{Apply}$ function for OBDDs, which is largely responsible for their widespread applicability. Using $\texttt{Apply}$, one can convert a CNF to an SDD by first converting each of its clauses into an SDD (which can be done easily) and then conjoining the resulting SDDs. Similarly, one can convert a DNF into an SDD by disjoining the SDDs corresponding to its terms.

The SDD returned by $\texttt{Apply}(\alpha, \beta, \circ)$ is not guaranteed to be compressed even if the input SDDs $\alpha$ and $\beta$ are compressed. In a compressed SDD, the subs of every decomposition must be distinct. This is potentially violated on Line 12 as the computed sub $s$ may already exist in $\gamma$. One can easily modify $\texttt{Apply}$ so it returns a compressed SDD. In particular, when adding element $(p, s)$ on Line 12, if an element $(q, s)$ already exists in $\gamma$, simply replace it with $(\texttt{Apply}(p, q, \vee), s)$ instead of adding $(p, s)$.[2] The augmented version of $\texttt{Apply}$ has proved harder to analyze though since the additional recursive call involves SDDs that may not be part of the input SDDs $\alpha$ and $\beta$. In our implementation, however, compression has proved critical for the efficiency of $\texttt{Apply}$. We have identified strong properties of the additional recursive call $\texttt{Apply}(p, q, \vee)$, but do not yet have a characterization of the complexity of $\texttt{Apply}$ with compression. We will, however, provide in Section 7 an upper bound on the size of compressed SDDs.

## 6 Every OBDD is an SDD

A vtree is said to be *right-linear* if each left-child is a leaf. The vtree in Figure 3(a) is right-linear. The compressed and trimmed SDD in Figure 3(b) respects this right-linear vtree. Every decomposition in this SDD has the form $\{(X, \alpha), (\neg X, \beta)\}$, which is a Shannon decomposition. This is not a coincidence as it holds for every compressed and trimmed SDDs that respects a right-linear vtree. In fact, such SDDs correspond to reduced OBDDs in a precise sense; see Figure 3(c). In particular, consider a reduced OBDD that is based on the total variable order induced by the given right-linear vtree. Then every decomposition in the SDD corresponds to a decision node in the OBDD and every decision node in the

[1]The consistency test on Line 10 can be implemented in linear (even constant) time since SDDs are DNNFs.

[2]Line 10 can be replaced with $p \neq \bot$ in this case.

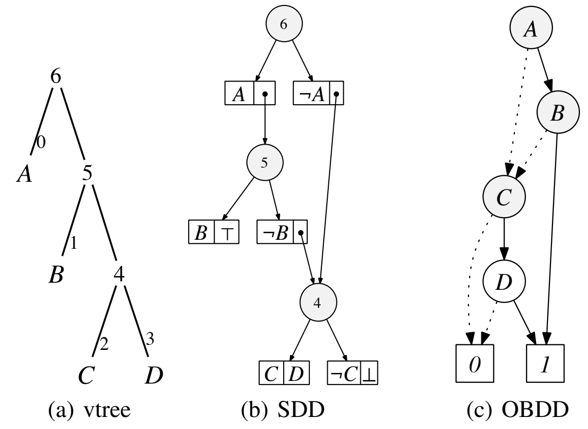


Figure 3: A vtree, SDD and OBDD for $(A \wedge B) \vee (C \wedge D)$.

$A \vee B \vee \neg C$
$\neg A \vee C \vee D$
$A \vee B \vee \neg E$
$B \vee F$
(a) CNF

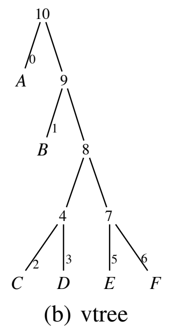


Figure 4: A CNF and a corresponding nice vtree.

OBDD corresponds to a decomposition or literal in the SDD. In a reduced OBDD, a literal is represented by a decision node with 0 and 1 as its children (e.g., the literal $D$ in Figure 3(c)). However, in a compressed and trimmed SDD, a literal is represented by a terminal SDD (e.g., the literal $D$ in Figure 3(b)).

## 7 An upper bound on SDDs

Consider the vtree in Figure 4(b) and node $v = 8$. Variable $A$ is an *ancestor's left child* of node $v$ and will be called an ALC variable of $v$. Variable $B$ is also an ALC of $v$. Consider now the CNF in Figure 4(a). If we condition the CNF on the ALC variables of $v = 8$, the simplified CNF will decompose into independent components, one over variables in $v^l = 4$ and the other over variables in $v^r = 7$. For example, if we condition on $A, \neg B$, the first component will be $C \vee D$ and the second component will be $F$. When a function $f(\mathbf{X}, \mathbf{Y})$ decomposes into independent components over variables $\mathbf{X}$ and $\mathbf{Y}$, it can be written as $f = (\exists \mathbf{X} f) \wedge (\exists \mathbf{Y} f)$, where $\exists \mathbf{X} f$ is the $\mathbf{Y}$-component and $\exists \mathbf{Y} f$ is the $\mathbf{X}$-component. This motivates the following definition, identified in [Pipatsrisawat and Darwiche, 2010b].

**Algorithm 2** $\texttt{sdd}(f, v, \mathbf{z})$ : $f$ is a Boolean function, $v$ is a nice vtree and $\mathbf{z}$ is a variable instantiation.

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

```
1: if $v$ is a leaf node then
2:     return $\alpha, \beta$: terminal SDDs, $\langle\alpha\rangle = f$ and $\langle\beta\rangle = \neg f$
3: else if $v^l$ is a leaf node with variable $X$ then
4:     $s_1, \neg s_1 \leftarrow \texttt{sdd}(f|X, v^r, \mathbf{z}X)$
5:     $s_2, \neg s_2 \leftarrow \texttt{sdd}(f|\neg X, v^r, \mathbf{z}\neg X)$
6:     return $\texttt{UniqueD}\{(X, s_1), (\neg X, s_2)\}$,
7:            $\texttt{UniqueD}\{(X, \neg s_1), (\neg X, \neg s_2)\}$
8: else
9:     $g \leftarrow \exists \mathbf{X} f$, where $\mathbf{X}$ are the variables of $v^r$
10:    $h \leftarrow \exists \mathbf{Y} f$, where $\mathbf{Y}$ are the variables of $v^l$
11:    $p, \neg p \leftarrow \texttt{sdd}(g, v^l, \mathbf{z})$
12:    $s, \neg s \leftarrow \texttt{sdd}(h, v^r, \mathbf{z})$
13:    return $\texttt{UniqueD}\{(p, s), (\neg p, \bot)\}$,
14:           $\texttt{UniqueD}\{(p, \neg s), (\neg p, \top)\}$
15: end if
```

**Definition 11** *Let $f$ be a Boolean function. A vtree for function $f$ is nice if for each internal node $v$ in the vtree, either $v^l$ is a leaf or $f|\mathbf{z} = \exists \mathbf{X}(f|\mathbf{z}) \wedge \exists \mathbf{Y}(f|\mathbf{z})$, where $\mathbf{X}$ are the variables of $v^r$, $\mathbf{Y}$ are the variables of $v^l$ and $\mathbf{Z}$ are the ALC variables of node $v$. The width of node $v$ is the number of distinct sub-functions $f|\mathbf{z}$. The width of the nice vtree is the maximum width of any node.*

The vtree in Figure 4 is nice for the CNF in that figure, since the left child of every internal node is leaf, except for node $v = 8$ which satisfies the second condition of a nice vtree. Nicety comes with the following guarantee.

**Theorem 12** *Let $f$ be a Boolean function and $v$ be a nice vtree with width $w$. There is a compressed and trimmed SDD for function $f$ with size $O(nw)$, where $n$ is the number of variables in the vtree.*

**Proof** Algorithm 2 computes such an SDD. In particular, the call $\texttt{sdd}(f, v, \mathit{true})$ returns two compressed and trimmed SDDs $\alpha, \beta$ such that $\langle\alpha\rangle = f$ and $\langle\beta\rangle = \neg f$. The correctness of the algorithm can be shown by induction on node $v$, using the properties of nice vtrees and the observations made after Theorem 2 about negation. The use of $\texttt{UniqueD}$ guarantees that the returned SDDs are compressed and trimmed. Note that compression here is easy since primes are always terminal SDDs. One can show by induction that every recursive call $\texttt{sdd}(f_\star, v_\star, \mathbf{z})$ is such that $\mathbf{Z}$ are the ALC variables of node $v_\star$ and $f_\star = \exists \mathbf{W}(f|\mathbf{z})$, where $\mathbf{W}$ are all other variables outside node $v$. Hence, the number of distinct recursive calls for node $v_\star$ is no more than the number of distinct subfunctions $f|\mathbf{z}$, which is the width of node $v_\star$. Moreover, for each distinct recursive call, the algorithm constructs at most two decompositions, each of size at most two. Since the number of vtree nodes is $O(n)$, the size of constructed SDDs is $O(nw)$. $\square$

If the function $f$ is in CNF, and with appropriate caching, Algorithm 2 can be adjusted so it runs in $O(nw)$ time as well. That will take away from its clarity, however, so we skip this adjustment here. The importance of nice vtrees is the following result.

**Theorem 13** *A CNF with $n$ variables and treewidth $w$ has a nice vtree with width $\leq 2^{w+1}$. Hence, the CNF has a compressed and trimmed SDD of size $O(n2^w)$.*

Nice vtrees can be constructed easily from appropriate *dtrees* of the given CNF [Darwiche, 2001], but we leave out the details for space limitations.

A CNF with $n$ variables and pathwidth $pw$ is known to have an OBDD of size $O(n2^{pw})$. Pathwidth $pw$ and treewidth $w$ are related by $pw = O(w \log n)$ (e.g., [Bodlaender, 1998]). Hence, a CNF with $n$ variables and treewidth $w$ has an OBDD of size polynomial in $n$ and exponential in $w$ [Ferrara *et al.*, 2005]. Theorem 13 is tighter, however, since the SDD size is *linear* in $n$ and exponential in $w$. BDD-trees are also canonical and come with a treewidth guarantee. Their size is also linear in $n$ but at the expense of being *doubly* exponential in treewidth [McMillan, 1994]. Hence, SDDs come with a tighter treewidth bound than BDD-trees.

## 8 Preliminary Experimental Results

We present in this section some preliminary empirical results, in which we compare the size of SDDs and OBDDs for CNFs of some circuits in the ISCAS89 suite.

OBDDs are characterized by the use of variable orders, and effective methods for generating good variable orders have been critical to the success and wide adoption of OBDDs in practice. This includes orders that are generated statically (by analyzing the input CNF) or dynamically (by constantly revising them during the compilation process); see, e.g., [Meinel and Theobald, 1998]. In contrast, SDDs are characterized by the use of vtrees, which relax the need for total variable orders in OBDDs. This allows one to potentially identify more compact compilations, while maintaining canonicity and a polytime `Apply` operation (the key factors behind the success and adoption of OBDDs in practice).

In the preliminary experimental results that we present next, we hope to illustrate two points: (1) even simple heuristics for statically constructing vtrees can allow SDDs that are more compact than OBDDs, and (2) SDDs have the potential to subsume the use of OBDDs in practice. The latter point however depends critically on the development of more sophisticated heuristics for statically or dynamically identifying good vtrees.

The main static method we used for generating vtrees is a dual of the method used for generating dtrees in [Darwiche, 2001], with the roles of clauses and variables exchanged.[3] This method requires a variable elimination heuristic, which was the minfill heuristic in our experiments. We also ensured that the left child of a vtree node is the one with the smallest number of descendants. The resulting vtree is referred to as "Minfill vtree" in Table 1. The main static method we used for generating variable orders is the MINCE heuristic [Aloul *et al.*, 2001]. This is referred to as "MINCE order." To provide more insights, we considered two additional types of vtrees and variable orders. In particular, we considered the variable order obtained by a left-right traversal of the Minfill vtree, referred to as "Minfill order" (see end of Section 2). We also considered the vtrees obtained by recursively dissecting the MINCE order in a balanced or random way, referred to as "MINCE balanced vtree" and "MINCE random vtree.[4]" We used the same implementation for both SDDs and OBDDs, obtaining OBDDs by using right-linear vtrees of the corresponding variable orders. The size of a compilation (whether SDD or OBDD) is computed by summing the sizes of decomposition nodes.

Table 1 reveal some patterns. First, Minfill vtrees give the best results overall, leading to smaller compilations almost consistently. Second, linearizing such vtrees, to obtain OBDDs as in Column 5, always leads to larger compilations in these experiments. Third, OBDDs based on MINCE orders are typically larger than SDDs based on Minfill vtrees. Finally, dissecting these orders into vtrees (whether balanced or random) produces better compilations in the majority of cases.

As mentioned earlier, these results are preliminary as they do not exhaust enough the class of vtrees and variable orders used, nor do they exhaust enough the suite of CNFs used. One interesting aspect of these limited experiments, however, is the close proximity they reveal between the structures characterizing SDDs (vtrees) and the structures characterizing OBDDs (variable orders). In particular, the experiments suggest simple methods for turning one structure into the other. They also suggest that methods for generating good variable orders may form a basis for generating good vtrees.

Within the same vein, the experiments suggest that the practice of using SDDs can subsume the practice of using OBDDs as long as one includes right-linear vtrees in the space of considered vtrees. This is particularly important since the literature contains many generalizations of OBDDs that promise smaller representations, whether theoretically or empirically, yet require a substantial shift in practice to realize such potentials. A notable example here is the FBDD, which relaxes the variable ordering property of an OBDD, by requiring what is known as an "FBDD type" [Gergov and Meinel, 1994]. Using such

[3]One could also use nice vtrees, but these appear to be less competitive based on a preliminary empirical evaluation.

[4]In the case of randomly dissecting a variable order, we used five trials and reported the best size obtained.

Table 1: A preliminary empirical evaluation of SDD and OBDD sizes. A "$\star$" indicates out of time or memory.

| CNF | Vars | Clauses | Minfill | | MINCE | | | Model count |
|---|---|---|---|---|---|---|---|---|
| | | | vtree | order | order | balanced vtree | random vtree | |
| s208.1 | 122 | 285 | 2380 | 3468 | 2104 | 2936 | 2865 | 262144 |
| s298 | 136 | 363 | 5159 | 17682 | 19288 | 12055 | 12989 | 131072 |
| s344 | 184 | 429 | 5904 | 36098 | 20138 | 10309 | 12397 | 16777216 |
| s349 | 185 | 434 | 4987 | 50156 | 25754 | 16374 | 24202 | 16777216 |
| s382 | 182 | 464 | 5980 | 14540 | 17506 | 13438 | 11258 | 16777216 |
| s386 | 172 | 506 | 18407 | 115994 | 28148 | 12748 | 18537 | 8192 |
| s400 | 189 | 486 | 6907 | 18904 | 24126 | 20210 | 12279 | 33554432 |
| s420.1 | 252 | 601 | 6134 | 24908 | 7372 | 6928 | 9647 | 17179869184 |
| s444 | 205 | 533 | 6135 | 18364 | 13408 | 7255 | 10675 | 16777216 |
| s510 | 236 | 635 | 10645 | 38764 | 34724 | 13925 | 17438 | 33554432 |
| s526 | 217 | 638 | 11562 | 87208 | 47296 | 22950 | 33405 | 16777216 |
| s641 | 433 | 918 | 19482 | 370832 | 646386 | 318301 | 142425 | 18014398509481984 |
| s713 | 447 | 984 | 24491 | 407250 | 396607 | 82955 | 85743 | 18014398509481984 |
| s820 | 312 | 1046 | 58935 | 1024111 | 458163 | $\star$ | $\star$ | 8388608 |
| s832 | 310 | 1056 | 61043 | 894904 | 383228 | $\star$ | $\star$ | 8388608 |
| s838.1 | 512 | 1233 | 14062 | 57182 | 29180 | 19033 | 23964 | 73786976294838206464 |
| s953 | 440 | 1138 | 167356 | $\star$ | 876544 | $\star$ | $\star$ | 35184372088832 |

types, FBDDs are known to be canonical and to support a polytime Apply operation. Yet, one rarely finds practical methods for constructing FBDD types and, hence, one rarely finds practical applications of FBDDs. Another broad class of OBDD generalizations is based on using decompositions that generalize or provide variants on the Shannon decomposition; see [Meinel and Theobald, 1998] for some examples. Again, none of these generalizations proved to be as influential as OBDDs in practice.

The close proximity of vtrees to variable orders is perhaps the most striking feature of SDDs, in comparison to other generalizations of OBDDs, as it stands to minimize the additional investment needed to identify good vtrees (given what we know about the generation of good variable orders). Clearly, the extent to which SDDs will eventually improve on OBDDs in practice will depend on making such an investment. This is why the generation of good vtrees, whether statically or dynamically, is a key priority of our future work on the subject.

**Acknowledgement** I wish to thank Arthur Choi, Knot Pipatsrisawat and Yexiang Xue for valuable discussions and for comments on earlier drafts of the paper.

## References

[Aloul *et al.*, 2001] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Faster SAT and smaller BDDs via common function structure. In *ICCAD*, pages 443–448, 2001.

[Bodlaender, 1998] Hans L. Bodlaender. A partial -arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998.

[Bryant, 1986] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Tran. Com.*, C-35:677–691, 1986.

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

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

[Darwiche, 2001] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Ferrara *et al.*, 2005] Andrea Ferrara, Guoqiang Pan, and Moshe Y. Vardi. Treewidth in verification: Local vs. global. In *LPAR*, pages 489–503, 2005.

[Gergov and Meinel, 1994] Jordan Gergov and Christoph Meinel. Efficient boolean manipulation with OBDD's can be extended to FBDD's. *IEEE Transactions on Computers*, 43(10):1197–1209, 1994.

[McMillan, 1994] K. McMillan. Hierarchical representations of discrete functions, with application to model checking. In *Lecture Notes In Computer Science; Vol. 818*, 1994.

[Meinel and Theobald, 1998] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD Foundations and Applications*. Springer, 1998.

[Pipatsrisawat and Darwiche, 2008] Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In *AAAI*, pages 517–522, 2008.

[Pipatsrisawat and Darwiche, 2010a] Knot Pipatsrisawat and Adnan Darwiche. A lower bound on the size of decomposable negation normal form. In *AAAI*, 2010.

[Pipatsrisawat and Darwiche, 2010b] Knot Pipatsrisawat and Adnan Darwiche. Top-down algorithms for constructing structured DNNF: Theoretical and practical implications. In *ECAI*, pages 3–8, 2010.