OBDDs, SDDs, and circuits of bounded width: completeness matters

Alexis de Colnet[1] Sebastian Ordyniak[2]
Stefan Szeider[1]

[1] Algorithms and Complexity Group, TU Wien, Vienna, Austria
[2] Algorithms and Complexity Group, University of Leeds, Leeds, UK

Abstract

Ordered Binary Decision Diagrams (OBDDs) are dynamic data structures with many application areas. The literature suggested that OBDDs of bounded width equate to Boolean circuits of bounded pathwidth. In this paper, we show that this relationship holds only for complete OBDDs. Additionally, we demonstrate that similar limitations affect the claimed equivalence between Sentential Decision Diagrams (SDDs) of bounded width and Boolean circuits of bounded treewidth.

1 Introduction

Ordered Binary Decision Diagrams (OBDDs) are a fundamental data structure used for efficiently representing and manipulating Boolean functions [10]. OBDDs have found widespread application in many areas of computer science, including hardware verification [11] and design [24], feature models representation [25, 30, 20], model checking [35, 14], optimization [4, 3, 5, 31], and proof complexity [2, 12, 21]. By providing a canonical representation for Boolean functions, OBDDs enable efficient equality testing and Boolean operations, making them a key tool in formal methods and verification. Sentential Decision Diagrams (SDDs) are a more recently proposed data structure that can be seen as a generalization of OBDDs [17]. SDDs provide a canonical representation for Boolean functions that can be exponentially more compact than OBDDs in certain cases [7]. Like OBDDs, SDDs support efficient Boolean operations and have been used in knowledge compilation [15, 26, 18].

One of the most important aspects of OBDDs and languages for knowledge representation, in general, is the trade-off between succinctness (the encoding should be small) and tractability (the language should support efficient reasoning algorithms). Because of this, the expressive power of OBDDs has been extensively studied. Entire book chapters have been dedicated to studying the OBDD-size of specific functions [34]. Generally, the OBDD-size of a function is large with respect to the number n of variables. Simple functions whose OBDD-size is exponential in n were found early [10]. Almost all Boolean functions have the same OBDD-size (up to an $1+o(1)$ factor) as the hardest Boolean function for OBDD, so exponential in n, something known as the *Shannon effect* [23, 33]. Therefore, functions with small OBDD-size are rare, and even rarer are function whose OBDD-size is linear in n. These functions can be described in terms of OBDD-width: the maximum number of nodes labeled with the same variable in an OBDD representation of the function. Linear OBDD-size essentially coincides with constant OBDD-width. It was claimed that OBDDs of bounded width have the same expressive power as Boolean circuits of bounded pathwidth [22, Theorem 2.15]. This article aims to show that this result does not hold for general OBDDs but merely for
textcolorblacksmooth or complete OBDDs.
textcolorblackSmoothness and completeness for an OBDD refers to the property that every variable of the function is read along every computation path. General OBDDs need not have this property

as they can represent functions with irrelevant variables. Typically, *reduced* OBDDs (ROBDD) are generally not
textcolorblacksmooth. Smoothness and completeness are mostly convenient for proofs. Indeed it is well-known that every OBDD can be made complete at the cost of increasing its size by a factor of n, so completeness is usually just assumed without loss of generality in scenarios where it only matters whether the OBDD-size is polynomial in n or not. However, this is not the case for the claim that bounded circuit pathwidth coincides with bounded OBDD-width. Here, completeness matters. Jha and Suciu prove that bounded complete-OBDD-width coincides with bounded circuit pathwidth but stated their result for general OBDD-width. On the other hand, we show that there are OBDDs of bounded width that cannot be translated to complete OBDDs of bounded width. Thus, we show that there are OBDDs of bounded width that cannot be translated to Boolean circuits of bounded pathwidth.

We also show that a similar caveat applies to the alleged equivalence between Sentential Decision Diagrams (SDDs) of bounded width and bounded treewidth Boolean circuits [9]. Namely, the functions computable by complete,
textcolorblackor smooth, SDDs of bounded width coincide with the functions computable by circuits of bounded treewidth, and again completeness, or
textcolorblacksmoothness, is a necessary property. However, there is the technical issue that width and
textcolorblacksmoothness are not defined for SDDs, or rather, the current the definition of SDDs does not allow for satisfactory definitions of these notions. This is an issue worth addressing as
textcolorblacksmoothness is a property of circuits that is often convenient and sometimes required for solving certain problems.
textcolorblackSmoothness has been introduced by Darwiche [16] as a useful property for doing model counting on Boolean circuits called d-DNNF circuits but the notion extends applicable to various kind of circuits, it has for instance been shown to be a crucial property for computing marginals on sum-product networks [28, 27].
textcolorblackSmoothness is important enough to motivate some work on efficient ways to make a circuit complete (or smooth) [29]. Thus, it is a problem that we cannot define
textcolorblacksmoothness on SDDs. In this paper, we tweak definition of SDDs in a way that is inconsequential for all significant properties of SDDs proved so far, and yet allows us to define
textcolorblacksmooth SDDs naturally.

We hope that this paper will help dissipate any confusion about the expressive power of bounded-width OBDDs and SDDs. The takeaway is that functions computable by circuits of bounded pathwidth (resp. treewidth) do not coincide with functions computable by OBDDs (resp. SDD) of bounded width, which are already quite rare, but with
textcolorblacksmooth or complete OBDDs (resp. SDDs) of bounded width, which are strictly rarer. This is informally summarized as

$$\begin{aligned} \mathrm{CPWD}(O(1)) &= \text{complete-OBDD}(O(1)) \neq \mathrm{OBDD}(O(1)) \quad \text{and} \\ \mathrm{CTWD}(O(1)) &= \text{complete-SDD}(O(1)) \neq \mathrm{SDD}(O(1)), \end{aligned}$$

where (complete-)OBDD($O(1)$) refers to the class of Boolean functions computable by bounded-width (complete) OBDDs, (complete-)SDD($O(1)$) refers to the class of Boolean functions computable by bounded-width (complete) SDDs, and CPWD($O(1)$) and CTWD($O(1)$) refer the classes of Boolean functions computable by circuits of bounded pathwidth and treewidth, respectively.

2 Preliminaries

Boolean variables take their values in $\{0, 1\}$ where 0 is interpreted as *false* and 1 is interpreted as *true*. A literal for a Boolean variable x is either x, or the negation of x, denoted by $\neg x$. We write $lit(X) = X \cup \{\neg x \mid x \in X\}$. Let X be a set of Boolean variables. An assignment to X is a mapping

$\alpha : X \to \{0,1\}$. A Boolean function f over X associates a value from $\{0,1\}$ to every assignment to X. An assignment $\alpha \in f^{-1}(1)$ is said to *satisfy* f. An assignment $\alpha \in f^{-1}(0)$ is said to *falsify* f. The set of variables of a function f (resp. an assignment α), when not explicit, is denoted by $var(f)$ (resp. $var(\alpha)$).
textcolorblackA variable $x \in var(f)$ is relevant for f when there exists two assignments α and β to $var(f)$ that differ only on x such that $f(\alpha) \neq f(\beta)$. A variable that is not relevant for f is said *irrelevant* for f. For two assignments $\alpha_1 : X_1 \to \{0,1\}$ and $\alpha_2 : X_2 \to \{0,1\}$ with $X_1 \cap X_2 = \emptyset$, we denote by $\alpha_1 \cup \alpha_2$ the assignment $\alpha : X_1 \cup X_2 \to \{0,1\}$ with $\alpha(x) = \alpha_i(x)$ if $x \in X_i$ for every variable $x \in X_1 \cup X_2$.

2.1 Graph and Width Parameters

The root node of a tree T is denoted by $root(t)$. For t a node in T, $T|_t$ denotes the subtree of T rooted at t (so $root(T|_t) = t$).

A *tree decomposition* of an undirected graph G is a pair (T, λ), where T is a tree and $\lambda : V(T) \to 2^{V(G)}$ such that for every edge $uv \in E(G)$, there is a node $t \in V(T)$ with $u, v \in \lambda(t)$ and, for every vertex $v \in V(G)$, the set $\{\, t \in V(T) \mid v \in \lambda(t) \,\}$ forms a non-empty connected subtree of T. To distinguish between the vertices of G and the vertices of T, we refer to the latter ones as nodes or bags. The *width* of (T, λ) is equal to the maximum of $|\lambda(t)| - 1$ over all nodes of T. We say that a tree decomposition (T, λ) is a *path decomposition* if T is a path. The *treewidth* (*pathwidth*) of a graph G is the minimum width of any tree decomposition (path decomposition) of G.

A binary decomposition tree of a graph G is a pair (T, μ) where T is binary tree and μ is a bijection from T's leaves to $V(G)$ [32, Definition 3.1.3]. Removing any edge e from T splits T into two trees and thus induces a bipartition (L_e, R_e) of $V(G)$. The *maximum matching width* of (T, μ), denoted by $mmw(T, \mu)$ is the size of the largest matching between $G[L_e]$ and $G[R_e]$ in G when e ranges over all edges of T. The maximum matching width of G, denoted by $mmw(G)$ is defined as the minimum $\min_{(T,\mu)} mmw(T, \mu)$ over all possible binary decomposition tree of G. The maximum matching width of G is known to equal the treewidth of G up to a constant factor.

Lemma 1 ([32, Theorem 4.2.5])**.** *Let G be a graph, then $\frac{1}{3}(tw(G) + 1) \leq mmw(G) \leq tw(G) + 1$.*

We say that a binary tree is *right-linear* if every internal node of T has two children and the left one is a leaf. We say that a binary tree decomposition (T, λ) is *right-linear* if so is T. Given a total ordering π of $V(G)$, D_π denotes the rooted and right-linear binary decomposition tree of G where π coincides with the ordering of the variables obtained by reading them on the leaves of the decomposition from left to right.

A (c, d)-*expander graph* G is a d-regular graph such that for any $S \subseteq V(G)$ of size $|S| \leq |V(G)|/2$, it holds that $|N(S)| \geq c|S|$, where $N(S) := \{v \in V(G) \setminus S \mid (u, v) \in E(G), u \in S\}$. It is known that there are infinitely many 3-regular expander graphs.

Lemma 2 ([1, Section 9.2])**.** *There is, for some $c > 0$, an infinite sequence of $(c, 3)$-expander graphs $(G_i)_{i \in \mathbb{N}}$.*

It is also known that expander graphs have large treewidth so, by Lemma 1, they also have large maximum matching width.

Lemma 3 ([19, Proposition 1])**.** *Let $c > 0$ and $d \in \mathbb{N}$ be fixed. There is a constant $\gamma > 0$ such that every (c, d)-expander graph on n vertices has treewidth and maximum matching width at least γn.*

2.2 DNNF circuits

A *circuit (or expression DAG)* is a directed acyclic graph D with a unique sink vertex o (output gate) such that every vertex $v \in V(D) \setminus \{o\}$ is either:

- an *input gate (IN-gate)* with no incoming arcs,

- an *AND-gate* with at least one incoming arc,
- an *OR-gate* with at least one incoming arc,
- a *NOT-gate* with exactly one incoming arc.

Every input gate corresponds to a Boolean variable and no two input gates correspond to the same variable. We denote by $var(D)$ the set of variables for the input gates of D. More generally, for g a gate of D, $var(g)$ is the set of variables for the input gets appearing below g. If g is an input gate for the variable x then $var(g) = \{x\}$. A circuit is in *negation normal form* (NNF) when the incoming neighbor of every NOT-gate is an input gate. A circuit is in *decomposable negation normal form* (DNNF) when for every AND-gate g, no two distinct incoming neighbor of g share a variable. Formally, if c and c' are two incoming neighbors of g, $c \neq c'$, then $var(c) \cap var(c') = \emptyset$. A circuit in DNNF is
textcolorblacksmooth when, for every OR-gate g, its incoming neighbors have identical variable sets. Formally, if c and c' are two incoming neighbors of g, then $var(c) = var(c')$.

For an assignment $\alpha : var(D) \to \{0,1\}$ and a vertex $v \in V(D)$, we denote by $val(v, D, \alpha)$ the value of the gate v after assigning all input gates according to α. That is, $val(v, D, \alpha)$ is recursively defined as follows: If v is an input gate for the variable x, then $val(v, D, \alpha) = \alpha(x)$, if v is an AND-gate (OR-gate), then $val(v, D, \alpha) = \bigwedge_{n \in N_D^-(v)} val(n, D, \alpha)$ ($val(v, D, \alpha) = \bigvee_{n \in N_D^-(v)} val(n, D, \alpha)$). Here and in the following $N_D^-(v)$ denotes the set of all incoming neighbors of v in D. We set $\mathsf{O}(D, \alpha) = val(o, D, \alpha)$ and we say that a Boolean function f over a set of variables X is represented by a circuit D if $var(D) = X$ and $\mathsf{O}(D, \alpha) = f(\alpha)$ for every assignment α of the variables in X. The *pathwidth* $pw(D)$ of a Boolean circuit D is equal to the pathwidth of the underlying undirected graph of D. Similarly, the *treewidth* $tw(D)$ of D is equal to the treewidth of the underlying undirected graph of D.

2.3 OBDDs

A *binary decision diagram* (BDD) is a directed acyclic graph (DAG) with a single source, two sinks labeled 0 and 1, and internal nodes labeled with Boolean variables. Each internal node has two distinct successors called its 0-child and its 1-child. Let B be a BDD. Its set of variables is denoted by $var(B)$. A BDD represents a Boolean function over $var(B)$. A BDD made of a single sink 0 (resp. 1) represents the constant 0-function (resp. 1-function). If however the source node of a BDD B is a decision node labeled with x whose 0-child is the BDD B_0 and whose 1-child is the BDD B_1, then B represents the function recursively defined by $ite(x, B_1, B_0) = (x \wedge B_1) \vee (\neg x \wedge B_0)$ ($if\, x\, then\, B_1\, else\, B_0$). Graphically, every complete assignment α to $var(B)$ corresponds to a unique path in B: starting from the source, if the path reaches a node v labeled with x, then the path continues to the 0-child of v if $\alpha(x) = 0$, and to the 1-child of v otherwise, and so on until reaching a sink. The value $B(\alpha)$ computed by B's function on α is the value of the sink reached by the path for α. An *ordered BDD* B (OBDD) is a BDD such that, on every path from the source to a sink, every variable labels at most one node and such that the order of appearance of the variables along any path is consistent with a single total ordering π of $var(B)$. We also say that B is a π-OBDD.

Definition 1 (Width of an OBDD)**.** The *width* of an OBDD is the maximum number of its nodes labeled with the same variable.

For B a π-OBDD over n variables and $i \in [n]$, we denote by $L_i(B)$ the i-th layer of B, that is, the set of its decision nodes labeled with the i-th variable in the ordering π. We also denote by $L_{n+1}(B)$ the set of B's sinks. The width of B is then $\max_{i \in [n]} |L_i(B)|$.

A *complete* OBDD is an OBDD where, on every path from the source to a sink, each variable appears *exactly once* [34, Definition 3.2.1]. When B is complete, we have that for every $i \in [n]$, the children of every node in $L_i(B)$ are in $L_{i+1}(B)$. An OBDD is *smooth* when, for every decision node $ite(x, B_1, B_0)$, we have $var(B_1) = var(B_0)$. There is a subtle difference between smoothness and completeness. Complete OBDDs are smooth but the converse does not hold. If an OBDD B represents

f, then completeness forces every source-to-sink path in B to contain one decision node per variable in $var(f)$, even those irrelevant for f. In contrast, B may be smooth while not having a single decision node for irrelevant variables.

Proposition 1. *Let f be a Boolean function over X. Suppose that every variable $x \in X$ is relevant in f. Then every smooth OBDD representing f is also complete.*

Proof. Write $X = \{x_1, \ldots, x_n\}$. Suppose toward a contradiction that B represents f and is smooth but not complete. Suppose, without loss of generality, that the variable ordering for B is $(x_1, \ldots, x_n)$. Since x_1 is relevant in f, the source node of B is labeled with x_1. By assumption there is a source-to-sink path $(v_1, v_2, \ldots)$ and an integer $i > 1$ such that for all $j < i$, v_j is a decision node labeled with x_j but v_i is not a decision node labeled with x_i. Thus, $x_i \notin var(v_i)$. For every $2 \leq j \leq i$ let u_j be the node such that $v_{j-1} = ite(x_{j-1}, v_j, u_j)$ or $v_{j-1} = ite(x_{j-1}, u_j, v_j)$. By smoothness we have that $var(u_j) = var(v_j)$. Therefore $var(v_{j-1}) = var(v_j) \cup \{x_{j-1}\}$ contains x_i if and only if $x_i \in var(v_j)$. We thus conclude that, $x_i \notin var(v_i)$ implies $x_i \notin var(v_1) = var(B)$. This means that x_i is irrelevant in the function represented by B, a contradiction. □

Every OBDD can be made complete and thus smooth without changing its variable ordering and without increasing its width by more than a factor n. The trick is to add "dummy nodes" whose 0-child and 1-child are the same. Proposition 1 implies that, given a function f, the minimal width of a complete OBDD representing f equals the minimal width of a smooth OBDD representing f (plus 1 when f is a constant function). Indeed a smooth OBDD can be made complete by removing all nodes labeled with irrelevant variables and by adding a chain of dummy nodes for those irrelevant variables at the source. Given this equivalence and given that the term *complete OBDD* is more widely used than *smooth OBDD*, we use the notion of complete OBDD in the remainder of the paper.

Proposition 2 ([34, Theorem 3.2.3])**.** *Let B be an OBDD over n variables, one can construct in time $O(n \cdot |B|)$ a* complete *OBDD B' with the same variable ordering, whose width is at most $n \cdot width(B)$, and that represents the same function.*

Every π-OBDD B can be transformed in linear time into an equivalent minimal-size π-OBDD that is unique and called *reduced*. There is also a unique complete π-OBDD whose size is minimal among all complete π-OBDD representing the same function. This complete π-OBDD is called *quasi-reduced*.

3 Complete OBDDs and Circuits of Bounded Pathwidth

Earlier work from Jha and Suciu tried to draw a connection between small-width OBDD and circuits of bounded pathwidth [22]. One of their results was that, for a class $\mathcal{F}$ of functions, there exists a constant c such that all functions $f \in \mathcal{F}$ admit OBDD representations of width at most c, if and only if there exists a constant c' such that all $f \in \mathcal{F}$ admit circuit of pathwidth[1] at most c'. This correspondence is summarized in [22] as

$$\boxed{!} \quad \text{CPWD}(O(1)) = \text{OBDD}(O(1)). \tag{1}$$

In this section, we argue that this result is not correct, and we give the corrected variant. We use the symbol $\boxed{!}$ to warn the reader of false claims.

3.1 OBDD and Circuits of Bounded Pathwidth: a Correction

Let OBDD(w) be the set of of Boolean functions that admit OBDD representations of width at most w and let CPWD(w) be the set of Boolean functions that admit circuits of pathwidth at most w. The erroneous claim (1) was derived from the following two results:

[1] Jha and Suciu actually refer to circuit pathwidth as *expression pathwidth*.

! **Claim 1.** [22, Theorem 2.15] *If there exists an OBDD for f with width w, then there exists a circuit of pathwidth at most $5w$ representing f. This implies* $\mathrm{OBDD}(w) \subseteq \mathrm{CPWD}(5w+1)$.

Claim 2. [22, Corollary 2.13] *If there exists a circuit for f with pathwidth w, then there exists an OBDD of width at most $2^{(w+1)2^{w+1}}$ representing f. This implies* $\mathrm{CPWD}(w) \subseteq \mathrm{OBDD}(2^{(w+1)2^{w+1}})$.

Unfortunately, Claim 1 does not hold unless the OBDD is complete,
textcolorblackor at least smooth. Since OBDDs can be
textcolorblacksmoothed in polynomial-time, one may deem this subtlety inconsequential. But
textcolorblacksmoothing OBDDs can multiply the width by a factor $\Omega(n)$, where n is the number of variables.

[22] is rather ambiguous about the properties of the OBDD. Whereas the introduction mentions that OBDD may not be complete,

> *on any path from the root to a sink every variable appears at most once and in the order* Π *(variables may be skipped)*

the proof of Theorem 2.15, though Lemma 4.1, implicitly uses complete OBDD. The lemma's statement is that, if $\bar{f} = f_1, f_2, \ldots, f_w$ are formulae with a shared OBDD of width w, then there exists a shared expression (or circuit) for them having pathwidth $5w$ s.t. all root nodes $\bar{f}$ occur on a leaf of the path decomposition. The proof given is by induction and reads as follows.

> *Let the first variable in the variable order of OBDD be X_1 and denote the formulae at the first level by $g_1, g_2, \ldots, g_w$. Then every f_i can be written as $(\neg X_1 \wedge g_j) \vee (X_1 \wedge g_k)$ for some j, k. Denote the nodes corresponding to new $\wedge$, $\neg$ operators by $\bar{op}$. Now, by induction hypothesis, $\bar{g}$ have a path-decomposition with width $5w$ one of whose leaves contains $\bar{g}$. We connect that leaf to a new node which contains $\bar{g}, \bar{f}$, X_1, $\bar{op}$. The resulting path-decomposition of $\bar{f}$ has width $5w$.*

The implicit assumption that the OBDD is complete occurs when saying that the g_is are in the first level of the OBDD. An OBDD level is a set of decision nodes that reads the same variable. In a non-complete OBDD, f_i can be $(\neg X_1 \wedge g'_j) \vee (X_1 \wedge g'_k)$ for g'_j and g'_k in the first layer or in layers below. If instead the level $g_1, \ldots, g_w$ is defined as the immediate successors of the nodes for $f_1, \ldots, f_w$, then nodes from one level need not all read the same variable and thus we cannot assume that there are at most w nodes per level.

We insist that the proofs of [22] are all correct under the assumption of complete, or quasi-reduced, OBDD. Thus, let us then rewrite Claim 1 with quasi-reduced OBDD. Let $\mathrm{qROBDD}(w)$ denote the set of Boolean functions that can be represented by a quasi-reduced OBDD, that is, a minimal-size complete OBDD, of width w. The proof of Claim 2 needs no correction and can even be strengthened to quasi-reduced OBDD.

Lemma 4. *If there exists a* complete *OBDD for f with width w, then there exists a circuit D representing f s.t. $pw(D) \leq 5w$. This implies* $\mathrm{qROBDD}(w) \subseteq \mathrm{CPWD}(5w+1)$.

Lemma 5. *If there exists a circuit for f with pathwidth w, then there exists a* complete *OBDD of width at most $2^{(w+1)2^{w+1}}$ representing f. This implies* $\mathrm{CPWD}(w) \subseteq \mathrm{qROBDD}(2^{(w+1)2^{w+1}})$.

At that point one may think that Claim 1 may still hold but requires a different proof. But we will refute both Claim 1 and (1) with the following theorem.

Theorem 1. *There is an infinite set $\mathcal{F}$ of Boolean functions and two constants $c \in \mathbb{N}$ and $\gamma \in (0,1]$ such that, for every $f \in \mathcal{F}$ over n variables, the OBDD-width of f is at most c and the textcolorblacksmooth-OBDD-width of f is at least γn.*

The combination of Lemmas 4 and 5 implies that

$$\mathrm{CPWD}(O(1)) = \mathrm{qROBDD}(O(1)) \tag{2}$$

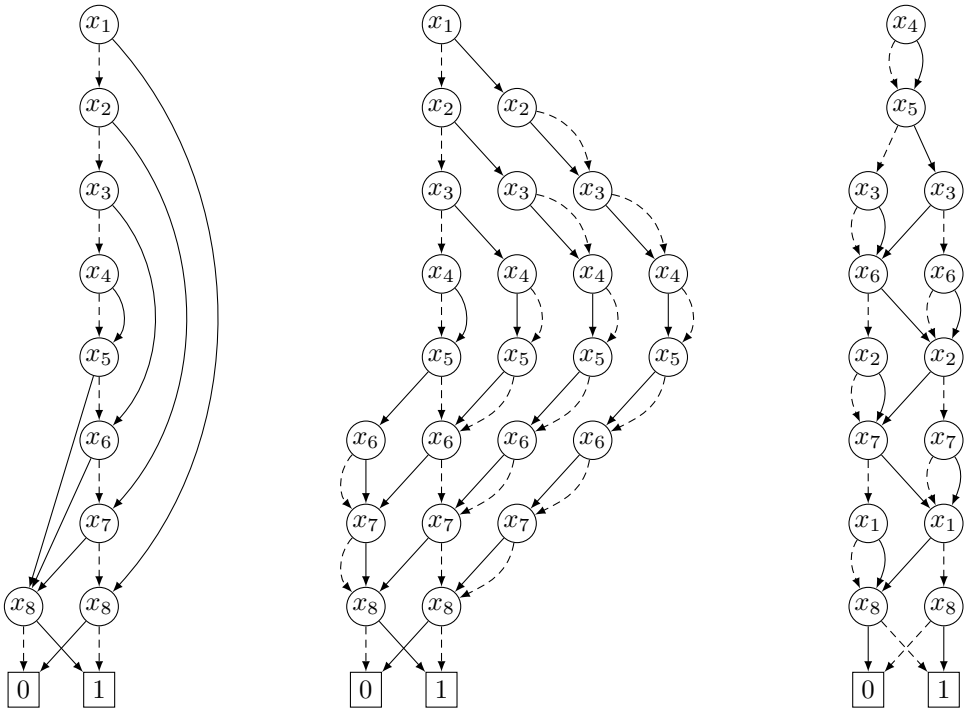


Figure 1: Three OBDD representing the same function.

while Theorem 1 essentially says that

$$\text{qROBDD}(O(1)) \subsetneq \text{OBDD}(O(1)) \tag{3}$$

The next section is dedicated to the proof of Theorem 1

3.2 Width Difference between OBDD and Complete OBDD

3.2.1 Related work

In Proposition 2, the bound on the width of B' is tight. It is not hard to find OBDDs whose widths are bounded by a constant but such that the width of the complete OBDD *with the same variable ordering* is linear in the number of variable. See for instance the first two OBDDs in Figure 1. They represent the same function. The first one has width 2 but is not complete, the second is complete, respects the same variable ordering $x_1, x_2, \ldots, x_n$, and has width $\Theta(n)$. In this example, if we are allowed to change the variable ordering, then we can find an equivalent complete OBDD of width 2 as shown in Figure 1 on the right. For other functions, changing the ordering does not help. This was shown by Bollig and Wegener.

Proposition 3 ([6, Theorem 3])**.** *There exists functions over n variables represented by an OBDD of width $O(n)$ for some variable ordering, and whose representations by complete OBDDs all have width at least $\Omega(n^2)$ for* all *variable orderings.*

Proposition 3 in itself already shows that, if Claim 1 was true, then we could not prove it by first making the OBDD complete and next using the proof of Jha and Suciu [22] on complete OBDDs. However, Proposition 3 is not sufficient in itself to refute Claim 1 because one can envision that the claim holds true with a more complex proof that does not use complete OBDDs at all. Since Proposition 3 does not apply to OBDDs of constant width, it cannot be used to prove (3). Theorem 1 is the equivalent to Proposition 3 for OBDDs of constant width.

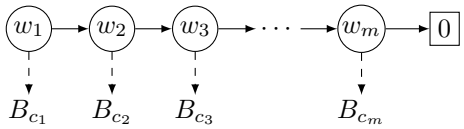
Capelli and Mengel [13] have shown a result similar to Theorem 1. We can prove a statement in the vein of Theorem 1 using two lemmas from [13]. In [13, Lemma 13], they show that for every n there is an $O(1)$-width OBDD B over the set of variables $X \cup Z$ with $n = |X \cup Z|$ such that the function $\exists Z.B(X,Z)$ (that is, $\bigvee_{a_Z \in \{0,1\}^Z} B(X, a_Z)$) does not have an OBDD of size $2^{o(n)}$. Crucially, B is not complete. In [13, Lemma 1], they show that if B was equivalent to a complete OBDD B' of width $O(1)$, then there would be an OBDD of width $O(1)$, and thus of size $O(n)$, computing $\exists Z.B(X,Z)$. Hence, the function represented by B does not admit a complete OBDD representations of width $O(1)$: the width must depend on n. Thus, one can already prove (3) from their results. However, Capelli and Mengel do not precisely quantify the dependence on n of the complete OBDD (it could be that the non-complete OBDD has constant width while the complete OBDD has width $o(n)$). Theorem 1 gives a more precise statement. We use the same construction for Theorem 1 as used by Capelli and Mengel for [13, Lemma 13].

3.2.2 Proof Strategy

The proof relies on properties of expander graphs. We consider a collection of constraints $c_1, \ldots, c_m$ and its primal graph, that is, the graph whose vertices are the variables and where two variables are connected by an edge if and only if they belong to the scope of a common constraint. From $c_1, \ldots, c_m$, one constructs a function f that uses m additional variables to select one and only one constraint to evaluate. The additional variables form a unary encoding of the index i of the constraint c_i to evaluate. Let us call $w_1, \ldots, w_m$ the additionnal variables, then f looks something like that

$$\bigvee_{i=1}^{m} ((w_1 w_2 \ldots w_{i-1} \neg w_i) \wedge c_i).$$

Assuming each constraint c_i is individually representable by a constant-width OBDD B_{c_i} for some common variable ordering, the function f has polynomial-size OBDDs of constant width, as shown in the next figure.



But such OBDDs are not complete. For well-chosen $c_1, \ldots, c_m$, the width of a complete OBDD for the function is not constant. Recall that the width of an n-variable complete OBDD is the maximum number of different functions obtainable from assigning the first k variables of its ordering, with k varying from 1 to n. The partition of the variables that splits the first k variables on one side and the $n-k$ last variables on the other is seen as a bipartition of the primal graph of $c_1, \ldots, c_m$. We choose $c_1, \ldots, c_m$ so that, the bigger the matching between the two sides of the bipartition, the more functions can be obtained by assigning the variables from one side. Since the result must hold for all possible variable orderings, making the primal graph an expander graph is a sound choice because expanders have the properties that any balanced bipartition of the vertices have many edges crossing from one side to the other. By ensuring that the primal graph is an expander and has bounded degree, we guarantee that every balanced bipartition has a large matching between its two sides.

3.2.3 Proof of Theorem 1

We need the following standard result on the width of complete OBDDs.

Lemma 6 ([34, Theorem 3.2.2])**.** *Let B be a complete π-OBDD over n variables representing the function f. Let $X_0 = \emptyset$ and, for every $j \in [n]$, let X_j be the set of the first j variables in the ordering π. The number of nodes of B labeled with the j-th variable in the ordering π is at least the number of*

different functions $f|\alpha$ for α a full assignment to X_{j-1}, where the function $f|\alpha : var(f)\setminus X_{j-1} \to \{0,1\}$ is given by $(f|\alpha)(\beta) = f(\alpha \cup \beta)$.

Let $W = (w_1, \ldots, w_m)$, $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$ be three sequences of Boolean variables such that $|set(W)| = m$, and $set(W) \cap (set(U) \cup set(V)) = \emptyset$ and such that, for all $i \in [m]$, $u_i \neq v_i$. Note that we can have $v_i = v_j$ or $u_i = u_j$ or $u_i = v_j$ for some $i \neq j$. Let

$$f(U,V,W) := \bigvee_{i=1}^{m} ((w_1 w_2 \ldots w_{i-1} \neg w_i) \wedge (u_i \leftrightarrow v_i)).$$

Definition 2. Let X be a set and π be a total ordering of X, a *cut* of π is a bipartition (X_1, X_2) of X such that, for some integer i, X_1 is exactly the set of the first i elements in π. Then, (X_1, X_2) is called the *i-th cut* of π. A pair of elements $\{x, y\} \subseteq X$ is *split by the cut* (X_1, X_2) if $x \in X$ and $y \in X_2$, or if $y \in X_1$ and $x \in X_2$.

Lemma 7. *Let B be a complete π-OBDD representing $f(U,V,W)$, with $W = (w_1, \ldots, w_m)$, $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$. Suppose there exists a cut of π and a set $J \subseteq [m]$ such that, for all $j \in J$, π splits $\{u_j, v_j\}$ and such that, for all $i, j \in J$, $i \neq j$, we have $\{u_i, v_i\} \cap \{u_j, v_j\} = \emptyset$. Then the width of B is at least $|J|$.*

Proof. Let (X_1, X_2) be the cut of π described in the statement and suppose it is the i-th cut of π. For every $j \in J$, let $\hat{u}_j$ be the element in $\{u_j, v_j\} \cap X_1$ and let $\hat{v}_j$ be the element in $\{u_j, v_j\} \cap X_2$. We have $\{\hat{u}_j, \hat{v}_j\} = \{u_j, v_j\}$.

For every $j \in J$, let a_j be the assignment to $U \cup V \cup W$ that sets w_j, $\hat{u}_j$ and $\hat{v}_j$ to 0 and all other variables to 1. Let a_j^1 be the restriction of a_j to X_1 and let a_j^2 be the restriction of a_j to X_2. Since a_j is a model of f, a_j^2 is a model of $f|a_j^1$. In addition, since all $\hat{u}_j$ for $j \in J$ belong to X_1 and are pairwise distinct, the assignments a_j^1 are pairwise distinct.

We claim that no assignment $\alpha := a_j^1 \cup a_{j'}^2$ is a model of f for any $j' \in J \setminus \{j\}$. On the one hand, $\alpha(\hat{u}_j) = a_j^1(\hat{u}_j) = 0$ and $\alpha(\hat{v}_j) = a_{j'}^2(\hat{v}_j) = 1$, so α falsifies the term $(w_1 w_2 \ldots w_{j-1} \neg w_j) \wedge (u_j \leftrightarrow v_j)$. On the other hand, $\alpha(\hat{u}_{j'}) = a_j^1(\hat{u}_{j'}) = 1$ and $\alpha(\hat{v}_{j'}) = a_{j'}^2(\hat{v}_{j'}) = 0$, so α falsifies the term $(w_1 w_2 \ldots w_{j'-1} \neg w_{j'}) \wedge (u_{j'} \leftrightarrow v_{j'})$. Finally, among $w_1, \ldots, w_m$, α only assigns w_j and/or $w_{j'}$ to 0, so it falsifies all terms $w_1 w_2 \ldots w_{i-1} \neg w_i$ for all $i \notin \{j, j'\}$. Therefore α is not a model of f.

Thus, for every $j' \in J \setminus \{j\}$, $a_{j'}^2$ is not a model of $f|a_j^1$. It follows that the functions $f|a_j^1$ for $j \in J$ are pairwise distinct. Hence, by Lemma 6 there are at least $|J|$ nodes at the $(i+1)$-th level of the OBDD. □

The sequences U and V are filled with the vertices of an expander graph in the following way. Let G be a graph over n vertices without self loop. Write the edges of G in an arbitrary order in the sequence: $(e_1, \ldots, e_m)$. Let $u(e_i)$ and $v(e_i)$ be the two endpoints of e_i. For each i, put $u(e_i)$ to U and $v(e_i)$ to V. Note that it is fine for the same vertex two be placed in U or in V and possibly in both U and V (the definition of $f(U,V,W)$ allows it). Observe also that the constraint of no element appearing in U and V at the same index is satisfied since, G having no self loop, we have $u(e_j) \neq v(e_j)$ for all $j \in [m]$. We rename U as U_G and V as V_G to render the dependence on G explicit. The function used in Theorem 1 is $f(U_G, V_G, W)$, where G is a $(c, 3)$-expander graph for some $c > 0$ and W is a set of $|E(G)|$ fresh Boolean variables.

Lemma 8. *For every 3-regular graph G, there is an OBDD of width at most 6 representing $f(U_G, V_G, W)$.*

Proof. For the OBDD, consider a variable ordering π where the variables W comes before the variables/vertices $V(G)$. Let $E(G) = \{e_1, \ldots, e_m\}$. For every i, let $u(e_i)$ (resp. $v(e_i)$) be the endpoint of e_i in U_G (resp. $V(G)$). There is an OBDD B_{e_i} representing $u(e_i) \leftrightarrow v(e_i)$ with $var(B_{e_i}) = \{u(e_i), v(e_i)\}$, of width 2, and whose variable ordering is consistent with π. Therefore, the OBDD shown in Figure 2 is a π-OBDD that represents $f(U_G, V_G, W)$. Each variable w_i has exactly one decision node and the number of nodes for each variable/vertex v_i is at most twice the number of edges containing it. Since G is 3-regular the width of the OBDD is at most 6. □

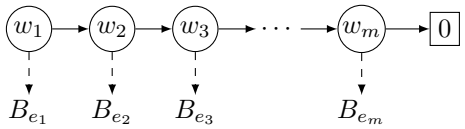


Figure 2: An OBDD representation of $f(U_G, V_G, W)$, where $B_{e_i} = u(e_i) \leftrightarrow v(e_i)$

It may look like the straightforward rewriting of the OBDD for $f(U_G, V_G, W)$ shown in Figure 2 as a circuit would give a circuit of small pathwidth. But keep in mind that all input variables are merged in the circuit, so in fact the pathwidth would not be small at all. Now we show that every *complete* OBDD for $f(U_G, V_G, W)$ has a non-constant width when G is a 3-regular expander graph.

Lemma 9. *Let $c > 0$. There is a constant $\alpha > 0$ such that, for every $(c, 3)$-expander n-vertex graph G, all complete OBDDs representing $f(U_G, V_G, W)$ have width at least $\alpha \cdot n$.*

Proof. Let B be a complete π-OBDD representing $f(U_G, V_G, W)$. Let π_G be the restriction of π to the variables $V_G \cup U_G$. Recall the definition of the rooted right-linear binary decomposition tree $D_{\pi_G} = (T, \lambda)$ corresponding to π_G and recall that removing an edge from T breaks T into two trees and thus induces a partition (L, R) of V_G. By definition of D_{π_G} one sees that removing an edge between two internal nodes of T yields the same bipartition as a cut of π_G. Hence, by Lemma 3 on the maximum matching width of G, there must be a cut of π_G that splits at least γn pairs $\{u(e_j), v(e_j)\}$. We say in this case that the edge e_j is *split* (by the cut). Let E be these split edges. Since G is 3-regular, a fifth of these edges are pairwise disjoint. To see this, initialize two sets E' and E'' to $\emptyset$ and visit the edges of E one by one, if $e \in E$ share not endpoint with any edge in E' then add it to E', otherwise add it to E''. By 3-regularity of G we have $|E''| \leq 4|E'|$ so, since $|E'| + |E''| = |E|$ we have that E' is a set of at least $|E|/5 \geq \gamma n/5$ pairwise disjoint edges split by a cut of π_G. Since π_G is the restriction of π to $V_G \cup U_G$, there is a cut of π that splits the same edges E'. Invoking Lemma 7 finishes the proof. □

Theorem 1 follows directly from Lemmas 2, 8, and 9.

4 Complete SDDs and Circuits of Bounded Treewidth

Inspired by the work relating circuit pathwidth to OBDDs, Bova and Szeider have proved an analogous relationship between circuit treewidth and SDDs (Sentential Decision Diagrams) [9]. SDDs were introduced by Darwiche as Boolean functions representations that extend OBDDs while preserving many of its advantages [17]. A notion of SDD-width can be defined similarly to that of OBDD-width and related to circuit treewidth. In this section, we show that if completeness is essential in proving that the class of functions with constant circuit pathwidth coincides with the class of functions with constant complete OBDD width, the same subtlety applies to SDD-width and circuit treewidth. Namely, the class of functions with constant circuit treewidth (denoted by CTWD($O(1)$)) coincides with the class of functions with constant complete SDD width (denoted by complete-SDD($O(1)$)), and completeness is not optional.
textcolorblackIn fact, what we really need is the *smoothness* property, which is almost equivalent. The purpose of this section is to explain that

$$\text{CTWD}(O(1)) = \text{complete-SDD}(O(1)) \subsetneq \text{SDD}(O(1)).$$

where SDD($O(1)$) refers to the class of functions with constant SDD width. One issue here is that completeness is not defined for SDD, or rather, that the usual definition of SDD does not let us define completeness
textcolorblackor smoothness in a natural way. We thus use a slightly modified definition of SDD that is equivalent to Darwiche's original definition and enables us to define complete
textcolorblackand smooth SDD in a satisfactory manner.

This section is organized as follows. In Subsection 4.1, we present the definition of SDD that we use and explain in what sense it differs from the original definition. In Subsection 4.2 we recall the translations between OBDDs, SDDs and DNNF circuits, which we use in Subsection 4.3 to motivate our definition of SDDs. In Subsection 4.3 we introduce completeness
textcolorblackand smoothness for SDDs and explain how to make an SDD
textcolorblacksmooth. We then show the separation between CTWD($O(1)$) and SDD($O(1)$) in Subsection 4.4.

4.1 SDD Definition(s)

Vtrees. A vtree (variable tree) over a set X of Boolean variables is a pair $\mathcal{T} = (T, \phi)$ where T is a binary tree and ϕ is a bijection from T's leaves to X. The children of a vertex t in T are ordered: we distinguish the left child t_ℓ from the right child t_r. We write $var(\mathcal{T})$ when X is not explicit. For every $t \in T$ define $var(t) = \{\, \phi(l) \mid l \text{ a leaf descendant of } t \,\}$ when t is an internal node of T, and $var(t) = \{\phi(t)\}$ when t is a leaf. In the degenerate case where $X = \emptyset$, we use the empty tree (no vertex, no edge) and the vacuous bijection from $\emptyset$ to $\emptyset$ to define a vtree over $\emptyset$. For $t \in T$, we write $\mathcal{T}|_t = (T|_t, \phi|_t)$. Note that $\mathcal{T}|_t$ is a vtree over $\phi(t)$.

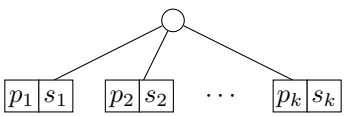
SDD. Our definition of SDDs is slightly different from Darwiche's original definition [17]. An SDD S *respects* a vtree $\mathcal{T} = (T, \phi)$, or *is structured by* $\mathcal{T}$. An SDD contains two kinds of nodes: constant nodes and decomposition nodes. Each node η is interpreted as a function $\langle \eta \rangle$ over $var(\eta)$. A *constant c-node* η with $c \in \{0, 1\}$ is interpreted as $\langle \eta \rangle = c$ with $var(\eta) = \emptyset$. A *decomposition node* η is represented by a set $\{(p_1, s_1), \ldots, (p_k, s_k)\}$ where

- each p_i is either a literal or a decomposition node, and
- each s_i is either a constant node or a decomposition node, and
- $\langle p_i \rangle \wedge \langle p_j \rangle = 0$ for every $i \neq j$, and
- $\bigvee_{i=1}^{k} \langle p_i \rangle = 1$.

We have $\langle \eta \rangle = \bigvee_{i=1}^{k} \langle p_i \rangle \wedge \langle s_i \rangle$, where $\langle p_i \rangle = \ell$ if p_i is a literal ℓ, and $var(\eta) = \bigcup_{i=1}^{k} var(p_i) \cup var(s_i)$, where $var(p_i) = \{x\}$ if p_i is a literal in $\{x, \neg x\}$. Let $\lambda_{S,\mathcal{T}}$ be the function mapping every decomposition node η to the deepest node t of T such that $var(\eta) \subseteq var(t)$. For S to be an SDD that respects $\mathcal{T}$, we must have that for every decomposition node η,

- either η is of the form $\{(x, c_1), (\neg x, c_0)\}$ with c_1 and c_0 two constant nodes and $\lambda_{S,\mathcal{T}}(\eta) = \phi^{-1}(x)$,
- or η is $\{(p_1, s_1), \ldots, (p_k, s_k)\}$ where some p_i or s_i is a decomposition node, and $\lambda_{S,\mathcal{T}}(\eta)$ is an internal node of T with two children t_ℓ and t_s such that, for all i, first, $var(p_i) \subseteq var(t_\ell)$ (so, when p_i is a decomposition node, $\lambda_{S,\mathcal{T}}(p_i)$ is either t_ℓ or a descendant of t_ℓ); second, $var(s_i) \subseteq var(t_r)$ (so, when s_i is a decomposition node, $\lambda_{S,\mathcal{T}}(s_i)$ is either t_r or a descendant of t_r).

An SDD S with root node η represents the function $\langle \eta \rangle$ over $var(\eta)$. Note that $var(\eta) \subseteq var(\mathcal{T})$. The size of S, denoted by $|S|$, is its number of decomposition nodes. The p_i's and s_i's are called *primes* and *subs*, respectively. A pair (p_i, s_i) is called a *prime-sub pair*. Graphically, a decomposition node is represented as follows



where each p_i, s_i is 0, 1, x, $\neg x$, or a pointer to another decomposition node.

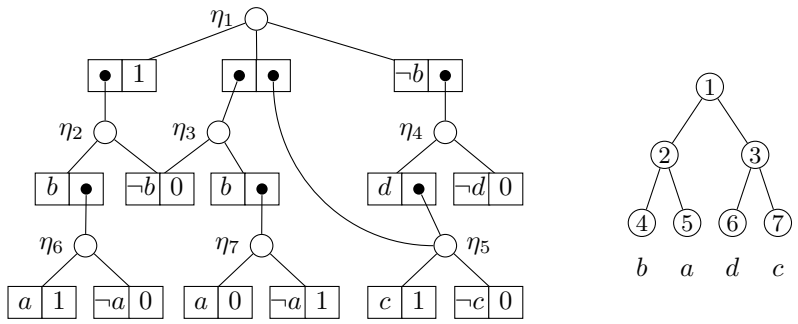


Figure 3

Example 1. The SDD of Figure 3 (left) represents the function $(b \wedge a) \vee (b \wedge \neg a \wedge c) \vee (\neg b \wedge d \wedge c)$. It is structured by the vtree shown on the same figure (right).

$\eta_1 = \{(\eta_2, 1), (\eta_3, \eta_5), (\neg b, \eta_4)\}$	$\langle \eta_1 \rangle = (b \wedge a) \vee (b \wedge \neg a \wedge c) \vee (\neg b \wedge d \wedge c)$	$\lambda_{S,\mathcal{T}}(\eta_1) = ①$
$\eta_2 = \{(b, \eta_6), (\neg b, 0)\}$	$\langle \eta_2 \rangle = b \wedge a$	$\lambda_{S,\mathcal{T}}(\eta_2) = ②$
$\eta_3 = \{(\neg b, 0), (b, \eta_7)\}$	$\langle \eta_3 \rangle = b \wedge \neg a$	$\lambda_{S,\mathcal{T}}(\eta_3) = ②$
$\eta_4 = \{(d, \eta_5), (\neg d, 0)\}$	$\langle \eta_4 \rangle = d \wedge c$	$\lambda_{S,\mathcal{T}}(\eta_4) = ③$
$\eta_5 = \{(c, 1), (\neg c, 0)\}$	$\langle \eta_5 \rangle = c$	$\lambda_{S,\mathcal{T}}(\eta_5) = ⑦$
$\eta_6 = \{(a, 1), (\neg a, 0)\}$	$\langle \eta_6 \rangle = a$	$\lambda_{S,\mathcal{T}}(\eta_6) = ⑤$
$\eta_7 = \{(a, 0), (\neg a, 1)\}$	$\langle \eta_7 \rangle = \neg a$	$\lambda_{S,\mathcal{T}}(\eta_7) = ⑤$

Note that the pre-image of some vtree nodes through λ is empty.

Darwiche's definition. Darwiche's definition of SDDs [17] is seemingly simpler, but less convenient to define concepts like completeness. Every SDD following Darwiche's definition can be transformed into an SDD following ours. First, we recall Darwiche's definition. Let $\mathcal{T} = (T, \phi)$ be a vtree and λ be a mapping from the SDD's nodes to T's nodes. An SDD node η structured by $(\mathcal{T}, \lambda)$ can be of three types: (1) η can be a constant $c \in \{0, 1\}$ (interpretation: $\langle \eta \rangle = c$, $var(\eta) = \emptyset$); (2) η can be a literal $\ell \in \{x, \neg x\}$, then $\lambda(\eta)$ is a leaf and $\phi(\lambda(\eta)) = x$ (interpretation: $\langle \eta \rangle = \ell$, $var(\eta) = \{x\}$); (3) n can be a *decomposition node* $\{(p_1, s_1), \ldots, (p_k, s_k)\}$ and $\lambda(\eta) = t$ is an internal node of T with children t_ℓ, t_r (interpretation: $\langle \eta \rangle = \bigvee_{i=1}^{k} \langle p_i \rangle \wedge \langle s_i \rangle$, $var(\eta) = \bigcup_{i=1}^{k} var(p_i) \cup var(s_i)$). In case (3), each p_i is an SDD node respecting $(\mathcal{T}, \lambda)$ with $\lambda(p_i)$ equal to t_ℓ or to a descendant of t_ℓ, each s_i is an SDD node respecting $(\mathcal{T}, \lambda)$ with $\lambda(s_i)$ equal to t_r or to a descendant of t_r, and it must hold that $\langle p_i \rangle \wedge \langle p_j \rangle = 0$ for every $i \neq j$, and also that $\bigvee_{i=1}^{k} \langle p_i \rangle = 1$.

A few remarks help understanding the differences between Darwiche's definition and ours.

Remark 1. In our definition, it is forbidden for a sub s_i to be a literal. To obtain $\langle s_i \rangle = x$, one sets $s_i = \{(x, 1), (\neg x, 0)\}$. This means, for instance, that the SDD $\{(a, \neg b), (\neg a, b)\}$ representing $a \oplus b$ in Figure 5 is correct in Darwiche's definition but not in ours. In our definition, one has to rewrite like in Figure 4. Similarly, in our definition a single literal is not an SDD, whereas it is in Darwiche's definition.

Remark 2. In our definition, a decomposition node of the form $\{(x, c_0), (\neg x, c_1)\}$, with c_1 and c_0 constant nodes, is mapped to a leaf of T by $\lambda_{S,\mathcal{T}}$ whereas in Darwiche's definition, decomposition nodes are always mapped to internal nodes of T. It follows that our definition allows for SDDs that are not "valid" under Darwiche's definition, for instance the SDD of Figure 4. Indeed, under Darwiche's definition, the root decomposition node of this SDD is mapped to the vtree node ①, but then the two other decomposition nodes must be mapped to children of ①, which cannot be since these are leaves.

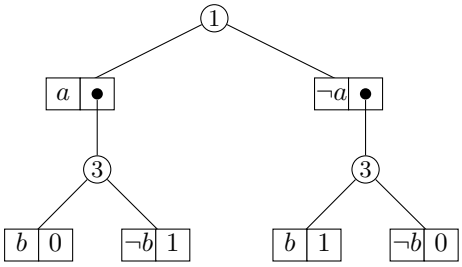


Figure 4: An SDD for $a \oplus b$

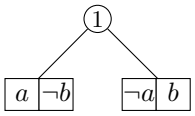


Figure 5: An SDD for $a \oplus b$ (Darwiche's representation)

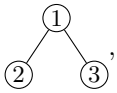


Figure 6: A vtree for the two SDDs

Remark 3. In both definitions, constants are SDDs. Constant nodes enable us represent the constants 0 and 1 without having to introduce any variable. Arguably we could write $\{(x, 0), (\neg x, 0)\}$ to represent the function over x that is uniformly 0, but this is not exactly the same as the constant 0 since $var(\{(x, 0), (\neg x, 0)\}) = \{x\}$ while $var(0) = \emptyset$.

Remark 4. In Darwiche's definition, there may be several λ that work for the same vtree $\mathcal{T} = (T, \phi)$ and the same SDD S. On the other hand, $\lambda_{S,\mathcal{T}}$ is uniquely defined. This allows us to say that S is structured by $\mathcal{T}$ rather than by $(\mathcal{T}, \lambda_{S,\mathcal{T}})$. Note that $\lambda_{S,\mathcal{T}}$ only maps decomposition nodes to nodes of T and ignores constant nodes and literals.

The differences between Darwiche's definition are minor and one can efficiently rewrite an SDD in Darwiche's definition to fit ours, and vice-versa.

Lemma 10. *Every SDD following Darwiche's definition can be rewritten to follow our definition in linear time. And, conversely, every SDD following our definition can be rewritten to follow Darwiche's definition in linear time.*

Proof. To go from Darwiche's definition to ours, first one repeatedly gets rid of the prime-sub pairs $(0, s_i)$ which are irrelevant to compute the function represented by the SDD, and replaces the prime-sub pairs $(1, s_i)$ by s_i. This is part of the operation called *trimming* in [17], which preserves the SDD format (in Darwiche's definition) and the function computed. Hence, we have an SDD where no prime is a constant. Next, every sub that is a literal x is replaced by $\{(x, 1), (x, 0)\}$ and every sub that is a literal $\neg x$ is replaced by $\{(x, 0), (\neg x, 1)\}$.

To go from our definition to Darwiche's definition, it is sufficient to replace the decomposition nodes $\{(x, 1), (\neg x, 0)\}$ by x, the decomposition nodes $\{(x, 0), (\neg x, 1)\}$ by $\neg x$, the decomposition nodes $\{(x, 0), (\neg x, 0)\}$ by 0, and the decomposition nodes $\{(x, 1), (\neg x, 1)\}$ by 1. These are the only decomposition nodes that are mapped to leaves of the vtree in our definition. Once they are replaced by literals or constants, all remaining decomposition nodes are mapped to internal nodes of the vtree by λ. The result is an SDD respecting Darwiche's definition for the mapping λ' that is consistent with λ on decomposition nodes, and that maps literals for x to $\phi^{-1}(x)$, and that maps constants to some child of their parent decomposition node. □

Since the difference between the two definition amount to a linear-time rewriting, all properties of SDD advertised by Darwiche carry on to our settings. In particular, all polynomial-time transformations involving two SDDs respecting a common vtree (conjunction, disjunction, etc.) are still feasible in polynomial time in our settings, and the canonicity of so-called compressed and trimmed SDDs in Darwiche settings (that is, two compressed and trimmed SDDs are equivalent if and only if they are syntactically identical) can be rephrased under our definition.

Now we move to SDD-width. There have already been attempts at defining the SDD-width of functions [9], or the SDD-width of functions relative to a vtree [17]. We believe that the width of an SDD should be defined as a parameter of an SDD independently of the function it represents. This would be consistent with how the width of an OBDD is defined.

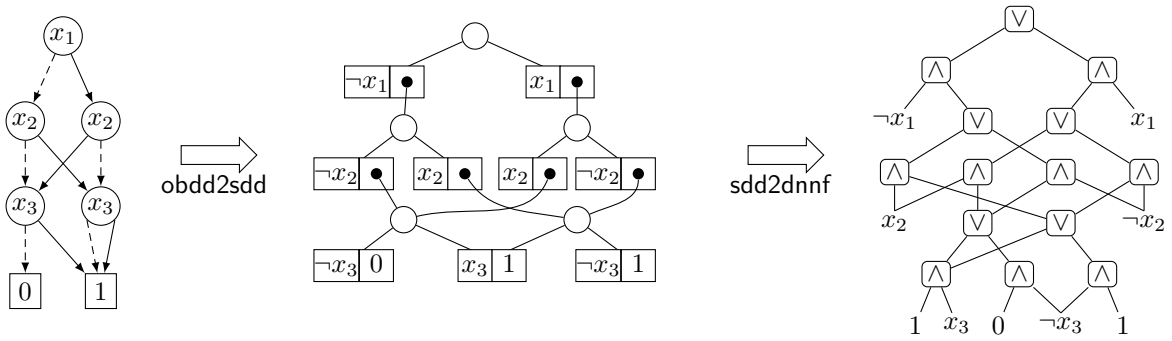


Figure 7: Standard rewriting from OBDD to SDD to DNNF circuit

Definition 3 (Width of an SDD)**.** Let S be an SDD structured by the vtree $\mathcal{T} = (T, \phi)$. The *width* of S is $width(S) := \max_{t \in T} |\lambda^{-1}_{S,\mathcal{T}}(t)|$.

For instance, the SDD of Figure 4 has width 2 since two nodes are mapped to the node ③ of Figure 6, and only one is mapped to node ①. Note that no node is mapped to ②. Next, the SDD-width of a function is naturally defined as the smallest width of an SDD representing that function.

Definition 4 (SDD-Width of functions)**.** Let f be a Boolean function, the *SDD-width* of f is the minimum width of an SDD computing f.

4.2 Relations and Translations between OBDDs, SDDs and DNNF circuits

Here we recall the standard rewriting of OBDDs into SDDs and of SDDs into DNNF circuits.

OBDDs to SDDs. OBDDs are easily rewritten as SDDs respecting *linear vtrees*. A vtree (T, ϕ) is (right-)linear when, for every internal node t of T with children t_ℓ and t_r, we have that t_ℓ is a leaf. The internal nodes of a linear vtree are organized on a path. Given a variable ordering π of the set X of variables, $\mathcal{T}_\pi$ is the right-linear vtree over X such the left-to-right ordering of the variables on its leaves is precisely π. For B a π-OBDD, $\mathsf{obdd2sdd}(B)$ returns an SDD respecting $\mathcal{T}_\pi$ defined as follows

- $\mathsf{obdd2sdd}(0\text{-sink}) =$ constant 0 node
- $\mathsf{obdd2sdd}(1\text{-sink}) =$ constant 1 node
- $\mathsf{obdd2sdd}(ite(x, B_1, B_0)) = \{(x, \mathsf{obdd2sdd}(B_1)), (\neg x, \mathsf{obdd2sdd}(B_0))\}$

Caching is implicit in $\mathsf{obdd2sdd}$. For instance, the OBDD $ite(x, B_1, ite(y, B_1, B_0))$ is turned into $\{(x, \mathsf{obdd2sdd}(B_1)), (\neg x, \{(y, \mathsf{obdd2sdd}(B_1)), (\neg y, \mathsf{obdd2sdd}(B_0))\})\}$ where the two occurrences of $\mathsf{obdd2sdd}(B_1)$ refer to the same SDD. $\mathsf{obdd2sdd}$ creates one SDD decomposition node per OBDD decision node, so the size of $\mathsf{obdd2sdd}(B)$ is in $O(|B|)$.

SDD to DNNF. SDDs can be seen as particular DNNF circuits. For S an SDD, $\mathsf{sdd2dnnf}(S)$ is a DNNF circuit defined as follows

- $\mathsf{sdd2dnnf}(c) = c$ when $c \in \{0, 1\}$ is a constant node
- $\mathsf{sdd2dnnf}(\ell) = \ell$ when $\ell \in \{x, \neg x\}$ is a literal
- $\mathsf{sdd2dnnf}(\{(p_1, s_1), \ldots, (p_m, s_m)\}) = \bigvee_{i=1}^{m} \mathsf{sdd2dnnf}(p_i) \wedge \mathsf{sdd2dnnf}(s_i)$

Again, we have left caching implicit.

4.3 Complete and Smooth SDDs

To the best of our knowledge, complete and smooth SDDs have not been defined before.
textcolorblackLike with OBDDs, completeness implies smoothness and the reverse implication holds when there are no irrelevant variables.

Definition 5 (smooth SDD)**.** An SDD is *smooth* when, all its decomposition nodes $\{(p_1, s_1), \ldots, (p_m, s_m)\}$ satisfy $var(p_1) = \cdots = var(p_m)$ and $var(s_1) = \cdots = var(s_m)$.

Definition 6 (complete SDD)**.** An SDD S that respects $\mathcal{T} = (T, \phi)$ is *complete* when the source node of S is mapped to the root of T by $\lambda_{S,\mathcal{T}}$ and when, for every decomposition nodes η,

- either η is of the form $\{(x, c_1), (\neg x, c_0)\}$ with c_1 and c_0 two constant nodes and $\lambda_{S,\mathcal{T}}(\eta) = \phi^{-1}(x)$,
- or η is $\{(p_1, s_1), \ldots, (p_m, s_m)\}$ where some p_i or s_i is a decomposition node and $\lambda_{S,\mathcal{T}}(\eta)$ is an internal node t of T and we have that $s_1, \ldots, s_m$ are all mapped to the right child of t and $p_1, \ldots, p_m$ are all mapped to the left child of t.

Proposition 4. *Let f be a Boolean function over X and $\mathcal{T} = (T, \phi)$ be a vtree over X. Suppose that every variable $x \in X$ is relevant in f. Then every smooth SDD S representing f is also complete.*

Proof. Since all variables are relevant in f, the source node of S is mapped to the source node of T by $\lambda_{S,\mathcal{T}}$. By assumption there is a decomposition node $\eta = \{(p_1, s_1), \ldots, (p_m, s_m)\}$ such that $\lambda_{S,\mathcal{T}}(\eta) = t$ and some p_i is not mapped by $\lambda_{S,\mathcal{T}}$ to the left child t_ℓ of t or some s_i is not mapped by $\lambda_{S,\mathcal{T}}$ to the right child t_r of t. It follows that $var(p_i) \cup var(s_i) \subsetneq var(t_\ell) \cup var(t_r) = var(t)$ for some i. By smoothness, we thus have the strict inclusion $var(\eta) \subsetneq var(t)$. Let $(\eta_1, \ldots, \eta_k = \eta)$ be a sequence of decomposition nodes with η_1 the source node of S and where each η_i, $i > 1$, is a sub or a prime of η_{i-1}; we denote by η'_i the prime or sub of η_{i-1} paired with η_i. Smoothness ensures that $var(\eta_{i-1}) = var(\eta_i) \cup var(\eta'_i)$. Since $var(\lambda_{S,\mathcal{T}}(\eta_i)) \cap var(\eta'_i) = \emptyset$ we find that if $x \in var(\lambda_{S,\mathcal{T}}(\eta_i)) \setminus var(\eta_i)$, then $x \notin var(\eta_{i-1})$ and thus $x \in var(\lambda_{S,\mathcal{T}}(\eta_{i-1})) \setminus var(\eta_{i-1})$. But then, knowing that $var(\eta) = var(\eta_k) \subsetneq var(t)$, we conclude that there is an $x \in var(t)$ that is not in $var(\eta_1)$. Since η_1 is the source node of S, it follows that S represents a function for which x is irrelevant, a contradiction. □

Thus smoothness and completeness are equivalent for functions without irrelevant variables and generally, the minimal width of a smooth SDD representing a function (not uniformly 0 or 1) equals the minimal width of a complete SDD representing the same function. In this section we prefer working with the notion of smooth SDD.

The reason our definition of SDD differs from Darwiche's definition is so that we can define textcolorblacksmooth and complete SDD for every function in a satisfactory way. Formally, we want the following to hold.

(1) Feeding a
textcolorblacksmooth OBDD to obdd2sdd yields a
textcolorblacksmooth SDD.

(2) Feeding a
textcolorblacksmooth SDD to sdd2dnnf yields a
textcolorblacksmooth DNNF circuit.

To see the small issue with the original definition of SDDs, consider the function $a \wedge b$. The only way to represent it as an SDD respecting the vtree of Figure 6 following Darwiche's definition is $\{(a, b), (\neg a, 0)\}$. One can check to sdd2dnnf$(\{(a, b), (\neg a, 0)\})$ is not a
textcolorblacksmooth DNNF. Naturally, one wants to replace the 0 in this SDD by $\{(b, 0), (\neg b, 0)\}$ and then sdd2dnnf would return a
textcolorblacksmooth DNNF. But this is not compatible with Darwiche's definition because this definition does not allow that $\{(b, 0), (\neg b, 0)\}$ respects the vtree node ③. As a decomposition node,

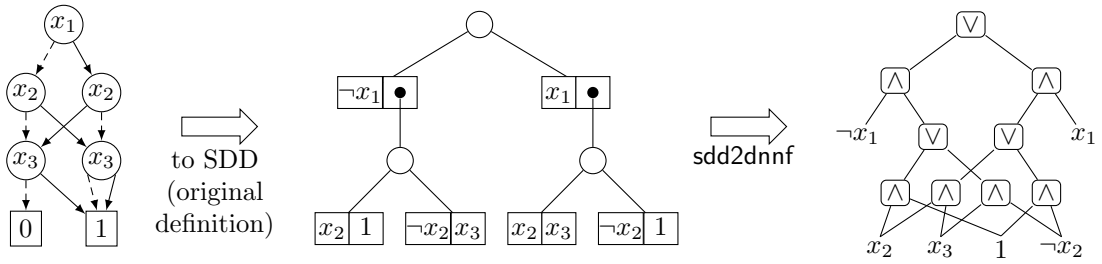


Figure 8: An example of loss of completeness when using the original definition of SDD by Darwiche

$\{(b, 0), (\neg b, 0)\}$ has to respect an internal node of the vtree, but it does not respect the only internal node in this case, namely, ①. Another example is given in Figure 8. Here we have written a textcolorblacksmooth OBDD as an SDD following Darwiche's definition and next applied sdd2dnnf to obtain a DNNF circuit that is not textcolorblacksmooth. We believe that if B is a textcolorblacksmooth OBDD, then $\mathsf{sdd2dnnf}(\mathsf{obdd2sdd}(B))$ should be a textcolorblacksmooth DNNF. With our definition of SDDs, the same textcolorblacksmooth OBDD is turned into a textcolorblacksmooth SDD which is turned into a textcolorblacksmooth DNNF as can be verified from Figure 7.

Lemma 11. *Let B be a textcolorblacksmooth OBDD, then $\mathsf{obdd2sdd}(B)$ is a textcolorblacksmooth SDD representing the same function. Let S be a textcolorblacksmooth SDD, then $\mathsf{sdd2dnnf}(S)$ is a textcolorblacksmooth DNNF circuit representation the same function.*

Proof. For obdd2sdd, the lemma is immediate if $B = c$-sink for $c \in \{0, 1\}$. Suppose the lemma holds for B_1 and B_0, the textcolorblacksmoothness of B implies $var(B_0) = var(B_1)$. By induction the variable sets are preserved so $var(\mathsf{obdd2sdd}(B_1)) = var(\mathsf{obdd2sdd}(B_0))$. The induction also ensures that $\mathsf{obdd2sdd}(B_1)$ and $\mathsf{obdd2sdd}(B_0)$ are textcolorblacksmooth, so $\mathsf{obdd2sdd}(B)$ is textcolorblacksmooth and computes $(x \wedge \langle \mathsf{obdd2sdd}(B_1) \rangle) \vee (\neg x \wedge \langle \mathsf{obdd2sdd}(B_0) \rangle)$, which by induction is $(x \wedge B_1) \vee (\neg x \wedge B_0)$.

For sdd2dnnf, the lemma is immediate for constants and literals. Let $S = \{(p_1, s_1), \ldots, (p_m, s_m)\}$ and suppose the lemma holds for every p_i and s_i. The textcolorblacksmoothness of S implies $var(p_1) = \cdots = var(p_m)$ and $var(s_1) = \cdots = var(s_m)$. Since sdd2dnnf preserves the variable set, we have that $var(\mathsf{sdd2dnnf}(p_i) \wedge \mathsf{sdd2dnnf}(s_i)) = var(\mathsf{sdd2dnnf}(p_j) \wedge \mathsf{sdd2dnnf}(s_j))$ for every $i \neq j$. By induction all $\mathsf{sdd2dnnf}(p_i)$ and $\mathsf{sdd2dnnf}(s_i)$ are textcolorblacksmooth, so $\mathsf{sdd2dnnf}(S)$ is textcolorblacksmooth and computes $\bigvee_{i=1}^{m} \mathsf{sdd2dnnf}(p_i) \wedge \mathsf{sdd2dnnf}(s_i)$, which by induction is $\bigvee_{i=1}^{m} \langle p_i \rangle \wedge \langle s_i \rangle = \langle S \rangle$. □

Similarly to OBDD and DNNF, an SDD can be made textcolorblacksmooth such that the width of the textcolorblacksmooth SDD is at most the number of variables times the width of the initial SDD (times a constant factor). To create a procedure textcolorblacksmooth, we first need the procedure negate, that returns the negation of an SDD. $\mathsf{negate}(S)$ starts from the root of S and proceeds as follows, using caching implicitly.

- $\mathsf{negate}(0) = 1$

- $\mathsf{negate}(1) = 0$
- $\mathsf{negate}(\{(p_1, s_1), \ldots, (p_k, s_k)\}) = \{(p_1, \mathsf{negate}(s_1)), \ldots, (p_k, \mathsf{negate}(s_k))\}$

Note that subs are either constants or decomposition node, so the input of negate is never a literal.

Lemma 12. *For S an SDD structured by $\mathcal{T} = (T, \phi)$, $\mathsf{negate}(S)$ returns an SDD over $var(S)$, equivalent to $\neg\langle S\rangle$, structured by $\mathcal{T}$. Moreover, the $\mathsf{negate}(S)$ has size at most $2|S|$ and of width at most $2 \cdot width(S)$. Finally, if S is*
textcolorblacksmooth, then so is $\mathsf{negate}(S)$.

Proof. By induction on the size of S. The base case is when S is a constant; this case is clear. For the inductive case, let $\{(p_1, s_1), \ldots, (p_k, s_k)\}$ be the root node of S. We have $\mathsf{negate}(S) = \{(p_1, \mathsf{negate}(s_1)), \ldots, (p_k, \mathsf{negate}(s_k))\}$. The condition on the primes holds trivially. By induction $var(\mathsf{negate}(s_i)) = var(s_i)$ and $\mathsf{negate}(s_i)$ respects the same vtree as s_i, so $\mathsf{negate}(\{(p_1, s_1), \ldots, (p_k, s_k)\})$ is an SDD that respects the same vtree as S.

Now, we have $\langle \mathsf{negate}(\{(p_1, s_1), \ldots, (p_k, s_k)\})\rangle = \bigvee_{i=1}^{k} \langle p_i\rangle \wedge \langle \mathsf{negate}(s_i)\rangle = \bigvee_{i=1}^{k} \langle p_i\rangle \wedge \neg\langle s_i\rangle$, where the last equality holds by induction. Since $\bigvee_{i=1}^{k} \langle p_i\rangle = 1$ and $\langle p_i\rangle \wedge \langle p_j\rangle = 0$ for every $i \neq j$, we have that $\bigvee_{i=1}^{k} \langle p_i\rangle \wedge \neg\langle s_i\rangle = \neg \bigvee_{i=1}^{k} \langle p_i\rangle \wedge \langle s_i\rangle = \neg\langle S\rangle$.

Next, for the size and the width, observe that each node in $\mathsf{negate}(S)$ is an node η originally in S, or a node $\mathsf{negate}(\eta)$ for η a node originally in S. Thus $\mathsf{negate}(\eta)$ has size most twice as many nodes as S. Since $var(\eta) = var(\mathsf{negate}(\eta))$, $\lambda_{\mathsf{negate}(S),\mathcal{T}}$ maps both η and $\mathsf{negate}(\eta)$ to the same node of T. Hence $width(\mathsf{negate}(S)) \leq 2 \cdot width(S)$.

Finally,
textcolorblacksmoothness is preserved due to negate preserving the variable set. □

Next, we describe textcolorblacksmooth, a procedure that takes in an SDD S and a vtree $\mathcal{T} = (T, \phi)$ over $X \supseteq var(S)$ such that S respects $\mathcal{T}$, and returns a
textcolorblacksmooth SDD over X, structured by $\mathcal{T}$, and that computes the function over X that accepts an assignment α if and only if the restriction of α to S is accepted by $\langle S\rangle$. For convenience, we write $\mathsf{textcolorblacksmooth}(S, t)$ for $\mathsf{textcolorblacksmooth}(S, \mathcal{T}|_t)$. Let us deal with constant nodes first.

- if t is a leaf with $\phi(t) = x$, then

$$\mathsf{textcolorblacksmooth}(1, t) = \{(x, 1), (\neg x, 1)\}$$
$$\mathsf{textcolorblacksmooth}(0, t) = \{(x, 0), (\neg x, 0)\}$$

- if t is an internal node with children t_ℓ and t_r, then

$$\mathsf{textcolorblacksmooth}(1, t) = \{(\mathsf{textcolorblacksmooth}(1, t_\ell),\ \mathsf{textcolorblacksmooth}(1, t_r))\}$$
$$\mathsf{textcolorblacksmooth}(0, t) = \{(\mathsf{textcolorblacksmooth}(1, t_\ell), \mathsf{textcolorblacksmooth}(0, t_r))\}$$

Now, in the case where S is not a constant node. Let $\eta = \{(p_1, s_1), \ldots, (p_k, s_k)\}$ be S's root node, let t be the root of T and let $t^* = \lambda_{S,\mathcal{T}}(\eta)$.

- If t is a leaf of T, then $\eta = \{(x, c_1), (\neg x, c_0)\}$ for some variable x and some $c_0, c_1 \in \{0, 1\}$. In that case $\mathsf{textcolorblacksmooth}(\eta, t)$ returns η.
- If t is an internal node of T with children t_ℓ and t_r, then we have four cases.
  1. If t^* is t_ℓ or a descendant of t_ℓ, then we compute $\eta' = \mathsf{textcolorblacksmooth}(S, t_\ell)$, $S_1 = \mathsf{textcolorblacksmooth}(1, t_r)$ and $S_0 = \mathsf{textcolorblacksmooth}(0, t_r)$ and return

$$\eta_1 = \{(\eta', S_1), (\mathsf{negate}(\eta'), S_0)\}$$

2. If t^* is t_r or a descendant of t_r, then we compute $\eta' = \mathsf{textcolorblacksmooth}(S, t_r)$, $S_1 = \mathsf{textcolorblacksmooth}(1, t_\ell)$ and return

$$\eta_2 = \{(S_1, \eta')\}$$

3. If $t^* = t$ and t_ℓ is not a leaf, then let $p'_i = \{(x, 1), (\neg x, 0)\}$ if $p_i = x$, $p'_i = \{(\neg x, 1), (x, 0)\}$ if $p_i = \neg x$ and $p'_1 = p_i$ otherwise. Return

$$\eta_3 = \{(\mathsf{textcolorblacksmooth}(p'_1, t_\ell), \mathsf{textcolorblacksmooth}(s_1, t_r)), \ldots, (\mathsf{textcolorblacksmooth}(p'_k, t_\ell), \mathsf{textcolorblack}$$

4. If $t^* = t$ and t_ℓ is a leaf with $\phi(t_\ell) = x$, then let $p'_i = \{(x, c), (\neg x, c)\}$ if p_i is the constant node c and $p'_i = p_i$ otherwise. Return

$$\eta_4 = \{(p'_1, \mathsf{textcolorblacksmooth}(s_1, t_r)), \ldots, (p'_k, \mathsf{textcolorblacksmooth}(s_k, t_r))\}$$

Lemma 13. *Let $c \in \{0, 1\}$, and $\mathcal{T} = (T, \phi)$ be a vtree. $\mathsf{textcolorblacksmooth}(c, \mathcal{T})$ returns an SDD structured by $\mathcal{T}$, with variable set $var(\mathcal{T})$, of size $O(|var(\mathcal{T})|)$, of width at most 2, and that computes the constant c function.*

Proof. Two SDD nodes are created per node t of T namely, $\mathsf{textcolorblacksmooth}(0, t)$ and $\mathsf{textcolorblacksmooth}(1, t)$. Thus $\mathsf{textcolorblacksmooth}(c, \mathcal{T})$ contains $O(|var(\mathcal{T})|)$ nodes. We prove the other properties by induction on the depth of T. When $t = root(T)$ is a leaf with $\phi(t) = x$, then by definition $var(\mathsf{textcolorblacksmooth}(c, \mathcal{T}))$ is an SDD respecting $\mathcal{T}$ with variable set $\{x\}$ and that computes c. When t has children t_ℓ and t_r, then by induction $\mathsf{textcolorblacksmooth}(1, t_\ell)$ is an SDD structured by $\mathcal{T}|_{t_\ell}$, with variable set $var(\mathcal{T}|_{t_\ell})$ and that computes the constant 1 function; whereas $\mathsf{textcolorblacksmooth}(c, \mathcal{T}|_{t_r})$ is an SDD structured by $\mathcal{T}|_{t_r}$, with variable set $var(\mathcal{T}|_{t_r})$ and that computes the constant c function. Thus, $\mathsf{textcolorblacksmooth}(c, \mathcal{T}) = \{(\mathsf{textcolorblacksmooth}(1, t_\ell), \mathsf{textcolorblacksmooth}(c, t_r)\}$ is an SDD that respects $\mathcal{T}$, with variable set $var(\mathcal{T}|_{t_\ell}) \cup var(\mathcal{T}|_{t_r}) = var(\mathcal{T})$ and that computes the function $1 \wedge c = c$. Finally, for the width, we have that $\lambda(\mathsf{textcolorblacksmooth}(1, t)) = t$, so only two nodes can be map to any given node of T, hence $width(\mathsf{textcolorblacksmooth}(c, \mathcal{T})) \leq 2$. □

Lemma 14. *Let S be an SDD respecting $\mathcal{T} = (T, \phi)$ and which root node is not a constant node. There is a smooth SDD structured by $\mathcal{T}$, with variable set $var(\mathcal{T})$, of size $O(n|S|)$, of width at most $2n \cdot width(S)$, and that computes the same function as S.*

Proof. The smooth SDD is be the output S' of $\mathsf{textcolorblacksmooth}(S, \mathcal{T})$. We show by induction on T's depth that S' is an SDD, that it is
textcolorblacksmooth, that it is structured by $\mathcal{T}$, that $var(S') = X = var(\mathcal{T})$ and that $\langle S' \rangle$ is the function that maps assignments to X to the value returned by $\langle S \rangle$ on their restrictions to $var(S)$. Let $t = root(T)$. We already know by Lemma 13 that S_1 and S_0 verify this claim.

Base case. If t is a leaf with $\phi(t) = x$, then $S = \{(x, c_1), (\neg x, c_0)\}$, for $c_0, c_1 \in \{0, 1\}$ and $\mathsf{textcolorblacksmooth}(S, t) = S$. Here it is clear that the lemma holds.

Inductive case. If t has two children t_ℓ and t_r then we consider the four cases.

1. The conditions on the primes of η_1 are verified: $\langle \eta' \rangle \wedge \langle \mathsf{negate}(\eta') \rangle = 0$ and $\langle \eta' \rangle \vee \langle \mathsf{negate}(\eta') \rangle = 1$. By induction, η' is
textcolorblacksmooth and $var(\eta') = var(t_l)$ and thus $var(\mathsf{negate}(\eta')) = var(t_l)$. We know that $var(S_1) = var(S_0) = var(t_r)$ (Lemma 13). So η_1 is
textcolorblacksmooth and $var(\eta_1) = var(t_l) \cup var(t_r) = var(t)$. By induction, $\langle \eta' \rangle$ computes the function $\langle S \rangle$ over $var(t_\ell)$, so $\langle \eta \rangle$ computes $\langle \eta' \rangle \wedge \langle S_1 \rangle \vee \langle \mathsf{negate}(\eta') \rangle \wedge \langle S_0 \rangle = \langle \eta' \rangle \wedge \langle S_1 \rangle = \langle \eta' \rangle$ over $var(t_\ell)$. By induction, η' and $\mathsf{negate}(\eta')$ are structured by $\mathcal{T}|_{t_\ell}$ and S_1 and S_0 are structured by $\mathcal{T}|_{t_r}$, so η_1 is structured by $\mathcal{T}|_t$.

2. The unique prime S_1 computes the constant 1 function over $var(t)$ (Lemma 13), so the partition condition of the primes is met. By induction, η' is
textcolorblacksmooth and $var(\eta') = var(t_r)$. We know that $var(S_1) = var(t_\ell)$. So η_2 is
textcolorblacksmooth and $var(\eta_2) = var(t_l) \cup var(t_r) = var(t)$. By induction, $\langle \eta' \rangle$ computes the function $\langle S \rangle$ over $var(t_r)$, so $\langle \eta \rangle$ computes $\langle \eta' \rangle \wedge \langle S_1 \rangle = \langle \eta' \rangle$ over $var(t_r)$. By induction, η' is structured by $\mathcal{T}|_{t_r}$ and S_1 is structured by $\mathcal{T}|_{t_\ell}$, so η_1 is structured by $\mathcal{T}|_t$.

3. Let $p_i'' = \mathsf{textcolorblacksmooth}(p_i', t_\ell)$ and $s_i'' = \mathsf{textcolorblacksmooth}(s_i, t_r)$. By induction, p_i'' is a textcolorblacksmooth SDD structured by $\mathcal{T}|_{t_\ell}$ that computes $\langle p_i' \rangle = \langle p_i \rangle$ over $var(t_\ell)$ and s_i'' is a textcolorblacksmooth SDD structured by $\mathcal{T}|_{t_r}$ that computes $\langle s_i \rangle$ over $var(t_r)$. Since $\langle p_i \rangle \wedge \langle p_j \rangle = 0$ for all $i \neq j$, we also have $\langle p_i'' \rangle \wedge \langle p_j'' \rangle = 0$, and $\langle p_1 \rangle \vee \cdots \vee \langle p_k \rangle = 1$ implies $\langle p_1'' \rangle \wedge \langle p_k'' \rangle = 1$. Thus, η_3 is a
textcolorblacksmooth SDD structured by $\mathcal{T}|_t$ that computes $\bigvee_{i=1}^k \langle p_i'' \rangle \wedge \langle s_i'' \rangle = \bigvee_{i=1}^k \langle p_i \rangle \wedge \langle s_i \rangle = \langle S \rangle$ over $var(t)$.

4. Since $\phi(t_\ell) = x$, p_i is either a constant node $c \in \{0, 1\}$ or a decomposition node $\{(x, c_1), (\neg x, c_0)\}$ for some constants $c_0, c_1 \in \{0, 1\}$. In both cases p_i' is a
textcolorblacksmooth SDD structured by $\mathcal{T}|_{t_\ell}$ that computes $\langle p_i \rangle$ over $var(t_r)$. Let $s_i' = \mathsf{textcolorblacksmooth}(s_i, t_r)$. By induction, s_i' is a
textcolorblacksmooth SDD structured by $\mathcal{T}|_{t_r}$ that computes $\langle s_i \rangle$ over $var(t_r)$. It follows that η_4 is a
textcolorblacksmooth SDD structured by $\mathcal{T}|_t$ that computes $\bigvee_{i=1}^k \langle p_i' \rangle \wedge \langle s_i' \rangle = \bigvee_{i=1}^k \langle p_i \rangle \wedge \langle s_i \rangle = \langle S \rangle$ over $var(t_\ell) \cup var(t_r) = var(t)$.

To finish the proof, we bound the width of S'. We dismiss decomposition nodes that are mapped by $\lambda_{S',\mathcal{T}}$ to leaves of T, as they are only at most four decomposition nodes of the form $\{(x, c_1), (\neg x, c_0)\}$, $c_0, c_1 \in \{0, 1\}$ for a fixed x. Every other node in S' is either a node η from S, or is the root of $\mathsf{textcolorblacksmooth}(\eta, t)$, or is the root of $\mathsf{negate}(\mathsf{textcolorblacksmooth}(\eta, t))$. Since the variable sets are preserved, at most $width(S)$ nodes from S are mapped to t, by $\lambda_{S',\mathcal{T}}$. To these we must add the roots of $\mathsf{textcolorblacksmooth}(\eta, t)$ and $\mathsf{negate}(\mathsf{textcolorblacksmooth}(\eta, t))$ for different η. So we have to count the maximum number of calls $\mathsf{textcolorblacksmooth}(\eta, t)$ for the same vtree node t. For t an internal vtree node, we may have computed $\mathsf{textcolorblacksmooth}(1, t)$ and $\mathsf{textcolorblacksmooth}(0, t)$, which both contribute one to the the number of nodes mapped to t in S'. Next, we count the number of calls $\mathsf{textcolorblacksmooth}(\eta, t)$ with η different from 0 and 1. A node η_3 or η_4 mapped to t is created for every node η of S with $\lambda_{S,\mathcal{T}}(\eta) = t$, so this is at most $width(S)$ nodes of the form η_3 or η_4. Nodes of the form η_1 and η_2 are created when calling $\mathsf{textcolorblacksmooth}(\eta, t)$ with η a node of S mapped to a descendant of t by $\lambda_{S,\mathcal{T}}$. There are at most $2n$ nodes in T, so we have at most $2n \cdot width(S)$ candidates for η. Thus, the number of nodes of S' mapped to t by $\lambda_{S',\mathcal{T}}$ is at most $2n \cdot width(S)$. It follows that $width(S') \leq 2n \cdot width(S)$ and $|S'| \in O(n|S|)$. □

In the following section, we will show that, similarly to OBDDs, the linear increment in the width when making SDDs
textcolorblacksmooth is sometimes unavoidable.

4.4 Width Difference between SDD and textcolorblackSmooth SDD

Definition 7 (
textcolorblackSmooth-SDD-Width of Functions)**.** The
textcolorblacksmooth-SDD-width of f is the minimum width of a
textcolorblacksmooth SDD computing f.

The point of this section is to prove Theorem 2, which is the counterpart of Theorem 1 for SDDs.

Theorem 2. *There is an infinite set $\mathcal{F}$ of Boolean functions and two constants $c \in \mathbb{N}$ and $\gamma \in (0, 1]$ such that, for every $f \in \mathcal{F}$ over n variables, the SDD-width of f is at most c and the textcolorblacksmooth-SDD-width of f is at least γn.*

4.4.1 Connecting textcolorblackSmooth-SDD-Width and Circuit Treewidth

Before proving Theorem 2, we discuss its implications on the connection between SDD-width and circuit treewidth.

Bova and Szeider [9] defined the SDD-width of a function as the width of a particular canonical SDD representing that function. They use a notion of width different from ours. First, they consider the width of a specific SDD per function and second, they essentially define the SDD-width as the maximum number of prime-sub pairs belonging to decomposition nodes mapped to a common vtree node. Formally, our width is $\max_{t \in T} |\lambda^{-1}_{S,\mathcal{T}}(t)|$ while their width is $\max_{t \in T} |\bigcup_{\eta \in \lambda^{-1}_{S,\mathcal{T}}(t)} \eta|$ where each decomposition node η is interpreted as a set of prime-sub pairs. Another way to interpret this difference is to say that if one converts the SDD to a DNNF circuit using sdd2dnnf, then our width counts $\vee$-gates of the circuit while theirs counts $\wedge$-gates. This is not a big difference since one shows, using arguments similar to that used by Capelli and Mengel [13, Observation 3], that the two widths w (ours) and w' (theirs) verify $w' \leq O(w^2)$.

The canonical SDDs of Bova and Szeider are defined in a recursive manner. They map nodes of their SDD to the vtree nodes. If t is an internal node of the vtree T with left child t_ℓ and right child t_r, then all decomposition nodes η mapped to t contain only prime-sub pairs (p_i, s_i) where the prime is a node mapped to t_r and the sub is a node mapped to t_ℓ. And if t is a leaf of the vtree with $\phi(t) = x$, then all nodes η mapped to v are either 0 or 1 or x or $\neg x$. To rewrite their SDD so that they fit our definition, it is sufficient to rewrite the nodes η mapped to a leaf t (with $x = \phi(t)$): if $\eta = 0$ then replace it by $\{(x, 0), (\neg x, 0)\}$, if $\eta = 1$ then replace it by $\{(x, 1), (\neg x, 1)\}$, if $\eta = x$ the replace it by $\{(x, 1), (\neg x, 0)\}$, and if $\eta = \neg x$ then replace it by $\{(x, 0), (\neg x, 1)\}$. This rewriting ensures that no sub is a literal and that no prime is a constant, so the SDD now fits our definition. In addition, a simple inductive argument now shows that the SDD is
textcolorblacksmooth. Indeed the rewriting ensures that every decomposition node η mapped to a leaf t of T verifies $var(\eta) = \{\phi(t)\}$ and, if η is a decomposition node mapped to an internal node t, then all its prime-sub pairs (p_i, s_i) verify $var(p_i) = var(t_\ell)$ and $var(s_i) = var(t_r)$ by induction and thus $var(\eta) = var(t_\ell) \cup var(t_r) = var(t)$.

Therefore, Bova and Szeider's canonical SDD are
textcolorblacksmooth SDDs in our framework, and their notion of SDD-width differs only polynomially from ours. They have shown the following.

Theorem 3 ([9, Theorem 4, Eq. (27)])**.** *If there exists a circuit for f with treewidth w, then there exists a*
textcolorblacksmooth SDD of width at most $2^{2^{w 2^{O(w)}}}$.

Let smooth-SDD(w) be the set of Boolean functions that admit
textcolorblacksmooth-SDD representations of width at most w and let CTWD(w) be the set of Boolean functions that admit circuit representations of treewidth at most w. Their results can then be summarized informally as:

$$\text{CTWD}(O(1)) = \text{smooth-SDD}(O(1)).$$

Now let SDD(w) be the set of Boolean functions that admit SDD representations of width at most w. Then Theorem 2 essentially says that

$$\text{SDD}(O(1)) \subsetneq \text{smooth-SDD}(O(1)).$$

Hence, just like completeness matters when connecting the OBDD-width of a function to its circuit pathwidth, here
textcolorblacksmoothness matters when connecting the SDD-width of a function to its circuit treewidth.

4.4.2 Proof of Theorem 2

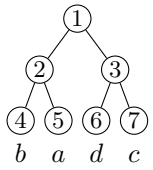
The proof of Theorem 2 is very similar to that of Theorem 1, in that we use the same function, but it requires some preliminaries on techniques to prove lower bounds on the SDD-width of a Boolean function.

Definition 8 (Rectangle)**.** Let X be a set of Boolean variables and let $\Pi = (X_1, X_2)$ be a bipartition of X. A Π-*rectangle* is a function $r(X) = \rho_1(X_1) \wedge \rho_2(X_2)$ where ρ_1 and ρ_2 are Boolean functions over X_1 and X_2, respectively.

Definition 9 (Rectangle Cover)**.** Let X be a set of Boolean variables and f be a Boolean function over X. Let Π be a bipartition of X. A Π-*rectangle cover of* f is a set R of Π-rectangles such that $f = \bigvee_{r \in R} r$.

Definition 10 (Induced Bipartitions)**.** Let $\mathcal{T} = (T, \phi)$ be a vtree over X. A bipartition $\Pi = (X_1, X_2)$ over X is *induced by* $\mathcal{T}$ when there is a node $t \in T$ such that $X_i = var(t)$ for some $i \in \{1, 2\}$.

For instance, consider the following vtree over $\{a, b, c, d\}$. The partition $(\{a, b\}, \{c, d\})$ is induced by the vtree due to node ②. The partition $(\{b\}, \{a, c, d\})$ is induced by the vtree due to node ④. But the partition $(\{a, c\}, \{b, d\})$ is not induced by the vtree.



In the next subsection we will prove the following.

Lemma 15. *Let S be a*
textcolorblacksmooth SDD representing a function $f(X)$ and structured by the vtree $\mathcal{T} = (T, \phi)$ over $var(S)$. Then there is a bipartition Π of X induced by $\mathcal{T}$ such that f has a Π-rectangle partition of size $width(S)$.

For now, we use Lemma 15 as a tool to prove Theorem 2. Recall the function

$$f(U, V, W) \quad = \quad \bigvee_{i=1}^{n} ((w_1 w_2 \ldots w_{i-1} \neg w_i) \wedge (u_i \leftrightarrow v_i))$$

where U, V and W are sequences of variables such that $|set(W)| = n$ and $set(W) \cap (set(U) \cup set(V)) = \emptyset$. Recall that a pair $\{u_i, v_i\}$ is split by a bipartition of (X_1, X_2) of $set(U) \cup set(V) \cup set(W)$ if $u_i \in X_1$ and $v_i \in X_2$, or if $v_i \in X_1$ and $u_i \in X_2$.

Lemma 16. *Suppose there exists a bipartition Π of $set(U) \cup set(V) \cup set(W)$ and a set $J \subseteq [n]$ such that, for all $j \in J$, Π splits $\{u_j, v_j\}$ and such that, for all $i, j \in J$, $i \neq j$, we have $\{u_i, v_i\} \cap \{u_j, v_j\} = \emptyset$. Then every Π-rectangle cover of $f(U, V, W)$ contains at least $|J|$ rectangles.*

Proof. Let $\Pi = (X_1, X_2)$. Use the definitions of a_j, a_j^1 and a_j^2 from Lemma 7. For $j, j' \in J$, $j \neq j'$, let $r = \rho_1(X_1) \wedge \rho_2(X_2)$ be a Π-rectangle in a cover of f. Suppose r is satisfied by a_j, so $\rho_1(a_j^1) = 1$ and $\rho_2(a_j^2) = 1$. Recall from Lemma 7 that, for every $j, j' \in J$, $j \neq j'$, $f(a_j^1 \cup a_{j'}^2) = 0$. It follows that r cannot be satisfied by $a_{j'}$ for otherwise we would have $\rho_1(a_{j'}^1) = 1$ and $\rho_2(a_{j'}^2) = 1$ and thus $r(a_j^1 \cup a_{j'}^2) = \rho_1(a_j^1) \wedge \rho_2(a_{j'}^2)$ would be 1. Hence, every rectangle in a cover of $f(U, V, W)$ can cover at most one model a_j for some $j \in J$. Since all a_j must be all covered by at least one rectangle and since the a_j are pairwise distinct, at least $|J|$ rectangles are necessary in the cover. □

Next, for G a graph with edge set $\{e_1, \ldots, e_m\}$, we again introduce the function $f(U_G, V_G, W)$, with $U_G = (u(e_1), \ldots, u(e_m))$ and $V_G = (v(e_1), \ldots, v(e_m))$. Here, vertices of G correspond to Boolean variables. We say that an edge e is split by a bipartition of $V(G) \cup W$ if $\{u(e), v(e)\}$ is split by that bipartition.

$$f(U_G, V_G, W) \quad = \quad \bigvee_{e_i \in E(G)} ((w_1 w_2 \dots w_{i-1} \neg w_i) \wedge (u(e_i) \leftrightarrow v(e_i))).$$

obdd2sdd preserves the width so, by Lemma 8, we already have that $f(U_G, V_G, W)$ admits representations as SDD of width $O(1)$ when G is a 3-regular graph.

Lemma 17. *For every 3-regular graph G, there exists an SDD of width $O(1)$ representing $f(U_G, V_G, W)$.*

We again use that G is an expander graph to show that, every vtree over $W \cup V(G)$ induces a cut of $V(G)$ that splits many edges.

Lemma 18. *Let $c > 0$. There is a constant $\alpha > 0$ such that, for every $(c, 3)$-expander n-vertex graph G, all*
textcolorblacksmooth SDDs representing $f(U_G, V_G, W)$ have width at least αn.

Proof. Let S be a
textcolorblacksmooth SDD structured by the vtree $\mathcal{T} = (T, \phi)$ over $V(G) \cup W$ (note that $V(G) = set(U_G) \cup set(V_G)$). Let ϕ_G be the restriction of ϕ to $\phi^{-1}(V(G))$. The restriction T_G of T is constructed as follows: first, repeatedly remove the leaves of T not mapped to $V(G)$ by ϕ, second, get rid of every node that has a single child by contracting the edge between that node and its child. $\mathcal{T}_G = (T_G, \phi_G)$ is a vtree over $V(G)$, so it is a binary decomposition tree of G. By Lemma 3, there is an edge e of T_G whose removal induces a bipartition of G's vertices that split $\Omega(n)$ of G's edges. Because G is 3-regular, a fifth of these edges are pairwise disjoint. The edges of T_G form a subset of the edges of T e originates from T, so the removal of e in T gives a bipartition of $V(G) \cup W$ that split $\Omega(n)$ pairwise disjoint edges of G. It follows from Lemmas 16 and 15 that $width(S) = \Omega(n)$. □

Theorem 2 follows directly from Lemmas 17 and 18. It only remains to prove Lemma 15.

4.4.3 Proof of Lemma 15

We construct rectangles from an SDD using functions called *certificates* that capture the variable assignments accepted by the SDD. The notion of certificates for DNNF circuits has been used in [8] to derive a variant of Lemma 15 for DNNF circuits. We use the same concepts and the same proof ideas adapted to SDDs.

Definition 11 (Certificates)**.** Let S be an SDD and $X = var(S)$. A *certificate* C of S is a pair (T, ψ) with T a binary tree and ψ a mapping from T to $nodes(S) \cup lit(X) \cup \{0, 1\}$ that verifies the following

- the root of T is mapped to the root node of S
- for all $t \in T$ such that $\psi(t)$ is the decomposition node $\{(p_1, s_1), \dots, (p_k, s_k)\}$, t has two children t_ℓ and t_r and there is an i between 1 and k such that $\psi(t_\ell) = p_i$ and $\psi(t_r) = s_i$.
- if $\psi(t)$ is a literal or a constant, then t is a leaf

We write $lit(C) = \psi^{-1}(lit(X))$ the set of literals mapped to leaves of T, $var(C) = \{var(l) \mid l \in lit(C)\}$ the set of variables mapped to leaves of T, and $constant(C) = \psi^{-1}(\{0, 1\})$ the set of constants mapped to leaves of T. By construction, $var(C) \subseteq var(\psi(root(T))$. We define the function represented by c as

$$\langle C \rangle = \bigwedge_{l \in lit(C)} l$$

The set of certificates of S is denoted by $cert(S)$. For convenience, we extend the notion of certificate to literals and constant. The only certificate of a literal l (resp. a constant c) is (T, ψ) where T is made of a single node t and ψ maps t to l (resp. c).

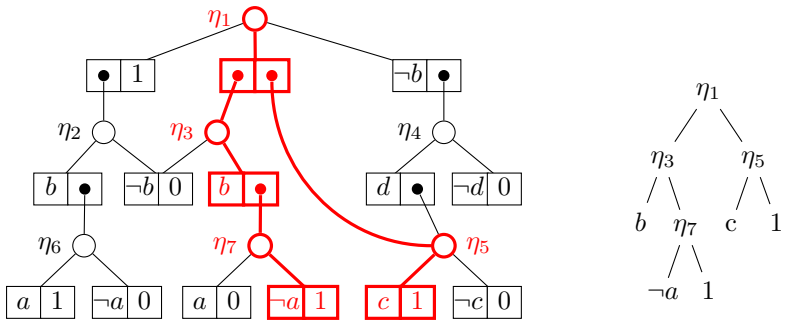


Figure 9

A graphical way to construct certificates is to start from S's root and, whenever we are at a decomposition node, to choose a single prime-sub pair (p, s) of that node and proceed recursively on p and s.

Example 2. Figure 9 shows a certificate (as a labeled binary tree) for the SDD of Figure 3. This certificate represents the function $b \wedge \neg a \wedge c$. The trace of the graphical construction for this certificate is highlighted in red in the SDD.

All certificates that contain the constant 0 represent the null function. Technically we could focus only on the 1-certificates, that is, the certificates for which $const(C) = \{1\}$, but this is not necessary for our results. Let $C \in cert(S)$. Following notations used so far, for $t \in T$ we write $C|_t = (T|_t, \psi|_t)$, where $\psi|_t$ is the restriction of ψ to the nodes of $T|_t$. It follows from Definition 11 that $C|_t$ is a certificate of $\psi(t)$. Certificates are interchangeable in the sense that "replacing $C|_t$ in C by any other certificate of $\psi(t)$" yields a certificate of S. Formally, let $C' = (T', \psi')$ is a certificate of $\psi(t)$, let T^* be the binary tree obtained by replacing $T|_t$ by T' in T, and let ψ^* be the mapping defined by $\psi^*(\tau) = \psi'(\tau)$ if $\tau \in T'$ and $\psi^*(\tau) = \psi(\tau)$ otherwise. Then (T^*, ψ^*) is a certificate of S.

Let $\eta = \{(p_1, s_1), \ldots, (p_k, s_k)\}$. For a fixed i, let $C_p = (T_p, \psi_p)$ be a certificate of p_i and let $C_s = (T_s, \psi_s)$ be a certificate of s_i. Let $merge(C_p, C_s)$ be the pair (T, ψ) where T is the binary tree whose root t has T_p for its leftsubtree and T_l for its right subtree, and where ψ maps t to η, is consistent with ψ_p on T_p, and is consistent with ψ_s on T_s. It follows from Definition 11 that (T, ψ) is a certificate of η and that

$$cert(\eta) = \bigcup_{i=1}^{k} \bigcup_{C_p \in cert(p_i)} \bigcup_{C_s \in cert(s_i)} merge(C_p, C_s)$$

Note that $\langle merge(C_p, C_s) \rangle = \langle C_p \rangle \wedge \langle C_s \rangle$. From there, we show that the set of certificates of S covers exactly the assignments accepted by S.

Lemma 19. *Let S be an SDD structured by (T, ϕ), then $\langle S \rangle = \bigvee_{C \in cert(S)} \langle C \rangle$, with the convention that this formula is 0 when $cert(S) = \emptyset$.*

Proof. By induction on the depth of T. For the base case. If T is made of one leaf t with $\phi(t) = x$, then S is made of a unique decomposition node $\{(x, c_1), (\neg x, c_0)\}$. One can check that for all four values of c_0 and c_1, $\bigvee_{C \in cert(S)} \langle C \rangle = (x \wedge c_1) \vee (\neg x \wedge c_0)$. In addition, the way we have defined $cert(l)$ for a literal l and $cert(c)$ for a constant c, allows one to check that $l = \bigvee_{C \in cert(l)} \langle C \rangle$ and that $b = \bigvee_{C \in cert(c)} \langle C \rangle$.

Next, for the inductive case, suppose S's root is a decomposition node $\eta = \{(p_1, s_1), \ldots, (p_k, s_k)\}$ and that $\phi(\eta) = t$ has children t_ℓ and t_r. We have that $\langle S \rangle = \bigvee_{i=1}^{k} \langle p_i \rangle \wedge \langle s_i \rangle$. By induction, $\langle p_i \rangle = \bigvee_{C_p \in cert(p_i)} \langle C_p \rangle$ and $\langle s_i \rangle = \bigvee_{C_s \in cert(s_i)} \langle C_s \rangle$.

$$\langle S \rangle = \bigvee_{i=1}^{k} \bigvee_{C_p \in cert(p_i)} \bigvee_{C_s \in cert(s_i)} \langle C_p \rangle \wedge \langle C_s \rangle = \bigvee_{i=1}^{k} \bigvee_{C_p \in cert(p_i)} \bigvee_{C_s \in cert(s_i)} \langle merge(C_p, C_s) \rangle = \bigvee_{i=1}^{k} \bigvee_{C \in cert(\eta)} \langle C \rangle$$

□

Lemma 20. *Let S be a*
textcolorblacksmooth SDD structured by the vtree $\mathcal{T} = (T, \phi)$ over $var(S)$. Let $C = (T', \psi')$ be a certificate of S, then $var(C) = var(S)$ and, for every $t \in T$ that is not a leaf, T' contains a node t' with $\lambda_{S,\mathcal{T}}(\psi(t')) = t$.

Proof. By induction on T's depth. When T is made of a single leaf we have that $S = \{(x, c_1), (\neg x, c_0)\}$ for some $c_1, c_0 \in \{0, 1\}$. By Definition 11, $root(T')$ is mapped to S's root by ψ' and $var(C) = \{x\} = var(S)$.

When T's root is a node t with children t_ℓ and t_r, consider the root decomposition node $\eta = \{(p_1, s_1), \ldots, (p_k, s_k)\}$ of S. We have that $\lambda_{S,\mathcal{T}}(\eta) = root(T)$ because $\mathcal{T}$ is over $var(S)$. By definition, $root(T')$ is mapped to η by ψ'. S is
textcolorblacksmooth, so $var(p_1) = \cdots = var(p_k)$ and $var(s_1) = \cdots = var(s_k)$. Since $\mathcal{T}$ is over $var(S)$ we have $var(p_i) = var(t_\ell)$ and $var(s_i) = var(t_r)$ for every i. Each s_i is a
textcolorblacksmooth SDD structured by $(T|_{t_r}, \phi|_{t_r})$ over $var(s_i)$. If t_ℓ is not a leaf of T, then $|var(t_\ell)| = |var(p_i)| \geq 2$ and thus, p_i is not a literal but a
textcolorblacksmooth SDD structured by $(T|_{t_\ell}, \phi|_{t_\ell})$ over $var(p_i)$. It follows by induction that every certificate $C'' = (T'', \psi'')$ of p_i, resp.s_i, contains a node $t'' \in T''$ with $\lambda_{S,\mathcal{T}}(\psi''(t'')) = \tau$ for every internal node τ of $T|_{t_\ell}$, resp. $T|_{t_r}$. Since the restriction of C to t'_ℓ, resp. t'_r, is a certificate for some p_i, resp. some s_i, the result follows. □

To finish this section, we provide the proof of Lemma 15.

Proof of Lemma 15. Fix t an internal node of T such that $|\lambda^{-1}_{S,\mathcal{T}}(t)| = width(S)$. Let $\Pi = (var(t), var(S) \setminus var(t))$. Let $N = \lambda^{-1}(t)$ and, for every $\eta \in N$, let Σ_η be the set of certificates of S whose tree has a node mapped to η. We claim that $\bigvee_{C \in \Sigma_\eta} \langle C \rangle$ is a Π-rectangle $r_\eta = \rho_1 \wedge \rho_2$. We prove this by constructing ρ_1 and ρ_2.

For $C \in \Sigma_\eta$, let τ be the node of C's tree mapped to η, then we call $C|_\eta$ for $C|_\tau$. Note that, by interchangeability of the certificates, if $C, C' \in \Sigma_\eta$, then replacing $C|_\eta$ by $C'|_\eta$ in C yields a certificate in Σ_η. Further, let $\langle C \setminus C|_\eta \rangle = \bigwedge_{l \in lit(C) \setminus lit(C|_\eta)} l$. Then $\langle C \rangle = \langle C|_\eta \rangle \wedge \langle C \setminus C|_\eta \rangle$. We define

$$\rho_1 = \bigvee_{C \in \Sigma_\eta} \langle C|_\eta \rangle \quad \text{and} \quad \rho_2 = \bigvee_{C \in \Sigma_\eta} \langle C \setminus C|_\eta \rangle$$

We have that $\rho_1 \wedge \rho_2 = \bigvee_{C, C' \in \Sigma_\eta} \langle C|_\eta \rangle \wedge \langle C' \setminus C'|_\eta \rangle$. By interchangeability of the certificates, for every $C, C' \in \Sigma_\eta$ there are $C'', C''' \in \Sigma_\eta$ such that $\langle C'' \rangle = \langle C|_\eta \rangle \wedge \langle C' \setminus C'|_\eta \rangle$ and $\langle C''' \rangle = \langle C'|_\eta \rangle \wedge \langle C \setminus C|_\eta \rangle$. It follows that

$$\rho_1 \wedge \rho_2 = \bigvee_{C, C' \in \Sigma_\eta} \langle C|_\eta \rangle \wedge \langle C' \setminus C'|_\eta \rangle = \bigvee_{C \in \Sigma_\eta} \langle C \rangle$$

Thus, for each node η mapped to t by λ, we have a rectangle $r_\eta = \bigvee_{C \in \Sigma_\eta} \langle C \rangle$. Next, by Lemma 20, every certificate of S contains a node mapped to t by λ. Thus,

$$\bigvee_{\eta \in N} r_\eta = \bigvee_{\eta \in N} \bigvee_{C \in \Sigma_\eta} \langle C \rangle = \bigvee_{C \in cert(S)} \langle C \rangle$$

and we conclude using Lemma 19. □

5 Conclusion

We have provided crucial clarifications and corrections regarding the relationships between knowledge representation formalisms, specifically OBDDs and SDDs of bounded width, and Boolean circuits of

bounded pathwidth and treewidth. By demonstrating that previously claimed equivalences only hold for
textcolorblacksmooth versions of OBDDs and SDDs, we reveal subtle but crucial distinctions that had been overlooked. Our rigorous proofs and counterexamples help to establish fundamental limits and connections between these widely used formalisms and provide a sound foundation for future work.

Acknowledgments

Sebastian Ordyniak acknowledges support from the Engineering and Physical Sciences Research Council (EPSRC, project EP/V00252X/1). Stefan Szeider acknowledges support from the Austrian Science Fund (FWF, projects P36420 and P36688), and from the Vienna Science and Technology Fund (WWTF, project ICT19-065).

References

[1] N. Alon and J. H. Spencer. *The Probabilistic Method, Second Edition*. John Wiley, 2000.

[2] A. Atserias, P. G. Kolaitis, and M. Y. Vardi. Constraint propagation as a proof system. In M. Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2004.

[3] D. Bergman, C. Cardonha, and S. Mehrani. Binary decision diagrams for bin packing with minimum color fragmentation. In L. Rousseau and K. Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 57–66. Springer, 2019.

[4] D. Bergman, A. A. Ciré, W. van Hoeve, and J. N. Hooker. *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016.

[5] D. Bergman and L. Lozano. Decision diagram decomposition for quadratically constrained binary optimization. *INFORMS J. Comput.*, 33(1):401–418, 2021.

[6] B. Bollig and I. Wegener. Asymptotically optimal bounds for obdds and the solution of some basic OBDD problems. *J. Comput. Syst. Sci.*, 61(3):558–579, 2000.

[7] S. Bova. SDDs are exponentially more succinct than OBDDs. In D. Schuurmans and M. P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 929–935. AAAI Press, 2016.

[8] S. Bova, F. Capelli, S. Mengel, and F. Slivovsky. Knowledge compilation meets communication complexity. In S. Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 1008–1014. IJCAI/AAAI Press, 2016.

[9] S. Bova and S. Szeider. Circuit treewidth, sentential decision, and query compilation. In E. Sallinger, J. V. den Bussche, and F. Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 233–246. ACM, 2017.

[10] R. E. Bryant. Verification of synchronous circuits by symbolic logic simulation. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects, Mathematical Science Institute Workshop, Cornall University, Ithaca, New York, USA, July 5-7, 1989, Proceedings*, volume 408 of *Lecture Notes in Computer Science*, pages 14–24. Springer, 1989.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In R. C. Smith, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*, pages 46–51. IEEE Computer Society Press, 1990.

[12] S. Buss, D. Itsykson, A. Knop, and D. Sokolov. Reordering rule makes OBDD proof systems stronger. In R. A. Servedio, editor, *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA*, volume 102 of *LIPIcs*, pages 16:1–16:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[13] F. Capelli and S. Mengel. Tractable QBF by knowledge compilation. In R. Niedermeier and C. Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany*, volume 126 of *LIPIcs*, pages 18:1–18:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[14] S. Chaki and A. Gurfinkel. BDD-based symbolic model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 219–245. Springer, 2018.

[15] A. Choi, D. Kisa, and A. Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In L. C. van der Gaag, editor, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 12th European Conference, ECSQARU 2013, Utrecht, The Netherlands, July 8-10, 2013. Proceedings*, volume 7958 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 2013.

[16] A. Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Appl. Non Class. Logics*, 11(1-2):11–34, 2001.

[17] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In T. Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 819–826. IJCAI/AAAI, 2011.

[18] G. V. den Broeck and A. Darwiche. On the role of canonicity in knowledge compilation. In B. Bonet and S. Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 1641–1648. AAAI Press, 2015.

[19] M. Grohe and D. Marx. On tree width, bramble size, and expansion. *J. Comb. Theory, Ser. B*, 99(1):218–228, 2009.

[20] T. Heß, S. N. Semmler, C. Sundermann, J. Torán, and T. Thüm. Towards deterministic compilation of binary decision diagrams from feature models. In M. Cordy, D. Strüber, M. Pinto, I. Groher, D. Dhungana, J. Krüger, J. A. Pereira, M. Acher, T. Thüm, M. H. ter Beek, J. Galasso-Carbonnel, P. Arcaini, M. R. Mousavi, X. Tërnava, J. Galindo, T. Yue, L. Fuentes, and J. M. Horcas, editors, *Proceedings of the 28th ACM International Systems and Software Product Line Conference - Volume A, SPLC 2024, Dommeldange, Luxembourg, September 2-6, 2024*, pages 136–147. ACM, 2024.

[21] D. Itsykson, A. Knop, A. E. Romashchenko, and D. Sokolov. On obdd-based algorithms and proof systems that dynamically change the order of variables. *J. Symb. Log.*, 85(2):632–670, 2020.

[22] A. K. Jha and D. Suciu. On the tractability of query compilation and bounded treewidth. In A. Deutsch, editor, *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 249–261. ACM, 2012.

[23] H. Liaw and C. Lin. On the OBDD-representation of general boolean functions. *IEEE Trans. Computers*, 41(6):661–664, 1992.

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

[25] M. Mendonça, A. Wasowski, K. Czarnecki, and D. D. Cowan. Efficient compilation techniques for large scale feature models. In Y. Smaragdakis and J. G. Siek, editors, *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, pages 13–22. ACM, 2008.

[26] U. Oztok and A. Darwiche. A top-down compiler for sentential decision diagrams. In Q. Yang and M. J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3141–3148. AAAI Press, 2015.

[27] R. Peharz, S. Tschiatschek, F. Pernkopf, and P. M. Domingos. On theoretical properties of sum-product networks. In G. Lebanon and S. V. N. Vishwanathan, editors, *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2015, San Diego, California, USA, May 9-12, 2015*, volume 38 of *JMLR Workshop and Conference Proceedings*. JMLR.org, 2015.

[28] H. Poon and P. M. Domingos. Sum-product networks: A new deep architecture. In *IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops, Barcelona, Spain, November 6-13, 2011*, pages 689–690. IEEE Computer Society, 2011.

[29] A. Shih, G. V. den Broeck, P. Beame, and A. Amarilli. Smoothing structured decomposable circuits. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11412–11422, 2019.

[30] T. Thüm. A BDD for linux?: the knowledge compilation challenge for variability. In R. E. Lopez-Herrejon, editor, *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, pages 16:1–16:6. ACM, 2020.

[31] W. van Hoeve. Graph coloring with decision diagrams. *Math. Program.*, 192(1):631–674, 2022.

[32] M. Vatshelle. *New Width Parameters of Graphs*. PhD thesis, Department of Informatics, University of Bergen, 2012.

[33] I. Wegener. The size of reduced obdds and optimal read-once branching programs for almost all boolean functions. In J. van Leeuwen, editor, *Graph-Theoretic Concepts in Computer Science, 19th International Workshop, WG '93, Utrecht, The Netherlands, June 16-18, 1993, Proceedings*, volume 790 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 1993.

[34] I. Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.

[35] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of bdd-based model checking. In G. Gopalakrishnan and P. J. Windley, editors, *Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Palo Alto, California, USA, November 4-6, 1998, Proceedings*, volume 1522 of *Lecture Notes in Computer Science*, pages 255–289. Springer, 1998.