ECAI 2010
H. Coelho et al. (Eds.)
IOS Press, 2010
© 2010 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-60750-606-5-3

Top-Down Algorithms for Constructing Structured DNNF: Theoretical and Practical Implications

Knot Pipatsrisawat and **Adnan Darwiche**[1]

Abstract. We introduce a top-down compilation algorithm for constructing structured DNNF for any Boolean function. With appropriate restrictions, the algorithm can produce various subsets of DNNF such as deterministic DNNF and OBDD. We derive a size upper bound for structured DNNF based on this algorithm and use the result to generalize similar upper bounds known for several Boolean functions in the case of OBDD. We then discuss two realizations of the algorithm that work on CNF formulas. We show that these algorithms have time and space complexities that are exponential in the treewidth and the dual treewidth of the input.

1 Introduction

Decomposability is a property that underlies many well-known compilation languages. Languages such as OBDD [1], AOMDD [7], and deterministic DNNF [4] can be obtained by imposing additional properties on top of decomposability. Recently, it was shown that a refinement of this property known as *structured decomposability* allows for an efficient conjoin operation between formulas satisfying such a property [8].

A bottom-up algorithm for constructing structured DNNFs was conceived based on this conjoin operation [8]. Such an algorithm works by first converting each clause of the input (usually given in CNF) into a structured DNNF, then repeatedly conjoin them to form a whole formula. This bottom-up algorithm may yield some very large intermediate formulas, thus preventing it from compiling certain formulas with compact structured DNNF representations (see [6] for a study on the relative merits of top-down versus bottom-up compilation in the case of OBDD).

In this paper, we present a top-down algorithm that constructs structured DNNF form for any Boolean function. This is a general algorithm that can be used to produce DNNFs, deterministic DNNFs, AOMDDs (Boolean domains), and even OBDDs. The proposed algorithm provides a unified framework for establishing upper bounds for various compilation languages and forms a basis for practical knowledge compilation algorithms. We demonstrate the value of this algorithm in the following ways. On the theoretical side, we derive a general upper bound for structured DNNF based on the algorithm. We show that our result, when interpreted in a certain context, subsumes the influential upper bound given by Sieling and Wegener for OBDD [12]. Moreover, we apply our result to prove upper bounds for various Boolean functions, generalizing some known upper bounds for OBDD. On the practical side, we present two concrete top-down algorithms for compiling formulas in conjunctive normal form (CNF) into structured DNNFs. We analyze their time and space complexities and show that, in one case, the algorithm is exponential in the treewidth of the CNF, while, on the other case, it is exponential in the dual treewidth. These are the first bounds of this type on the time and space complexity of constructing structured DNNFs. We provide the proofs of some of our results in this paper, leaving others to the full version for space limitations.

[1] Computer Science Department, University of California, Los Angeles, email: {thammakn,darwiche}@cs.ucla.edu

2 Basic Definitions

In this section, we present basic notations and definitions that will be used throughout the paper. We use an upper case letter to denote a variable (e.g., X) and a lower case letter to denote its instantiation (e.g., x). Moreover, we use a bold upper case letter to denote a set of variables (e.g., $\mathbf{X}$) and a bold lower case letter to denote their instantiations (e.g., $\mathbf{x}$).

A *Boolean function* (or simply function) over a set of variables $\mathbf{Z}$ is a function that maps each complete assignment of variables $\mathbf{Z}$ to either *true* or *false*. The *conditioning* of function f on variable assignment $\mathbf{x}$ (of variables $\mathbf{X}$) is defined as $f|\mathbf{x} = \exists\mathbf{X}(f \wedge \mathbf{x})$. If f is represented by a propositional formula, we can obtain $f|\mathbf{x}$ by replacing each occurrence of variable $X \in \mathbf{X}$ by its value in $\mathbf{x}$. We also refer to $\mathbf{x}$ as an *instantiation* of variables $\mathbf{X}$. A function f *depends only* on variables $\mathbf{Z}$ iff for any variable $X \notin \mathbf{Z}$, we have $f|X = f|\neg X$. We will write $f(\mathbf{Z})$ to mean that f is a function that depends only on variables $\mathbf{Z}$. Note that $f(\mathbf{Z})$ may not necessary depend on every variable in $\mathbf{Z}$.

A conjunction is *decomposable* if each pair of its conjuncts share no variables. A disjunction is *deterministic* if any two of its disjuncts are inconsistent with each other. A *negation normal form* (NNF) is a DAG whose internal nodes are labelled with disjunctions and conjunctions and whose leaf nodes are labeled with literals or the constants *true* and *false*; see Figure 2. An NNF is decomposable (called a DNNF) iff each of its conjunctions is decomposable; see Figure 2(a). A DNNF is deterministic (called a d-DNNF) iff each of its disjunctions is deterministic; see Figure 2(b). We use $vars(N)$ to denote the set of variables mentioned by an NNF node N.

3 OBDDs and the Sieling and Wegener Bound

An OBDD is a DAG whose non-leaf nodes are labeled with variables, and whose leaf nodes are labeled with Boolean constants; see Figure 1(a). Every non-leaf node in an OBDD has exactly one high child (pointed to by a solid edge) and one low child (pointed to by a dotted edge). An OBDD respects a variable ordering if the order of variables on any path from the root to a leaf is consistent with the given order.

One key concept that underlies OBDD is the *Shannon decomposition*, which states that every function f can always be written as

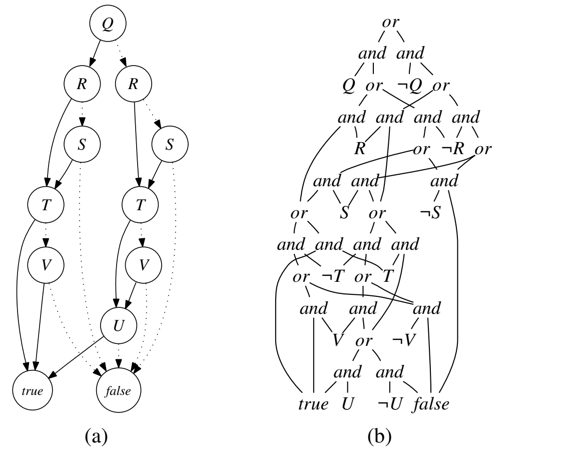

Figure 1. (a) an OBDD in the conventional representation (b) the same OBDD in the NNF representation.

$f = (X \wedge f|X) \vee (\neg X \wedge f|\neg X)$, where X is any variable. Since $f|X$ and $f|\neg X$ no longer depend on X, the conjunctions of the Shannon decomposition must be decomposable. Any OBDD can be viewed as the result of applying this decomposition recursively on some Boolean function according to a given variable ordering.

An OBDD can also be interpreted as an NNF as shown in Figure 1(b). Every OBDD node labeled with X is expanded into the form $(X \wedge h) \vee (\neg X \wedge l)$, where h and l are the results of expanding the high and low children of the node. The resulting NNF are decomposable and deterministic (see [4] for more details).

An OBDD is an attractive representation of Boolean functions because of its ability to represent some functions compactly and its polytime support for many logical operations (see [1]). Since most applications of OBDDs rely on the compactness of the representation, much work exists on bounding the size of OBDDs for various Boolean functions. The following (paraphrased) result by Sieling and Wegener provides a basis for numerous work in this direction. Central to this result is the notion of a *sub-function* $f|\mathbf{x}$, which is a function that is obtained by conditioning the given function f on a variable instantiation $\mathbf{x}$.

Theorem 1 ([12]) *Let f be a function over $X_1, \ldots, X_n$ and m be the number of distinct sub-functions of f obtained by conditioning on $X_1, \ldots, X_{i-1}$ that depend on X_i. A reduced*[2] *OBDD for f using variable ordering $X_1, \ldots, X_n$ contains exactly m nodes labeled with X_i.*

The main use of the Sieling and Wegener's result is in showing which variable orderings lead to efficient OBDD representations (upper bound), and which ones lead to exponential OBDDs (lower bound). Consider the Boolean function $f = (X_1 \wedge Y_1) \vee \ldots \vee (X_n \wedge Y_n)$, for example. If we use variable ordering $X_1, Y_1, \ldots, X_n, Y_n$, we find that the number of distinct sub-functions that depend on X_i or Y_i (by conditioning f on all preceding variables) is no more than 2 for any i. Hence, this variable order will lead to an efficient OBDD representation. However, if we use the variable ordering $X_1, \ldots, X_n, Y_1, \ldots, Y_n$, we find that we have $\Omega(2^n)$ distinct sub-functions that depend on variable Y_1, when we condition

[2] An OBDD is reduced iff no two distinct nodes in the OBDD represent the same Boolean function.

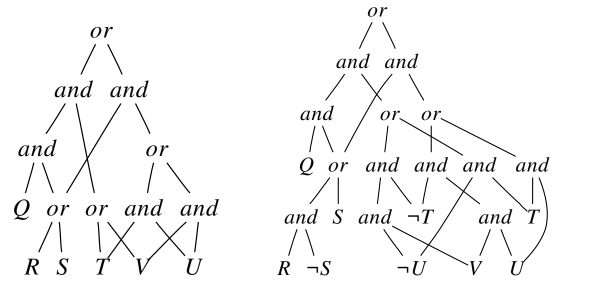

Figure 2. (a) A DNNF (b) a d-DNNF. These DNNFs respect the vtree in Figure 3(a)

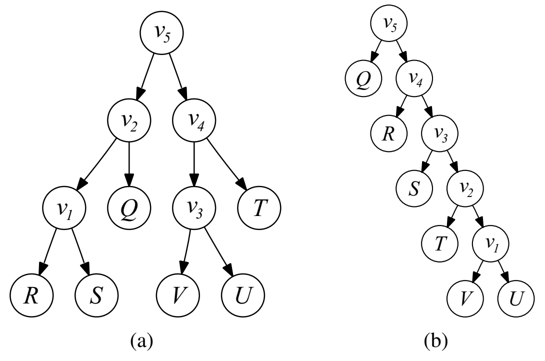

Figure 3. A vtree in (a) and a linear vtree in (b).

f on $X_1, \ldots, X_n$. Hence, this variable ordering will lead to an exponentially-sized OBDD.

4 Structured DNNFs

Our goal is to derive an analogue of the upper bound in the Sieling and Wegener result, but for a more general class of representations that include OBDDs. In particular, our focus is on *structured DNNFs.* A structured DNNF is a DNNF that respects a *vtree* [8].

Definition 1 (Vtree) *A vtree for a set of variables $\mathbf{Z}$ is a full, rooted binary tree whose leaves are in one-to-one correspondence with the variables in $\mathbf{Z}$.*

Figure 3 depicts two example vtrees for the same set of variables. Given an internal node v in a vtree for variables $\mathbf{Z}$, we use v^l and v^r to refer to its left and right children, use $vars(v)$ to denote the set of variables at or below v in the tree. We can now define what it means for a DNNF to respect a vtree.

Definition 2 *A DNNF respects a vtree iff every and-node has exactly two children N^l and N^r, and we have $vars(N^l) \subseteq vars(v^l)$ and $vars(N^r) \subseteq vars(v^r)$ for some vtree node v.*

The DNNF in Figure 2(a) and the d-DNNF in Figure 2(b) respect the vtree in Figure 3(a). The language of structured DNNF simply contains all DNNFs that respects some vtree.[3]

[3] Some DNNFs, such as $(((a \wedge b) \wedge (\neg c \wedge d)) \vee ((\neg a \wedge c) \wedge (b \wedge \neg d)))$, do not respect any vtree.

Note that a variable ordering corresponds to a *linear vtree* as shown in Figure 3(b).Moreover, every OBDD is a DNNF that respects the corresponding linear vtree. We later present an algorithm for constructing DNNFs that respects the given vtrees and derive an upper bound on the time and space complexity of the algorithm. When applied to an OBDD, our bound can be shown to subsume the Sieling and Wegener bound.

5 Decompositions of Boolean Functions

We now review a key notion, called a decomposition, which can be used to characterize various subsets of DNNF. This notion was introduced previously in [9] for the purpose of establishing lower bounds on DNNFs and we shall use it in this paper for establishing upper bounds. In the rest of the paper, we will assume that variables $\mathbf{X}$ and $\mathbf{Y}$ form a partition of variables $\mathbf{Z}$.

Definition 3 *An $\underline{\mathbf{X}}$-decomposition of function $f(\mathbf{Z})$ is a collection of functions (a.k.a. elements) $f^1(\mathbf{Z}), \ldots, f^m(\mathbf{Z})$ such that (i) $f = f^1 \vee \ldots \vee f^m$ and (ii) each f^i can be expressed as follows:*

$$f^i(\mathbf{Z}) = g^i(\mathbf{X}) \wedge h^i(\mathbf{Y}).$$

The number m is called the size of the decomposition in this case. A decomposition is minimal if no other decomposition has a smaller size. A decomposition is deterministic if $f^i \wedge f^j$ is inconsistent for all $i \neq j$.

Note that an $\mathbf{X}$-decomposition for $f(\mathbf{Z})$ is also a $\mathbf{Y}$-decomposition for $f(\mathbf{Z})$. We will typically just say "decomposition" when variables $\mathbf{X}$ and $\mathbf{Y}$ are clear from the context.

Consider the Boolean function $f = (X_1 \wedge Y_1) \vee (X_2 \wedge Y_2) \vee (X_2 \wedge Y_3)$ and the partition $\mathbf{X} = \{X_1, X_2\}, \mathbf{Y} = \{Y_1, Y_2, Y_3\}$. The following are two decompositions of this function:

$g(\mathbf{X})$	$h(\mathbf{Y})$
X_1	Y_1
X_2	$Y_2 \vee Y_3$

$g(\mathbf{X})$	$h(\mathbf{Y})$
X_1	Y_1
$\neg X_1 \wedge X_2$	$Y_2 \vee Y_3$
$X_1 \wedge X_2$	$\neg Y_1 \wedge (Y_2 \vee Y_3)$

Each row corresponds to an element of the decomposition. Moreover, we present each element in terms of its $\mathbf{X}$ and $\mathbf{Y}$ components; the element can be recovered by simply conjoining these components together. Note that the left decomposition is non-deterministic, while the right decomposition is deterministic. One can always find an $\mathbf{X}$-decomposition for any function $f(\mathbf{Z})$ if one is not concerned about the size of the decomposition. In particular, the models of $f(\mathbf{Z})$ can be the basis for a rather trivial $\mathbf{X}$-decomposition (for any partition $\mathbf{X}, \mathbf{Y}$ of $\mathbf{Z}$).

The notion of a decomposition generalizes the Shannon decomposition as used in the OBDD literature. According to the Shannon decomposition, every function $f(\mathbf{Z})$ can be expressed as $f = (X \wedge f|X) \vee (\neg X \wedge f|\neg X)$. If we let $\mathbf{X} = \{X\}, \mathbf{Y} = \mathbf{Z} \setminus \mathbf{X}$, then this can be thought of as the following decomposition:

$g(\mathbf{X})$	$h(\mathbf{Y})$
X	$f\|X$
$\neg X$	$f\|\neg X$

The Shannon decomposition is always of size two, deterministic, and is completely determined by the choice of variable X.

Algorithm 1: $DNNF(v, f)$: keeps a cache $cache(.,.)$ where the first argument is a vtree node and the second argument is a function. The cache is initialized to nil.

```
input:
  $v$:  vtree node
  $f$:  function that depends only on $vars(v)$
output: DNNF for function $f$ respecting vtree rooted at node $v$
main:
1:  If $f = true$ or $f = false$ or $v$ is a leaf node, return $f$
2:  If $cache(v, f) \neq nil$, return $cache(v, f)$
3:  $\mathbf{X} \leftarrow$ variables in the vtree rooted at $v^l$
4:  $\mathbf{Y} \leftarrow$ variables in the vtree rooted at $v^r$
5:  $g^1(\mathbf{X}) \wedge h^1(\mathbf{Y}), \ldots, g^m(\mathbf{X}) \wedge h^m(\mathbf{Y}) \leftarrow$ a decomposition of $f$
6:  $\alpha \leftarrow false$
7:  for $i = 1$ to $m$ do
8:      $\alpha \leftarrow \alpha \vee (DNNF(v^l, g^i(\mathbf{X})) \wedge DNNF(v^r, h^i(\mathbf{Y})))$
9:  end for
10: $cache(v, f) \leftarrow \alpha$
11: return $\alpha$
```

6 An Algorithm for Constructing DNNF

In this section, we present a top-down algorithm for constructing DNNFs that respect a given vtree (Algorithm 1). This algorithm works by applying a decomposition to the input Boolean function (Lines 3-5). Then, it invokes itself recursively to decompose each function used in the decomposition (Line 8). The disjunction of all elements of the decomposition (α) is then returned. In the base cases (Line 1), the input function must be equivalent to either a truth constant or a literal. We return such a value. Note that caching is used in this algorithm to avoid decomposing the same function more than once at a given vtree node (Lines 2, 10). In each function call, if there is no cache hit, a total of m and-nodes and one or-node will be constructed by the algorithm (m is the size of the decomposition used).[4] Note that this version of the algorithm uses the input Boolean function as a cache key. This caching scheme is not practical, but it suffices for our discussion in the next two sections. We will present more practical caching schemes when we discuss more concrete versions of the algorithm in Section 9.

The value of this algorithm is two-fold. First, it provides, a top-down algorithm for constructing structured DNNF that does not commit to any particular vtree or any particular type of decomposition. Thus, by varying the type of vtree and the type of decompositions used on Line 5, formulas from various languages including DNNF, deterministic DNNF and OBDD can be constructed. Secondly, it can be used as a tool for deriving upper bounds on the size of structured DNNFs. We will demonstrate this usage later. We now state the correctness of the algorithm.

Proposition 1 *Algorithm 1 returns a DNNF for function f that respects the vtree rooted at node v. If the decompositions computed on Line 5 are deterministic, the returned DNNF will be deterministic.*

In the rest of the paper, we implicitly assume that $DNNF(v, \mathcal{F})$ is the first call to Algorithm 1 and use $DNNF(v, .)$ to refer to a (recursive) call to Algorithm 1 made with vtree node v as the first argument. We say that the and-nodes and or-nodes constructed during a call $DNNF(v, .)$ are *computed at node v*. Lastly, we use the term

[4] The number of edges constructed in each function call is bounded by $3m$.

number of distinct calls at node v to denote the number of recursive calls $DNNF(v, f)$ made with distinct functions as the second argument. Once a call $DNNF(v, f)$ has been made, all subsequent calls with the same vtree node and function yield no work because of the cache. Therefore, the number of distinct calls is an indicator of how much time (and space) is used by the algorithm.

7 An Upper Bound

In this section, we present our main result, which is an upper bound on the size of structured DNNF based on Algorithm 1.

Theorem 2 *Consider calling Algorithm 1 on a function with n variables. Let v be a node in the vtree used, k_v be the number of distinct calls $DNNF(v, f)$ and let m_v be the size of largest decomposition computed at node v. The number of nodes computed at node v is $O(k_v m_v)$. Moreover, the size of the DNNF returned by the algorithm is in $O(Kn)$, where $K = \max_v k_v, m_v$.*

This result shows that the amount of work done (and the size of the resulting DNNF) with respect to a particular vtree node can be upper bounded by the product of (i) how many times recursive calls (on distinct functions) are made on a particular vtree node and (ii) how large the decompositions computed at node v are. Moreover, we can use this result to bound the size of the whole DNNF returned by the algorithm. We point out that this version of the upper bound is not as specific as the upper bound given in Theorem 1, because this general result does not make any assumption on the type of vtree or the type of decompositions used in the algorithm. The upper bound of Theorem 1 is obtained when one uses Shannon decompositions and linear vtrees.[5]

Consider Algorithm 1 again. After all the recursive calls are made on Line 8, α (which is an or-node) may contain disjuncts of the following form: $(g^i \wedge h^i) \vee (g^j \wedge h^j)$ where $g^i = g^j$. One can generate smaller DNNFs by factoring this expression to yield the more compact expression $g^i \wedge (h^i \vee h^j)$. The following two propositions assume this type of factorization. The cost of applying this technique is irrelevant to these propositions as they are only concerned with bounding the size of resulting DNNFs. Yet, this factorization allows us to bound the size of the DNNF fragment constructed at vtree node v in terms of the number of distinct calls made to the children of node v, therefore, removing the need to reference the size of decompositions computed at vtree node v. The next result utilizes this fact to provide a special upper bound for functions with certain properties.

Proposition 2 *Consider calling Algorithm 1 on a function over n variables. If, for every internal vtree node v, the number of distinct calls at node v is bounded by K, then the size of the output DNNF is in $O(K^2 n)$.*

Another interesting scenario is when the number of distinct calls at each vtree node v is linear in the number of variables in the sub-vtree rooted at v. Even though the number of distinct calls is not bounded by a constant in this case, we can still obtain an interesting bound on the overall DNNF size.

Proposition 3 *Consider calling Algorithm 1 on a function over n variables. If, for every internal vtree node v, the number of distinct calls at node v is in $O(|vars(v)|)$, then the size of the output DNNF is in $O(n^2)$.*

Note that the claim made by this proposition is stronger than the $O(n^3)$ bound that is immediate from Proposition 2. We will utilize these results to bound the size of structured DNNFs for certain Boolean functions in the next section.

8 Example Applications of the Upper Bound

In this section, we demonstrate several example applications of our upper bound results presented in the previous section. The bounds presented here do not require any assumption on the format of the input functions. We take advantage of the knowledge about the Boolean functions considered to tailor appropriate decompositions for deriving the bounds. In each example, we present the decompositions to be used (by Algorithm 1). Then, we derive a bound by reasoning about the sizes of these decompositions and the number of distinct calls at each vtree node. All of the results presented in this section generalize well-known OBDD upper bounds for the corresponding Boolean functions.

8.1 Total Symmetric Boolean Functions

We start by deriving an upper bound on the DNNF size of a well-known class of Boolean functions. A Boolean function $\mathcal{F}$ is said to be *total symmetric* if exchanging the values of any variables does not affect the value of $\mathcal{F}$. It is well-known that the size of any OBDD (any ordering) representing a total symmetric Boolean function is upper bounded by $O(n^2)$, where n is the number of variables [14]. We will generalize this result for structured DNNF respecting *any* vtree (i.e., any tree structure, any placement of variables). We first state this result.

Proposition 4 *Let $\mathcal{F}$ be any total symmetric function over n variables. For any given vtree, there exists a deterministic DNNF for $\mathcal{F}$ that respects the vtree with size in $O(n^2)$.*

Proof. Let a vtree be given. Because $\mathcal{F}$ is total symmetric, for any variable partition $\mathbf{X}, \mathbf{Y}$, we can always apply the following decomposition: $f^{=0}(\mathbf{X}) \wedge h^0(\mathbf{Y}), \ldots, f^{=|\mathbf{X}|}(\mathbf{X}) \wedge h^{|\mathbf{X}|}(\mathbf{Y})$, where $f^{=i}$ is the *exactly i function* ($f^{=i}(\mathbf{z}) = true$ iff $\mathbf{z}$ sets exactly i variables to *true*) and h^i is such that $\mathbf{y} \models h^i$ iff $\mathbf{y} \wedge f^{=i}(\mathbf{X}) \models \mathcal{F}$, where $\mathbf{y}$ is an instantiation of $\mathbf{Y}$. [6] Note that each $f^{=i}$ is also total symmetric. Moreover, because f is total symmetric, each h^i must be total symmetric as well. This implies that this decomposition can be recursively applied to each $f^{=i}$ and each h^i. It is not hard to see that, at each vtree node v, we need at most $|vars(v)| + 1$ distinct function calls (either $f^{=0}, \ldots, f^{=|vars(v)|}$ or $h^0, \ldots, h^{|vars(v)|}$). Clearly, the number of distinct function calls at each vtree node is linear in the number of variables in that sub-vtree. Therefore, by Proposition 3, the size of structured DNNF for this function can be upper bounded by $O(n^2)$. Notice that, since the decompositions used in the above proof are deterministic, the resulting DNNF is also deterministic. □

[5] We provide a proof sketch here and leave the details for the full version of the paper, because of space limitations. When Shannon decompositions and linear vtrees are used, we can show that the distinct calls made at node v induce an $\mathbf{X}$-decomposition ($\mathbf{X} = vars(v)$) whose size can be bounded by the number of sub-functions $\mathcal{F}|\mathbf{y}$, where $\mathbf{Y}$ is the set of variables outside of $vars(v)$. Here, $\mathbf{Y}$ corresponds to the variables $X_1, \ldots, X_{i-1}$ in Theorem 1. Each distinct call made at node v returns an or-node that corresponds to an OBDD node labeled with the first variable in v (nearest to the root).This variable corresponds to X_i in Theorem 1. As a result, we obtain the upper bound result of Theorem 1.

[6] This decomposition is valid only because f is total symmetric. It is possible for some h^i to be *false*.

This result has an important implication. It shows that we can always conjoin a structured DNNF representation of *any* function with that of a total symmetric function in polytime. Given a structured DNNF representation of a function using any vtree, we can use the above construction to create a structured DNNF representation of the considered total symmetric function and conjoin it with the function (conjoin is a quadratic time operation for structured DNNFs that respect the same vtree [8]).

In the next two examples, we consider specific functions in this class and show that even tighter upper bounds can be derived for structured DNNF.

8.2 Odd/Even Parity Functions

In this example, we will prove a result that generalizes the linear OBDD upper bound for the parity functions [13].

Proposition 5 *Given any vtree, there exist DNNFs for the odd and even parity functions over n variables that respect the vtree with size in $O(n)$.*

Proof. Consider the odd parity (f^o) and the even parity (f^e) functions. Let $\mathbf{X}$ and $\mathbf{Y}$ be the variable partition at the vtree node considered. We adopt the following decompositions:

$$\text{For } f^o(\mathbf{Z}) : f^o(\mathbf{X}) \wedge f^e(\mathbf{Y}), f^e(\mathbf{X}) \wedge f^o(\mathbf{Y})$$
$$\text{For } f^e(\mathbf{Z}) : f^e(\mathbf{X}) \wedge f^e(\mathbf{Y}), f^o(\mathbf{X}) \wedge f^o(\mathbf{Y})$$

Since these decompositions utilize only the odd and even parity functions, they can be applied recursively. As a result, we have that, at each vtree node v, (i) only two distinct functions (f^o, f^e over $vars(v)$) are needed and (ii) the size of any decomposition is exactly two. Therefore, by Theorem 2, for every vtree node v, Algorithm 1 only constructs a constant number of DNNF nodes. Moreover, by Corollary 1 the total size of the DNNF is in $O(n)$. □

Again, this bound applies to *any* vtree. Notice also that, since the decompositions used are deterministic, the constructed DNNF is also deterministic. This result generalizes the well-known OBDD upper bound for parity functions in [13].

8.3 Threshold Functions

In this last example, we present an upper bound for threshold functions. A threshold function $f^{\geq k}$ evaluates to *true* iff at least k of its inputs are *true*.

Proposition 6 *Let $f^{\geq k}$ be a threshold function over n variables. For any vtree, there exists a DNNF for $f^{\geq k}$ that respects the vtree with size in $O(k^2 n)$.*

Proof. Let $f^{=k}$ be the *exactly* k function. Now, consider any vtree. We adopt the following decomposition on Line 5 of Algorithm 1:

- If $f = f^{\geq k}(\mathbf{Z})$ and assuming $|\mathbf{X}| \leq |\mathbf{Y}|$, choose the decomposition $f^{=0}(\mathbf{X}) \wedge f^{\geq k}(\mathbf{Y})$, $f^{=1}(\mathbf{X}) \wedge f^{\geq k-1}(\mathbf{Y})$, $\ldots$, $f^{=k-1}(\mathbf{X}) \wedge f^{\geq 1}(\mathbf{Y})$, $f^{\geq k}(\mathbf{X})$.
- If $f = f^{=k}(\mathbf{Z})$, choose the decomposition $f^{=0}(\mathbf{X}) \wedge f^{=k}(\mathbf{Y})$, $f^{=1}(\mathbf{X}) \wedge f^{=k-1}(\mathbf{Y}), \ldots, f^{=k}(\mathbf{X}) \wedge f^{=0}(\mathbf{Y})$.

Note that $f^{=k}(\mathbf{Z}) = \mathit{false}$ and $f^{\geq k}(\mathbf{Z}) = \mathit{false}$ when $|\mathbf{Z}| < k$.

It is not hard to see that the number of distinct function calls at each vtree node is $\leq k + 1$. By invoking Proposition 2, we obtain that the size of the resulting DNNF must be in $O(k^2 n)$, where n is the total number of variables. □

Again, the resulting DNNF is deterministic. The bound presented above is applicable to any vtree. Yet, the bound can be tightened even further for certain types of vtrees. For example, if the vtree is linear, the proposed decompositions will reduce to the following:

- If $f = f^{\geq k}(\mathbf{Z})$, choose the decomposition $f^{=0}(X) \wedge f^{\geq k}(\mathbf{Y})$, $f^{=1}(X) \wedge f^{\geq k-1}(\mathbf{Y})$.
- If $f = f^{=k}(\mathbf{Z})$, choose the decomposition $f^{=0}(X) \wedge f^{=k}(\mathbf{Y})$, $f^{=1}(X) \wedge f^{=k-1}(\mathbf{Y})$.

Notice that $f^{=0}(X) = \neg X$ and $f^{=1}(X) = X$. Hence, these decompositions are in fact Shannon decompositions. Since the size of each decomposition is two and the number of distinct calls at each vtree node is $\leq k + 1$, we get a bound of $O(kn)$ for the total size of the DNNF, which is also in the same order as the bound known for OBDD [11].[7]

9 Practical Compilation Algorithms and Their Complexities

We now present two practical versions of Algorithm 1 and address their time and space complexities. In particular, we consider the cases when the inputs are expressed in conjunctive normal form (CNF). We present two types of decompositions for CNF inputs along with concrete caching schemes. Base on these choices, we present results on the time and space complexities of the resulting algorithms using *treewidth* [10]. Due to space constraints, we leave the proofs of the claims made here to the full paper.

In the first case, we show that the time and space complexities of the algorithm is exponential in the treewidth of the constraint graph of the given CNF. A related result was stated in [8], but concerned only the size of a structured DNNF. Here, we provide an algorithm with this time and space guarantee. In the second case, we present a compilation algorithm that establishes (structured) decomposability without imposing determinism. We will show that the presented algorithm has time and space complexity that is exponential only in the treewidth of the dual constraint graph of the CNF (a.k.a. the *dual treewidth*). The dual constraint graph is a graph in which each clause of the CNF corresponds to a distinct vertex and an edge exists between two vertices iff their clauses share a variable [5]. In general, neither the treewidth of the constraint graph nor the treewidth of the dual constraint graph dominates each other (i.e., there exists a CNF whose treewidth is smaller than the dual treewidth and vice versa). In what follows, given a partition of variables $\mathbf{X}, \mathbf{Y}$, we write each CNF Δ as a conjunction of three components: (i) $\Delta(\mathbf{X})$, the clauses over $\mathbf{X}$, (ii) $\Delta(\mathbf{Y})$, the clauses over $\mathbf{Y}$, and (iii) $\Delta(\mathbf{X}, \mathbf{Y})$, the clauses that mention variables in $\mathbf{X}$ and $\mathbf{Y}$. We refer to $\Delta(\mathbf{X}, \mathbf{Y})$ as the *cutset clauses* of the CNF (with respect to the partition). Moreover, we use $|\Delta|$ to refer to the number of clauses in the CNF Δ.

9.1 An Algorithm with a Treewidth Bound

We now describe a variation on Algorithm 1 with a treewidth guarantee. This variation is defined by a specific class of decompositions and a specific caching scheme that we shall describe next. The resulting algorithm, which is based on the ideas underlying [2], will be referred to as Algorithm 1(TW).

[7] A more refined bound of $kn - k^2 + k$ can be obtained with a more detailed analysis.

Definition 4 *Consider a CNF $\Delta(\mathbf{Z}) = \Delta(\mathbf{X}) \wedge \Delta(\mathbf{Y}) \wedge \Delta(\mathbf{X}, \mathbf{Y})$. The set of cutset-variables is the subset of variables $\mathbf{X}$ mentioned by $\Delta(\mathbf{X}, \mathbf{Y})$. We define the cutset-variable decomposition to be $g^1(\mathbf{X}) \wedge h^1(\mathbf{Y}), \ldots, g^m(\mathbf{X}) \wedge h^m(\mathbf{Y})$, where, for each instantiation $\mathbf{v}^i$ of the cutset-variables, $g^i = (\mathbf{v}^i \wedge \Delta(\mathbf{X})|\mathbf{v}^i)$ and $h^i = (\Delta(\mathbf{Y}) \wedge \Delta(\mathbf{X}, \mathbf{Y})|\mathbf{v}^i)$.*[8]

Since we consider only CNF inputs, the functions g^i and h^i in this decomposition must be given in CNF as well. We can obtain CNF representations of $\Delta(\mathbf{X})|\mathbf{v}^i$ and $\Delta(\mathbf{X}, \mathbf{Y})|\mathbf{v}^i$ in time that is linear in the sizes of these formulas. Therefore, the time complexity for producing a cutset-variable decomposition is proportional to the decomposition size and the size of CNF Δ.

To fully specify our new algorithm, we need to specify the cache key. For this purpose, we assume that a third argument, which is an instantiation, is passed to each function call (in addition to vtree node v and the CNF f). Considering Definition 4, the third argument for $DNNF(v^l, g^i, .)$ is defined to be the third argument of the current function call, while the third argument for $DNNF(v^r, h^i, .)$ is defined to be the third argument of the current function call conjoined with the instantiation $\mathbf{v}^i$. For each vtree node v, we can pre-compute the set of *context variables*, which are variables outside of $vars(v)$ that appear in the same clause (of $\mathcal{F}$) as some variable in $vars(v)$. Then, during actual call $DNNF(v, f, .)$, the cache key is then simply the values of the context variables of v according to the third argument. The following proposition provides time and space guarantee for this algorithm.

Proposition 7 *Given a CNF Δ over n variables and an elimination order of Δ with width w, we can construct a vtree such that a call to Algorithm 1(TW) on Δ using the vtree can have a time and space complexity in $O(n|\Delta|2^w)$.*

Hence, given an appropriate elimination order of the input CNF, Algorithm 1(TW) produces a DNNF in time and space that are only exponential in the treewidth of the CNF (see [3] for a review of elimination orders and treewidth).

9.2 An Algorithm with a Dual Treewidth Bound

We now describe another variation on Algorithm 1 with a dual treewidth guarantee. This variation is defined again by a specific class of decompositions and a specific caching scheme that we shall describe next. The resulting algorithm will be referred to as Algorithm 1(DTW).

Definition 5 *Consider a CNF $\Delta(\mathbf{Z}) = \Delta(\mathbf{X}) \wedge \Delta(\mathbf{Y}) \wedge (\bigwedge_{i=1}^{k}(\alpha_i(\mathbf{X}) \vee \beta_i(\mathbf{Y})))$, where α_i, β_i are clauses over $\mathbf{X}$ and $\mathbf{Y}$. The cutset-clause decomposition of Δ is defined as*

$$\left\{ \left(\Delta(\mathbf{X}) \wedge (\bigwedge_{i \in S} \alpha_i) \right) \wedge \left(\Delta(\mathbf{Y}) \wedge (\bigwedge_{j \notin S} \beta_j) \right) \middle| S \subseteq [k] \right\},$$

where $[k]$ is defined to be $\{1, 2, \ldots, k\}$.

In this definition, the cutset clauses are viewed as disjunctions of sub-clauses over $\mathbf{X}$ and over $\mathbf{Y}$. Each element of the decomposition is a CNF consisting of $\Delta(\mathbf{X}), \Delta(\mathbf{Y})$, and an element of the cross-product of the cutset clauses. The size of this decomposition is exponential in the number of cutset clauses. The time needed to compute each clause-cutset decomposition is proportional to the size of the decomposition and $|\Delta|$. It is important to note that the decomposition described above may not be deterministic in general. This variation on Algorithm 1 uses the input CNF as a cache key.

Proposition 8 *Given a CNF Δ over n variables and an elimination order of its dual constraint graph with width w, we can construct a vtree such that a call to Algorithm 1(DTW) on Δ using the vtree can have a time and space complexity in $O(n|\Delta|3^w)$.*

With the right elimination order, Algorithm 1(DTW) produces a DNNF in time and space exponential in the dual treewidth of the CNF.

10 Conclusion

We presented a top-down algorithm for constructing structured DNNF. We then derived a general upper bound for structured DNNF based on this algorithm. We showed that this result, when considered in the right context, subsumed the Siegling and Wegener OBDD upper bound. Then, we demonstrated how our result could be used to upper bound the size of structured DNNF for various Boolean functions, generalizing the known OBDD upper bounds for these functions. We then presented practical variations of the proposed algorithm. We showed that, in one variation, we could obtain a treewidth guarantee on both time and space, and, in the other variation, the time and space complexities were exponential in the treewidth of the dual constraint graph.

REFERENCES

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

[2] Adnan Darwiche, 'New advances in compiling CNF to decomposable negational normal form', in *Proceedings of European Conference on Artificial Intelligence, Valencia, Spain*, pp. 328–332, (2004).

[3] Adnan Darwiche, *Modeling and Reasoning with Bayesian Networks*, Cambridge University Press, 2009.

[4] Adnan Darwiche and Pierre Marquis, 'A knowledge compilation map', *JAIR*, **17**, 229–264, (2002).

[5] Rina Dechter, *Constraint Processing*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[6] Jinbo Huang and Adnan Darwiche, 'Using DPLL for efficient OBDD construction', in *SAT 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pp. 157–172, (2005).

[7] Robert Mateescu and Rina Dechter, 'Compiling constraint networks into and/or multi-valued decision diagrams (AOMDDs)', in *Proc. of CP-06*, pp. 329–343, (2006).

[8] Knot Pipatsrisawat and Adnan Darwiche, 'New compilation languages based on structured decomposability', in *Proc. of AAAI-08*, pp. 517–522, (2008).

[9] Knot Pipatsrisawat and Adnan Darwiche, 'A lower bound on the size of decomposable negation normal form', in *Proceedings of AAAI-10, to appear*, (July 2010).

[10] Neil Robertson and P. D. Seymour, 'Graph minors. ii. algorithmic aspects of tree-width', *Journal of Algorithms*, **7**(3), 309 – 322, (1986).

[11] Don E. Ross, Kenneth M. Bulter, and M. Ray Mercer, 'Exact ordered binary decision diagram size when representing classes of symmetric functions', *J. Electron. Test.*, **2**(3), 243–259, (1991).

[12] Detlef Sieling and Ingo Wegener, 'Nc-algorithms for operations on binary decision diagrams', *Parallel Processing Letters*, **3**, 3–12, (1993).

[13] Ingo Wegener, *The complexity of Boolean functions*, John Wiley & Sons, Inc., New York, NY, USA, 1987.

[14] Ingo Wegener, *Branching programs and binary decision diagrams: theory and applications*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[8] When $\Delta(\mathbf{X}, \mathbf{Y}) = true$, we simply adopt the decomposition $\Delta(\mathbf{X}) \wedge \Delta(\mathbf{Y})$. There are other definitions of cutset-variables which could lead to smaller decompositions. We chose this one for simplicity.