



A canonical generalization of OBDD

Florent Capelli  

Univ. Artois, CNRS, UMR 8188, CRIL, F-62300 Lens, France

YooJung Choi  

Arizona State University

Stefan Mengel  

Univ. Artois, CNRS, UMR 8188, CRIL, F-62300 Lens, France

Martín Muñoz  

Univ. Artois, CNRS, UMR 8188, CRIL, F-62300 Lens, France

Guy Van den Broeck  

University of California, Los Angeles

Abstract

We introduce Tree Decision Diagrams (TDD) as a model for Boolean functions that generalizes OBDD. They can be seen as a restriction of structured d-DNNF; that is, d-DNNF that respect a vtree T . We show that TDDs enjoy the same tractability properties as OBDD, such as model counting, enumeration, conditioning, and apply, and are more succinct. In particular, we show that CNF formulas of treewidth k can be represented by TDDs of FPT size, which is known to be impossible for OBDD. We study the complexity of compiling CNF formulas into deterministic TDDs via bottom-up compilation and relate the complexity of this approach with the notion of factor width introduced by Bova and Szeider.

2012 ACM Subject Classification Computing methodologies → Knowledge representation and reasoning; Theory of computation → Constraint and logic programming

Keywords and phrases Boolean functions, Model counting, Knowledge Compilation

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.9

Related Version A full version of this paper with every proof can be found on arXiv [15].

Full Version: <https://arxiv.org/abs/2604.05537>

Funding *Florent Capelli:* ANR-25-CE23-3078 CERADOC

YooJung Choi: AFOSR grant FA9550-25-1-0320

Stefan Mengel: ANR-25-CE23-3078 CERADOC

Guy Van den Broeck: DARPA ANSR program under award FA8750-23-2-0004.

1 Introduction

Knowledge compilation is the systematic study of different representations of knowledge, often in the form of Boolean functions, but also for preferences [23], actions in planning [28], product configuration [36, 7], databases [25], etc. To compare different data structures representing the same type of data, following the groundbreaking work of Darwiche and Marquis [21], one analyzes them with respect to a list of potential desirable properties that they might have, generally a set of tractable operations and queries on them. There is a general observed trade-off between usefulness (what can one do efficiently with the data structure?) and succinctness (how small is the representation in a specific form?): on one end of the spectrum, there are representations like OBDD that allow many useful operations but are rather verbose. On the other end, there are, e.g., DNNF that are far more succinct but allow only a few operations efficiently. Knowledge compilation explores the space between

43 these two extremes and aims to provide representation languages with different trade-offs for
 44 different applications.

45 An important operation in knowledge compilation is the *apply* operation which, given
 46 two representations of the same format, computes a representation of a target Boolean
 47 combination: most importantly, their conjunction. OBDD [14] and SDD [20] are the most
 48 prominent knowledge compilation languages that support it. The apply operation is of special
 49 practical importance because it is often used as the basis of *bottom-up* compilation of systems
 50 of constraints into a different target representation. The idea is to first compile the individual
 51 constraints into the target format and then iteratively conjoin them with the apply operation.
 52 In particular, this is the most common approach for constructing OBDD [35] and SDD [16].
 53 To avoid size blow-ups during bottom-up compilation, it is common to try to shrink the
 54 currently compiled form, which for OBDD is possible because they can be turned into a
 55 canonical minimal form, i.e., a form of minimal size and unique, up to isomorphism, among
 56 all equivalent OBDDs with the same variable order. Canonicity is also useful to efficiently
 57 test equivalence between two given OBDDs. For SDD, which are in general exponentially
 58 smaller than OBDD [11], the situation is more complicated [38]: while they have a canonical
 59 form, it is not minimal and can, in fact, be exponentially larger than the smallest equivalent
 60 SDD. Moreover, canonical SDDs are not stable under conjunction, as the conjunction of two
 61 canonical SDDs can become exponentially larger than each after canonization.

62 In this paper, we introduce and analyze a new knowledge compilation language which we
 63 call Tree Decision Diagrams (TDD). We show that TDD have various desirable properties of
 64 OBDD, such as having an efficient apply algorithm, a canonical form that is also minimal, and
 65 an efficient algorithm to find it for any given non-minimal TDD. We show that, as is the case
 66 for OBDD, the size of a canonical TDD can be characterized by certain subfunction counts
 67 which gives a very clean understanding of which functions can efficiently be compiled into a
 68 TDD. We highlight that, in contrast to SDD, canonical TDD can be efficiently combined via
 69 apply into a new canonical TDD.

70 Since TDD have efficient apply and minimization algorithms, they are a good target
 71 language for bottom-up compilation. As a proof of concept, we present a simple algorithm
 72 that efficiently compiles CNF formulas and circuits of bounded treewidth. While compilation
 73 results in this setting were known before [19, 27, 12, 4, 13], our approach compiles into a
 74 more restricted language with better properties. We highlight that these previous results had
 75 rather involved dynamic programming solutions, in contrast to our compilation algorithm
 76 which simply performs apply and minimization in a bottom-up fashion. Our results depend
 77 on the characterization by subfunction counts. However, crucially, this argument is only used
 78 in the analysis and not in the algorithm.

79 The paper is organized as follows: Section 2 introduces necessary preliminaries. Section 3
 80 defines the notion of TDD, Section 4 presents the transformations that are tractable for TDD.
 81 Section 5 contains the minimization procedures for TDD and shows that they are canonical.
 82 Section 6 establishes bottom-up compilation of TDD and uses it to compile bounded treewidth
 83 formulas and circuits. Finally, Section 7 compares TDD with other representation languages.
 84 Due to page limit, most of the proofs do not appear in this conference version. The proofs
 85 can be found in the arXiv version of this paper [15].

86 **2 Preliminaries**

87 **Assignments and Boolean functions.** Given two sets A and B , we denote by B^A the set
 88 of functions from A to B . When $B = \{0, 1\}$, we will often write 2^A to denote the set of

89 assignments from a set A to $\{0, 1\}$. An element $\tau \in 2^A$ is called a *Boolean assignment*
 90 *over variables* A , and we will often just write “assignment” when it is clear from context
 91 that it is Boolean. A *partial (Boolean) assignment over variables* X is an element of 2^Y for
 92 some $Y \subseteq X$. Given two assignments $\tau_1 \in 2^X$ and $\tau_2 \in 2^Y$ with $X \cap Y = \emptyset$, we denote by
 93 $\tau_1 \times \tau_2$ the assignment over variables $X \cup Y$ such that $(\tau_1 \times \tau_2)(z) = \tau_1(z)$ if $z \in X$ and
 94 $(\tau_1 \times \tau_2)(z) = \tau_2(z)$ if $z \in Y$. We denote by $\langle x/0 \rangle$ (resp. $\langle x/1 \rangle$) the assignment in $2^{\{x\}}$
 95 mapping x to 0 (resp. to 1). We will also use the notation $\langle x_1/b_1, \dots, x_k/b_k \rangle$ to denote the
 96 assignment in $2^{\{x_1, \dots, x_k\}}$ mapping x_i to b_i . Given $\tau \in 2^X$ and $Y \subseteq X$, we denote by $\tau|_Y$ the
 97 assignment in 2^Y such that $\tau|_Y(y) = \tau(y)$ for every $y \in Y$.

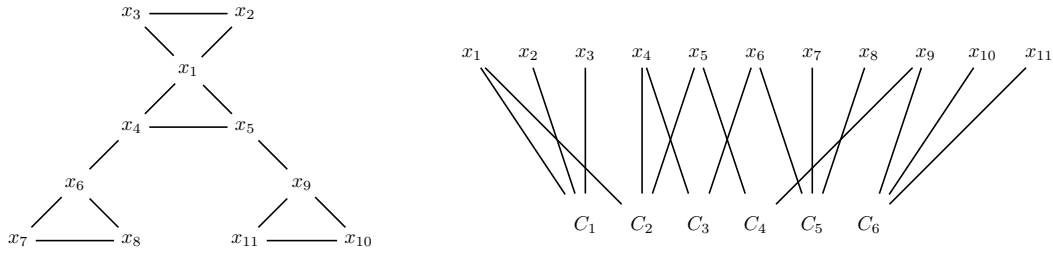
98 A *Boolean function* f over variables X is a mapping from 2^X to $\{0, 1\}$. An assignment τ
 99 such that $f(\tau) = 1$ is said to *satisfy* f and is alternatively called a *satisfying assignment* or
 100 a *model*. Given a Boolean function f over variables X and $Y \subseteq X$, we denote by $f|_Y$ the
 101 Boolean function over variables Y whose models are $\{\tau|_Y \mid f(\tau) = 1\}$. We denote by $\neg f$ the
 102 negation of f and by $f \wedge g$ (resp. $f \vee g$) the conjunction (resp. disjunction) of f and g .

103 **Conjunctive Normal Form Formulas.** Given a set X of variables, a literal over X is either
 104 $x \in X$ or $\neg x$. We let $\text{lit}(X)$ be the set of literals over X , and for $\ell \in \text{lit}(X)$, we denote by
 105 $\text{var}(\ell)$ its underlying variable (that is, $x = \text{var}(x) = \text{var}(\neg x)$). For an assignment $\tau \in 2^X$, we
 106 naturally extend it to literals by defining $\tau(\neg x) = 1 - \tau(x)$. A *clause* c is a set of literals,
 107 interpreted as their disjunction and written as $c = \ell_1 \vee \dots \vee \ell_k$; we let $\text{var}(c) = \{\text{var}(\ell) \mid \ell \in c\}$.
 108 An assignment τ *satisfies a clause* c if there exists $\ell \in c$ such that τ is defined on $\text{var}(\ell)$ and
 109 $\tau(\ell) = 1$. A *Conjunctive Normal Form (CNF) formula* F is a set of clauses, interpreted
 110 as their conjunction and often denoted $F = c_1 \wedge \dots \wedge c_m$. We let $\text{var}(F) = \bigcup_{c \in F} \text{var}(c)$.
 111 An assignment τ *satisfies* F if for every clause $c \in F$, τ satisfies c . The Boolean function
 112 defined by a CNF formula is the Boolean function over $\text{var}(F)$ whose models are exactly the
 113 assignments over $\text{var}(F)$ that satisfy F . We will often identify a CNF formula or a clause
 114 with the Boolean function it represents and use the notation we defined for Boolean functions
 115 directly on formulas. For example, we write $F \models c$ whenever every satisfying assignment of
 116 F is also a satisfying assignment for c . The *size* $\|F\|$ of F is defined as $\sum_{c \in F} |\text{var}(c)|$.

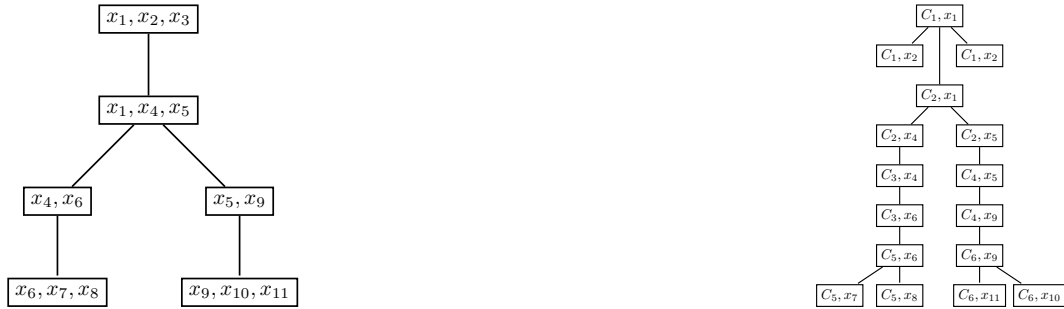
117 A CNF formula can be conditioned by a partial assignment: for $\tau \in 2^Y$, we let $F[\tau]$ be
 118 the CNF formula obtained as follows. We remove from F every clause c containing a literal
 119 ℓ such that $\tau(\ell) = 1$. In the remaining clauses, we remove every literal ℓ such that $\tau(\ell) = 0$.
 120 An assignment $\sigma \in 2^{\text{var}(F) \setminus Y}$ satisfies $F[\tau]$ if and only if $\sigma \times \tau$ satisfies F .

121 **Graphs of CNF formulas.** We characterize the structure of CNF formulas using graphs.
 122 Given a CNF formula F over variables X , the *primal graph* of F , denoted by $\text{Prim}(F) =$
 123 (X, E) , is the graph whose vertices are the variables of F and which has an edge $\{x, y\}$ if
 124 and only if there is a clause $c \in F$ such that $x, y \in \text{var}(c)$. The *incidence graph* of F , denoted
 125 by $\text{Inc}(F) = (X \cup F, E)$ is the graph whose vertices are both the variables and the clauses of
 126 F and which contains the edge $\{x, c\}$ for $x \in X$ and $c \in F$ if and only if $x \in \text{var}(c)$. Observe
 127 that $\text{Inc}(F)$ is bipartite. See Figure 1 for an example.

128 **Treewidth.** We study the structure of F by analyzing the structure of $\text{Prim}(F)$ or $\text{Inc}(F)$,
 129 using the notion of *treewidth*. A *tree decomposition* \mathcal{T} of a graph $G = (V, E)$ is a tree such
 130 that each node t of \mathcal{T} is labeled by a subset B_t of V , called a *bag at node* t . Moreover, \mathcal{T}
 131 has the following properties: it is **connected**, that is, for every $x \in V$, the set $\{t \mid x \in B_t\}$
 132 is connected in \mathcal{T} and it is **complete**, that is, for every edge e of G , there exists a node
 133 t such that $e \subseteq B_t$. Figure 2 shows examples of tree decompositions. The *width of a tree*



■ **Figure 1** The primal graph $\text{Prim}(F)$ (left) and incidence graph $\text{Inc}(F)$ (right) of $F = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$ where $C_1 = (x_1 \vee x_2 \vee x_3)$, $C_2 = (x_1 \vee x_4 \vee x_5)$, $C_3 = (x_4 \vee x_6)$, $C_4 = (\neg x_5 \vee x_9)$, $C_5 = (x_6 \vee x_7 \vee x_8)$, $C_6 = (x_9 \vee x_{10} \vee x_{11})$.



■ **Figure 2** Tree decompositions for $\text{Prim}(F)$ (left) and $\text{Inc}(F)$ (right) from Figure 1.

134 decomposition \mathcal{T} of G , denoted by $\text{tw}(G, \mathcal{T})$ is defined as $\max_{t \in \mathcal{T}} |B_t| - 1$ and the *treewidth*
 135 of G , denoted by $\text{tw}(G)$, is defined to be $\min_{\mathcal{T}} \text{tw}(G, \mathcal{T})$, where \mathcal{T} runs over every valid tree
 136 decomposition of G .

137 We apply the notion of treewidth to CNF formulas as follows. The *primal treewidth*
 138 $\text{ptw}(F)$ of a CNF formula F is defined as the treewidth of $\text{Prim}(F)$, while the *incidence*
 139 *treewidth* $\text{itw}(F)$ of a CNF formula F is the treewidth of $\text{Inc}(F)$.

140 It is not hard to see that for every CNF formula F , we have $\text{itw}(F) \leq \text{ptw}(F) + 1$ and that
 141 for every $n \in \mathbb{N}$, there exists a CNF formula F_n such that $\text{itw}(F_n) = 1$ and $\text{ptw}(F_n) = n - 1$.

142 **OBDD.** An *Ordered Binary Decision Diagram (OBDD)* over variables X is a directed
 143 acyclic graph C such that:

- 144 ■ Every node with outdegree 0 is labeled by a constant 0 or 1 and is called a *sink*.
- 145 ■ Every other node is called a decision-node. It is labeled by a variable $x \in X$ and has two
 146 outgoing edges labeled by 0 and 1 respectively. We say that the decision-node *tests* x .
- 147 ■ C has a unique node with indegree 0 called the *source*.

148 Moreover, there is an order (x_1, \dots, x_n) on X such that if g is a decision-node on x_i , then
 149 every decision node that can be reached from g by a path tests a variable x_j with $j > i$.

150 An OBDD C over variables X represents a Boolean function over variables X as follows:
 151 an assignment $\tau \in 2^X$ satisfies C if and only if there is a path $P = (g_0, \dots, g_k)$ from the
 152 source g_0 to a 1-sink g_k of C such that for every $i < k$, the edge (g_i, g_{i+1}) is labeled by $\tau(x)$
 153 where x is the variable tested by g_i .

154 **DNNF.** We assume the reader to be familiar with the notion of Boolean circuits, see [6]
 155 for details. A Boolean circuit C is in *Negation Normal Form (NNF)* if it only contains
 156 \wedge -gates and \vee -gates, and its inputs are labeled by literals. Given a gate g of C , we denote

157 by $\text{var}(g)$ the set of variables appearing in the subcircuit rooted in g . We say that an \wedge -gate
 158 g with inputs g_1, \dots, g_k is *decomposable* if and only if $\text{var}(g_i) \cap \text{var}(g_j) = \emptyset$ for every $i < j$.
 159 A *Decomposable NNF (DNNF) circuit* is a circuit where every \wedge -gate is decomposable. An
 160 \vee -gate g with inputs g_1, \dots, g_k is said to be *deterministic* if and only if for every $i < j$,
 161 the models of g_i and g_j are disjoint. In other words, g is deterministic if for every model
 162 $\tau \in 2^{\text{var}(g)}$ of g , there exists a unique $i \leq k$ such that τ is a model of g_i . A *deterministic*
 163 *DNNF (d-DNNF) circuit* is a DNNF where every \vee -gate is deterministic. Observe that
 164 determinism is a semantic notion. It is actually coNP-complete to decide whether a given
 165 \vee -gate in a DNNF is deterministic.

166 In this paper, we are interested in a restriction of DNNF called *structured DNNF*
 167 (*SDNNF*) [26]. Structuredness is a syntactic restriction of the way an \wedge -gate can split
 168 variables in a DNNF. It is based on the notion of *variable trees* (vtree for short): a vtree
 169 over X is a rooted binary tree T such that the leaves of T are in one-to-one correspondence
 170 with X . Given a node t of T , we denote by $\text{var}(t) \subseteq X$ the set of variables labeling the leaves
 171 of the subtree of T rooted at t . Let t be a node of T with children t_1, t_2 . Given an \wedge -gate
 172 g with two inputs g_1, g_2 , we say that g *respects* t if and only if it has two inputs g_1, g_2 and
 173 $\text{var}(g_1) \subseteq \text{var}(t_1)$ and $\text{var}(g_2) \subseteq \text{var}(t_2)$. A DNNF circuit *respects a vtree* T if for every \wedge -gate
 174 g of C , there is a node t of T such that g respects t . If a DNNF circuit C respects a vtree T ,
 175 we say that C is a *structured DNNF circuit*.

176 **SDD.** A *Sentential Decision Diagram (SDD)* [20] is a restriction of structured deterministic
 177 DNNF enjoying more tractable operations and some form of canonicity (though the canonical
 178 circuit is not the minimal one in this case). Proofs and definitions regarding SDD can be
 179 found in the arXiv version of this paper [15].

180 3 Tree Decision Diagrams

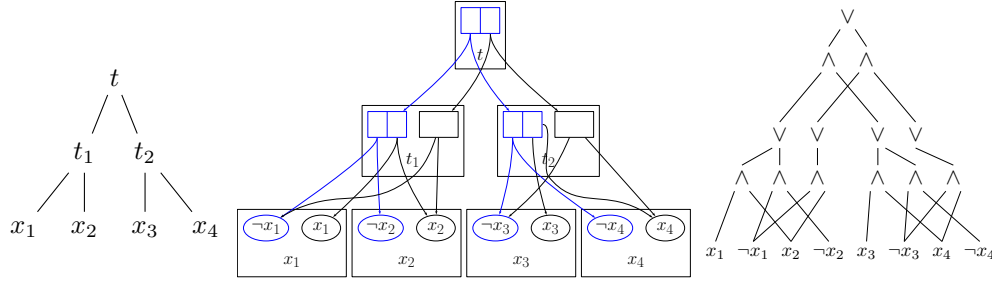
181 Let T be a vtree whose leaves are labeled by a set of variables X . A *Non-deterministic Tree*
 182 *Decision Diagram (nTDD for short)* $C = (N, E)$ *respecting the vtree* T , is defined as follows:

- 183 ■ $N = \bigsqcup_{t \in T} N_t$ is a set of nodes, partitioned into disjoint sets N_t for each node t of T . The
 184 elements of N_t are called *t-nodes*.
- 185 ■ If t is a leaf labeled by x , then every node in N_t is labeled by either x , $\neg x$, 1 or 0.
- 186 ■ E maps every t -node g to its *inputs*: if t is a leaf, then $E(g) = \emptyset$. Otherwise, if t has
 187 children t_1, t_2 , $E(g) \subseteq N_{t_1} \times N_{t_2}$, that is, $E(g)$ is a set of pairs (g_1, g_2) such that $g_1 \in N_{t_1}$
 188 is a t_1 -node and $g_2 \in N_{t_2}$ is a t_2 -node.
- 189 ■ There is one distinguished r -node $\text{out}(C)$, called *the output of* C , where r is the root of T .

190 An nTDD C computes a Boolean function over X defined inductively as follows. Each
 191 t -node g computes a Boolean function f_g over variables X_t where $X_t = \text{var}(t)$:

- 192 ■ If t is a leaf, then g computes the Boolean function defined by its label: that is, if g is
 193 labeled by 0 then f_g has no model, if g is labeled by 1 then every assignment of x is a
 194 model of f_g , and if g is labeled by $\ell \in \{x, \neg x\}$, the only model of f_g is the assignment
 195 $\tau \in 2^{\{x\}}$ such that $\tau(\ell) = 1$.
- 196 ■ If t is an internal node with children t_1, t_2 , then τ is a model of f_g if and only if there
 197 exists $(g_1, g_2) \in E(g)$ such that $\tau|_{X_{t_1}}$ is a model of f_{g_1} and $\tau|_{X_{t_2}}$ is a model of f_{g_2} . If
 198 $E(g)$ is empty, we make the convention that $f_g = \emptyset$ is the 0 constant function.

199 An nTDD C computes the Boolean function f_C defined as $f_{\text{out}(C)}$, the function computed in
 200 its output. We often abuse notation and say a model of g instead of a model of f_g . Another
 201 way of defining f_g is as $f_g = \bigvee_{(g_1, g_2) \in E(g)} (f_{g_1} \wedge f_{g_2})$. This definition allows us to see that



■ **Figure 3** A vtrees, an nTDD respecting it and the corresponding structured DNNF.

202 an nTDD is just a structured DNNF written in a slightly different way. We chose this
 203 presentation however because it is more convenient to define TDDs. In fact, every structured
 204 DNNF can also be rewritten as an nTDD by smoothing the circuit and ensuring that \wedge -gates
 205 and \vee -gates alternate. The *size* $|C|$ of nTDD $C = (N, E)$ is defined as $\sum_{n \in N} |E(n)|$. The
 206 *width* of nTDD $C = (N, E)$ respecting vtrees T is defined as $\max_{t \in T} |N_t|$.

207 Figure 3 shows a vtrees T over variables $X = \{x_1, \dots, x_4\}$, an nTDD C respecting T , and
 208 the interpretation of C as a DNNF. Its width is 2, and its size is 8. We grouped together the
 209 set of t -nodes for every node t of T . The assignment defined as $\tau(x) = 0$ for every $x \in X$ is a
 210 model of C because it is a model of every node pictured in blue.

211 Another way of characterizing the models of C is via the notion of certificates. Given an
 212 assignment $\tau \in 2^X$, a *certificate for τ in C* is an nTDD \mathcal{P} formed by picking exactly one
 213 t -node $g_t^{\mathcal{P}}$ of C for every node t of T such that:

- 214 ■ If t is a leaf of T , then $g_t^{\mathcal{P}}$ is either labeled by 1 or by a literal ℓ such that $\tau(\ell) = 1$.
- 215 ■ If t is a node of T with children t_1, t_2 , then $(g_{t_1}^{\mathcal{P}}, g_{t_2}^{\mathcal{P}}) \in E(g_t^{\mathcal{P}})$.

216 The blue part of Figure 3 represents the certificate for τ , where τ is the assignment setting
 217 every variable to 0, which is indeed a model of the circuit. More generally, a certificate for τ
 218 in C is a witness of the fact that τ is a model of C :

219 ► **Proposition 1** (\star). *Let T be a vtrees over X and C an nTDD respecting T . For every*
 220 *$\tau \in 2^X$, τ is a model of C if and only if there exists a certificate \mathcal{P} for τ in C . In particular,*
 221 *for every node t of T , $\tau|_{X_t}$ satisfies $g_t^{\mathcal{P}}$.*

222 **Determinism.** A TDD $C = (N, E)$ is an nTDD respecting the following extra properties
 223 (which we will sometimes refer to as *determinism*) for every node t of T :

- 224 ■ If t is a leaf labeled by x , then no two nodes of N_t have the same label. Moreover, if one
 225 t -node is labeled by \top , then no t -node can be labeled by a literal. In particular, it means
 226 that two distinct t -nodes cannot be satisfied simultaneously.
- 227 ■ For all distinct $g, g' \in N_t$, we have $E(g) \cap E(g') = \emptyset$. That is, every pair of nodes (g_1, g_2)
 228 is the input of *at most* one node.

229 One can verify that the nTDD from Figure 3 is also a TDD. Our notion of determinism is
 230 similar to others in the literature. First, it resembles the notion of determinism for bottom-up
 231 tree automata [17] where a pair of states from children nodes gives at most one state in the
 232 parent node. Similar constructions have also been used in probabilistic circuits to guarantee
 233 determinism, see for example [33] and MDNets in [40].

234 Contrary to the notion of determinism for DNNF, the notion of determinism for TDD is
 235 syntactic. Therefore, it can be checked in polynomial time whether a given non-deterministic
 236 TDD respects the determinism property. Moreover, it induces a very strong form of determ-
 237 inism. We prove this with a bottom-up induction along the vtrees.

238 ▶ **Theorem 2** (\star). *Let $C = (N, E)$ be a TDD respecting a vtree T . For every node t of T*
 239 *and t -nodes g, g', f_g and $f_{g'}$ have disjoint models. As a consequence, for every model τ of C ,*
 240 *there exists a unique certificate $\mathcal{P}_C(\tau)$ for τ in C .*

241 Observe that a TDD of width k has size at most $2|X| \cdot k^2$. Indeed, T has at most $2|X|$
 242 nodes and each t -node can contain at most k^2 pairs.

243 Interestingly, we can construct the certificate for an assignment τ of C efficiently in a
 244 bottom-up way, or report that τ is not a model of C . To do so, we select the unique leaf
 245 nodes satisfied by τ and construct the certificate bottom up by selecting the unique t -node
 246 whose input contains the pair g_1, g_2 of t_1 -node and t_2 -nodes inductively constructed so far. If
 247 no such node exists, we report that τ is not a model of C . Using appropriate data structures
 248 to represent the input of each t -node, we can find the right t -node in constant time. Hence
 249 we can construct a certificate for τ in time $O(|X|)$ if it exists.

250 A consequence of Theorem 2 is that the DNNF interpretation of a TDD is deterministic,
 251 which proves that TDD is a subclass of structured d-DNNF:

252 ▶ **Theorem 3** (\star). *Given a TDD C respecting a vtree T , one can construct a structured*
 253 *d-DNNF C' respecting T and computing the same function as C in time $O(|C|)$.*

254 In particular, every tractable query for structured d-DNNF is also tractable for TDD.
 255 For example, we can efficiently compute the number of models of a TDD [21], enumerate
 256 them with delay $O(|X|)$ [3] and so on.

257 4 Tractable Transformations

258 Since the publication of the knowledge compilation map [21], it is common in the field to
 259 compare newly introduced representations to others by analyzing them with respect to a
 260 set of standard queries and transformations. Since, due to Theorem 3, every TDD can be
 261 efficiently transformed into a deterministic, structured DNNF (d-SDNNF) and on those one
 262 can already perform all queries from [21] efficiently [26], TDDs inherit all these efficient
 263 queries. So we here only focus on the transformations, showing that TDD allow more efficient
 264 transformations than d-SDNNF, and both canonical and general SDD. In fact, TDD allow
 265 the same efficient transformations as OBDD.

266 We give a compact description of the standard transformations in the first table of
 267 Figure 4; for additional discussion and justifications of these transformations see [21]. The
 268 main result of this section is the following.

269 ▶ **Theorem 4** (\star). *The efficient transformations that TDD allow are as described in Figure 4.*

270 The proof of Theorem 4 is not too hard but rather long and tedious, so we left the details out
 271 (they can be found in [15]). We give some intuition here. Conditioning (CD) for a variable
 272 x and $b \in \{0, 1\}$ is obtained as usual in circuits, by replacing inputs labeled by x with b
 273 and inputs labeled by $\neg x$ with $1 - b$. The important observation is to see that it preserves
 274 determinism: indeed, if t is the node of the vtree labeled by x , and if there are inputs labeled
 275 by x or $\neg x$, then we know that there is no t -node labeled by 1. Then replacing inputs will
 276 create exactly one t -node labeled by 1, which is consistent with the definition of determinism.

277 Bounded Conjunction ($\wedge C$) is exactly the same algorithm as the one for structured
 278 d-DNNF circuits (see [26]), and one just has to be careful that it preserves the syntactic
 279 properties of TDDs. For negation ($\neg C$), the main idea is to first make the TDD *full*: if C is
 280 a TDD respecting vtree T , we ensure that for every node t of T and assignment τ of $\text{var}(t)$,
 281 there is exactly one t -node that is satisfied by τ . This can be ensured bottom-up by creating

Transformation Name	Description
Conditioning (CD)	given a variable x and $a \in \{0, 1\}$ compute representation for $f[x/a]$
Forgetting (FO)	given a list x_1, \dots, x_ℓ of variables compute $\exists x_1 \dots \exists x_\ell f$
Singleton Forgetting (SFO)	same as FO, but only for a single variable
Conjunction ($\wedge C$)	compute representation for $\bigwedge_{i \in [\ell]} f_i$
Bounded Conjunction ($\wedge BC$)	same as $\wedge C$, but only two input representations
Disjunction ($\vee C$)	compute representation for $\bigvee_{i \in [\ell]} f_i$
Bounded Disjunction ($\vee BC$)	same as $\vee C$, but only two input representations
Negation ($\neg C$)	compute representation for $\neg f$

	CD	FO	SFO	$\wedge C$	$\wedge BC$	$\vee C$	$\vee BC$	$\neg C$	references
TDD	✓	•	✓	•	✓	•	✓	✓	this paper
OBDD	✓	•	✓	•	✓	•	✓	✓	[21]
SDD	✓	•	✓	•	✓	•	✓	✓	[38]
canonical SDD	•	•	•	•	•	•	•	✓	[38]
d-SDNNF	✓	•	•	•	✓	•	•	•	[26, 39]

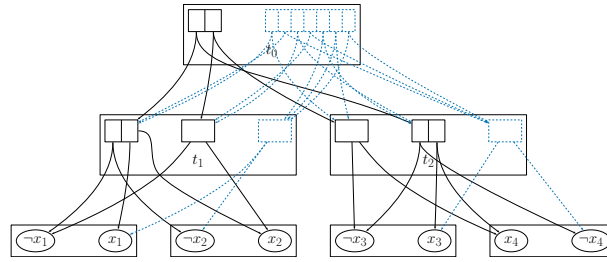
■ **Figure 4** Overview of the transformations from the knowledge compilation map [21] that can be performed efficiently on different representation languages. The first table describes the transformations. For all of them, either one input representation of a Boolean function f or a list of representations of such functions f_1, \dots, f_ℓ is given. Some transformations take additional inputs that are stated explicitly. In the second table, a ✓ means that the operation can be performed in polynomial time on representations from the language, whereas a • means that it takes super-polynomial time. All negative results are unconditionally true. For all transformations, we require that all inputs and outputs have the same vtree, resp. variable order.

282 a new t -node n_t whose input is the list of pairs (n_1, n_2) which are not inputs of any other
 283 t -node. In the end, if r is the root of T , this creates an r -node which computes the negation
 284 of the TDD. See Example 5 for details. The other transformations follow from those we have
 285 just described. For example, ($\vee BC$) follows from ($\neg C$) and ($\wedge BC$) since $f \vee g = \neg(\neg f \wedge \neg g)$.
 286 Similarly, SFO follows from ($\vee BC$) and (CD) since $\exists x.f = f[x/0] \vee f[x/1]$.

287 ► **Example 5.** Figure 5 shows a TDD, in black, computing $((x_1 = x_2) \wedge (x_3 = x_4)) \vee (\neg x_1 \wedge$
 288 $x_2 \wedge \neg x_3 \wedge x_4)$. The pair $(x_1, \neg x_2)$ is not the input of any t_1 -node and the pair $(x_3, \neg x_4)$ is
 289 not the input of any t_2 -node. To make the circuit full, we introduce a new t_1 -node and a new
 290 t_2 -node whose input are $(x_1, \neg x_2)$ and $(x_3, \neg x_4)$ respectively. These new nodes and their
 291 connections are represented in dashed blue in Figure 5. The models of these new nodes are
 292 exactly the assignments that are not a model of any other t_1 -node or t_2 -node. Similarly, we
 293 add a new t_0 -node (again, represented in dashed blue) whose inputs are all pairs of t_1 -nodes
 294 and t_2 -nodes that are not inputs of the only t_0 -node. One can see that the models of this
 295 new node are exactly the assignments that are not models of the original black TDD. Hence,
 296 this new t_0 -node computes the negation of the black TDD.

297 5 Minimization and canonicity

298 One of the most interesting features of TDD is that they can be minimized in polynomial
 299 time and that the minimal circuit is unique up to isomorphism, a property called *canonicity*.
 300 The minimization algorithm is similar to the minimization for OBDD: we identify in the
 301 circuit pairs of gates that we call twins and which can be merged without changing the



■ **Figure 5** A full TDD. The black nodes are the original nodes while the dashed blue ones have been added to make the TDD full.

302 function computed by the circuits. We repeat this merging procedure until no twins can be
 303 found anymore. The circuit we obtain is then shown to be the minimal TDD computing the
 304 same Boolean function.

305 We fix a vtree T over variables X and a TDD C respecting T . Let t_1 be a node of T that
 306 is not the root of T , let t be its parent and t_2 its sibling. For a t_1 -node g_1 and a t -node g , we
 307 define the *siblings of g_1 with respect to g* , denoted by $\text{sib}(g_1, g)$ to be $\{g_2 \mid (g_1, g_2) \in E(g)\}$,
 308 i.e., the set of t_2 -nodes that appear together with g_1 in the inputs of g .

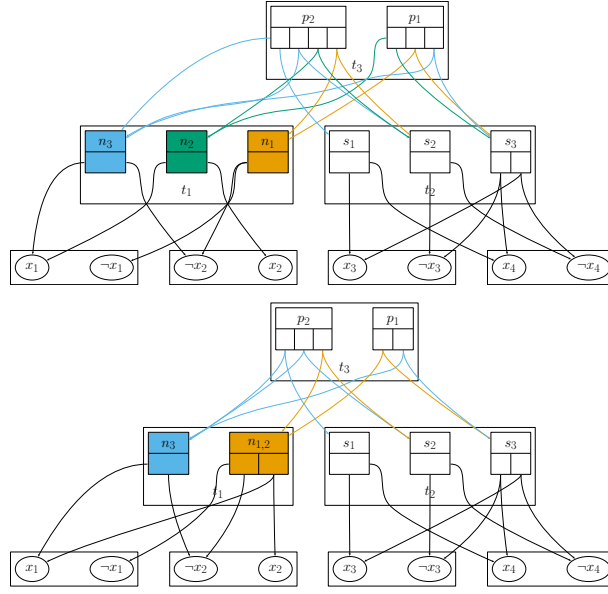
309 We say that two t_1 -nodes g_1, g'_1 are *twins* if for every t -node g , we have $\text{sib}(g_1, g) =$
 310 $\text{sib}(g'_1, g)$. For twins g_1 and g'_1 , we define the *twin contraction of g_1, g'_1* to be the operation
 311 where we replace g_1, g'_1 in C by a new gate v_{g_1, g'_1} such that $E(v_{g_1, g'_1}) = E(g_1) \cup E(g'_1)$.
 312 Moreover, for any t -node g , we replace any pair of the form (g_1, g_2) in $E(g)$ by (v_{g_1, g'_1}, g_2) and
 313 remove every pair of the form (g'_1, g_2) . Observe that since g_1 and g'_1 are twins, $(g_1, g_2) \in E(g)$
 314 if and only if $(g'_1, g_2) \in E(g)$ by definition. Intuitively, two nodes are twins if the way they
 315 are used by the rest of the circuit is completely the same, hence contracting them does not
 316 change the function computed by the circuit.

317 ► **Example 6.** Top of Figure 6 depicts a TDD with two nodes n_1 (orange) and n_2 (green)
 318 that are twins. It can be checked because $\text{sib}(n_1, p_1) = \{s_3\} = \text{sib}(n_2, p_1)$ and $\text{sib}(n_1, p_2) =$
 319 $\text{sib}(n_2, p_2) = \{s_2\}$. Observe, however, that n_1 and n_3 (blue) are not twins. Indeed, even if
 320 $\text{sib}(n_1, p_1) = \{s_3\} = \text{sib}(n_3, p_1)$, we have $\text{sib}(n_3, p_2) = \{s_1, s_2\} \neq \text{sib}(n_2, p_2)$. We can hence
 321 contract n_1 and n_2 into a single node $n_{1,2}$ as depicted in the bottom of Figure 6. We will see
 322 that this results in a canonical TDD since no other pairs are twins.

323 ► **Lemma 7** (\star). *After contracting a pair of twins, the function computed by a circuit is not*
 324 *changed. Moreover, the circuit is still a TDD.*

325 We now define $m(C)$ to be the circuit obtained by the following transformation: first, if r is
 326 the root of T , we remove every r -node but $\text{out}(C)$. We also remove every node that is not
 327 connected to the output of the circuit by a path. This does not change the function computed
 328 by C since these gates are not used in any certificate. We then apply twin contraction to
 329 C until no twins exist anymore. This process terminates since the number of nodes in C
 330 decreases by 1 with each contraction. Moreover, identifying and contracting twins can be
 331 done in polynomial time, hence we can construct $m(C)$ in polynomial time. We now prove
 332 that $m(C)$ is minimal and canonical by semantically characterizing the t -nodes of $m(C)$.

333 We will describe the gates of $m(C)$ from the subfunctions they compute, which is similar
 334 to the description of canonical OBDD [34]. A *subfunction of f induced by Y* , or Y -subfunction
 335 for short, is a Boolean function over $X \setminus Y$ of the form $f[\tau]$ for some $\tau \in 2^Y$. Observe that f
 336 has at most $2^{|Y|} \leq 2^{|X|}$ distinct Y -subfunctions, but it could have fewer. Indeed, two distinct
 337 assignments $\tau_1, \tau_2 \in 2^Y$ could be such that $f[\tau_1]$ and $f[\tau_2]$ have the same models over $2^{X \setminus Y}$,



■ **Figure 6** A TDD with two twins (top) and resulting TDD after contraction (bottom).

338 hence defining the same subfunction. A subfunction is said to be *non-trivial* if it has at
 339 least one model. Given a vtree T and a node t of T , we will mostly be interested in the
 340 X_t -subfunctions of f . For example, consider the Boolean function $PARITY_X$ whose models
 341 are the assignments of X having an even number of variables set to one and let $Y \subseteq X$.
 342 Then $PARITY_X$ has two Y -subfunctions: indeed, if $\tau \in 2^Y$ sets an even number of variables
 343 to 1, then $PARITY_X[\tau] = PARITY_{X \setminus Y}$. Otherwise $PARITY_X[\tau] = \neg PARITY_{X \setminus Y}$.

344 Now, we observe that a t -node in a TDD C naturally defines an X_t -subfunction. Indeed,
 345 if τ_1, τ_2 are two models of a t -node g and τ is a model of C such that $\tau|_{\text{var}(g)} = \tau_1$, then we
 346 can change the value of τ over $\text{var}(g)$ to τ_2 , and it remains a model of C because we only
 347 change the part of the certificate of τ below g . Hence, we have:

348 ► **Lemma 8** (\star). *For a vtree node t of T and g a t -node of C , let τ_1, τ_2 be two models of g .
 349 We have that $f_C[\tau_1]$ and $f_C[\tau_2]$ define the same X_t -subfunction, denoted by sub_g . Moreover,
 350 for every model τ of C such that g is in the certificate of τ , $\tau|_{X \setminus X_t}$ is a model of sub_g .*

351 By Lemma 8, we can map every t -node g of C to an X_t -subfunction sub_g of f_C defined
 352 as $f_C[\tau]$ for some arbitrary model τ of g . This directly gives a lower bound on the number
 353 of t -nodes in a TDD representing a Boolean function f : it must be at least the number of
 354 non-trivial X_t -subfunctions of f_C . Indeed, if $\tau_1, \tau_2 \in 2^{X_t}$ are such that $f_C[\tau_1]$ and $f_C[\tau_2]$
 355 define two distinct X_t -subfunctions, then they cannot be models of the same t -node. Now,
 356 if $f_C[\tau_1]$ is non-trivial, then there must be a t -node g_1 such that τ_1 is a model of g_1 , since
 357 there exists at least one $\sigma \in 2^{X \setminus X_t}$ such that $\sigma \times \tau_1$ is a model of C . Similarly, if $f_C[\tau_2]$ is
 358 non-trivial, there is a t -node g_2 such that τ_2 is a model of g_2 . Since $\text{sub}_{g_1} \neq \text{sub}_{g_2}$, we have
 359 at least one gate per non-trivial X_t -subfunction of f_C .

360 ► **Theorem 9**. *Given a Boolean function f over variables X and a vtree T over X , the
 361 smallest TDD computing f has at least S_t t -nodes for every node t of T where S_t is the
 362 number of non-trivial X_t -subfunctions of f .*

363 The following proves that $m(C)$ matches the lower bound from Theorem 9. The proof boils
 364 down to showing that if there are more than S_t number of t -nodes, then by the pigeonhole

365 principle, two t -nodes must be mapped to the same subfunction and thus can be merged. If
 366 t is the shallowest node where it happens, we can show that such t -nodes must be twins.

367 ▶ **Theorem 10** (\star). *Let T be a vtree over X and C a TDD. Then $m(C)$ has exactly S_t*
 368 *t -nodes, where S_t is the number of non-trivial X_t -subfunctions of f_C .*

369 Theorems 9 and 10 together prove that $m(C)$ has minimal size. Moreover, this minimal
 370 circuit is unique because each gate is uniquely defined by the X_t -subfunction it computes.
 371 TDD can therefore be minimized in polynomial time to a canonical minimal circuit. The
 372 time needed to compute $m(C)$ is polynomial in the width k of C and linear in the number of
 373 variables. Indeed, removing non-accessible nodes can be done in time linear in $|C| \leq k^2|X|$.
 374 Contracting twins in a t -node can be done in time polynomial in the number of t -nodes,
 375 that is, in polynomial time in k , and we have to do it for every node t of T and there are
 376 at most $2|X|$ such nodes. The exact complexity of the minimization depends on the data
 377 structures used to represent t -nodes and their inputs. We leave fine-grained analysis for
 378 practical implementations.

379 ▶ **Theorem 11.** *Given a TDD C of width k over variables X , we can compute a minimal*
 380 *canonical representation $m(C)$ of C in time $\text{poly}(k) \cdot |X|$.*

381 **Learnability.** An interesting application of canonicity is that it allows us to design efficient
 382 L^* -style learning for TDD, as for finite automata and OBDD [5]. This result is framed in
 383 the *Minimally Adequate Teacher* model: there is a hidden Boolean function $f : 2^X \rightarrow \{0, 1\}$
 384 which the learning agent can only access via two types of queries: *membership queries*, in
 385 which it tests some assignment on X , and an oracle answers whether it is a model or not;
 386 and *equivalence queries*, in which the agent tests a TDD, and an oracle answers whether it
 387 represents f , and in the negative case provides a counterexample. The goal of the process is
 388 to construct a minimal size TDD for f with a low number of queries.

389 ▶ **Proposition 12** (\star). *Fix a vtree T on X . There is an algorithm that learns the canonical*
 390 *TDD C respecting T in polynomial time in $|C|$ and with a polynomial number of oracle calls*
 391 *to membership and equivalence queries.*

392 6 Bottom up compilation

393 The tractability (\wedge BC) for TDDs gives a natural algorithm for compiling a CNF formula into
 394 a TDD, whose pseudo-code is given in Algorithm 1. The idea is to order the clauses of F as
 395 c_1, \dots, c_m , build a TDD T_i computing c_i for every $i \leq m$ and then, iteratively construct a
 396 TDD C_i computing $F_i := c_1 \wedge \dots \wedge c_i$ by observing that $F_i = F_{i-1} \wedge c_i$, using the algorithm
 397 for bounded conjunction. In the worst case, we could have $|C_i| = |c_i| \cdot |C_{i-1}|$, leading to an
 398 exponential blow-up in the size of the circuit. To avoid this if possible, we minimize the
 399 circuit after each conjunction. The only missing piece here is the fact that we can efficiently
 400 construct a TDD given a clause c . This can be done with a TDD of width 2. For every node
 401 t of the vtree T , we have two t -nodes. One computes $c_t := c|_{\text{var}(t)}$ and the other computes
 402 $d_t := (\neg c)|_{\text{var}(t)}$. The circuit is constructed by induction using the fact that, if t has children
 403 t_1, t_2 then $d_t = d_{t_1} \wedge d_{t_2}$ and $c_t = (c_{t_1} \wedge c_{t_2}) \vee (c_{t_1} \wedge d_{t_2}) \vee (d_{t_1} \wedge c_{t_2})$.

404 This kind of algorithm is usually referred to as “bottom-up compilation” and has been used
 405 for OBDD [35] and SDD [16]. In this section, we investigate the complexity of Algorithm 1
 406 depending on the structure of the input CNF formula. We recover in a clean and modular
 407 way the fact that CNF formulas having bounded primal or incidence treewidth have TDDs
 408 of FPT size [12] and are able to easily generalize to bounded treewidth circuits [13, 4].

■ **Algorithm 1** Bottom-up compilation into TDD.

Input: A CNF formula $F = c_1 \wedge \dots \wedge c_m$, a vtree T over $\text{var}(F)$.

Output: A TDD computing F respecting T .

```

1: procedure CNF-TO-TDD( $F, T$ )
2:    $C \leftarrow$  TDD computing 1 respecting  $T$ 
3:   for  $i = 1$  to  $m$  do
4:      $D \leftarrow$  TDD computing  $c_i$ 
5:      $C \leftarrow$  construct a TDD for  $C \wedge D$ 
6:     minimize( $C$ )
7: return  $C$ 

```

409 To this end, we use the notion of *factor width* [13]. Given a Boolean function f and a
410 vtree T over X , Theorems 9 and 10 allow us to prove that the size of the smallest TDD
411 for f respecting T is equal to $\sum_t S_t$ where the sum is over every node t of T and S_t is the
412 number of non-trivial X_t -subfunctions of f . Hence, the *factor width of f with respect to T* ,
413 $\text{fw}(f, T)$ for short, is defined as $\max_t S_t$ where the maximum is over all nodes t of T [13].
414 From what precedes, we know that the smallest TDD computing f and respecting T has
415 width $\text{fw}(f, T)$ and size $O(|X| \cdot \text{fw}(f, T))$ since T has at most $2|X| - 1$ nodes. Factor width
416 thus provides a good proxy for the size of the smallest TDD computing f . We also define
417 the *factor width of f* to be $\min_T \text{fw}(f, T)$, where T goes over every vtree over the variables
418 X (the definition of factor width from [13] allows vtrees over variables $Z \supseteq X$ but these
419 extra variables do not change the value of $\text{fw}(f)$). We slightly abuse notation and for a given
420 CNF formula F and a vtree T over $\text{var}(F)$, write $\text{fw}(F, T)$ to denote the factor width of the
421 Boolean function represented by F with respect to T . We can then bound the runtime of
422 Algorithm 1 as follows:

423 ► **Theorem 13** (\star). *Given a CNF formula F , a vtree T over variables X and an order*
424 *c_1, \dots, c_m on the clauses of F , Algorithm 1 runs in time $m \cdot |X| \cdot \text{poly}(k)$ where $k =$*
425 *$\max_{i=1}^m \text{fw}(c_1 \wedge \dots \wedge c_i, T)$.*

426 The proof of Theorem 13 is based on the fact that minimizing a TDD C can be done in time
427 $|X| \cdot \text{poly}(w)$ where w is the width of C . Similarly, computing a TDD for $C \wedge D$ can be done
428 in time $|X| \cdot \text{poly}(w)$. The result follows from the fact that the width of every intermediate
429 circuit built in the main loop of Algorithm 1 will never exceed k . We now explore a few
430 applications of Theorem 13.

431 **Bounded primal and incidence treewidth.** CNF formulas of bounded primal or incidence
432 treewidth have long been known to be tractable. It has long been known that SAT can be
433 solved in time $2^{O(k)} \|F\|$. The earliest reference of this fact seems to be in a paper by Dantsin
434 from 1979 [18], though it is not specifically stated with the treewidth terminology, later
435 improved by Alekhovich and Razborov [1, 2], where the result is expressed in terms of the
436 equivalent branch-width measure, and Szeider [37]. The generalization for the tractability
437 of #SAT has first been observed by Sang, Bacchus, Beame, Kautz, and Pitassi in [32] and
438 later generalized to the case of incidence treewidth by Samer and Szeider [31]. The existence
439 of small d-DNNFs for such formulas is implicit in Darwiche's early contribution [19] and
440 explicit in a collaboration with Pipatsrisawat [27] for primal treewidth. The case for incidence
441 treewidth has been formally proven along more general results in [12]. We revisit these
442 results by showing that such formulas have small factor width. More precisely:

443 ▶ **Theorem 14** (\star). *Given a CNF formula F and a tree decomposition \mathcal{T} of $\text{Prim}(F)$ of*
 444 *width k , we can construct a vtree T over $\text{var}(F)$ such that for every $F' \subseteq F$, $\text{fw}(F', T) \leq 2^k$.*

445 ▶ **Theorem 15** (\star). *Given a CNF formula F and a tree decomposition \mathcal{T} of $\text{Inc}(F)$ of width*
 446 *k , we can construct a vtree T over $\text{var}(F)$ such that for every $F' \subseteq F$, $\text{fw}(F', T) \leq 2^k$.*

447 We give an intuition on the proof of Theorem 14. For a node t of \mathcal{T} , let Y_t be the set of
 448 variables of F appearing in a bag below t . We claim that F has at most 2^k Y_t -subfunctions.
 449 Indeed, assume that a clause c of F has variables both in Y_t and in $X \setminus Y_t$. Then we must
 450 have $\text{var}(c) \cap Y_t \subseteq B_t$, where B_t is the bag at node t of \mathcal{T} . Let $\tau \in 2^{Y_t}$. Remove from $F[\tau]$
 451 every clause already satisfied. Then $F[\tau]$ contains either clauses without variables in Y_t
 452 and clauses having both in Y_t and in $X \setminus Y_t$. From what precedes, $\text{var}(c) \cap Y_t \subseteq B_t$, hence
 453 $F[\tau] = F[\tau|_{B_t}]$. Hence there are at most $2^{|B_t|} \leq 2^k$ Y_t -subfunctions. This proof works for
 454 every $F' \subseteq F$. It remains to build a vtree which induces roughly the same partitions as Y_t ,
 455 see [15].

456 The case of incidence treewidth is very similar. In this case however, a Y_t -subfunction
 457 induced by an assignment $\tau \in 2^{Y_t}$ is completely defined by the subset of clauses in B_t that
 458 are satisfied by τ and by the value of τ in $B_t \cap X$. It still gives at most 2^k Y_t -subfunctions.

459 The new connection established by Theorems 14 and 15 allows us to nicely recover the
 460 tractability results discussed before. Indeed, it is straightforward to see that if a CNF
 461 formula has primal or incidence treewidth k , then every sub-formula of F has treewidth
 462 at most k . Hence, the bottom-up compilation to TDD from Algorithm 1 runs in time
 463 $\text{poly}(2^k) \cdot mn = 2^{O(k)} \cdot mn$ on a formula with n variables, m clauses and of primal or incidence
 464 treewidth k by Theorem 13, as long as we start from the vtree given by Theorems 14 and 15.

465 ▶ **Theorem 16**. *Given a CNF formula F of primal or incidence treewidth k , one can*
 466 *construct a TDD of width at most 2^k computing F in time $2^{O(k)} \cdot mn$.*

467 Algorithm 1 gives a conceptually simpler algorithm than the bottom-up dynamic pro-
 468 gramming on tree decomposition from [31] and serves as a nice example of the power of
 469 TDDs and minimization. The complexity of this approach is however not as good as earlier
 470 work where the dependency on the size of the CNF formula is linear. Maybe it can be fixed
 471 by minimizing the circuits while computing $C \wedge D$ in Algorithm 1 and to have a dedicated
 472 algorithm to compute $C \wedge D$ in the case where D represents a clause but we leave this
 473 question for further investigation.

474 **Circuit treewidth.** Another interesting application of Algorithm 1 is related to the com-
 475 pilation of bounded treewidth Boolean circuits, which can be seen as a generalization of
 476 incidence treewidth. The treewidth of a Boolean circuit C is defined as the treewidth of
 477 its underlying graph. For the notion to make sense, one needs to first assume that for any
 478 variable $x \in X$, there is at most one input of the circuit labeled by x . From [13], we know
 479 that the factor width of a Boolean circuit of treewidth k is bounded by $2^{2^{O(k)}}$. We improve
 480 to 3^{k+2} , getting a single exponential in k and show that bottom-up compilation can be used
 481 to recover a result from [4] showing that bounded treewidth circuit can be compiled into
 482 structured d-DNNF of size $2^{O(k)}|C|$. For a Boolean circuit, a *subcircuit* C' of C is a subset
 483 of nodes and edges of C forming a valid Boolean circuit (that is, its inputs are labeled with
 484 variables).

485 ▶ **Theorem 17** (\star). *Let C be a Boolean circuit over variables X and let \mathcal{T} be a tree*
 486 *decomposition of C of treewidth k . We can construct a vtree T such that for every subcircuit*
 487 *C' of C computing a function f' , we have $\text{fw}(f', T) \leq 3^{k+2}$.*

488 It gives a straightforward way of constructing a TDD of size $2^{O(k)}|C|$ computing f_C from
 489 a tree decomposition \mathcal{T} of treewidth k of a Boolean circuit C . We first extract a vtree T as
 490 in Theorem 17 and, for every gate g of C , we build a TDD T_g computing the same Boolean
 491 function as g . For example, for a \wedge -gate g of C with input g_1, \dots, g_p , construct T_g as follows:
 492 inductively construct T_{g_1}, \dots, T_{g_p} , then iteratively build $((T_{g_1} \wedge T_{g_2}) \wedge T_{g_3}) \wedge \dots \wedge T_{g_p}$. For
 493 every $i \leq p$, the circuit having g as output and g_1, \dots, g_i as input is a subcircuit of C , hence
 494 the resulting intermediate TDD have width at most 3^{k+2} . We proceed similarly for \neg -gates
 495 and \vee -gates. Since computing an optimal tree decomposition can be done in FPT linear
 496 time [9], we have:

497 **► Theorem 18.** *Given a Boolean circuit C of treewidth k , we can compute a vtree T and a*
 498 *TDD respecting T computing f_C of width $2^{O(k)}$ and in time $2^{O(k)} \cdot |X| \cdot |C|$.*

499 Treewidth is not the most general parameter for which we can build polynomial-size
 500 d-DNNF. CNF formulas of bounded MIM-width, for example, may have unbounded treewidth
 501 but have polynomial-size deterministic DNNF circuits [12, 30]. That said, it is not clear
 502 whether they have bounded factor width. Such a result would allow to show that Algorithm 1
 503 works in polynomial time on bounded MIM-width instances, simplifying the convoluted
 504 algorithm from the literature. We leave the study of such graph measures for future work.

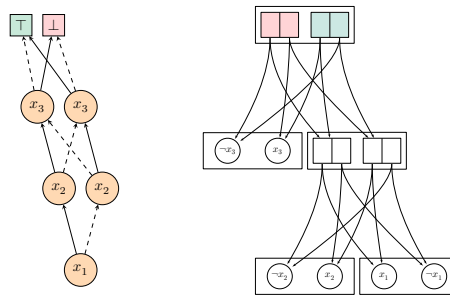
505 **7** Comparing TDD with other data structures

506 **OBDD.** Tractable queries and transformations for TDD are similar to those for OBDD.
 507 We can actually see OBDD as a particular subclass of TDD where the underlying vtree
 508 T is *linear*, that is, for every internal node t of T , one child of t is a leaf. A linear vtree
 509 T over variables X naturally induces an order $\pi_T = (x_1, \dots, x_n)$ on X defined as follows:
 510 x_1 is the leaf attached to the root of T , and (x_2, \dots, x_n) is the order induced by the other
 511 subtree attached to the root. Similarly, an order $\pi = (x_1, \dots, x_n)$ can be mapped naturally
 512 to the linear vtree T_π defined as the vtree whose root has one leaf child labeled by x_1 , and
 513 its other child is the vtree for the order (x_2, \dots, x_n) . For an order $\pi = (x_1, \dots, x_n)$, we let
 514 $\pi^{-1} = (x_n, \dots, x_1)$. The class of TDD with linear vtrees corresponds exactly to OBDD in
 515 the following sense:

516 **► Theorem 19 (*)**. *Given an OBDD C over variables X and order $\pi = (x_1, \dots, x_n)$, one can*
 517 *construct an equivalent TDD respecting $T_{\pi^{-1}}$ of size at most $3|C|$ in time $O(|C|)$. Similarly,*
 518 *let T be a linear vtree and C be a TDD respecting T . Then one can construct an equivalent*
 519 *OBDD in time $O(|C|)$ respecting order π_T^{-1} .*

520 The proof of Theorem 19 mainly boils down to rooting an OBDD in its 1-sink, as
 521 illustrated in Figure 7. We observe however that Theorem 9 offers a way of getting the
 522 correspondence of Theorem 19 in a non-constructive way. Indeed, it is known [24] that the
 523 width of the minimal OBDD is exactly the maximum number of subfunctions one can get
 524 by fixing variables x_1, \dots, x_i for some $i \leq n$. This exactly corresponds to the number of
 525 subfunctions we can have with a linear vtree. The previous constructions can be extended to
 526 the case of non-deterministic TDD and non-deterministic OBDD.

527 OBDDs are however weaker than TDDs. This follows as a corollary of [29] and Theorem 16.
 528 In [29], Razgon proves that there is a family of CNF formulas $(F_n)_{n \in \mathbb{N}}$ where F_n has n variables
 529 and treewidth $O(\log n)$ such that every OBDD representing F_n must have size $n^{\Omega(\log n)}$, while
 530 Theorem 16 shows that such instances have polynomial-size TDD representation.



■ **Figure 7** An OBDD (left) represented as a TDD (right). The output of the TDD is represented in green, and each node t of the vtree is made explicit by a rectangle around t -nodes.

531 ► **Theorem 20.** *There exists a family $(F_n)_{n \in \mathbb{N}}$ of CNF formulas such that F_n can be*
 532 *represented by a polynomial-size TDD while every OBDD representing F_n has size at least*
 533 *$n^{c \log n}$ for some constant c .*

534 The separation given by Theorem 20 is only quasi-polynomial, and one can wonder
 535 whether a truly exponential separation is possible. It may seem possible that TDDs can
 536 be quasi-polynomially simulated by OBDDs, in the same way FBDDs quasi-polynomially
 537 simulate decision-DNNF circuits [8]. We leave this question for future investigation.

538 **Deterministic DNNF.** As we have already observed, TDD is a subclass of structured
 539 deterministic DNNF. We show in this section that structured deterministic DNNF may be
 540 exponentially smaller than TDD. To do so, we are interested in the *Hidden Weighted Bit*
 541 functions, which are known to be hard for OBDD [41]. Given $n \in \mathbb{N}$, define $\text{HWB}_n(x_1, \dots, x_n) = 1$
 542 if and only if the value assigned to x_S is 1, where $S = \sum_{i=1}^n x_i$. The Boolean function
 543 HWB_n can easily be computed by a structured d-DNNF, by first guessing the number S
 544 of variables set to 1 and then checking that this is indeed the case and that $x_S = 1$ with a
 545 small OBDD. However, HWB_n does not admit polynomial-size OBDD [41]. We adapt the
 546 HWB_n lower bound for OBDD to TDD. The proof relies on adapting [41, Lemma 4.10.1].
 547 In a nutshell, this lemma shows that if we pick $Y \subseteq \{x_1, \dots, x_n\}$ of size $n/2$, then HWB_n
 548 has an exponential number of Y -subfunctions. We generalize it to show that if Y has size
 549 between $n/3$ and $2n/3$ then we still have an exponential number of Y -subfunctions. We then
 550 apply the lemma by finding a node t in the vtree such that X_t has size between $n/3$ and
 551 $2n/3$ which gives an exponential lower bound on the size of a TDD computing HWB_n .

552 ► **Theorem 21** (\star). *Let $n \in \mathbb{N}$ be a multiple of 7. Then $\text{fw}(\text{HWB}_n) \geq 2^{cn}$ for some constant c .*
 553 *In particular, any TDD computing HWB_n has size at least 2^{cn} .*

554 **SDD.** TDDs and SDDs have a lot in common: they are both restrictions of structured
 555 deterministic DNNF with canonical representations, efficient negations and efficient apply.

556 Bova [11] proved that HWB_n can be computed by an SDD of size $O(n^3)$. He also
 557 constructs a Boolean function $F(X, Y)$ such that $F(X, 1, \dots, 1) = \text{HWB}_n(X)$ which has a
 558 compressed SDD of size $O(n^3)$. This establishes:

559 ► **Theorem 22.** *Compressed SDD cannot be polynomially simulated by TDD.*

560 The other way around, it turns out that we can always simulate a TDD by a polynomial-size
 561 SDD. If we allow encoding variables, since a TDD and its negation are both polynomial-size

562 structured d-DNNFs, it follows by [10, Theorem 1]. We show below that this is possible even
 563 without the encoding variables:

564 ► **Theorem 23** (\star). *Given a TDD C respecting vtree T , one can construct a vtree T' and an*
 565 *SDD C' respecting T' such that C' computes the same function as C and $|C'| = O(|C|^2)$.*

566 The resulting SDD, whose construction is given in [15], does not respect the same
 567 vtree as the original TDD. This is unavoidable. Indeed, consider the *Multiplexer* function
 568 $\text{MUX}_n(x_0, \dots, x_{k-1}, y_0, \dots, y_{n-1})$ which has $k + n$ variables where $n = 2^k$ and is satisfied if
 569 and only if $y_{[x]_2} = 1$ where $[x]_2 = \sum_{i=0}^{k-1} x_i \cdot 2^i$. Let π be the order $(x_0, \dots, x_{k-1}, y_0, \dots, y_{n-1})$.
 570 It is not hard to see that there is an OBDD respecting π of size $O(n)$ computing MUX_n [41,
 571 Theorem 4.3.2]. Hence, there is an SDD of size $O(n)$ respecting T_π computing MUX_n .
 572 However, one can also prove that any OBDD respecting π^{-1} and computing MUX_n must
 573 have size 2^n . Indeed, we have one Y -subfunction per assignment of the Y variables: if
 574 $\tau, \sigma \in 2^Y$ are distinct, then let y_i be such that, wlog, $1 = \tau(y_i) \neq \sigma(y_i) = 0$. Then
 575 $\text{MUX}_n[\tau] \neq \text{MUX}_n[\sigma]$ because they differ on α_i , the assignment of X variables encoding i in
 576 binary. In other words, every SDD respecting $T_{\pi^{-1}}$ and computing MUX_n must have size 2^n .
 577 But there is a TDD of size $O(n)$ computing MUX_n and respecting $T_{\pi^{-1}}$ by Theorem 19.

578 We note however that the construction from Theorem 23 does not give a compressed,
 579 hence not canonical, SDD. It is not clear whether we can always build a polynomial-size
 580 canonical SDD equivalent to a given TDD. We leave open the question of comparing canonical
 581 SDD and TDD.

582 8 Conclusion

583 In this paper, we have introduced a new data structure for representing Boolean functions
 584 that offers advantages similar to OBDD but can handle bounded treewidth instances. The
 585 main advantage of these data structures over the existing ones such as d-DNNF or SDD
 586 is that they can be minimized into a canonical circuit for which the size and width can be
 587 easily understood. While SDDs also have canonical representations, those are not minimal,
 588 and the canonical representation may be exponentially larger than the minimal one [38]. Our
 589 approach allows us to recover compilation results in a clean and modular way.

590 Several research directions remain concerning TDD. First, it would be interesting to
 591 implement a bottom-up compiler with TDD as a target language and perform a comparison
 592 with OBDD and SDD compilers. To make TDD competitive, it might be necessary to
 593 study a variant that is non-smooth, i.e., allowing t -nodes to have as inputs u -nodes where
 594 u is a descendant of t in the vtree but not necessarily a child. While this can only lead to
 595 polynomial size gains, it could make a big difference in practice. Second, and related to
 596 this, the question of finding a good vtree in practice remains open. The SDD compiler uses
 597 local changes in the vtree to find a better one, and adapting it to TDD may be promising.
 598 Vtree changes for the related model of probabilistic circuits have also been studied [42].
 599 Generally, we think it would be useful to understand the complexity of transforming a TDD
 600 respecting vtree T into an equivalent canonical TDD respecting vtree T' , measured in the
 601 input and output size. Finally, an interesting application of TDD that we feel is worth
 602 exploring is extending bottom-up compilation to the setting of [22] where a conjunction of
 603 constraints represented as OBDDs is compiled into a d-DNNF. When the incidence graph of
 604 this conjunction has bounded incidence treewidth, and the constraints can be represented by
 605 OBDDs with any order, then it is possible to construct an FPT size d-DNNF. It seems that
 606 if the constraints are all represented by TDDs using the same vtree, a bottom-up compilation
 607 could give similar and slightly more general results.

References

- 608 —
- 609 1 Michael Alekhnovich and Alexander Razborov. Satisfiability, Branch-Width and Tseitin
610 tautologies. *Computational Complexity*, 20(4):649–678, November 2011.
- 611 2 Michael Alekhnovich and Alexander A Razborov. Satisfiability, branch-width and tseitin
612 tautologies. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002.*
613 *Proceedings.*, pages 593–603. IEEE, 2002.
- 614 3 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach
615 to efficient enumeration. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca
616 Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming,*
617 *ICALP 2017, Warsaw, Poland, July 10-14, 2017*, volume 80 of *LIPICs*, pages 111:1–111:15.
618 Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. URL: [https://doi.org/10.4230/](https://doi.org/10.4230/LIPICs.ICALP.2017.111)
619 [LIPICs.ICALP.2017.111](https://doi.org/10.4230/LIPICs.ICALP.2017.111), doi:10.4230/LIPICs.ICALP.2017.111.
- 620 4 Antoine Amarilli, Florent Capelli, Mikael Monet, and Pierre Senellart. Connecting knowledge
621 compilation classes and width parameters. *Theory Comput. Syst.*, 64(5):861–914, 2020. URL:
622 <https://doi.org/10.1007/s00224-019-09930-2>, doi:10.1007/S00224-019-09930-2.
- 623 5 Dana Angluin. Learning regular sets from queries and counterexamples. *Information and*
624 *computation*, 75(2):87–106, 1987.
- 625 6 Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge
626 University Press, New York, NY, USA, 1st edition, 2009.
- 627 7 Jean Marc Astesana, Laurent Cosserat, and Hélène Fargier. Constraint-based vehicle config-
628 uration: A case study. In *Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 68–75.
629 IEEE, 2010.
- 630 8 Paul Beame, Jerry Li, Sudeepa Roy, and Dan Suciu. Lower bounds for exact model counting
631 and applications in probabilistic databases. In *Proceedings of the Twenty-Ninth Conference on*
632 *Uncertainty in Artificial Intelligence*, 2013.
- 633 9 Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth.
634 In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC
635 '93, pages 226–234. ACM, 1993.
- 636 10 Beate Bollig and Martin Fahrenholtz. On the relation between structured d-dnnfs and sdds.
637 *Theory Comput. Syst.*, 65(2):274–295, 2021. URL: <https://doi.org/10.1007/s00224-020-10003-y>,
638 doi:10.1007/S00224-020-10003-Y.
- 639 11 Simone Bova. Sdds are exponentially more succinct than obdds. In Dale Schuurmans and
640 Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial*
641 *Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 929–935. AAAI Press, 2016.
642 URL: <https://doi.org/10.1609/aaai.v30i1.10107>, doi:10.1609/AAAI.V30I1.10107.
- 643 12 Simone Bova, Florent Capelli, Stefan Mengel, and Friedrich Slivovsky. On Compiling CNFs into
644 Structured Deterministic DNNFs. In *Theory and Applications of Satisfiability Testing*, Lecture
645 Notes in Computer Science, pages 199–214. Springer International Publishing, September
646 2015.
- 647 13 Simone Bova and Stefan Szeider. Circuit treewidth, sentential decision, and query compilation.
648 In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the*
649 *36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS*
650 *2017, Chicago, IL, USA, May 14-19, 2017*, pages 233–246. ACM, 2017. doi:10.1145/3034786.
651 3034787.
- 652 14 Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams.
653 *ACM Computing Surveys*, 24:293–318, 1992.
- 654 15 Florent Capelli, YooJung Choi, Stefan Mengel, Martín Muñoz, and Guy Van den Broeck.
655 A canonical generalization of obdd, 2026. URL: <https://arxiv.org/abs/2604.05537>, arXiv:
656 2604.05537.
- 657 16 Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams.
658 *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1):187–194, Jun. 2013. URL:
659 <https://ojs.aaai.org/index.php/AAAI/article/view/8690>, doi:10.1609/aaai.v27i1.8690.

- 660 17 Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof
661 Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2008.
- 662 18 E Dantsin. Parameters defining the time of tautology recognition by the splitting method.
663 *Semiotics and information science*, 12:8–17, 1979.
- 664 19 Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form.
665 In *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004*, pages
666 328–332, 2004.
- 667 20 Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases.
668 In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference
669 on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 819–826.
670 IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-143.
- 671 21 Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *Journal of Artificial
672 Intelligence Research*, 17:229–264, 2002.
- 673 22 Alexis de Colnet, Stefan Szeider, and Tianwei Zhang. Compilation and fast model counting
674 beyond CNF. In *Proceedings of the Thirty-Third International Joint Conference on Artificial
675 Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, pages 3315–3323. ijcai.org,
676 2024. URL: <https://www.ijcai.org/proceedings/2024/367>.
- 677 23 Hélène Fargier, Stefan Mengel, and Jérôme Mengin. An extended knowledge compilation
678 map for conditional preference statements-based and generalized additive utilities-based
679 languages. *Ann. Math. Artif. Intell.*, 92(5):1161–1196, 2024. URL: [https://doi.org/10.1007/
680 s10472-024-09935-9](https://doi.org/10.1007/s10472-024-09935-9), doi:10.1007/S10472-024-09935-9.
- 681 24 Kazuyoshi Hayase and Hiroshi Imai. Obdds of a monotone function and its prime implicants.
682 *Theory of Computing Systems*, 31(5):579–591, 1998.
- 683 25 Dan Olteanu and Jakub Zavodny. Factorised representations of query results: size bounds
684 and readability. In Alin Deutsch, editor, *15th International Conference on Database Theory,
685 ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 285–298. ACM, 2012. doi:10.1145/
686 2274576.2274607.
- 687 26 Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured de-
688 composability. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence,
689 AAAI*, pages 517–522, 2008.
- 690 27 Knot Pipatsrisawat and Adnan Darwiche. Top-down algorithms for constructing structured
691 DNNF: Theoretical and practical implications. In *ECAI*, pages 3–8, 2010.
- 692 28 Alberto Pliego Marugán, Fausto Pedro García Márquez, and José Lorente. Decision making
693 process via binary decision diagram. *International Journal of Management Science and
694 Engineering Management*, 10(1):3–8, 2015.
- 695 29 Igor Razgon. No small nondeterministic read-once branching programs for cnfs of bounded
696 treewidth. In *Parameterized and Exact Computation - 9th International Symposium, IPEC*,
697 pages 319–331, 2014.
- 698 30 S. Hortemo Sæther, J.A. Telle, and M. Vatshelle. Solving MaxSAT and #SAT on structured
699 CNF formulas. In *Theory and Applications of Satisfiability Testing*, pages 16–31, 2014.
- 700 31 M. Samer and S. Szeider. Algorithms for propositional model counting. *Journal of Discrete
701 Algorithms*, 8(1):50–64, 2010.
- 702 32 Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining
703 component caching and clause learning for effective model counting. *Theory and Applications
704 of Satisfiability Testing*, 4, 2004.
- 705 33 Andy Shih and Stefano Ermon. Probabilistic circuits for variational inference in discrete
706 graphical models. *Advances in neural information processing systems*, 33:4635–4646, 2020.
- 707 34 Detlef Sieling and Ingo Wegener. Nc-algorithms for operations on binary decision diagrams.
708 *Parallel Processing Letters*, 3(01):3–12, 1993.
- 709 35 Fabio Somenzi. Cudd: Cu decision diagram package release 2.3. 0. *University of Colorado at
710 Boulder*, 621, 1998.

- 711 **36** Chico Sundermann, Elias Kuitert, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas
712 Thüm. On the benefits of knowledge compilation for feature-model analyses: C. sundermann
713 et al. *Annals of Mathematics and Artificial Intelligence*, 92(5):1013–1050, 2024.
- 714 **37** Stefan Szeider. On fixed-parameter tractable parameterizations of SAT. In Enrico Giunchiglia
715 and Armando Tacchella, editors, *Theory and Applications of Satisfiability, 6th International*
716 *Conference*, volume 2919 of *LNCIS*, pages 188–202. Springer, 2004.
- 717 **38** Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation.
718 In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.
- 719 **39** Harry Vinall-Smeeth. Structured d-DNNF is not closed under negation. In *Proceedings of*
720 *the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju,*
721 *South Korea, August 3-9, 2024*, pages 3593–3601. ijcai.org, 2024. URL: [https://www.ijcai.org/](https://www.ijcai.org/proceedings/2024/398)
722 [proceedings/2024/398](https://www.ijcai.org/proceedings/2024/398).
- 723 **40** Benjie Wang and Marta Kwiatkowska. Compositional probabilistic and causal inference using
724 tractable circuit models. In *International Conference on Artificial Intelligence and Statistics*,
725 pages 9488–9498. PMLR, 2023.
- 726 **41** Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.
- 727 **42** Honghua Zhang, Benjie Wang, Marcelo Arenas, and Guy Van den Broeck. Restructur-
728 ing tractable probabilistic circuits. In Yingzhen Li, Stephan Mandt, Shipra Agrawal,
729 and Mohammad Emtiyaz Khan, editors, *International Conference on Artificial Intelli-*
730 *gence and Statistics, AISTATS 2025, Mai Khao, Thailand, 3-5 May 2025*, volume 258
731 of *Proceedings of Machine Learning Research*, pages 2566–2574. PMLR, 2025. URL:
732 <https://proceedings.mlr.press/v258/zhang25f.html>.