On Compiling CNF into Decision-DNNF

Umut Oztok and Adnan Darwiche

Computer Science Department, University of California,
Los Angeles, CA 90095, USA
{umut,darwiche}@cs.ucla.edu

Abstract. Decision-DNNF is a strict subset of decomposable negation normal form (DNNF) that plays a key role in analyzing the complexity of model counters (the searches performed by these counters have their traces in Decision-DNNF). This paper presents a number of results on Decision-DNNF. First, we introduce a new notion of CNF width and provide an algorithm that compiles CNFs into Decision-DNNFs in time and space that are exponential only in this width. The new width strictly dominates the treewidth of the CNF primal graph: it is no greater and can be bounded when the treewidth of the primal graph is unbounded. This new result leads to a tighter bound on the complexity of model counting. Second, we show that the output of the algorithm can be converted in linear time to a sentential decision diagram (SDD), which leads to a tighter bound on the complexity of compiling CNFs into SDDs.

1 Introduction

Decision-DNNF is a tractable propositional language that is a strict subset of DNNF. One key role of this language is in the complexity analysis of modern model counters. We will therefore start with a motivation of model counters.

Model counting is the problem of counting the number of satisfying assignments of a Boolean formula. It has various applications, such as inference in Bayesian networks [1,5]. Although model counting has been shown to be a hard problem (#P-complete [21]), there are two common approaches that have proven effective in practice.

One approach is based on DPLL [11,10], which is a family of algorithms that were initially developed for SAT: the problem of deciding whether a Boolean formula has a satisfying assignment. In essence, it is a systematic search algorithm that searches the space of truth assignments until finding a satisfying one or identifying that such an assignment does not exist. This search method can easily be extended to compute the number of satisfying assignments of the formula. Simply, by not stopping the search when a single satisfying assignment is found, and exhaustively continuing to look for all other satisfying assignments, one can obtain a naive model counter. To make this approach more practical, various sophisticated techniques were incorporated into the core exhaustive DPLL algorithm, such as component analysis [13] and formula caching [14,1,20].

Another approach for model counting is based on *knowledge compilation*. The basic idea of knowledge compilation is to compile a Boolean formula represented

B. O'Sullivan (Ed.): CP 2014, LNCS 8656, pp. 42–57, 2014.
© Springer International Publishing Switzerland 2014

in a source language into a target language that supports model counting in polytime. Negation normal form (NNF) circuits have been established as the basis of a number of such languages [9]. These circuits have *and* nodes (representing conjunctions) and *or* nodes (representing disjunctions) as internal gates, and literals or constants as inputs (see Fig. 1(c)). In [9], two fundamental properties on NNF circuits are identified to ensure the tractability of model counting: *decomposability* and *determinism*. Decomposability is a property of *and* nodes, requiring that the children of *and* nodes share no variables. Determinism is a property of *or* nodes, requiring that each two children of an *or* node be mutually exclusive (i.e., contradict each other). Determinism and decomposability characterize deterministic-DNNF (d-DNNF), a strict subset of DNNF [6], which includes other languages such as sentential decision diagrams (SDD) [8], free binary decision diagrams (FBDD) [3], and ordered binary decision diagrams (OBDD) [4]. Although d-DNNF is the most general language known that supports efficient model counting, a strict subset, Decision-DNNF, has been used in state-of-the-art model counters based on knowledge compilers [7,15].

Although the approaches described above look conceptually different than each other, a strong connection between them has been established [12]. In particular, the *traces*[1] of the searches performed by state-of-the-art model counters has been shown to be in Decision-DNNF. In other words, model counters based on exhaustive DPLL effectively generates the compilation of the Boolean formula in Decision-DNNF. By this result, Decision-DNNF has the role of bridging model counters and knowledge compilers. More importantly, any new result pertaining to Decision-DNNFs will have a possibly significant further impact on model counters. For instance, the relationship between Decision-DNNFs and FBDDs has been recently studied in [2]. Accordingly, Decision-DNNFs can be converted into FBDDs with only a quasipolynomial increase in the representation size. This result allowed the authors to show new exponential lower bounds on Decision-DNNFs, by leveraging the existing lower bounds on FBDDs, which are immediately applicable to model counters.

In this work, we present new results on Decision-DNNFs. First, we introduce a new notion of width for CNFs, called *decision-width*. We show a compilation algorithm that can compile CNFs into Decision-DNNFs in time and space that are exponential only in decision-width. This new width is no greater than the treewidth of the CNF primal graph, and can be bounded while the latter is unbounded. This result not only improves the existing complexity results on d-DNNF compilation but also the existing results on the complexity of model counting. Second, we show that Decision-DNNFs constructed by our algorithm can be converted to SDDs in linear time. SDD is a recently discovered tractable language that is a strict superset of the influential language OBDD. It comes with many interesting properties, including a polytime `Apply`[2] operation and canonicity, which are two key features underlying the success of OBDDs. Our result leads to a tighter bound on the complexity of compiling CNFs into SDDs.

[1] See Section 4 for a detailed discussion of the trace of an exhaustive search algorithm.

[2] `Apply` takes two SDDs and uses a binary operator to combine them.

The rest of the paper is organized as follows. Section 2 starts with technical preliminaries. Section 3 then introduces decision-width and discusses a compilation algorithm that can compile CNFs into Decision-DNNFs in time and space that are exponential only in this width. This is followed by a comparison of decision-width and the treewidth of the CNF primal graph. Next, Section 4 discusses the importance of Decision-DNNFs in model counting by reviewing in detail the strong connection that has been established between Decision-DNNFs and model counters. Section 5 shows that the output of our algorithm for compiling CNFs into Decision-DNNFs can be transformed in linear time to SDDs. We close with a discussion of related work and some concluding remarks. Due to space limitations, some proofs are delegated to the full version of the paper.[3]

2 Technical Preliminaries

Upper case letters (e.g., X) will denote variables and lower case letters (e.g., x) will denote their instantiations. Bold upper case letters (e.g., $\mathbf{X}$) will denote sets of variables and bold lower case letters (e.g., $\mathbf{x}$) will denote their instantiations.

A *Boolean function* f over variables $\mathbf{Z}$ maps each instantiation $\mathbf{z}$ of variables $\mathbf{Z}$ to *true* or $false$. The *conditioning* of f on instantiation $\mathbf{x}$, written $f|\mathbf{x}$, is a *subfunction* that results from setting variables $\mathbf{X}$ to their values in $\mathbf{x}$. A conjunction is *decomposable* if each pair of its conjuncts share no variables. A disjunction is *deterministic* if each pair of its disjuncts are mutually exclusive. A *negation normal form* (NNF) circuit is a rooted DAG whose internal nodes correspond to disjunctions and conjunctions, and whose leaf nodes correspond to literals or the constants $\top$ (*true*) and $\bot$ (*false*). An NNF circuit is decomposable and deterministic (called a d-DNNF) iff each of its conjunctions is decomposable and each of its disjunctions is deterministic; see Fig. 1(c). We will identify an NNF circuit by its root node N, use $Vars(N)$ to denote the set of variables mentioned by circuit N, and $F(N)$ to denote the Boolean function represented by N. The *size* of NNF circuit N, denoted $|N|$, is the total number of edges in the circuit.

The literals of variable X are denoted by x and $\neg x$. A *conjunctive normal form* (CNF) is a set of clauses, where each clause is a disjunction of literals, and the set represents the conjunction of its clauses (e.g., $\{x \vee \neg y \vee \neg z,\ y \vee z,\ \neg x\}$ represents the Boolean formula $(x \vee \neg y \vee \neg z) \wedge (y \vee z) \wedge \neg x$). Conditioning a CNF Δ on literal ℓ, denoted $\Delta|\ell$, amounts to removing literal $\neg\ell$ from all clauses and then dropping all clauses that contain literal ℓ.

3 Compiling CNFs into Decision-DNNFs

The purpose of this section is to show an algorithm that compiles CNFs into Decision-DNNFs with a complexity guarantee. To analyze the complexity of the algorithm, we will also introduce a new notion of width and study its properties.

---

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

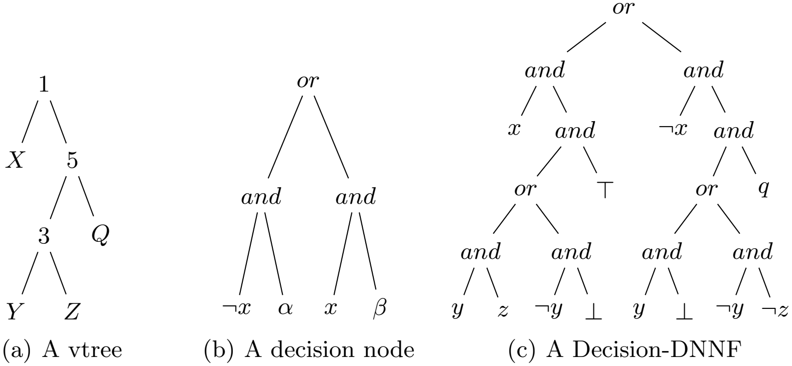


Fig. 1. A vtree, a decision node, and a Decision-DNNF

3.1 Decision-DNNF

A *decision node* is a special form of an *or* node, which is depicted in Fig. 1(b) where X is a variable, and α and β are arbitrary NNF nodes. A d-DNNF is called a Decision-DNNF iff each of its *or* nodes is a decision node; see Fig. 1(c). In this case, determinism is always ensured by the decision nodes.

3.2 Decision Vtrees

Both the width and the compilation algorithm we will present in this section are driven by a tree-structure, which is introduced next. A *vtree* for a set $\mathbf{Z}$ of variables is a rooted, full binary tree whose leaves are in one-to-one correspondence with variables in $\mathbf{Z}$. Figure 1(a) depicts an example vtree. We will use v^l and v^r to refer to the left and right children of an internal vtree node v. We will also use $Vars(v)$ to denote the set of variables at or below a vtree node v. A vtree node is called a *Shannon node* iff its left child is a leaf. A vtree in which every node is a Shannon node will be called *right-linear*. Given a vtree v, we will sometimes refer to v as the root of the vtree.

A vtree for a CNF is a vtree over the CNF variables. Our focus is on a special type of vtrees, defined next.

Definition 1 (decision vtree). *A clause is compatible with an internal vtree node v iff the clause mentions some variables inside v^l and some variables inside v^r. A vtree for CNF Δ is said to be a decision vtree for Δ iff every clause in Δ is compatible with only Shannon nodes.*[4]

Figure 1(a) depicts a decision vtree for the CNF $\{y \vee \neg z,\ \neg x \vee z,\ x \vee \neg y,\ x \vee q\}$. We will later show that one can always construct a decision vtree for any CNF.

[4] A unit clause (one containing a single literal) is not compatible with any vtree node. Hence, a unit clause trivially satisfies the condition of being compatible with only Shannon nodes.

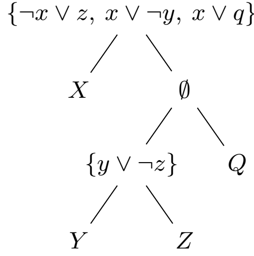


Fig. 2. Distributing the clauses of CNF $\{y \vee \neg z,\ \neg x \vee z,\ x \vee \neg y,\ x \vee q\}$ over a vtree

3.3 A Compilation Algorithm

We will next present an algorithm that compiles a CNF into a Decision-DNNF using a decision vtree for the CNF. This compilation method is given by Algorithm 1, which takes a decision vtree v and an auxiliary CNF S over the variables of vtree v (S is initially empty). The CNF Δ to be compiled is passed with the vtree as follows. Each clause of Δ is assigned to the lowest vtree node that contains the clause variables. Figure 2 depicts an example of how clauses are assigned to vtree nodes. Note that the (non-unit) clauses are assigned only to Shannon nodes as the vtree is a decision vtree. We use $Clauses(v)$ to denote the clauses assigned to a vtree node v. We also use $CNF(v)$ to denote the clauses assigned to all nodes in the vtree rooted at v. A recursive call $\texttt{c2d}(v, S)$ will return a Decision-DNNF for $CNF(v) \cup S$. The algorithm keeps a cache at every vtree node, which is indexed by S.

Consider now a run of Algorithm 1. An *or* node can only be constructed on Line 4. Accordingly, each *or* node created by this algorithm is a decision node. Using this fact with an inductive argument is enough to prove the soundness of the algorithm.

Lemma 1. *Let v be a decision vtree for $CNF(v)$. Let S be a CNF over $Vars(v)$ whose clauses are compatible with only Shannon nodes of v. The call* $\texttt{c2d}(v, S)$ *to Algorithm 1 returns a Decision-DNNF equivalent to $CNF(v) \cup S$.*

Proof. The proof is by induction on vtree nodes. The base case is when v is a leaf node. This case is trivially satisfied by Line 2. Assume now that v is an internal node. As an induction hypothesis, consider that for each vtree node v' below v, the call $\texttt{c2d}(v', S')$ computes a Decision-DNNF equivalent to $CNF(v') \cup S'$, where S' is a CNF over $Vars(v')$ whose clauses are compatible with only Shannon nodes of v'. During the call to v, we will compute a Decision-DNNF equivalent to $CNF(v) \cup S$ by utilizing the following decomposition of a Boolean function f (known as Shannon decomposition): $f = (x \wedge f|x) \vee (\neg x \wedge f|\neg x)$. Note that in our context $f = CNF(v) \cup S$. Assume v is a Shannon node. Then v^l is a leaf node (with variable X). The possible clauses that can be assigned to v^l are $\{x\}$ and $\{\neg x\}$. Lines 4–4 consider all four possible assignments of those two

Algorithm 1. $\texttt{c2d}(v, S)$
$cache(v, \Delta)$ is a hash table that maps v and Δ into a Decision-DNNF.
$terminal(\Delta)$ returns the literal or constant equivalent to Δ.

Input: v : a vtree node, S : a CNF over $Vars(v)$.
Output: A Decision-DNNF for $CNF(v) \cup S$.

1 **if** $cache(v, S) \neq nil$ **then return** $cache(v, S)$ $C \leftarrow Clauses(v)$
2 **if** *v is a leaf* **then return** $terminal(C \cup S)$ **if** *v is a Shannon node* **then**
3 $\quad X \leftarrow$ variable of v^l
4 $\quad$ **if** $\{x\}$ *and* $\{\neg x\}$ *assigned to* v^l **then** $\alpha \leftarrow \bot$ **else if** $\{x\}$ *assigned to* v^l **then**
$\quad \alpha \leftarrow x \wedge \texttt{c2d}(v^r, (C \cup S)|x)$ **else if** $\{\neg x\}$ *assigned to* v^l **then**
$\quad \alpha \leftarrow \neg x \wedge \texttt{c2d}(v^r, (C \cup S)|\neg x)$ **else**
$\quad \alpha \leftarrow \Big(x \wedge \texttt{c2d}(v^r, (C \cup S)|x)\Big) \vee \Big(\neg x \wedge \texttt{c2d}(v^r, (C \cup S)|\neg x)\Big)$
5 **else**
6 $\quad S_1 \leftarrow$ clauses in S that only mention variables in v^l
7 $\quad S_2 \leftarrow$ clauses in S that only mention variables in v^r
8 $\quad \alpha \leftarrow \Big(\texttt{c2d}(v^l, S_1) \wedge \texttt{c2d}(v^r, S_2)\Big)$
9 $cache(v, S) \leftarrow \alpha$
10 **return** α

clauses to v^l: (1) both $\{x\}$ and $\{\neg x\}$ are assigned and $f = \bot$, (2) only $\{x\}$ is assigned and $f = x \wedge f|x$, (3) only $\{\neg x\}$ is assigned and $f = \neg x \wedge f|\neg x$, and (4) no clause is assigned and $f = (x \wedge f|x) \vee (\neg x \wedge f|\neg x)$. Except for the first case, in which f is trivially computed as $\bot$, by the induction hypothesis, $\texttt{c2d}(v^r, (C \cup S)|x)$ and $\texttt{c2d}(v^r, (C \cup S)|\neg x)$ compute Decision-DNNFs for $f|x$ and $f|\neg x$, respectively.[5] Note that we construct an *or* node only in the last case, which is a decision node. So, when v is a Shannon node, we compute a Decision-DNNF equivalent to $CNF(v) \cup S$. Assume now that v is a non-Shannon node. In this case, C must be empty because the vtree is a decision vtree. Thus, $CNF(v) = CNF(v^l) \cup CNF(v^r)$. Also, S cannot contain any clause that mentions variables from both v^l and v^r as no clause in S can be compatible with v. Then, by the induction hypothesis, on Line 8, we compute a Decision-DNNF equivalent to $CNF(v) \cup S$. □

Corollary 1. *Let v be a decision vtree for $CNF(v)$. The call $\texttt{c2d}(v, \{\})$ to Algorithm 1 returns a Decision-DNNF that is equivalent to $CNF(v)$.*

For instance, when the vtree in Fig. 2 is passed to Algorithm 1, it computes the Decision-DNNF in Fig. 1(c). To analyze time and space complexities of the algorithm, we next introduce a new notion of width.

[5] If a clause is not compatible with a vtree node v then every conditioning of the clause will also not be compatible with v.

3.4 Decision-Width

Before defining the new notion of width, we will introduce two concepts.

Definition 2. *Consider a CNF and a corresponding vtree. Let v be an internal vtree node. The context clauses of v are the clauses that mention variables inside v and outside v.*

For example, consider the CNF $\{y \vee \neg z, \neg x \vee z, x \vee \neg y, x \vee q\}$ and the vtree node $v = 5$ in Fig. 1(a). The context clauses of v are $\{\neg x \vee z, x \vee \neg y, x \vee q\}$.

Definition 3. *Consider a CNF Γ and a set of variables $\mathbf{V}$. We denote by $CNFs(\Gamma, \mathbf{V})$ the set of CNFs that is obtained from conditioning Γ on each instantiation $\mathbf{v}$ of $\mathbf{V}$.*

Consider the CNF $\Gamma = \{x \vee y \vee z, x \vee y \vee q, \neg x \vee \neg y \vee z, x \vee \neg y \vee z\}$ and the set of variables $\mathbf{V} = \{Y\}$. Then,

$$\Gamma|y = \{\neg x \vee z, x \vee z\},$$
$$\Gamma|\neg y = \{x \vee z, x \vee q\},$$
$$CNFs(\Gamma, \mathbf{V}) = \{\{\neg x \vee z, x \vee z\}, \{x \vee z, x \vee q\}\}.$$

We are now ready to introduce the new notion of width.

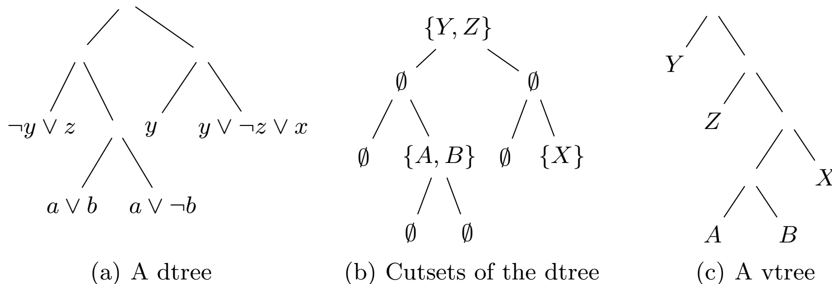
Definition 4 (decision-width). *Consider a CNF Δ and a corresponding decision vtree. Let v be an internal vtree node with context clauses Γ. Let $\mathbf{Y}$ be variables outside v. Then, the width of v is the ceiling of $\log(|CNFs(\Gamma, \mathbf{Y})|)$, where $\log 0$ is defined as 0. The decision-width of the decision vtree is the largest width of any of its internal nodes minus 1. The decision-width of the CNF is the smallest decision-width attained by any of its decision vtrees.*

For instance, consider the vtree in Fig. 1(a). Assuming that the vtree corresponds to the CNF $\{y \vee \neg z, \neg x \vee z, x \vee \neg y, x \vee q\}$, the width of node $v = 5$ is 1, and the decision-width of the vtree is 0.

Having defined decision-width, we can now establish the complexity of Algorithm 1.

Theorem 1. *If decision vtree v is over n variables and has decision-width w, and if $CNF(v)$ has size m, then the call* `c2d`$(v, \{\})$ *to Algorithm 1 takes time in $O(nm2^w)$ and returns a Decision-DNNF whose size is in $O(n2^w)$.*

Proof (Sketch). Each distinct call to a Shannon node (Lines 2–4) takes time in $O(2m)$: we perform at most two conditionings of $C \cup S$, which has at most m clauses, on a single literal. This process contributes to the size at most three nodes, each having two children. Each distinct call to a non-Shannon node (Lines 5–8) takes time in $O(m)$: we partition the set S, which has at most m clauses, into two subsets. This case contributes to the size one node with two children. Also, due to caching, the number of distinct calls to a vtree node v is at most 2^k where k is the width of v. As there are $O(n)$ nodes in the vtree, Algorithm 1 takes time in $O(nm2^w)$ and returns a Decision-DNNF whose size is in $O(n2^w)$. □



(a) A dtree (b) Cutsets of the dtree (c) A vtree

Fig. 3. A dtree, its cutsets, and a vtree for the CNF $\{y, \neg y \vee z, y \vee \neg z \vee x, a \vee b, a \vee \neg b\}$

3.5 Relationship to Treewidth

One of the classical parameters for characterizing the structural properties of a CNF is treewidth [19]. This is a property of some graph abstraction of the CNF, such as the primal, dual or incidence graph. We will now compare decision-width with the treewidth of the CNF primal graph, and show that decision-width strictly dominates the treewidth.

The *primal graph* of a CNF is obtained by treating CNF variables as graph nodes, while adding an edge between two variables iff they appear in the same clause. We will use *twp* to denote the treewidth of the primal graph.

We will now compare decision-width with *twp*. First, we will show that decision-width dominates *twp*.

Theorem 2. *Consider a CNF whose primal graph has treewidth w. We can construct a decision vtree of this CNF whose decision-width is no greater than w.*

Proof (Sketch). The primal graph must have a tree decomposition in the form of a dtree [6] that has width w. A *dtree* of a CNF is a rooted, full binary tree whose leaves are in one-to-one correspondence with the CNF clauses; see Fig. 3(a). We define the cutset of an internal dtree node d as the variables that appear in the left child of d and in the right child of d but not in any cutset of the ancestors of d. Also, the cutset of a leaf dtree node d is defined as the variables of d that do not appear in any cutset of the ancestors of d. Figure 3(b) shows the cutsets of a dtree. We construct decision vtrees from dtrees using the following recursive procedure. At a leaf dtree node d, we construct a right-linear vtree from the variables appearing in the cutset of d. At an internal dtree node d, we recursively construct a decision vtree v^l for the dtree d^l, and a decision vtree v^r for the dtree d^r. We then create another vtree node v by assigning v^l as v's left child and v^r as v's right child. Finally, we create and return a right-linear vtree using the variables in the cutset of dtree node d, with vtree node v as the right-most child of this right-linear vtree. Due to cutset properties, the resulting vtree is a decision vtree. Figure 3(c) shows a decision vtree obtained from the dtree in Fig. 3(a) using the method we just described. The full paper shows that the decision-width of the resulting decision vtree is no greater than w. □

The above result shows that decision-width dominates *twp*. It also implies that one can always construct a decision vtree for any CNF. We will next show that decision-width can be bounded when *twp* is unbounded.

Theorem 3. *The CNF* $\Delta_n = \{x_1 \vee \ldots \vee x_n\}$, $n \geq 1$, *has a primal graph with treewidth* $n-1$ *and decision-width* 0.

Proof. The primal graph of Δ_n is a complete graph, and complete graphs are known to have unbounded treewidth, which is $n-1$ in this case. Consider the right-linear vtree induced by the variable ordering $X_1, \ldots, X_n$. That is, the left child of the vtree root v is X_1. The left child of v^r is X_2, and so on. Consider a vtree node v' whose left child is X_i. Let Γ be the context clauses of v'. If $i = 1$, then Γ is empty and the width of v' is 0. Otherwise, $\Gamma = \{x_1 \vee \ldots \vee x_n\}$. Let $\mathbf{Y}$ be the variables outside v'. Then, the set of CNFs that are obtained from $\Gamma|\mathbf{y}$ is $\{\{\}, \{x_i \vee \ldots \vee x_n\}\}$. The width of v' is then 1. The decision-width of the vtree is then 0. □

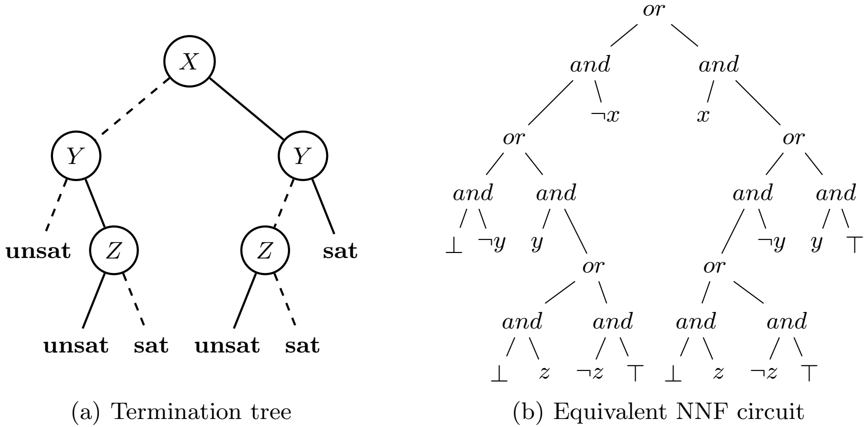
The best known upper bounds for the complexities of compiling CNFs into both Decision-DNNFs and d-DNNFs are exponential in the treewidth of the CNF primal graph [12]. As decision-width strictly dominates treewidth, and also Decision-DNNFs is a strict subset of d-DNNFs, by Theorem 1, we obtain a tighter upper bound both for Decision-DNNF and d-DNNF compilation. Furthermore, as the traces of most model counters are in Decision-DNNF (see next section), this result also provides a tighter bound for model counters based on Decision-DNNFs (under some assumptions to be discussed next).

4 Decision-DNNFs and Model Counters

In this section, we will discuss the close relationship between Decision-DNNFs and model counters based on exhaustive DPLL.

The original DPLL algorithm was developed for SAT. It is a systematic search algorithm that searches the space of truth assignments until finding a satisfying one or identifying that such an assignment does not exist. In particular, given a Boolean formula f, it chooses a variable X of the formula, and then considers two cases recursively, which correspond to $f|x$ and $f|\neg x$. It then decides that the formula f is satisfiable when at least one of $f|x$ and $f|\neg x$ is satisfiable. This method can easily be extended to compute the number of satisfying assignments of the formula by not stopping the search when a single satisfying assignment is found. That is, by exhaustively continuing to look for all other satisfying assignments, one can obtain a naive model counter.

The tree in Fig. 4(a) shows all the paths that are traversed during an exhaustive DPLL on a Boolean formula. Each circled node represents a variable on which two decisions are performed: the variable is set to false (dashed edge) or set to true (solid edge). This way, paths from the root to leaf nodes represent (partial) variable assignments. Each leaf node then represents the result of the search when the variable assignment on the path from the root to the leaf is applied on the Boolean formula, with the label **unsat** being unsatisfiable and the



(a) Termination tree (b) Equivalent NNF circuit

Fig. 4. The trace of an exhaustive DPLL

label **sat** being satisfiable. This tree is called the *trace* of the search performed by an exhaustive DPLL [12]. Note that one can think of each circled node of the tree as an *or* node, by utilizing the following conversion:

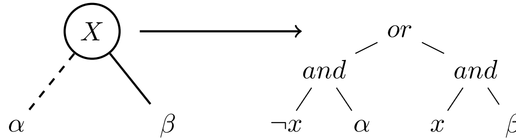


Figure 4(b) is the NNF circuit obtained from Fig. 4(a) using the above conversion, and also replacing each **sat** with $\top$, and each **unsat** with $\bot$. One can always interpret the trace of an exhaustive DPLL as an equivalent NNF circuit, which turns out to satisfy both decomposability and determinism. In fact, the traces of the searches performed by such model counters correspond to FBDDs [12]. Moreover, when these model counters are augmented with component analysis, their traces correspond to Decision-DNNFs [12].

This connection has two implications. First, it allows one to translate Decision-DNNF lower bounds immediately into lower bounds on the complexity of model counters. Second, but under some assumptions, it allows one to translate Decision-DNNF upper bounds into ones on the complexity of model counters. For example lower bounds, it was recently shown that Decision-DNNFs can be converted into FBDDs with only a quasipolynomial increase in size [2]. As a result, known lower bounds for FBDDs immediately translate into lower bounds on model counters whose traces are in Decision-DNNF (see [2] for examples).

Translating upper bounds on Decision-DNNF to upper bounds on arbitrary model counters, however, is not as direct. Here, one needs, for example, to assume that the traces of the model counter are optimal, and that the time complexity of the counter is polynomial in the size of the trace. Under these assumptions, an

upper bound on Decision-DNNF translates directly into an upper bound on the model counter. Interestingly enough, Algorithm 1 satisfies the second condition. The algorithm does not satisfy the first condition, but we know that its traces (i.e., compilations) are bounded exponentially only by the decision-width. Since this width dominates the primal graph treewidth, we now have a tighter upper bound on model counting in general (realized by Algorithm 1). We also have a tighter upper bound on any model counter that satisfies the previous conditions.

5 From Decision-DNNF to SDD

We finally show a new complexity result on compiling CNFs into SDDs.[6] The existing upper bound on SDDs is exponential in the treewidth of the CNF primal graph. We show a tighter bound that is exponential only in decision-width, which strictly dominates treewidth as shown in Sect. 3.5. We will obtain this result by showing that Decision-DNNFs generated by Algorithm 1 can be converted into compressed and trimmed SDDs in linear time and by at most doubling the size. That is, Algorithm 1 is effectively compiling SDDs.

Note that the output of Algorithm 1 is a special form of Decision-DNNF. In particular, the vtree used in the compilation provides the generated Decision-DNNF with a specific structure. That is, every node N in the Decision-DNNF is associated with some vtree node v in the following way:

- an *and* node is associated with v when $Vars(N^l) \subseteq Vars(v^l)$ and $Vars(N^r) \subseteq Vars(v^r)$, and
- an *or* node, $(x \wedge \alpha) \vee (\neg x \wedge \beta)$, is associated with v when $X \subseteq Vars(v^l)$ and $Vars(\alpha \cup \beta) \subseteq Vars(v^r)$.

For instance, each node of the Decision-DNNF in Fig. 1(c) is associated with a vtree node in Fig. 1(a).

Algorithm 2 shows how to convert a Decision-DNNF into an SDD. It takes a Decision-DNNF that is constructed by Algorithm 1 and computes two compressed and trimmed SDDs that are equivalent to the Boolean functions represented by N and the negation of N. The conversion is done in a bottom-up fashion. Terminal SDDs are obtained from leaf nodes (Line 1). Then, an *or* node (Lines 1–3) or an *and* node (Lines 4–6) is obtained from the results of recursive calls. To prevent redundant calculations, the results are cached (Line 7). The following theorem establishes the soundness and the complexity of the algorithm.

Theorem 4. *If N is a Decision-DNNF generated by Algorithm 1 and has size m, then the call* `d2sdd`(N) *takes time in $O(m)$, and returns two compressed and trimmed SDDs for $F(N)$ and $\neg F(N)$, whose sizes are in $O(m)$.*

Proof. The proof is by induction on NNF nodes. The base case is when N is a leaf node, which is satisfied by Line 1. As an induction hypothesis (IH), assume

[6] The definition of SDD and some of its properties are delegated to Appendix A.

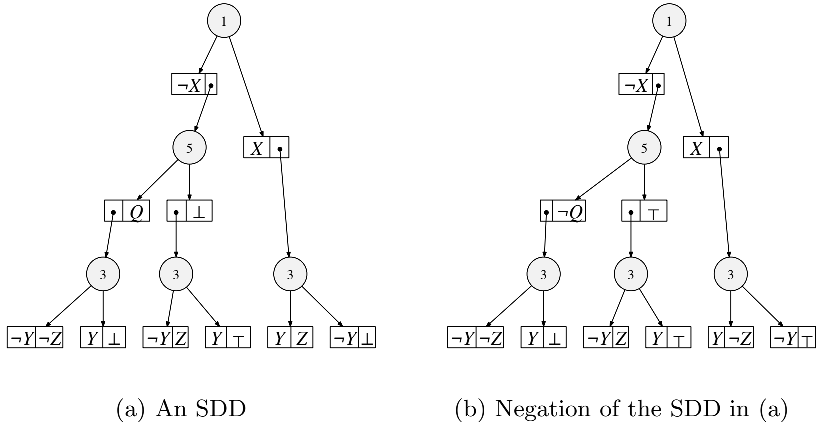
Algorithm 2. `d2sdd`(N)
cache(N) is a hash table that maps N to an SDD.
terminal(N) returns the terminal SDD equivalent to $F(N)$.
unique(α) removes an element from α if its prime is $\bot$. It then returns s if $\alpha = \{(p_1, s), (p_2, s)\}$ or $\alpha = \{(\top, s)\}$; p_1 if $\alpha = \{(p_1, \top), (p_2, \bot)\}$; else the unique SDD node with elements α.

Input: N : a Decision-DNNF generated by Algorithm 1.
Output: Two compressed and trimmed SDDs equivalent to $F(N)$ and $\neg F(N)$.

1 **if** $cache(N) \neq nil$ **then return** $cache(N)$ **if** N *is a leaf node* **then return** $terminal(N), terminal(\neg N)$ **if** $N = (x \wedge N_1) \vee (\neg x \wedge N_2)$ **then**
2 $\quad s_1, \neg s_1 \leftarrow$ `d2sdd`(N_1), $\quad s_2, \neg s_2 \leftarrow$ `d2sdd`(N_2)
3 $\quad \alpha \leftarrow unique(\{(x, s_1), (\neg x, s_2)\})$, $\quad \neg\alpha \leftarrow unique(\{(x, \neg s_1), (\neg x, \neg s_2)\})$
4 **else** // $N = N^l \wedge N^r$
5 $\quad p, \neg p \leftarrow$ `d2sdd`(N^l), $\quad s, \neg s \leftarrow$ `d2sdd`(N^r)
6 $\quad \alpha \leftarrow unique(\{(p, s), (\neg p, \bot)\})$, $\quad \neg\alpha \leftarrow unique(\{(p, \neg s), (\neg p, \top)\})$
7 $cache(N) \leftarrow \alpha, \neg\alpha$
8 **return** $\alpha, \neg\alpha$

that for every NNF circuit whose root N' is below N, and whose size $|N'|$ is k, the call `d2sdd`(N') takes time in $O(k)$, and returns two compressed and trimmed SDDs for $F(N')$ and $\neg F(N')$, whose sizes are in $O(k)$. Suppose that N is an *or* node, $(x \wedge N_1) \vee (\neg x \wedge N_2)$, where N_1 and N_2 are NNF nodes. Let $|N_1| = m_1$ and $|N_2| = m_2$. Then, $m = 6 + m_1 + m_2$. By the IH, the call `d2sdd`(N_1) (Line 2) takes time in $O(m_1)$, and returns the SDDs for $F(N_1)$ and $\neg F(N_1)$, whose sizes are in $O(m_1)$. Similarly, the call `d2sdd`(N_2) (Line 2) takes time in $O(m_2)$, and returns the SDDs for $F(N_2)$ and $\neg F(N_2)$, whose sizes are in $O(m_2)$. Then, using the structure of N, we construct the SDD for $F(N)$ (Line 3), whose size is at most $2 + O(m_1) + O(m_2) = O(m)$. To compute the SDD for $\neg F(N)$, we just need to negate the subs of the SDD for $F(N)$, which are already computed by the recursive calls. So, the call to an *or* node takes time in $O(m)$. Assume now that N is an *and* node, $N^l \wedge N^r$, where N^l and N^r are NNF nodes. Let $|N^l| = m_1$ and $|N^r| = m_2$. Then, $m = 2+m_1+m_2$. By the IH, the call `d2sdd`(N^l) (Line 5) takes time in $O(m_1)$, and returns the SDDs for $F(N^l)$ and $\neg F(N^l)$, whose sizes are in $O(m_1)$. Similarly, the call `d2sdd`(N^r) (Line 5) takes time in $O(m_2)$, and returns the SDDs for $F(N^r)$ and $\neg F(N^r)$, whose sizes are in $O(m_2)$. We then construct the SDD for $F(N)$ (Line 6) by making use of the following: $F(N) = \big(F(N^l) \wedge F(N^r)\big) \vee \big(\neg F(N^l) \wedge \bot\big)$. Thus, the constructed SDD has size at most $2 + O(2m_1) + O(m_2) = O(2m)$. Again, the SDD for $\neg F(N)$ is computed by negating the subs. Thus, the call to an *and* node takes time in $O(m)$. □

For instance, when we pass the Decision-DNNF in Fig. 1(c) to Algorithm 2, the algorithm computes the compressed and trimmed SDDs in Fig. 5.



(a) An SDD

(b) Negation of the SDD in (a)

Fig. 5. An SDD and its negation (both defined over the vtree in Fig. 1(a))

The result we obtained in this section shows that we can compile CNFs into SDDs in time and space that are exponential only in decision-width. As the best known upper bound for compiling CNFs into SDDs is exponential in the treewidth of the CNF primal graph [8], we obtain a tighter upper bound on the complexity of SDD compilation.

6 Related Work

The notion of decision-width is closely connected to another notion of width that we introduced recently [16], called *CV-width*. In this latter work, an algorithm was provided for compiling CNFs into structured DNNFs, based on vtrees, with space and time guarantees that are exponential only in CV-width. This width was defined over arbitrary vtrees. However, when restricted to decision vtrees, it reduces to the notion of decision-width defined in this paper. In fact, when the compilation algorithm of [16] is passed a decision vtree, it reduces to the compilation algorithm given in this paper. This is an interesting phenomenon, where language fragments are generated by restricting the type of vtrees passed to a compilation algorithm. In fact, in [16], we also showed that when right-linear vtrees are used, the compilation algorithm yields OBDD compilations (with new complexity bounds that improve on what existed before).

When the current work is considered from the viewpoint of [16], it corresponds to identifying a class of vtrees (decision vtree), together with a corresponding notion of width (decision-width) and a corresponding fragment of structured DNNF. These vtrees are particularly important given their role in model counting and sentential decision diagrams.

Interestingly enough, decision vtrees and the method for constructing them have been used twice in the past, yet without realizing the corresponding prop-

erties and guarantees that we discussed [18,8]. Moreover, no specific notion of width has been defined before based on this restricted type of vtrees. In [18], these vtrees were used to show an upper bound on the compilation of structured DNNFs. Similarly, in [8], these vtrees were used to show an upper bound on the compilation of SDDs. In both cases, the bounds were in terms of the primal graph treewidth. In this work, we used these vtrees to provide an upper bound on a subset of Decision-DNNF, which is included in both structured DNNF and SDDs. Moreover, our bound (based on decision-width) is tighter than the ones based on the primal graph treewidth. More importantly though, we have identified decision vtrees as a distinct class of vtrees for the first time, explicated their characteristic property (clauses are compatible only with Shannon nodes), equipped them with a corresponding notion of width, and characterized their corresponding compilations (as a subset of Decision-DNNF).

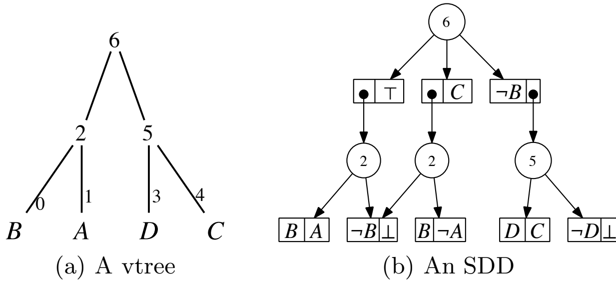
7 Conclusion

In this paper, we presented new results on Decision-DNNFs. We showed a compilation algorithm that compiles CNFs into Decision-DNNFs. To analyze the complexity of the algorithm, we defined a new notion of width, called decision-width, and showed that the algorithm has time and space complexities that are exponential only in decision-width. To better evaluate decision-width, we compared it with the treewidth of the CNF primal graph, and showed that decision-width strictly dominates treewidth. Not only did we obtain a tighter upper bound for compiling CNFs into d-DNNFs (through Decision-DNNFs) but we also obtained a tighter upper bound on model counting, for which the previously best known bounds were both exponential in the treewidth. We finally showed that Decision-DNNFs compiled by the algorithm can be transformed into SDDs in linear time. This led to a tighter upper bound on compiling CNFs into SDDs, for which the previously best known bound was exponential in the treewidth of the CNF primal graph.

Acknowledgements. This work has been partially supported by ONR grant #N00014-12-1-0423 and NSF grant #IIS-1118122.

A Sentential Decision Diagrams

Consider a Boolean function $f(\mathbf{X}, \mathbf{Y})$ with disjoint sets of variables $\mathbf{X}$ and $\mathbf{Y}$. If $f(\mathbf{X}, \mathbf{Y}) = (p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})) \vee \ldots \vee (p_n(\mathbf{X}) \wedge s_n(\mathbf{Y}))$, the set $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ is called an $(\mathbf{X}, \mathbf{Y})$-decomposition of the function f and each pair (p_i, s_i) is called an *element* of the decomposition [17]. The decomposition is further called an $(\mathbf{X}, \mathbf{Y})$-*partition* iff the p_i's form a partition [8]. That is, $p_i \neq \bot$ for all i; and $p_i \wedge p_j = \bot$ for $i \neq j$; and $\bigvee_i p_i = \top$. In this case, each p_i is called a *prime* and each s_i is called a *sub*. An $(\mathbf{X}, \mathbf{Y})$-partition is *compressed* iff its subs are distinct, i.e., $s_i \neq s_j$ for $i \neq j$ [8]. Compression can always be ensured by



(a) A vtree (b) An SDD

Fig. 6. Function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$

repeatedly disjoining the primes of equal subs. Moreover, a function $f(\mathbf{X}, \mathbf{Y})$ has a *unique,* compressed $(\mathbf{X}, \mathbf{Y})$-partition. Finally, the *size* of a decomposition, or partition, is the number of its elements.

Note that $(\mathbf{X}, \mathbf{Y})$-partitions generalize Shannon decompositions, which fall as a special case when $\mathbf{X}$ contains a single variable. OBDDs result from the recursive application of Shannon decompositions, leading to decision nodes that branch on the states of a single variable (i.e., literals). As we show next, SDDs result from the recursive application of $(\mathbf{X}, \mathbf{Y})$-partitions, leading to decision nodes that branch on the state of multiple variables (i.e., arbitrary sentences).

Consider the vtree in Fig. 6(a). Consider also the Boolean function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ over the same variables. Node $v = 6$ is the vtree root. Its left subtree contains variables $\mathbf{X} = \{A, B\}$ and its right subtree contains $\mathbf{Y} = \{C, D\}$. Decomposing function f at node $v = 6$ amounts to generating an $(\mathbf{X}, \mathbf{Y})$-partition of function f. The unique compressed $(\mathbf{X}, \mathbf{Y})$-partition here is

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

This partition is represented by the root node of Fig. 6(b). This node, which is a circle, represents a *decision node* with three branches. Each branch corresponds to one element $\boxed{p|s}$ of the above partition. Here, the left box contains a prime when the prime is a literal or a constant; otherwise, it contains a pointer to a prime. Similarly, the right box contains a sub or a pointer to a sub. The three primes are decomposed recursively, but using the vtree rooted at $v = 2$. Similarly, the subs are decomposed recursively, using the vtree rooted at $v = 5$. This recursive decomposition process moves down one level in the vtree with each recursion, terminating when it reaches leaf vtree nodes. The full SDD for this example is depicted in Fig. 6(b).

SDDs obtained from the above process are called *compressed* iff the $(\mathbf{X}, \mathbf{Y})$-partition computed at each step is compressed. These SDDs may contain trivial decision nodes which correspond to $(\mathbf{X}, \mathbf{Y})$-partitions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$. When these decision nodes are removed (by directing their parents to α), the resulting SDD is called *trimmed.* Compressed and trimmed SDDs are canonical for a given vtree [8].

References

1. Bacchus, F., Dalmao, S., Pitassi, T.: DPLL with Caching: A new algorithm for #SAT and Bayesian Inference. In: Electronic Colloquium on Computational Complexity (ECCC), vol. 10(003) (2003)
2. Beame, P., Li, J., Roy, S., Suciu, D.: Lower Bounds for Exact Model Counting and Applications in Probabilistic Databases. In: Proc. of UAI (2013)
3. Blum, M., Chandra, A.K., Wegman, M.N.: Equivalence of free boolean graphs can be decided probabilistically in polynomial time. Inf. Process. Lett. 10(2), 80–82 (1980)
4. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
5. Chavira, M., Darwiche, A.: On Probabilistic Inference by Weighted Model Counting. Artif. Intell. 172(6-7), 772–799 (2008)
6. Darwiche, A.: Decomposable Negation Normal Form. J. ACM 48(4), 608–647 (2001)
7. Darwiche, A.: New Advances in Compiling CNF into Decomposable Negation Normal Form. In: Proc. of ECAI, pp. 328–332 (2004)
8. Darwiche, A.: SDD: A New Canonical Representation of Propositional Knowledge Bases. In: Proc. of IJCAI, pp. 819–826 (2011)
9. Darwiche, A., Marquis, P.: A Knowledge Compilation Map. J. Artif. Intell. Res. (JAIR) 17, 229–264 (2002)
10. Davis, M., Logemann, G., Loveland, D.W.: A Machine Program for Theorem-Proving. Commun. ACM 5(7), 394–397 (1962)
11. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. J. ACM 7(3), 201–215 (1960)
12. Huang, J., Darwiche, A.: The Language of Search. J. Artif. Intell. Res. (JAIR) 29, 191–219 (2007)
13. Bayardo Jr., R.J., Pehoushek, J.D.: Counting Models Using Connected Components. In: AAAI/IAAI, pp. 157–162 (2000)
14. Majercik, S.M., Littman, M.L.: Using Caching to Solve Larger Probabilistic Planning Problems. In: AAAI/IAAI, pp. 954–959 (1998)
15. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: DSHARP: Fast d-DNNF Compilation with sharpSAT. In: Kosseim, L., Inkpen, D. (eds.) Canadian AI 2012. LNCS, vol. 7310, pp. 356–361. Springer, Heidelberg (2012)
16. Oztok, U., Darwiche, A.: CV-width: A New Complexity Parameter for CNFs. In: Proc. of ECAI (to appear, 2014)
17. Pipatsrisawat, K., Darwiche, A.: A Lower Bound on the Size of Decomposable Negation Normal Form. In: Proc. of AAAI (2010)
18. Pipatsrisawat, K., Darwiche, A.: Top-Down Algorithms for Constructing Structured DNNF: Theoretical and Practical Implications. In: Proc. of ECAI. pp. 3–8 (2010)
19. Robertson, N., Seymour, P.D.: Graph minors. III. Planar tree-width. J. Comb. Theory, Ser. B 36(1), 49–64 (1984)
20. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining Component Caching and Clause Learning for Effective Model Counting. In: Proc. of SAT (2004)
21. Valiant, L.G.: The complexity of computing the permanent. Theor. Comput. Sci. 8, 189–201 (1979)