On Compiling DNNFs without Determinism

Umut Oztok and Adnan Darwiche

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

Abstract. State-of-the-art knowledge compilers generate *deterministic* subsets of DNNF, which have been recently shown to be exponentially less succinct than DNNF. In this paper, we propose a new method to compile DNNFs without enforcing determinism necessarily. Our approach is based on compiling deterministic DNNFs with the addition of auxiliary variables to the input formula. These variables are then existentially quantified from the deterministic structure in linear time, which would lead to a DNNF that is equivalent to the input formula and not necessarily deterministic. On the theoretical side, we show that the new method could generate exponentially smaller DNNFs than deterministic ones, even by adding a single auxiliary variable. Further, we show that various existing techniques that introduce auxiliary variables to the input formulas can be employed in our framework. On the practical side, we empirically demonstrate that our new method can significantly advance DNNF compilation on certain benchmarks.

1 Introduction

Decomposability and determinism are two fundamental properties that underlie many tractable representations in propositional logic. Decomposability is the characteristic property of decomposable negation normal form (DNNF) [9], and adding determinism to DNNF leads to deterministic DNNF (d-DNNF) [10], which includes many other representations, such as sentential decision diagrams (SDDs) [14] and ordered binary decision diagrams (OBDDs) [5].

The key property of deterministic subsets of DNNF is their ability to render the query of model counting tractable, which is key to probabilistic reasoning (see, e.g., [29,11,6]). On the other hand, decomposability without determinism is also sufficient to ensure the tractability of many interesting queries, such as clausal entailment and cardinality minimization. Indeed, these queries are enough for various applications, which do not require efficient computation of model counting. For example, constructing DNNFs would suffice to perform required reasoning tasks efficiently for model-based diagnosis (e.g., [8,3,18]) and testing (e.g., [31,32]).

However, state-of-the-art knowledge compilers all generate deterministic subsets of DNNF (see, e.g., [12,23,25]). Yet, unsurprisingly, the addition of determinism comes with a cost of generating less succinct representations. In particular, as

recently shown [4], DNNF is exponentially more succinct than its deterministic subsets. Therefore, for those applications where only decomposability is sufficient, compiling a deterministic subset of DNNF not only implies performing more work than necessary, but it could also result in generating larger DNNFs which would make reasoning tasks less efficient (if compilation is possible at all). Still, all existing compilers that we know of to generate decomposability also ensure determinism.

In this paper, we focus on compiling DNNFs without enforcing determinism, and make several contributions in that matter. Our main contribution is a new methodology to compile DNNFs by leveraging existing knowledge compilers. The key insight behind our approach is a new type of equivalence relation between two Boolean functions: a Boolean function $f(\mathbf{X})$ over variables $\mathbf{X}$ is *equivalent modulo forgetting* to another Boolean function $g(\mathbf{X}, \mathbf{Y})$ over variables $\mathbf{X}$ and $\mathbf{Y}$ iff existentially quantifying (also known as, forgetting) variables $\mathbf{Y}$ from g results in a function equivalent to f. The relevance of this notion to DNNF compilation is the well-known result that one can forget arbitrarily many variables on a given DNNF in linear time in the DNNF size, without losing the property of decomposability but not necessarily determinism [9]. Thus, instead of compiling function f directly, one can compile function g into a deterministic DNNF using existing compilers, on which forgetting variables $\mathbf{Y}$ would result in a DNNF that is not necessarily deterministic and equivalent to f.

The usefulness of our new approach depends on two important questions, which we address in this paper both theoretically and empirically: (i) to what extend forgetting variables could lead to more compact DNNFs without determinism than deterministic DNNFs, and (ii) how can one identify functions that are equivalent modulo forgetting. On the theoretical side, we present two main results. First, we show that even forgetting a single auxiliary variable can lead to exponential difference between sizes of DNNFs with and without determinism. Second, we study various existing approaches, such as Tseitin transformation [33], extended resolution [34], and bounded variable addition (BVA) [22], where auxiliary variables are introduced to formulas, mostly to obtain an equisatisfiable formula so that SAT task can be performed or becomes easy.[1] We show that those existing techniques indeed correspond to generating functions that are equivalent modulo forgetting, and hence offering some practical ideas to apply to our approach. In particular, we show that BVA would generate CNFs without increasing the treewidth of the input CNF much in the worst case, and could potentially reduce it to a bounded value from an unbounded value. Since CNF-to-DNNF compilation is tractable for bounded treewidth [9], this result shows the potential of BVA on DNNF compilation. On the practical side, we demonstrate that BVA, which turns out to be useful for SAT solving, can significantly advance DNNF compilation.

This paper is structured as follows. We start with providing some technical preliminaries in Section 2. We then describe our new method in detail in Section 3. This is followed by showing that forgetting a single auxiliary variable can

[1] Two formulas are equisatisfiable when the satisfiability of one depends on the other.

lead to exponential separation between DNNFs with and without determinism in Section 4. We then make a treatment of various existing approaches in the literature as equivalent modulo forgetting transformations in Section 5. After providing an empirical evaluation of our new approach in Section 6, we continue with a discussion on related work in Section 7. We conclude the paper with a few remarks in Section 8.

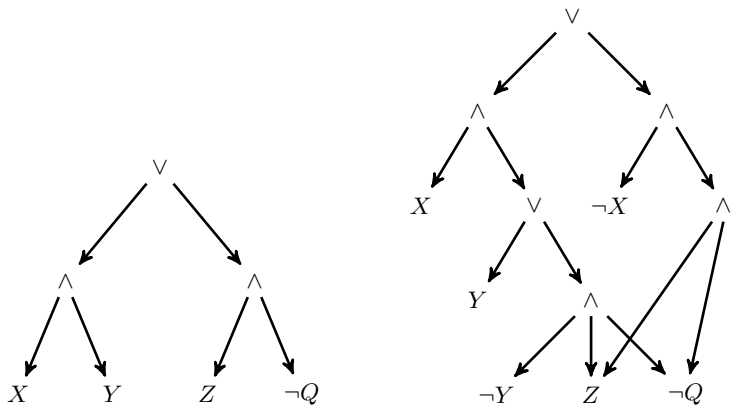
2 Technical Preliminaries

In this section, we will briefly introduce the concepts that will be used throughout the paper. We will use upper-case letters (e.g., X) to denote variables and lower-case letters (e.g., x) to denote their instantiations. That is, x is a *literal* denoting X or $\neg X$. We will use bold upper-case letters (e.g., $\mathbf{X}$) to denote sets of variables and bold lower-case letters (e.g., $\mathbf{x}$) to denote their instantiations.

A *Boolean function* f over variables $\mathbf{Z}$, denoted $f(\mathbf{Z})$, is a function that maps each instantiation $\mathbf{z}$ of variables $\mathbf{Z}$ to either 1/true or 0/false. A *trivial* Boolean function maps all its inputs to true (denoted $\top$) or maps them all to false (denoted $\bot$). An instantiation $\mathbf{z}$ *satisfies* function f iff f maps $\mathbf{z}$ to true. In this case, $\mathbf{z}$ is said to be a *model* of function f. The *model count* of function f is the number of models of f. Two functions f and g are logically equivalent, denoted $f \equiv g$, iff they have the same set of models. The *conditioning* of function f on instantiation $\mathbf{x}$, denoted $f|\mathbf{x}$, is the sub-function obtained by setting variables $\mathbf{X}$ to their values in $\mathbf{x}$. The *existential quantification* of variable X from function f, denoted $\exists X.\, f$, is the function obtained by disjoining functions $f|X$ and $f|\neg X$ (that is, $\exists X.\, f = f|X \vee f|\neg X$). Existential quantification is also known as *forgetting*, and can also be performed on a set of variables $\mathbf{X}$ by successively quantifying variables in $\mathbf{X}$. We will combine Boolean functions using the traditional Boolean operators, such as $\wedge$, $\vee$, $\oplus$, and $\Leftrightarrow$.

CNF: A *conjunctive normal form* (CNF) is a conjunction of clauses, where each clause is a disjunction of literals. For instance, $(X \vee \neg Y) \wedge (\neg X \vee Y \vee Z) \wedge \neg Z$ is a CNF with three clauses. Conditioning CNF Δ on literal ℓ amounts to removing literal $\neg\ell$ from all clauses and then dropping all clauses that contain literal ℓ.

NNF: A *negation normal form* (NNF) is a rooted, directed acyclic graph whose internal nodes are labeled with either conjunctions (i.e., $\wedge$) or disjunctions (i.e., $\vee$) and whose leaf nodes are labeled with either literals or constants $\top$ and $\bot$ [15]. A conjunction is *decomposable* iff each pair of its conjuncts share no variables [9]. A disjunction is *deterministic* iff each pair of its disjuncts are inconsistent with each other [10]. A *decomposable negation normal form* (DNNF) is an NNF whose conjunctions are decomposable [9]. A *deterministic* DNNF (d-DNNF) is a DNNF whose disjunctions are deterministic [10]. For instance, Fig. 1 illustrates a DNNF and a d-DNNF that are both equivalent to the CNF $(X \vee Z \wedge (X \vee \neg Q) \wedge (Y \vee Z) \wedge (Y \vee \neg Q)$ (note that the former is not necessarily deterministic).



(a) DNNF (b) d-DNNF

Fig. 1. A DNNF and a d-DNNF for the CNF $(X \vee \neg Q) \wedge (X \vee Z) \wedge (Y \vee \neg Q) \wedge (Y \vee Z)$.

3 Compiling DNNFs through Forgetting Variables

In this section, we will describe the proposed methodology, which is based on a new type of equivalence relation between two functions.

Definition 1. *Let $f(\mathbf{X})$ and $g(\mathbf{X}, \mathbf{Y})$ be two Boolean functions, where variables $\mathbf{X}$ and $\mathbf{Y}$ are disjoint. Then function f is said to be equivalent modulo forgetting (emf) to function g iff the following holds:*

$$f(\mathbf{X}) \equiv \exists\, \mathbf{Y}.\, g(\mathbf{X}, \mathbf{Y}).$$

Intuitively, the models of functions f and g match on their values over variables $\mathbf{X}$. Specifically, for each model $\mathbf{x}$ of f, there must exist an instantiation $\mathbf{y}$ such that $\mathbf{xy}$ is a model of g. Similarly, for each model $\mathbf{xy}$ of g, $\mathbf{x}$ must be a model of f. In other words, function f says everything function g says on variables $\mathbf{X}$. Hence, variables $\mathbf{Y}$ only act as *auxiliary* from the view of function f. We note that the model counts of f and g are not necessarily the same.

We utilize this notion in compiling DNNFs as shown in Algorithm 1. Here, to compile a DNNF representation of a function $f(\mathbf{X})$, we first obtain another function $g(\mathbf{X}, \mathbf{Y})$ that is emf to function f, with variables $\mathbf{Y}$ being auxiliary (Line 1). Clearly, the specific method to construct function g would depend on the input representation of f. We will discuss different ways for that later in Section 5 when the input is a CNF. Once function g is constructed, we compile a deterministic DNNF representation of it using an off-the-shelf knowledge compiler (Line 2). Finally, we forget auxiliary variables $\mathbf{Y}$ from the compiled structure (Line 3). This would generate a DNNF representation of the input as g is emf to function f.

Proposition 1. *Algorithm 1 returns a DNNF representation of its input.*

Algorithm 1: $DNNF(f)$

Input: $f(\mathbf{X})$: a Boolean function over variables $\mathbf{X}$
Output: constructs a DNNF representation of function f

1 $g(\mathbf{X}, \mathbf{Y}) \leftarrow emf(f)$
2 $\Delta \leftarrow$ compile $g(\mathbf{X}, \mathbf{Y})$ using a d-DNNF compiler
3 $\Gamma \leftarrow$ forget variables $\mathbf{Y}$ from Δ
4 **return** Γ

We remark that the last step of Algorithm 1 can be performed only in linear time in the size of the structure. This is due to the property of decomposability, which supports linear time multiple-variable forgetting: all one needs is to replace auxiliary variables with the constant $\top$ in the structure. An example of this procedure is depicted in Fig. 2, where we forget variables X, Z from a deterministic DNNF. What is crucial here is that the resulting structure does not enforce determinism anymore, but the decomposability property stays intact. In fact, as we will show in the next section, this could lead to exponentially more succinct representations, which can be thought of as a compensation for losing the ability of performing efficient model counting.

4 An Exponential Separation by Forgetting Variables

In this section, we address the following question: to what extend forgetting auxiliary variables could lead to more compact DNNFs without determinism than deterministic DNNFs?

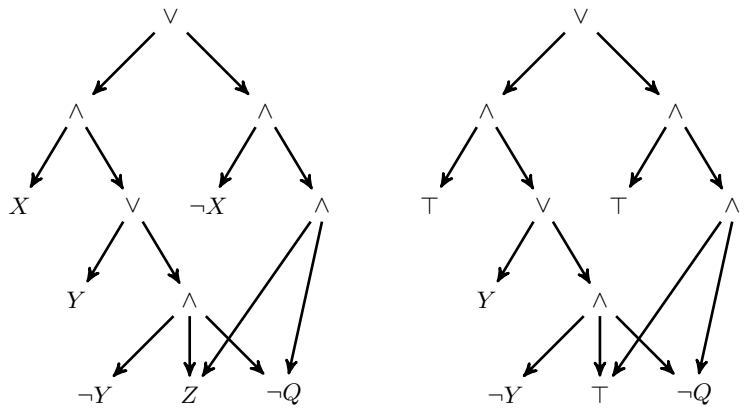
We next state our main result, showing that exponentially more compact representations can be obtained.

Theorem 1. *There exist two classes of Boolean functions $f_n(\mathbf{X})$ and $g_n(\mathbf{X}, Z)$ such that: (i) f_n is emf to g_n, (ii) the size of each d-DNNF computing f_n is at least exponential in n, and (iii) there is a d-DNNF computing g_n whose size is polynomial in n.*

In other words, it is not feasible to compile a deterministic DNNF representation of f_n, yet one can construct a compact DNNF representation of f_n through forgetting a single auxiliary variable from the compact deterministic DNNF representation of g_n. We remark that obtaining a DNNF computing f_n directly is not possible in practice as existing knowledge compilers generate deterministic subsets of DNNF.

We next present the proof of Theorem 1, where we make use of the function that has been shown to exponentially separate DNNFs from deterministic DNNFs [4].

Let $\mathbf{M}$ be an $n \times n$ matrix of Boolean variables. Let $\mathbf{R}_1, \ldots, \mathbf{R}_n$ be the rows of M and $\mathbf{C}_1, \ldots, \mathbf{C}_n$ be the columns of M. Let h_n be the class of functions over



(a) Before forgetting (b) After forgetting

Fig. 2. Forgetting variables X, Z from a DNNF.

n variables evaluating to 1 iff the sum of its inputs is divisible by 3. Consider the following function defined on the variables of $\mathbf{M}$ and variable Z:

$$g_n(\mathbf{M}, Z) = \big(Z \wedge row_n(\mathbf{M})\big) \vee \big(\neg Z \wedge col_n(\mathbf{M})\big),$$

where row_n and col_N are defined by

$$row_n(\mathbf{M}) = \bigoplus_{i=1}^{n} h_n(\mathbf{R}_i), \qquad col_n(\mathbf{M}) = \bigoplus_{i=1}^{n} h_n(\mathbf{C}_i).$$

Finally, let f_n be the following function defined on the variables of $\mathbf{M}$:

$$f_n(\mathbf{M}) = row_n(\mathbf{M}) \vee col_n(\mathbf{M}).$$

Clearly, $f_n(\mathbf{M}) \equiv \exists\, Z.\, g_n(\mathbf{M}, Z)$, and hence function f_n is emf to function g_n. Indeed, function f_n is the *Sauerhoff* function [30], which was used in the exponential separation of DNNFs from deterministic DNNFs [4]. That is, f_n has a polynomial size DNNF representation, but each deterministic DNNF computing it is exponential in size. Finally, since functions row_n and col_n both have polynomial size OBDDs (a subset of deterministic DNNF), function g_n has a polynomial size deterministic DNNF representation. Thus, Theorem 1 holds.

As a side note, this result implies that forgetting on deterministic DNNF and FBDD cannot be done in polynomial time, which was only known up to some standard complexity-theoretic assumptions (i.e., $\mathsf{P} \neq \mathsf{NP}$).[2]

Corollary 1. *d-DNNF and FBDD do not support polynomial time single-variable forgetting, as well as polynomial time multiple-variable forgetting.*

[2] The same function can also be used to show that d-DNNF and FBDD do not support polynomial time disjunction operation.

Theorem 1 reveals the usefulness of our new approach in theory. To make it useful in practice, we need to identify transformations that would produce emf formulas, which is discussed next.

5 EMF Transformations

In this section, we address the following question: how can one identify functions that are equivalent modulo forgetting?

We will study some existing techniques for CNFs that incorporate auxiliary variables, mostly to get an equisatisfiable CNF. For each technique, we will demonstrate that the produced equisatisfiable CNF is indeed emf to the input CNF. We first formally define a notion of transformation that will be used to identify methods producing emf formulas.

Definition 2. *Let T be an algorithm that takes as input a Boolean function $f(\mathbf{X})$ and outputs another Boolean function $g(\mathbf{X}, \mathbf{Y})$, where $\mathbf{X}$ and $\mathbf{Y}$ are disjoint. Then algorithm T is said to be an emf transformation iff function f is emf to function g.*

Given this definition, we next present some emf transformations that exist in the literature.

5.1 Tseitin Transformation

State-of-the-art SAT solvers require their input to be a Boolean formula in CNF. When this is not the case, one has to first transform the input into a CNF. The naive approach here is to use the famous De Morgan's law and the distributive property, which preserves logical equivalence. However, this can easily blow-up CNF size exponentially. Thus, one typically applies Tseitin transformation [33], which converts a Boolean formula into an equisatisfiable CNF by adding auxiliary variables with only a linear increase in size. In fact, Tseitin transformation does more than constructing an equisatisfiable CNF. In particular, it guarantees two more properties [33]:

(1) Dropping auxiliary variables from a model of the constructed CNF would yield a model of the input formula;
(2) Any model of the input formula can be extended to be a model of the constructed CNF.

As we prove next, these two properties make Tseitin transformation an emf transformation, as well as any other transformation that satisfies them.

Theorem 2. *Let T be a transformation that satisfies the two properties above. Then T is an emf transformation.*

Let $f(\mathbf{X})$ be the input function to transformation T, and let $g(\mathbf{X}, \mathbf{Y})$ be the function constructed for $f(\mathbf{X})$ by transformation T, where variables $\mathbf{Y}$ are introduced during the transformation. We want to show that $f(\mathbf{X}) \equiv \exists \mathbf{Y}. g(\mathbf{X}, \mathbf{Y})$.

Let $\mathbf{x}$ be a model of $\exists \mathbf{Y}. g(\mathbf{X}, \mathbf{Y})$. We will show that $\mathbf{x}$ is also a model of $f(\mathbf{X})$. Since $\mathbf{x}$ is a model of $\exists \mathbf{Y}. g(\mathbf{X}, \mathbf{Y})$, there must be an instantiation $\mathbf{y}$ such that $\mathbf{xy}$ is a model of $g(\mathbf{X}, \mathbf{Y})$. Then, by the first property above, $\mathbf{x}$ must be a model of $f(\mathbf{X})$.

Let $\mathbf{x}$ be a model of $f(\mathbf{X})$. We will show that $\mathbf{x}$ is also a model of $\exists \mathbf{Y}. g(\mathbf{X}, \mathbf{Y})$. Due to the second property above, there must exist an instantiation $\mathbf{y}$ such that $\mathbf{xy}$ is a model of $g(\mathbf{X}, \mathbf{Y})$. Then, as $\exists \mathbf{Y}. g(\mathbf{X}, \mathbf{Y})$ says everything $g(\mathbf{X}, \mathbf{Y})$ says on variables $\mathbf{X}$, $\mathbf{x}$ must be a model of $\exists \mathbf{Y}. g(\mathbf{X}, \mathbf{Y})$.

Therefore, Theorem 2 holds, which immediately implies that Tseitin transformation is an emf transformation.

Proposition 2. *Tseitin transformation is an emf transformation.*

Accordingly, we can apply Tseitin transformation to compile DNNF when the input is not in CNF, which is also the required format for most knowledge compilers.

5.2 Extended Resolution

Resolution is a powerful rule of inference that has been used in SAT solving [28]. Specifically, iterating the following rule repeatedly in a certain way would tell whether a CNF is satisfiable or not:

$$\frac{X \vee \alpha \quad \neg X \vee \beta}{\alpha \vee \beta},$$

where X is a variable and α and β are clauses. This rule states that whenever the clauses in the premise appear in a CNF, one can increment the CNF by adding the clause in the conclusion, without changing the logical content of the CNF (i.e., preserving logical equivalence). Here, $\alpha \vee \beta$ is called the *resolvent* obtained by *resolving* variable X on $X \vee \alpha$ and $\neg X \vee \beta$.

It turns out that resolution could generate only exponentially long proofs of unsatisfiability for certain families of formulas (see, e.g., the Pigeonhole principle [16]). To remedy this, *extended* resolution is introduced, which is a more powerful generalization of resolution that includes an additional rule, called the *extension* rule [33]. Accordingly, extended resolution allows one to increment the CNF with the addition of clauses of the form $X \Leftrightarrow \ell_1 \vee \ell_2$[3], where X is an auxiliary variable that does not appear in the CNF and literals ℓ_1 and ℓ_2 appear in the CNF. Then one can apply the resolution rule as before. This simple addition creates an exponentially more powerful proof system than resolution, as extended resolution could generate polynomial size proofs where the regular resolution can only generate exponential size proofs [7].

[3] More specifically, $X \Leftrightarrow \ell_1 \vee \ell_2$ can be replaced with the clauses $\neg X \vee \ell_1 \vee \ell_2$, $X \vee \neg \ell_1$, $X \vee \neg \ell_2$.

Indeed, extended resolution constructs an equisatisfiable CNF, and thus applying resolution on it produces correct results for SAT solving. This technique has also been shown to be useful in practice of SAT solving, where different schemes for applying the extension rule have been suggested [17,1,21]. Hence, its usage could potentially be extended to DNNF compilation, given that we will now show it is indeed an emf transformation.

We will now prove the following result, which generalizes extended resolution.

Theorem 3. *Let $f(\mathbf{X}), \alpha^1(\mathbf{X}), \ldots, \alpha^n(\mathbf{X})$ be Boolean functions. Consider the class of Boolean functions*

$$g_n(\mathbf{X}, \mathbf{Y}) = f(\mathbf{X}) \wedge (Y_1 \Leftrightarrow \alpha^1(\mathbf{X})) \wedge \ldots \wedge (Y_n \Leftrightarrow \alpha^n(\mathbf{X})),$$

where $\mathbf{Y} = \{Y_1, \ldots, Y_n\}$. Then function f is emf to function g_n.

We want to show that $f(\mathbf{X}) \equiv \exists \mathbf{Y}.\, g_n(\mathbf{X}, \mathbf{Y})$. For that, we will use the following simplification n times:

$$\begin{aligned}
\exists \mathbf{Y}.\, g_n &\equiv \exists Y_1, \ldots, Y_{n-1}.\ \exists Y_n.\, f(\mathbf{X}) \wedge \bigwedge_{i=1}^{n} Y_i \Leftrightarrow \alpha^i(\mathbf{X}) && (1)\\
&\equiv \exists Y_1, \ldots, Y_{n-1}.\ f(\mathbf{X}) \wedge \left(\bigwedge_{i=1}^{n-1} Y_i \Leftrightarrow \alpha^i(\mathbf{X}) \right) \wedge \exists Y_n.\, Y_n \Leftrightarrow \alpha^n(\mathbf{X}) && (2)\\
&\equiv \exists Y_1, \ldots, Y_{n-1}.\, f(\mathbf{X}) \wedge \bigwedge_{i=1}^{n-1} Y_i \Leftrightarrow \alpha^i(\mathbf{X}) && (3)\\
&\ldots\\
&\equiv \exists Y_1.\, f(\mathbf{X}) \wedge \big(Y_1 \Leftrightarrow \alpha_1(\mathbf{X})\big)\\
&\equiv f(\mathbf{X}).
\end{aligned}$$

Equation (1) is due to the definition of multiple-variable forgetting. Equation (2) holds as $f(\mathbf{X}) \wedge \bigwedge_{i=1}^{n-1} Y_i \Leftrightarrow \alpha_i(\mathbf{X})$ does mention variable Y_n. Equation (3) holds as forgetting variable Y_n from $Y_n \Leftrightarrow \alpha_n(\mathbf{X})$ is equivalent to the trivial function $\top$.

Assuming that $f(\mathbf{X})$ is a CNF, replacing each $\alpha^i(\mathbf{X})$ with a clause of two literals of variables $\mathbf{X}$ would clearly correspond to the extension rule of extended resolution.

Proposition 3. *Extended resolution is an emf transformation.*

5.3 Bounded Variable Addition

Bounded variable addition (BVA) is a preprocessing technique introduced for SAT solving [22]. The goal here is to reduce the sum of the number of variables and clauses of a CNF by introducing auxiliary variables, without losing the ability of answering the SAT query. It is based on resolution as described next.

Let C_X be a set of clauses containing literal X and $C_{\neg X}$ a set of clauses containing literal $\neg X$. Let $C_X \bowtie C_{\neg X}$ denote the set of resolvents one would obtain by resolving X on clauses in C_X and $C_{\neg X}$. Given a CNF Δ and an auxiliary variable X that does not appear in Δ, BVA looks for sets of clauses C_X and $C_{\neg X}$ such that $C_X \bowtie C_{\neg X}$ belongs to Δ and $|C_X \bowtie C_{\neg X}| > |C_X| + |C_{\neg X}|$. In this case, BVA replaces clauses $C_X \bowtie C_{\neg X}$ with clauses C_X and $C_{\neg X}$. For instance, consider the following CNF:

$$\Delta = (A \vee D) \wedge (B \vee D) \wedge (C \vee D) \wedge (A \vee E) \wedge (B \vee E) \wedge (C \vee E).$$

By adding an auxiliary variable X, we can obtain the following CNF which has fewer clauses than Δ:

$$\Sigma = (A \vee \neg X) \wedge (B \vee \neg X) \wedge (C \vee \neg X) \wedge (D \vee X) \wedge (E \vee X).$$

Indeed, Δ is equisatisfiable to Σ, and thus one can feed Σ to a SAT solver, instead of Δ.

The authors of [22] also developed a heuristic to apply the BVA transformation on CNFs, which is a greedy algorithm that searches for clause-patterns in the input CNF. This algorithm is shown to be useful in SAT solving, and thus offering a practical idea to apply to our DNNF compilation method due to the following result.

Proposition 4. *Bounded variable addition is an emf transformation.*

We will now prove the above proposition. In particular, let $\Delta(\mathbf{X})$ be a CNF and $\Sigma(\mathbf{X}, Y)$ be the CNF obtained by applying BVA on Δ, where Y is the auxiliary variable added during the process. Then we want to show that $\Delta(\mathbf{X}) \equiv \exists Y.\, \Sigma(\mathbf{X}, Y)$.

Let $\Gamma_Y = \wedge_{i=1}^{m} Y \vee \alpha_i$ and $\Gamma_{\neg Y} = \wedge_{j=1}^{k} \neg Y \vee \beta_j$ be the clauses containing literals Y and $\neg Y$ in CNF Σ, respectively. Then, due to the BVA process, we can rewrite CNF Δ as the CNF $\Phi \wedge \Gamma_Y \bowtie \Gamma_{\neg Y}$ where Φ is a CNF. Moreover, CNF Σ is equivalent to $\Phi \wedge \Gamma_Y \wedge \Gamma_{\neg Y}$. In this setting, we have the following equations:

$$\begin{aligned}
\exists Y.\, \Sigma(\mathbf{X}, Y) &\equiv \exists Y.\, \Phi \wedge \Gamma_Y \wedge \Gamma_{\neg Y} \\
&\equiv \Phi \wedge \exists Y.\, \Gamma_Y \wedge \Gamma_{\neg Y} \\
&\equiv \Phi \wedge \Big((\Gamma_Y \wedge \Gamma_{\neg Y})|Y \vee (\Gamma_Y \wedge \Gamma_{\neg Y})|\neg Y \Big) \\
&\equiv \Phi \wedge \Big(\Gamma_{\neg Y}|Y \vee \Gamma_Y|\neg Y \Big) \\
&\equiv \Phi \wedge \Big(\bigwedge_{j=1}^{k} \beta_j \vee \bigwedge_{i=1}^{m} \alpha_i \Big) \\
&\equiv \Phi \wedge \Big(\bigwedge_{j=1}^{k} \bigwedge_{i=1}^{m} \beta_j \vee \alpha_i \Big) \\
&\equiv \Phi \wedge \Gamma_Y \bowtie \Gamma_{\neg Y} \\
&\equiv \Delta(\mathbf{X}).
\end{aligned}$$

Treewidth and BVA We now identify another guarantee that comes with the BVA transformation. Treewidth[4] is a well-known graph-theoretic property [27], which has been extensively used as a parameter that renders many hard reasoning tasks tractable when being small. In the context of knowledge compilation, it is known that compiling a CNF into a deterministic DNNF can be done in the worst case in time that is linear in the number of variables and exponential in the treewidth of the CNF primal graph [9].[5] Therefore, a CNF with a bounded treewidth can easily be compiled into a deterministic DNNF.

We will next present two results regarding the effects of the BVA transformation on the primal treewidth of CNFs, whose proofs are delegated to Appendix A. Our first result is the following guarantee.

Theorem 4. *Let Δ be a CNF whose primal treewidth is w. Let Σ be the CNF obtained by applying the BVA transformation k times on CNF Δ. Then the primal treewidth of Σ is at most $w + k$.*

Hence, the BVA transformation would not affect the treewidth much in the worst case, when applied constant times. Moreover, as we present next, the BVA transformation could potentially reduce the treewidth from an unbounded value to a bounded value.

Theorem 5. *There exists a class of CNFs Δ_n over n^3 variables such that: (i) the primal treewidth of Δ_n is unbounded (i.e., at least n), and (ii) applying the BVA transformation 2 times on Δ_n can generate a CNF whose primal treewidth is bounded (i.e., at most 2).*

Theorem 5 implies that the BVA transformation can generate a CNF whose compilation to deterministic DNNF is easy, whereas this cannot be identified in the input CNF (as the treewidth is unbounded). Therefore, Algorithm 1 can easily compile a DNNF in this case, if the BVA transformation is applied. Yet, there is no guarantee on compiling a deterministic DNNF with existing compilers, without applying the BVA transformation. Indeed, we will confirm this empirically in our experiments, where the following class of CNFs Δ^a_n will be considered:

$$\bigwedge_{1 \leq i,j,k \leq n} X_i \vee Y_j \vee Z_k.$$

This class of CNFs has unbounded treewidth. On the other hand, the following class of CNFs Δ^b_n can be identified by the BVA transformation, which has bounded treewidth.

$$\Big(\bigwedge_{1 \leq i \leq n} A \vee X_i \Big) \wedge \Big(\bigwedge_{1 \leq j \leq n} \neg A \vee B \vee Y_j \Big) \wedge \Big(\bigwedge_{1 \leq k \leq n} \neg B \vee Z_k \Big).$$

[4] The definition of treewidth and some of its properties is delegated to Appendix A.

[5] Primal graph is a CNF abstraction that represents the connection between the variables and clauses of the CNF. In particular, each vertex of the graph represents a variable, and there is an edge between to vertices iff the corresponding variables appear together in one of the CNF clauses.

Table 1. Experimental results on CNFs Δ_n^a. `c2d_forget` is our approach, compiling DNNFs without determinism. All timings are in seconds.

	`c2d_forget`			`c2d`		
Δ_n^a	#node	#edge	Time	#node	#edge	Time
10	42	43	0.04	794	1,578	0.11
15	57	58	0.03	26,199	52,368	11.43
30	102	103	0.04	–	–	–
50	162	163	0.04	–	–	–
75	237	238	0.04	–	–	–
100	312	313	0.04	–	–	–

Note that we added two auxiliary variables A, B into CNF Δ_n^a, and reduced the number of clauses from n^3 to $3n$.

6 Experiments

In this section, we will empirically demonstrate the applicability of Algorithm 1 in compiling DNNFs, when coupled with the BVA transformation. In particular, we compile CNFs into DNNF and deterministic DNNF. For the latter we use the `c2d` compiler[6], and for the former we use the same compiler after preprocessing CNFs by the preprocessor `Coprocessor`[7] and forgetting auxiliary variables after the compilation.

We evaluated the mentioned systems on two different benchmarks. First, we used the manually constructed class of CNFs Δ_n^a (described in Section 5) for values of $n \in \{10, 15, 30, 50, 75, 100\}$. Second, we used some CNF encodings of wire routing problems in the channels of field-programmable gate arrays (FPGA) [24]. The goal here is to decide if a routing configuration is possible. That is, given m connections and k channels on an FPGA (denoted fpga_m_n), the satisfiability of the CNF encoding would imply that the routing of m connections through k channels is possible. Our experiments were performed on a 2.6GHz Intel Xeon E5-2670 CPU with a 1 hour time limit and a memory limit of 8GB RAM.

Table 1 highlights the results on CNFs Δ_n^a. According to this, our approach (`c2d_forget`) recognizes the tractability of the CNF instances by introducing two auxiliary variables (as shown in Section 5), and thus it compiles the instances quickly and compactly. On the other hand, the traditional approach (`c2d`) performed poorly as it could not finish compilation after $n = 20$.

For the FPGA routing problems, we first present some statistics of the CNF instances before and after the preprocessing in Table 2. We now highlight the

[6] Available at `http://reasoning.cs.ucla.edu/c2d`.
[7] Available at `http://tools.computational-logic.org/content/riss.php`.

Table 2. Some stats on CNF encodings of FPGA routing problems, before and after preprocessing.

	Before BVA		After BVA		
Instance	#variable	#clause	#variable	#clause	#aux_variable
fpga_10_8	120	448	158	290	38
fpga_10_9	135	549	174	330	39
fpga_12_8	144	560	188	356	44
fpga_12_9	162	684	207	405	45
fpga_12_11	198	968	269	503	71
fpga_12_12	216	1128	300	552	84
fpga_13_9	176	759	229	444	53

results in Table 3. Accordingly, our approach is clearly superior than the traditional approach as we can compile 5 instances which otherwise could not be compiled. In the remaining 2 instances, not only our approach produces DNNFs faster but also constructs more compact representations. Therefore, our approach improves performance of DNNF compilation on these FPGA problems.

7 Related Work

The closest related work to ours is perhaps the work of [26], in which the authors identified a subset of DNNF, called *structured* DNNF. The significance here is that this subset supports a polynomial time conjoin operation [26], while general DNNF do not support this (unless $\mathsf{P} = \mathsf{NP}$) [15]. Due to this operation, one can compile CNFs incrementally in a bottom-up fashion into a structured DNNF. That is, after representing each clause as a structured DNNF (which can be done easily), one can conjoin clauses one by one until a structured DNNF is compiled for the input CNF. Indeed, the compiled DNNF would not necessarily be deterministic (as the conjoin operation does not enforce this). However, building an efficient knowledge compiler based on this approach would require intensive engineering effort and has not been accomplished yet. Our work, on the other hand, leverages state-of-the-art knowledge compilers as it only depends on constructing emf formulas. Due to this, one can quickly build an efficient DNNF compiler, as we have done in this work. Moreover, DNNF could be exponentially more succinct than its structured subset, which could make the mentioned work more restrictive than our presented approach.

Another related work to ours is that of [19,20], in which the authors studied the effects of preprocessing CNFs for model counting. They considered various techniques from the literature and also introduced a few new ones, which resulted in an efficient preprocessor. Their focus was on constructing CNFs that

Table 3. Experimental results on FPGA routing problems. `c2d_forget` is our approach, compiling DNNFs without determinism. All timings are in seconds.

	`c2d_forget`			`c2d`		
Instance	#node	#edge	Time	#node	#edge	Time
fpga_10_8	38,601	116,399	0.66	122,106	398,915	97.63
fpga_10_9	37,528	107,316	0.84	199,563	695,470	661.85
fpga_12_8	215,790	595,522	26.87	–	–	–
fpga_12_9	428,340	1,303,189	92.68	–	–	–
fpga_12_11	491,225	1,428,101	99.43	–	–	–
fpga_12_12	389,274	1,115,493	207.58	–	–	–
fpga_13_9	1,149,770	3,133,399	268.34	–	–	–

are either equivalent or preserving the model count. Our work is based on preprocessing techniques that generate emf formulas, and targets compiling DNNFs, as opposed to counting the models.

Finally, [2] studied the problem of *projected* model counting, in which the goal is to compute the model count of a formula after forgetting certain variables. In their setting, auxiliary variables are named as "non-priority" variables. The main distinction here is that we are not interested in the model counting after forgetting variables. Because of this, interestingly enough, the forgetting operation helps in our setting to obtain more compact representations.

8 Conclusion

In this work, we studied compiling DNNFs without enforcing determinism. We presented a new methodology to relax determinism, which is based on introducing auxiliary variables and forgetting them from a deterministic DNNF. We demonstrated that several existing techniques that introduce auxiliary variables can be used in our framework, allowing us to exploit existing knowledge compilers. We further showed that our new approach can lead to exponentially more compact representations, and our experimental evaluation confirmed the applicability of the new technique on certain benchmarks, when bounded variable addition is employed to introduce auxiliary variables.

A Treewidth

In this section, we will define treewidth and present the proofs of Theorem 4 and Theorem 5. We start with a definition of the primal treewidth of a CNF, where we choose to use the one based on *jointrees* (e.g., [13]) among a number of ways.

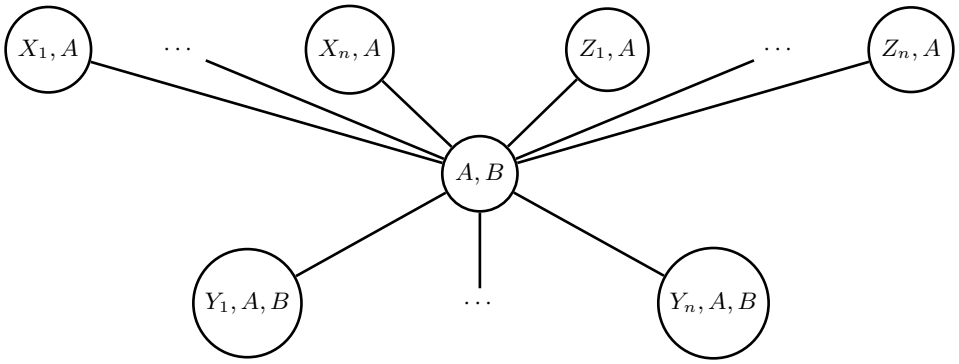


Fig. 3. A jointree for CNF Δ_n^b.

A jointree for a CNF Δ is a tree whose vertices are labeled with a subset of variables of Δ such that the following two conditions hold:

- For each clause γ of Δ, there is a vertex whose labels contain the variables of γ;
- If a variable X appears in the labels of two vertices V_1 and V_2, then each vertex on the path connecting V_1 and V_2 includes variable X in its labels.

The labels of a vertex of a jointree is called its *cluster*. The *width* of a jointree is the size of its largest cluster minus 1. The primal treewidth of a CNF is the smallest width attained by any of its jointrees. For instance, Fig. 3 depicts a jointree for Δ_n^b whose width is 2.

Proof of Theorem 4 Assume that we apply the BVA transformation on CNF Δ once, and constructed the CNF Δ^1. So, an auxiliary variable X is added to CNF Δ^1. Consider now the best jointree of Δ (i.e., the one whose width is w). If we add variable X to each label set of its vertices, the resulting tree will clearly be a jointree for Δ^1, with width $w+1$. So, the treewidth of Δ^1 will be at most $w+1$. Now, if we apply the same idea after each application of the BVA transformation, the treewidth will be at most $w+k$ after the k^{th} step.

Proof of Theorem 5 We first show that the treewidth of Δ_n^a is unbounded (i.e., at least $2n$). In the primal graph of Δ_n^a, each vertex will have a degree of $2n$. According to a known result (see, e.g., [13]), this implies that the treewidth of Δ_n^a is no less than $2n$.

We will now show that the treewidth of Δ_n^b, which can be obtained from Δ_n^a by the BVA transformation, is bounded (i.e., at most 2). Figure 3 depicts a jointree for Δ_n^b whose width is 2. Hence, the treewidth of Δ_n^b is at most 2.

References

1. Audemard, G., Katsirelos, G., Simon, L.: A Restriction of Extended Resolution for Clause Learning Sat Solvers. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. pp. 15–20 (2010)
2. Aziz, R.A., Chu, G., Muise, C., Stuckey, P.: #∃ SAT: Projected Model Counting. In: Proceedings of the Eighteenth International Conference on Theory and Applications of Satisfiability Testing. pp. 121–137 (2015)
3. Barrett, A.: Model Compilation for Real-Time Planning and Diagnosis with Feedback. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence. pp. 1195–1200 (2005)
4. Bova, S., Capelli, F., Mengel, S., Slivovsky, F.: Knowledge Compilation Meets Communication Complexity. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence. pp. 1008–1014 (2016)
5. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
6. Chavira, M., Darwiche, A.: On Probabilistic Inference by Weighted Model Counting. Artifical Intelligence 172(6-7), 772–799 (2008)
7. Cook, S.A.: A Short Proof of the Pigeon Hole Principle Using Extended Resolution. SIGACT News 8(4), 28–32 (1976)
8. Darwiche, A.: Model-Based Diagnosis under Real-World Constraints. AI Magazine 21(2), 57–73 (2000)
9. Darwiche, A.: Decomposable Negation Normal Form. Journal of the ACM 48(4), 608–647 (2001)
10. Darwiche, A.: On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision. Journal of Applied Non-Classical Logics 11(1-2), 11–34 (2001)
11. Darwiche, A.: A Logical Approach to Factoring Belief Networks. In: Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning. pp. 409–420 (2002)
12. Darwiche, A.: New Advances in Compiling CNF into Decomposable Negation Normal Form. In: Proceedings of the Sixteenth European Conference on Artificial Intelligence. pp. 328–332 (2004)
13. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009)
14. Darwiche, A.: SDD: A New Canonical Representation of Propositional Knowledge Bases. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence. pp. 819–826 (2011)
15. Darwiche, A., Marquis, P.: A Knowledge Compilation Map. Journal of Artificial Intelligence Research 17(1), 229–264 (2002)
16. Haken, A.: The Intractability of Resolution. Theoretical Computer Science 39, 297–308 (1985)
17. Huang, J.: Extended clause learning. Artificial Intelligence 174(15), 1277–1284 (2010)
18. Huang, J., Darwiche, A.: On Compiling System Models for Faster and More Scalable Diagnosis. In: Proceedings of the Twentieth AAAI Conference on Artificial Intelligence. pp. 300–306 (2005)
19. Lagniez, J.M., Marquis, P.: Preprocessing for Propositional Model Counting. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence. pp. 2688–2694 (2014)

20. Lagniez, J.M., Marquis, P.: On Preprocessing Techniques and Their Impact on Propositional Model Counting. Journal of Automated Reasoning 58(4), 413–481 (2017)
21. Manthey, N.: Extended Resolution in Modern SAT Solving. In: Joint Automated Reasoning Workshop and Deduktionstreffen (2014)
22. Manthey, N., Heule, M., Biere, A.: Automated Reencoding of Boolean Formulas. In: Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference. pp. 102–117 (2012)
23. Muise, C., Mcilraith, S.A., Beck, J.C., Hsu, E.: DSHARP: Fast d-DNNF Compilation with sharpSAT. In: Proceedings of the Twenty-Fifth Canadian Conference on Artificial Intelligence. pp. 356–361 (2012)
24. Nam, G., Aloul, F.A., Sakallah, K.A., Rutenbar, R.A.: A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints. IEEE Transactions on Computers 53(6), 688–696 (2004)
25. Oztok, U., Darwiche, A.: A Top-Down Compiler for Sentential Decision Diagrams. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence. pp. 3141–3148 (2015)
26. Pipatsrisawat, K., Darwiche, A.: New compilation languages based on structured decomposability. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence. pp. 517–522 (2008)
27. Robertson, N., Seymour, P.D.: Graph Minors. III. Planar Tree-width. Journal of Combinatorial Theory, Series B 36(1), 49–64 (1984)
28. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12(1), 23–41 (1965)
29. Roth, D.: On the hardness of approximate reasoning. Artificial Intelligence 82(1-2), 273–302 (1996)
30. Sauerhoff, M.: Approximation of boolean functions by combinatorial rectangles. Theoretical Computer Science 1–3(301), 45–78 (2003)
31. Schumann, A., Huang, J., Sachenbacher, M.: Computing Cost-Optimal Definitely Discriminating Tests. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (2010)
32. Schumann, A., Sachenbacher, M.: Computing Energy-Optimal Tests Using DNNF Graphs. In: Proceedings of the Twenty-First International Workshop on the Principles of Diagnosis (2010)
33. Tseitin, G.S.: On the Complexity of Derivations in the Propositional Calculus. Studies in Mathematics and Mathematical Logic Part II, 115–125 (1968)
34. Tseitin, G.S.: On the Complexity of Proofs in Propositional Logics. Seminars in Mathematics 8, 466–483 (1970)