A Compiler for Weak Decomposable Negation Normal Form

Petr Illner, Petr Kučera

Department of Theoretical Computer Science and Mathematical Logic, Faculty of Mathematics and Physics
Charles University, Czech Republic
illner3@gmail.com, kucerap@ktiml.mff.cuni.cz

Abstract

This paper integrates weak decomposable negation normal form (wDNNF) circuits, introduced by Akshay et al. in 2018, into the knowledge compilation map. This circuit type generalises decomposable negation normal form (DNNF) circuits in such a way that they allow a restricted form of sharing variables among the inputs of a conjunction node. We show that wDNNF circuits have the same properties as DNNF circuits regarding the queries and transformations presented in the knowledge compilation map, whilst being strictly more succinct than DNNF circuits (that is, they can represent Boolean functions compactly). We also present and evaluate a knowledge compiler, called Bella, for converting CNF formulae into wDNNF circuits. Our experiments demonstrate that wDNNF circuits are suitable for configuration instances.

Introduction

Knowledge compilation (Darwiche and Marquis 2002; Marquis 2015) concerns transforming a given propositional theory into a target language for efficient query answering and manipulation. Many target languages (circuit types) are defined as a restriction of negation normal form (NNF) circuits. One of the most general languages considered in the knowledge compilation map (Darwiche and Marquis 2002) is the language of decomposable NNF (DNNF) circuits (Darwiche 1999). A DNNF circuit is an NNF circuit in which every AND node is decomposable, meaning no variable is shared among its inputs. DNNF circuits form the most succinct language within the knowledge compilation map that supports efficient answering of the following queries: Consistency check, clause entailment check, and model enumeration. Additional restrictions are often put on DNNF circuits to support answering other queries efficiently. For example, let us note deterministic DNNF (d-DNNF) circuits (Darwiche 2001b) that put additional restrictions on OR nodes to allow model counting in linear time. For completeness, we mention other languages, which are restricted variants of d-DNNF circuits, such as decision-DNNF circuits (Huang and Darwiche 2007), SDDs (Darwiche 2011), and OBDDs (Bryant 1986). DNNF circuits have many applications, including diagnosis (Huang and Darwiche 2005; Siddiqi and Huang 2007; Siddiqi 2011), configuration (Voronov, Åkesson, and Ekstedt 2011; Zengler and Küchlin 2013), MaxSAT (Pipatsrisawat and Darwiche 2007; Ramírez and Geffner 2007), and planning (Bonet and Geffner 2006).

Because of its importance, decomposability has been intensively studied, and some restricted and generalised variants have been invented. For example, let us mention a restricted variant called structured decomposability (Pipatsrisawat and Darwiche 2008), which defines structured DNNF (SDNNF) circuits. This circuit type has been included in the knowledge compilation map. On the other hand, a generalised variant, which was utilised in Boolean functional synthesis, is called weak decomposability (Akshay et al. 2018). In this case, it is allowed for a variable x to be shared among the inputs of an AND node α if either only x or only $\neg x$ appears in the subcircuit rooted at α. An NNF circuit in which all AND nodes are weak decomposable is called a weak DNNF (wDNNF) circuit. To our knowledge, this alluring circuit type has not yet been studied in terms of the knowledge compilation map. For completeness, we remark that weak decomposability (also known as consistency) combined with smoothness (also called completeness) was studied for sum-product networks (SPNs) (Peharz et al. 2015).

A crucial feature of a target language is whether we can implement a (knowledge) compiler that compiles into that language. We are unaware of any compiler that compiles into wDNNF circuits. Because there is even no direct compiler for DNNF circuits, the most common way is to compile a CNF formula into a decision-DNNF circuit using a compiler such as C2D (Darwiche 2004), D4 (Lagniez and Marquis 2017), DSHARP (Muise et al. 2010), or SharpSAT-TD (Kiesel and Eiter 2023). Since wDNNF circuits are strictly more succinct than decision-DNNF circuits, the full potential of succinctness is not used by compiling them into decision-DNNF circuits. For completeness, we note that Oztok and Darwiche (2017) described an indirect method of compiling into DNNF circuits that uses a decision-DNNF circuit as an intermediary. A disadvantage of this method is that we still need to generate the decision-DNNF circuit in the compilation process.

The main contributions of this paper are as follows:

- We study wDNNF circuits in terms of the knowledge compilation map.
- We show that NNF circuits are strictly more succinct than wDNNF circuits, even if $\mathcal{P} = \mathcal{NP}$.

- We present and evaluate our compiler for wDNNF circuits.

Basic Definitions

In this section, we introduce notation and preliminaries.

Propositional Logic

Let X be a set of (propositional) *variables*. A *literal* is a variable $x \in X$ (a *positive literal*) or its complement $\neg x$ (a *negative literal*). For a literal ℓ, $\neg\ell$ denotes the complementary literal of ℓ. We assume that the reader is familiar with the definition of a (propositional) *formula*. A *term* is a conjunction of literals. Sometimes, we treat a term as the set of its literals. A term is *consistent* if it does not contain a pair of complementary literals. A *clause* is a disjunction of literals. A *conjunctive normal form* (CNF) *formula* is a conjunction of clauses. A variable x is positive (resp. negative) in a CNF formula if $\neg x$ (resp. x) does not appear in that formula.

An *assignment* is a function $\omega : X \to \{0, 1\}$. We also treat an assignment as the set of literals satisfied by it. We denote 0 (resp. 1) by *False* (resp. *True*). A *clause is satisfied* by an assignment if at least one of its literals is satisfied. A *CNF formula is satisfied* by an assignment if all its clauses are satisfied. An assignment that satisfies a formula is called a *model* of that formula. A formula with at least one model is called *consistent* (*satisfiable*). A formula is *valid* if it is satisfied by every assignment. We say that a formula ψ *is entailed* by a formula φ, denoted by $\varphi \models \psi$, if every model of φ is also a model of ψ. Two formulae φ and ψ are *equivalent*, denoted by $\varphi \equiv \psi$, if they have the same set of models (that is, $\varphi \models \psi$ and $\psi \models \varphi$).

Let φ be a formula, and let τ be a consistent term. The *conditioning* of φ on τ, denoted by $\varphi|\tau$, is the formula obtained from φ by replacing every literal $\ell \in \tau$ by *True* and every literal ℓ such that $\neg\ell \in \tau$ by *False*.

Let φ be a formula on the variables X, and let Y be a subset of X. The *forgetting* of Y from φ, denoted by $\exists Y.\varphi$, is a formula ψ on the variables $X \setminus Y$ such that for every formula χ on the variables $X \setminus Y$, we have that $\varphi \models \chi \iff \psi \models \chi$.

(Decomposable) Negation Normal Form Circuits

A *negation normal form* (NNF) *circuit* is a rooted directed acyclic graph. It has two types of inner nodes: Conjunctions (AND) and disjunctions (OR). Furthermore, each leaf is a literal, *True*, or *False*. For a node α, we denote by $\mathrm{Vars}(\alpha)$ the set of variables that appear in the subcircuit rooted at α. An AND node α is *decomposable* (Darwiche 1999) if no variable is shared among its child nodes (that is, $\mathrm{Vars}(\mathrm{child}_i) \cap \mathrm{Vars}(\mathrm{child}_j) = \emptyset$ for any pair of different child nodes child_i and child_j of α). An NNF circuit in which all AND nodes are decomposable is called a *decomposable NNF* (DNNF) *circuit* (Darwiche 1999). A variable x is positive (resp. negative) in a subcircuit if $\neg x$ (resp. x) does not appear in that subcircuit.

The size of a circuit Δ, denoted by $|\Delta|$, is measured by the number of edges in that circuit. Sometimes, we handle a node α of a circuit as the subcircuit rooted at α.

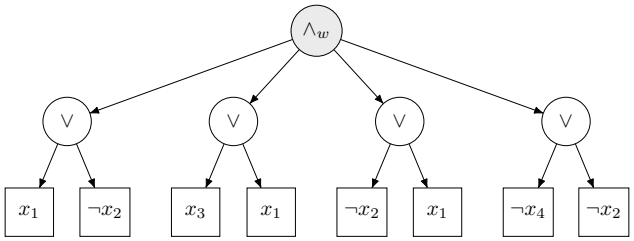

Figure 1: A wDNNF circuit. The root is weak decomposable because its four children share two variables: x_1, x_2, and all the occurrences of x_1 (resp. x_2) are positive (resp. negative).

Graph Theory

A *hypergraph* is an ordered pair (V, E) where V is a set of *vertices* and E is a set of *hyperedges* such that $E \subseteq \mathcal{P}(V)$. A (v_1, v_m)-*walk* is a sequence $(v_1, e_2, v_2, e_3, \ldots, e_m, v_m)$ such that $\forall i \in \{1, \ldots, m\} : v_i \in V, \forall j \in \{2, \ldots, m\} : e_j \in E$ and $\forall k \in \{2, \ldots, m\} : v_{k-1} \in e_k$ and $v_k \in e_k$. A hypergraph is said to be *connected* if there exists a (v_i, v_j)-walk for every two different vertices v_i and v_j; otherwise, we say that the hypergraph is *disconnected*. A *cut* of a connected hypergraph is a set of hyperedges such that if we remove those hyperedges from that hypergraph, the resulting hypergraph becomes disconnected.

The *dual hypergraph* $H(\varphi)$ of a CNF formula φ is a hypergraph where the vertices correspond to the clauses of φ and the hyperedges correspond to the variables of φ. For each variable, the corresponding hyperedge is a set of clauses containing that variable.

Weak DNNF Circuits

In this section, we recall the weak decomposability property and wDNNF circuits.

Definition 1. *(Akshay et al. 2018) An AND node α is* weak decomposable *if for every variable $x \in \mathrm{Vars}(\alpha)$ shared by at least two child nodes of α, we have that x is either positive or negative in the subcircuit rooted at α.*

Definition 2. *(Akshay et al. 2018) An NNF circuit in which all AND nodes are weak decomposable is called a* weak DNNF *(wDNNF)* circuit.

Figure 1 depicts a wDNNF circuit.

It is evident that DNNF circuits form a proper subset of wDNNF circuits (for example, $(x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_4)$ is a wDNNF circuit, but it is not a DNNF circuit).

The completeness of wDNNF circuits (that is, they can represent any Boolean function) directly follows from the fact that they include complete DNNF circuits (Darwiche 2001a).

Queries

In this section, we discuss wDNNF circuits with respect to the queries considered in the knowledge compilation map.

A query on a circuit can be viewed as a question we ask about the circuit without modifying the circuit. We say that a

circuit type satisfies a query if and only if there is a polynomial time algorithm for answering the query on the circuits of that type. We consider the following queries:

- *Consistency check* (CO): Is the circuit consistent?
- *Validity check* (VA): Is the circuit valid?
- *Clausal entailment check* (CE): Does the circuit entail a given clause?
- *Implicant check* (IM): Does a given term imply the circuit?
- *Model counting* (CT): How many models does the circuit have?
- *Model enumeration* (ME): Enumerate the circuit models with polynomial delay.
- *Sentential entailment check* (SE): Does one circuit entail another circuit?
- *Equivalence check* (EQ): Are two circuits equivalent?

Table 1 summarises the queries for wDNNF circuits.

Proposition 1. *The results for the queries in Table 1 hold.*

We can see that wDNNF circuits satisfy the same queries as DNNF circuits. VA, IM, CT, SE, and EQ are not satisfied by wDNNF circuits because they are also not satisfied by DNNF circuits (Darwiche and Marquis 2002), which form a proper subset of wDNNF circuits.

To check the consistency of a wDNNF circuit, we can use the predicate introduced for DNNF circuits (Darwiche 1999).

Definition 3. *(Darwiche 1999) For a node α, the $\mathrm{CO_p}(\alpha)$ predicate is defined as follows:*

(a) $\mathrm{CO_p}(\alpha = \mathit{True}) = \mathit{True}$.
(b) $\mathrm{CO_p}(\alpha = \mathit{False}) = \mathit{False}$.
(c) $\mathrm{CO_p}(\alpha) = \mathit{True}$ *if* α *is a literal.*
(d) $\mathrm{CO_p}(\alpha = \bigvee_i \alpha_i) = \mathit{True} \iff \mathrm{CO_p}(\alpha_i)$ *is True for at least one* i.
(e) $\mathrm{CO_p}(\alpha = \bigwedge_i \alpha_i) = \mathit{True} \iff \mathrm{CO_p}(\alpha_i)$ *is True for all* i.

It should be clear that $\mathrm{CO_p}(\alpha)$ can be evaluated in linear time in the size of the circuit rooted at α. Moreover, the $\mathrm{CO_p}(\alpha)$ predicate can trivially provide a model as a by-product. The following theorem shows that the $\mathrm{CO_p}(\alpha)$ predicate is sound and complete.

Theorem 1. *A wDNNF circuit with the root α is consistent $\iff$ $\mathrm{CO_p}(\alpha)$ is True.*

By Theorem 1, wDNNF circuits satisfy CO. In the next section, we show that this circuit type also allows conditioning in linear time. It follows that this circuit type satisfies CE and ME by the standard properties of queries and transformations (see Lemmata A.4 and A.3 in (Darwiche and Marquis 2002)).

Transformations

This part concerns the transformations presented in the knowledge compilation map.

Transformations, unlike queries, modify circuits. A transformation is a function from a circuit (resp. a set of circuits) into an appropriate circuit of the same type. We say that a *circuit type satisfies a transformation* if and only if there exists a polynomial time algorithm that carries out the transformation on the circuits of the given type. We consider the following transformations:

- *Conditioning* (CD): Given a circuit Δ and a consistent term τ, construct a circuit representing $\Delta|\tau$.
- *Forgetting* (FO): Given a circuit Δ and a set of variables Y, construct a circuit equivalent to $\exists Y.\Delta$.
- *Bounded conjunction* ($\wedge$BC): Given Δ_1 and Δ_2 of the same type, construct a circuit representing $\Delta_1 \wedge \Delta_2$.
- *Bounded disjunction* ($\vee$BC): Given Δ_1 and Δ_2 of the same type, construct a circuit representing $\Delta_1 \vee \Delta_2$.
- *Negation* ($\neg$C): Given a circuit Δ, construct a circuit equivalent to $\neg\Delta$.

Table 1 summarises the transformations for wDNNF circuits.

Proposition 2. *The results for the transformations in Table 1 hold.*

As we can notice, DNNF circuits satisfy the same transformations as wDNNF circuits.

The hardness of $\wedge$BC and $\neg$C for wDNNF circuits can be shown using the same arguments as in the proof of Proposition 5.1 in (Darwiche and Marquis 2002).

Forgetting a variable can be done in the same way as in DNNF circuits (see Darwiche 2001a). In particular, assume that α is the root of a wDNNF circuit Δ, and let Y be a set of variables. Then $\mathrm{FO_p}(\alpha, Y)$ denotes the wDNNF circuit that originates from Δ by replacing every literal on a variable from Y by *True*.

It should be evident that $\mathrm{FO_p}(\alpha, Y)$ preserves the type of the circuit and can be constructed in linear time in the size of Δ. The following theorem shows the correctness of this approach.

Theorem 2. *Let α be the root of a wDNNF circuit Δ, and let Y be a set of variables. Then $\mathrm{FO_p}(\alpha, Y)$ represents $\exists Y.\Delta$.*

Succinctness

In this section, we relate wDNNF circuits to NNF and DNNF circuits with respect to succinctness (Gogic et al. 1995).

Definition 4. *Let C_1 and C_2 be two circuit types. We say that C_1 is* at least as succinct as *C_2, denoted by $C_1 \leq C_2$, if and only if there exists a polynomial p such that for every circuit Δ_2 of type C_2, there exists an equivalent circuit Δ_1 of type C_1 for which $|\Delta_1| \leq p(|\Delta_2|)$. C_1 is said to be* strictly more succinct *than C_2, denoted by $C_1 < C_2$, if $C_1 \leq C_2$ and $C_2 \not\leq C_1$.*

It is common knowledge that wDNNF circuits are strictly more succinct than DNNF circuits (de Colnet and Mengel 2021).

Unless $\mathcal{P} = \mathcal{NP}$, NNF circuits are known to be strictly more succinct than wDNNF circuits (Akshay et al. 2019). The following theorem gives the unconditional relation.

Theorem 3. *NNF circuits are strictly more succinct than wDNNF circuits.*

Circuit type	CO	VA	CE	IM	CT	ME	SE	EQ	CD	FO	∧BC	∨BC	¬C
NNF circuits	○	○	○	○	○	○	○	○	✓	○	✓	✓	✓
wDNNF circuits	✓	○	✓	○	○	✓	○	○	✓	✓	○	✓	○
DNNF circuits	✓	○	✓	○	○	✓	○	○	✓	✓	○	✓	○

Table 1: ✓ denotes the fact that the circuit type satisfies the query/transformation. ○ means that the circuit type does not satisfy the query/transformation unless $\mathcal{P} = \mathcal{NP}$.

A Knowledge Compiler: Bella

This section presents our top-down compiler Bella[1] for converting CNF formulae into wDNNF circuits. Bella exploits the same techniques as the D4 compiler (Lagniez and Marquis 2017): Dynamic decomposition, component caching, conflict analysis, and non-chronological backtracking. Because of that, we delineate only the main distinctions between Bella and D4.

To make the experiments as comparable as possible, Bella uses:

- The same SAT solver as D4[2] (that is, a modified version of the SAT solver MINISAT 2.2 (Eén and Sörensson 2004) adapted for D4).
- The identical caching scheme i, which is defined in (Lagniez and Marquis 2020).
- The same external hypergraph partitioning software PaToH[3] (Çatalyürek and Aykanat 2011) for the following operating systems: Linux and macOS. Unlike D4, Bella also supports Windows, for which hMETIS[4] (Karypis et al. 1997) or KaHyPar[5] (Schlag et al. 2022) can be used.

Disjoint Component Analysis

The *disjoint component analysis* technique is typically used in top-down compilers. The main idea is to split the residual CNF formula (that is, the input CNF formula under the current partial assignment) during a search into multiple disjoint components that do not share variables and compile every component separately and combine the results. There are two approaches that differ when disjoint components are computed. In the *static decomposition* approach (used by C2D[6]), the disjoint components are computed before a search, whilst the disjoint components are computed during a search in the *dynamic decomposition* approach (used by DSHARP and D4). The latter approach has the advantage of resulting in more efficient decompositions. On the other hand, it is more time-consuming because decompositions are computed more often. We decided to use the dynamic decomposition approach for two reasons. It is better based on the experiments in (Lagniez and Marquis 2017), and intuitively, it seems less time-consuming for wDNNF circuits because of sharing positive and negative variables.

[1]https://github.com/Illner/BellaCompiler
[2]https://github.com/crillab/d4
[3]https://faculty.cc.gatech.edu/~umit/software.html
[4]http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview
[5]https://kahypar.org/
[6]http://reasoning.cs.ucla.edu/c2d/

Dynamic Decomposition in D4

We describe how dynamic decomposition is performed in D4. Let φ be a residual CNF formula which cannot be split into more components (that is, the clauses cannot be divided into at least two disjunctive sets that do not share variables). A set of variables as small as possible must be found such that if the variables are assigned (regardless of how they are assigned) in φ, the new residual CNF formula can be split into more components. To find such a set, hypergraph partitioning is used. First, the dual hypergraph of φ is constructed, and then PaToH is used to find a cut in that hypergraph. Since finding cuts in hypergraphs is time-consuming, D4 does not find a new cut every time, but only if the current cut is empty. Besides that, it uses other techniques to shrink the sizes of hypergraphs, such as simplification by considering literal equivalence (Lagniez and Marquis 2017, 2014) (we call that *the equivalence simplification method*).

Example 1. *Let* $\varphi = (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee x_5) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5)$ *be a residual CNF formula. First, we notice that* φ *cannot be split into multiple components. The dual hypergraph of* φ *is* $H(\varphi) = (\{c_1, c_2, c_3, c_4\}, \{\{c_1, c_2, c_4\}, \{c_1, c_3, c_4\}, \{c_2, c_3\}, \{c_1, c_3\}, \{c_2, c_4\}\})$ *where* c_i *represents the* i*-th clause and the* i*-th hyperedge corresponds to the variable* x_i*. The minimum cut of that hypergraph is* $\{x_1, x_2, x_3\}$.

Dynamic Decomposition in Bella

Foremost, we must realise that disjoint components can share variables as long as they are positive or negative.

Example 2. *Let* $\varphi = (x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_5 \vee \neg x_6) \wedge (\neg x_2 \vee \neg x_5 \vee x_6)$ *be a CNF formula.* φ *has two components:* $(x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_4)$ *and* $(x_1 \vee x_5 \vee \neg x_6) \wedge (\neg x_2 \vee \neg x_5 \vee x_6)$*, even though the variables* x_1 *and* x_2 *are shared since* x_1 *(resp.* x_2*) is positive (resp. negative) in* φ.

It should be clear that positive and negative variables can be omitted in dual hypergraphs. The consequence is that dual hypergraphs are smaller, making computing cuts faster.

Example 3. *Let* φ *be a residual CNF formula in Example 1. The variable* x_1 *(resp.* x_2*) is positive (resp. negative) in* φ*, meaning those variables will not occur in the dual hypergraph. The dual hypergraph of* φ *is* $H(\varphi) = (\{c_1, c_2, c_3, c_4\}, \{\{c_2, c_3\}, \{c_1, c_3\}, \{c_2, c_4\}\})$ *where* c_i *represents the* i*-th clause and the* i*-th hyperedge corresponds to* x_{2+i}*. We notice that this hypergraph is smaller than the one in Example 1. Moreover, the minimum cut is smaller as well; in this case, the cut contains only* x_3.

The Problem of Exactly-One Constraints

Exactly-one constraints (that is, exactly one of m variables must be *True*) are crucial for many real-world problems. We outline why such constraints are an Achilles heel of sharing variables. First, we notice that the Boolean functions representing exactly-one constraints are not monotone in any variable (for the definition, see, for example, Crama and Hammer 2011); therefore, no variable can be shared. Let $\mathcal{B}$ be the Boolean function that represents an exactly-one constraint on m variables. If a variable is assigned to *True*, all the remaining variables are trivially derived to *False*. Otherwise, if the variable is assigned to *False*, the simplified Boolean function on the remaining variables represents the exactly-one constraint on $(m-1)$ variables. Regardless of the encoding used for such constraints, no variable mentioned in such constraints will ever be shared during a search.

The Pseudocode of Bella

Algorithm 1 shows the pseudocode of Bella. An input CNF formula φ is compiled by calling $\mathrm{Bella}(\varphi, \emptyset)$.

The simplifications of formulae in the pseudocode are supposed to help the reader better understand the code. In reality, a more efficient approach is used.

At line 1, we check if the residual CNF formula φ is satisfiable. If not, φ can be represented by a *False* leaf returned at line 2. Otherwise, we continue to compile φ. At line 4, we get all the implied literals $\ell_1, \ldots, \ell_m$ derived from φ using unit propagation. And then, at line 5, an AND node containing the implied literals (that is, $\ell_1 \wedge \cdots \wedge \ell_m$) is created. At line 6, we remove the implied literals from φ (that is, $\varphi|(\ell_1, \ldots, \ell_m)$). At lines 7 - 9, we check if the simplified formula φ_i has any variable. If not, it is sufficient to return the AND node representing the implied literals (that is, $impliedLitsNode$). Otherwise, we continue to compile φ_i. Now comes the first difference between Bella and D4. At line 10, we compute the connected components of φ_i, and in addition, we get the set of pure clauses (a pure clause of φ_i is a clause that contains only variables that are positive or negative in φ_i). Now, we can trivially compile the pure clauses and remove them from φ_i, which is done at lines 11 and 12. Because all the clauses of φ_i could be pure, we again check if the simplified formula φ_{ip} has at least one variable. If not, a conjunction of the AND nodes representing the implied literals and the pure clauses (that is, $impliedLitsNode \wedge pureClsNode$) is returned. Otherwise, we continue to compile φ_{ip}. For each component, we do the following. First, we remove from φ_{ip} all the clauses that do not contain variables from c (line 19). At lines 20 - 24, we use component caching to check if φ_c has not already been compiled. If so, we add the corresponding cached node to the list $nodes$ and continue to compile another component. Otherwise, the variables of cut are restricted to those in c at line 25. A new cut is computed (as described above using the equivalence simplification method) if cut_r is empty and the current component c has more than five variables. The second condition is due to efficiency. At line 29, a decision variable is selected from cut_r or c if cut_r is empty. The VSADS heuristic (Sang, Beame, and Kautz 2005) is used. At lines 30 and 31, two recursive calls are made for the cases where the decision variable is assigned. And the results are used to create a decision node at line 32. The compiled component is added to the cache at line 34. At line 36, we return an AND node containing the nodes representing the compiled components and the AND nodes representing the implied literals and the pure clauses.

Algorithm 1: Bella(φ, cut)

Input: a current residual CNF formula φ
Input: a current cut
Output: the root of a wDNNF circuit representing φ

```
1: if not isSatisfiable(φ) then
2:     return createFalseLeaf()
3: end if
4: impliedLits ← getImpliedLiterals(φ)
5: impliedLitsNode ← createAndNode(impliedLits)
6: φ_i ← simplify(φ, impliedLits)
7: if getVars(φ_i) = ∅ then
8:     return impliedLitsNode
9: end if
10: (comps, pureCls) ← computeComponents(φ_i)
11: pureClsNode ← compilePureClauses(pureCls)
12: φ_ip ← removePureClauses(φ_i, pureCls)
13: nodes ← [impliedLitsNode, pureClsNode]
14: if getVars(φ_ip) = ∅ then
15:     return createAndNode(nodes)
16: end if
17:
18: for all component c in comps do
19:     φ_c ← simplify(φ_ip, c)
20:     cacheKey ← createCacheKey(φ_c)
21:     if cacheKeyExists(cacheKey) then
22:         nodes.append(getCachedNode(cacheKey))
23:         continue
24:     end if
25:     cut_r ← restrict(cut, c)
26:     if (cut_r = ∅) and (|c| > 5) then
27:         cut_r ← computeNewCut(φ_c)
28:     end if
29:     dec ← getDecisionVariable(c, cut_r)
30:     pNode ← Bella(φ_c|dec, cut_r)
31:     nNode ← Bella(φ_c|¬dec, cut_r)
32:     node ← createDecisionNode(dec, pNode, nNode)
33:     nodes.append(node)
34:     addToCache(cacheKey, node)
35: end for
36: return createAndNode(nodes)
```

Nondeterministic Disjunctions

During a compilation, two types of disjunctions can be created. First, when a decision is made, one deterministic disjunction (for the definition, see, for example, Darwiche 2001b) is created. Second, if a pure clause is compiled, one nondeterministic disjunction is created. Therefore, the compiled wDNNF circuit does not have to be purely determinis-

tic. For example, a monotone CNF formula is trivially compiled into a wDNNF circuit that does not contain a deterministic disjunction, since every clause is pure and no decision is made during the compilation.

Experimental Results

For the experiments, we considered the same data set[7] which is used in (Lagniez and Marquis 2017). The data set, which contains 701 instances, consists of the following instance types: Bounded model checking (BMC), Bayesian networks (BN), circuit, configuration, handmade, planning, quantitative information flow analysis - security (QIF), and random.

Since D4 has a fixed random generator seed for hypergraph partitioning, we randomised that seed. Moreover, we noticed that the first call of SAT solver is not included in the compilation time mentioned in the output report. We rectified[8] that. More details are given in the readme.md file.

All the experiments were conducted on a Linux (Debian 11) machine using an AMD EPYC™ 7543 2.8GHz processor and 512 GiB of RAM. The time-out (resp. memory-out) was set to two hours (resp. 16 GB). Due to randomness, every instance was compiled five times, and the results presented in this section are averages. The experiments[9] were performed for the following compilers (circuit types): Bella (wDNNF circuits), D4 (decision-DNNF circuits), and C2D (decision-DNNF circuits). Due to the space limitation, only the most important results are presented here.

Figures 2 and 3 show the results for wDNNF circuits (Bella) and decision-DNNF circuits (D4) regarding compilation times (in seconds) and circuit sizes. Logarithmic scales are used for both coordinates. First, the configuration instances are considerably better for wDNNF circuits. We explicitly mention that there are applications for configuration instances (Voronov, Åkesson, and Ekstedt 2011; Zengler and Küchlin 2013) for which wDNNF circuits are sufficient. Second, four configuration instances have noticeably better sizes for decision-DNNF circuits. Unlike D4, Bella guarantees that every variable is mentioned (that is, has at least one occurrence of positive or negative literal) in the compiled circuit. This is the reason why those four instances have better sizes for the decision-DNNF circuits generated by D4. Third, two BN instances are significantly worse for wDNNF circuits. The reason is that for those instances, at some point, it is better to select a decision variable which is positive or negative in the current residual CNF formula. Unfortunately, these variables are not included in cuts and, therefore, cannot be selected as decision variables. Last, most instances with similar compilation times and circuit sizes contain exactly-one constraints. This makes sense since the variables mentioned in such constraints cannot be shared, resulting in lower sharing, typically zero sharing.

Table 2 shows the compilation times (in seconds), the circuit sizes, and the number of successful compilations (maximum is five) of the four most challenging configuration instances. Indisputably, wDNNF circuits are orders of magnitude better.

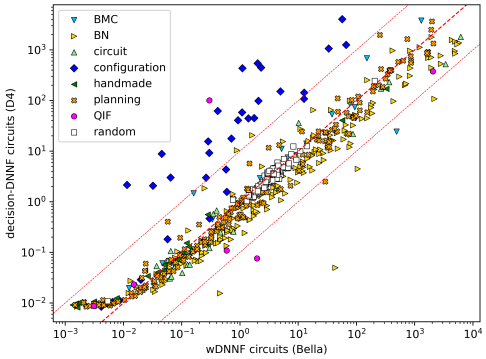

Figure 2: decision-DNNFs (D4) vs wDNNFs (Bella): time

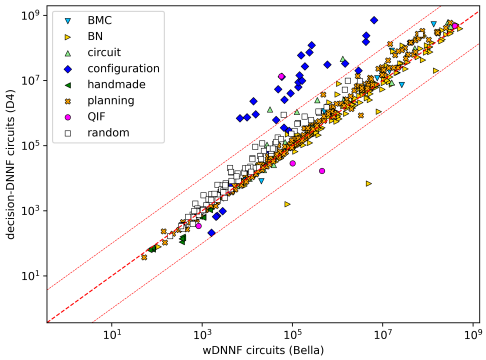

Figure 3: decision-DNNFs (D4) vs wDNNFs (Bella): size

Table 3 shows the number of successfully compiled instances for each instance type and compiler. We notice that Bella compiled every configuration instance.

Additionally, we note that, unlike D4, Bella uses *the unique-node technique* (see, for example, Brace, Rudell, and Bryant 1991), significantly reducing the number of nodes.

Conclusion

First, we have integrated wDNNF circuits into the knowledge compilation map. Second, we have proved the unconditional strict succinctness relation between NNF and wDNNF circuits. Third, our compiler for wDNNF circuits has been presented and evaluated.

Proofs

Succinctness

We use the notion of existential literal quantification (also called literal forgetting) introduced in (Lang, Liberatore, and Marquis 2003) and further studied in (Darwiche and Marquis 2022). If Δ is an NNF circuit, by *forgetting a literal* ℓ *from* Δ, we mean the operation $\exists \ell.\Delta \stackrel{def}{=} (\Delta|\ell) \vee (\neg\ell \wedge (\Delta|\neg\ell))$. Darwiche and Marquis (2022) showed that literal forgetting can be performed efficiently on decision-DNNF circuits by simply replacing every occurrence of ℓ by *True*. The same approach can be used for wDNNF circuits.

[7] http://www.cril.univ-artois.fr/kc/benchmarks.html
[8] https://github.com/Illner/BellaModifiedD4-AAAI24
[9] https://github.com/Illner/BellaExperimentalResults-AAAI24

Configuration instance	wDNNF circuits			decision-DNNF circuits						
	Bella			D4			SharpSAT-TD		C2D	
	time (s)	size	#	time (s)	size	#	time (s)	size	size	#
C202_FS	66.3	4 245 485	5	1 256.2	154 862 147	4	176.1	341 407 473	169 398 649	4
C202_FW	70.7	7 344 961	5	—	—	0	151.4	253 123 753	195 320 974	1
C210_FS	33.2	4 300 692	5	1 065.3	240 087 456	5	239.6	425 732 421	71 522 262	5
C210_FW	56.7	6 429 027	5	4 048.2	715 624 258	2	339.9	609 298 328	118 653 033	5

Table 2: The compilation times (in seconds), the circuit sizes, and the number of successful compilations (maximum is five) of the four most challenging configuration instances. Since SharpSAT-TD is deterministic, every instance was compiled only once. Due to overflowing, the compilation times for C2D are not provided.

Circuit type (compiler)	BMC (18)	BN (192)	circuit (39)	configuration (35)	handmade (58)	planning (248)	QIF (7)	random (104)
wDNNF circuits (Bella)	16	164	33	35	35	201	6	103
decision-DNNF circuits (D4)	16	175	32	34	35	200	6	102
decision-DNNF circuits (C2D)	8	125	32	35	31	200	5	102

Table 3: The number of successfully compiled instances for each instance type and compiler.

Lemma 1. *If Δ is a wDNNF circuit and Δ' originates from Δ by replacing every occurrence of ℓ by True, then Δ' is a wDNNF circuit equivalent to $\exists \ell.\Delta$.*

Proof. Δ' is clearly a wDNNF circuit. It remains to prove the equivalence with $\exists \ell.\Delta$. Let us denote the variable of ℓ by x. We proceed by induction on the structure of Δ'. Leaves: The proposition holds by Proposition 19 in (Darwiche and Marquis 2022). Disjunctions: Let $\alpha = \alpha_1 \vee \alpha_2$. Then we have $\exists \ell.\alpha = (\exists \ell.\alpha_1) \vee (\exists \ell.\alpha_2)$ by Proposition 20(a) in (Darwiche and Marquis 2022). Conjunctions: Let $\alpha = \alpha_1 \wedge \alpha_2$ be weak decomposable. If x is not shared between α_1 and α_2, then we have $\exists \ell.\alpha = (\exists \ell.\alpha_1) \wedge (\exists \ell.\alpha_2)$ by Proposition 20(c) in (Darwiche and Marquis 2022). Otherwise, let x be shared between α_1 and α_2. By weak decomposability, we have that either ℓ or $\neg\ell$ is not in the subcircuit rooted at α. If ℓ is not in α, then replacing ℓ by *True* does not change that subcircuit. Otherwise, assume that ℓ is in α. Since $\neg\ell$ does not occur in that subcircuit, we have that $(\alpha|\neg\ell)$ implies $(\alpha|\ell)$. Hence, $\exists \ell.\alpha = (\alpha|\ell) \vee (\neg\ell \wedge (\alpha|\neg\ell)) \equiv (\alpha|\ell)$. The proposition follows since $(\alpha|\ell)$ originates from α by setting ℓ to *True*. □

Let Δ be an NNF circuit, and let ℓ be a literal. We say that Δ *is monotone in* ℓ if $(\Delta|\neg\ell)$ implies $(\Delta|\ell)$. In particular, if an NNF circuit Δ is equivalent to a positive (resp. negative) CNF formula, then Δ is monotone in every positive (resp. negative) literal. Let us make the following observation.

Lemma 2. *Let Δ be an NNF circuit, and let ℓ be a literal. If Δ is monotone in ℓ, then Δ is equivalent to $\exists \neg\ell.\Delta$.*

Proof. We have that $\exists \neg\ell.\Delta = (\Delta|\neg\ell) \vee (\ell \wedge (\Delta|\ell))$. We show that $(\Delta|\neg\ell) \vee (\ell \wedge (\Delta|\ell)) \equiv \Delta$. If an assignment satisfies Δ, then it clearly satisfies the left-hand side. Assume now that the left-hand side is satisfied by an assignment ω. If $\neg\ell \in \omega$, then $(\Delta|\neg\ell)$ is satisfied by ω, and thus Δ is satisfied by ω. Otherwise, if $\ell \in \omega$, then $(\Delta|\ell)$ is satisfied since $(\Delta|\neg\ell)$ implies $(\Delta|\ell)$. Therefore, Δ is satisfied by ω. □

Proof of Theorem 3. Tardos (1988) described a class $\mathbb{C}_{\mathrm{T}}^{+}$ of positive Boolean functions that can be represented by non-monotone circuits (that is, NNF circuits) of size polynomial in the number of input bits n, whilst monotone circuits (that is, NNF circuits containing no negative literals) representing those functions have size exponential in n. Let Δ be a wDNNF circuit representing a positive function from $\mathbb{C}_{\mathrm{T}}^{+}$, and let Δ' originate from Δ by forgetting all negative literals. By Lemma 2, it holds that $\Delta \equiv \Delta'$. By Lemma 1, we get that $|\Delta| \geq |\Delta'|$. Since Δ' is a monotone circuit, its size must be exponential in n by (Tardos 1988). □

Queries and Transformations

Proof of Theorem 1. We proceed by induction on the structure of the circuit. Cases (a) to (d) follow directly. Case (e): Let $\alpha = \bigwedge_i \alpha_i$ be weak decomposable, and let X be the set of variables shared by at least two child nodes of α. Without loss of generality, let us assume that all variables from X are positive in the subcircuit rooted at α. It is clear that if α is satisfiable, then $\forall i : \alpha_i$ is satisfiable. Let ω_i be a model of α_i. If every variable from X is set to *True* in ω_i, then the changed assignment is also a model of α_i. It follows that if $\forall i : \alpha_i$ is satisfiable, then α is satisfiable. □

Proof of Proposition 1. The proof follows from Theorem 1 and the discussion below the proposition statement. □

Proof of Theorem 2. By Proposition 16 in (Darwiche and Marquis 2022), we have that forgetting a variable x is equivalent to forgetting both literals x and $\neg x$. Therefore, the correctness of the procedure follows from Lemma 1. □

Proof of Proposition 2. Most of the proof follows from Theorem 2 and the discussion below the proposition statement. CD: Conditioning means replacing some of the literals by *True* or *False*, but this cannot violate weak decomposability. $\vee$BC: Weak decomposability restricts only AND nodes. □

Acknowledgments

The authors would like to thank Petr Savický for the fruitful discussion about this paper.

This research was partially supported by SVV project number 260 698 and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

References

Akshay, S.; Arora, J.; Chakraborty, S.; Krishna, S.; Raghunathan, D.; and Shah, S. 2019. Knowledge compilation for boolean functional synthesis. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, 161–169. IEEE.

Akshay, S.; Chakraborty, S.; Goel, S.; Kulal, S.; and Shah, S. 2018. What's Hard About Boolean Functional Synthesis? In Chockler, H.; and Weissenbacher, G., eds., *Computer Aided Verification*, 251–269. Cham: Springer International Publishing. ISBN 978-3-319-96145-3.

Bonet, B.; and Geffner, H. 2006. Heuristics for Planning with Penalties and Rewards Using Compiled Knowledge. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'06, 452–462. AAAI Press. ISBN 9781577352716.

Brace, K. S.; Rudell, R. L.; and Bryant, R. E. 1991. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, 40–45. New York, NY, USA: Association for Computing Machinery. ISBN 0897913639.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8): 677–691.

Çatalyürek, Ü.; and Aykanat, C. 2011. *PaToH (Partitioning Tool for Hypergraphs)*, 1479–1487. Boston, MA: Springer US. ISBN 978-0-387-09766-4.

Crama, Y.; and Hammer, P. L. 2011. Boolean Functions - Theory, Algorithms, and Applications. In *Encyclopedia of mathematics and its applications*.

Darwiche, A. 1999. Compiling Knowledge into Decomposable Negation Normal Form. In *Proceedings of the 16th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'99, 284–289. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Darwiche, A. 2001a. Decomposable negation normal form. *Journal of the ACM (JACM)*, 48(4): 608–647.

Darwiche, A. 2001b. 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.

Darwiche, A. 2004. New Advances in Compiling CNF to Decomposable Negation Normal Form. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, 318–322. Amsterdam, The Netherlands: IOS Press. ISBN 978-1-58603-452-8.

Darwiche, A. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, 819–826. AAAI Press. ISBN 978-1-57735-514-4.

Darwiche, A.; and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17: 229–264.

Darwiche, A.; and Marquis, P. 2022. On Quantifying Literals in Boolean Logic and its Applications to Explainable AI. *Journal of Artificial Intelligence Research*, 72: 285–328.

de Colnet, A.; and Mengel, S. 2021. A Compilation of Succinctness Results for Arithmetic Circuits. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, 205–215.

Eén, N.; and Sörensson, N. 2004. An Extensible SAT-solver. In Giunchiglia, E.; and Tacchella, A., eds., *Theory and Applications of Satisfiability Testing*, 502–518. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-24605-3.

Gogic, G.; Kautz, H.; Papadimitriou, C.; and Selman, B. 1995. The Comparative Linguistics of Knowledge Representation. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, 862–869. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1558603638.

Huang, J.; and Darwiche, A. 2005. On Compiling System Models for Faster and More Scalable Diagnosis. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, AAAI'05, 300–306. AAAI Press. ISBN 157735236x.

Huang, J.; and Darwiche, A. 2007. The Language of Search. *J. Artif. Int. Res.*, 29(1): 191–219.

Karypis, G.; Aggarwal, R.; Kumar, V.; and Shekhar, S. 1997. Multilevel Hypergraph Partitioning: Application in VLSI Domain. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, 526–529. New York, NY, USA: Association for Computing Machinery. ISBN 0897919203.

Kiesel, R.; and Eiter, T. 2023. Knowledge Compilation and More with SharpSAT-TD. In *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*, 406–416.

Lagniez, J.-M.; and Marquis, P. 2014. Preprocessing for Propositional Model Counting. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28(1).

Lagniez, J.-M.; and Marquis, P. 2017. An Improved Decision-DNNF Compiler. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 667–673.

Lagniez, J.-M.; and Marquis, P. 2020. Enhanced Caching for #SAT Solving. Working paper or preprint.

Lang, J.; Liberatore, P.; and Marquis, P. 2003. Propositional independence: formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18(1): 391–443.

Marquis, P. 2015. Compile! In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, 4112–4118. AAAI Press. ISBN 0262511290.

Muise, C.; McIlraith, S.; Beck, J. C.; and Hsu, E. 2010. Fast d-DNNF Compilation with sharpSAT. In *Workshops at the twenty-fourth AAAI conference on artificial intelligence*.

Oztok, U.; and Darwiche, A. 2017. On Compiling DNNFs without Determinism. *arXiv preprint arXiv:1709.07092*.

Peharz, R.; Tschiatschek, S.; Pernkopf, F.; and Domingos, P. 2015. On Theoretical Properties of Sum-Product Networks. In Lebanon, G.; and Vishwanathan, S. V. N., eds., *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, 744–752. San Diego, California, USA: PMLR.

Pipatsrisawat, K.; and Darwiche, A. 2007. Clone: Solving Weighted Max-SAT in a Reduced Search Space. In Orgun, M. A.; and Thornton, J., eds., *AI 2007: Advances in Artificial Intelligence*, 223–233. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-76928-6.

Pipatsrisawat, K.; and Darwiche, A. 2008. New Compilation Languages Based on Structured Decomposability. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 1*, AAAI'08, 517–522. AAAI Press. ISBN 9781577353683.

Ramírez, M.; and Geffner, H. 2007. Structural Relaxations by Variable Renaming and Their Compilation for Solving MinCostSAT. In Bessière, C., ed., *Principles and Practice of Constraint Programming – CP 2007*, 605–619. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-74970-7.

Sang, T.; Beame, P.; and Kautz, H. 2005. Heuristics for Fast Exact Model Counting. In Bacchus, F.; and Walsh, T., eds., *Theory and Applications of Satisfiability Testing*, 226–240. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-31679-4.

Schlag, S.; Heuer, T.; Gottesbüren, L.; Akhremtsev, Y.; Schulz, C.; and Sanders, P. 2022. High-Quality Hypergraph Partitioning. *ACM J. Exp. Algorithmics*.

Siddiqi, S. 2011. Computing Minimum-Cardinality Diagnoses by Model Relaxation. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Two*, IJCAI'11, 1087–1092. AAAI Press. ISBN 9781577355144.

Siddiqi, S.; and Huang, J. 2007. Hierarchical Diagnosis of Multiple Faults. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, 581–586. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Tardos, É. 1988. The gap between monotone and non-monotone circuit complexity is exponential. *Combinatorica*, 8(1): 141–142.

Voronov, A.; Åkesson, K.; and Ekstedt, F. 2011. Enumeration of valid partial configurations. In *Proceedings of Workshop on Configuration, IJCAI 2011*, volume 755, 25–31.

Zengler, C.; and Küchlin, W. 2013. Boolean Quantifier Elimination for Automotive Configuration – A Case Study. In Pecheur, C.; and Dierkes, M., eds., *Formal Methods for Industrial Critical Systems*, 48–62. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-41010-9.