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 formulæ 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 \( \alpha \) if either only \( x \) or only \( \neg x \) appears in the subcircuit rooted at \( \alpha \). 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 \( \mathcal{P} = \mathcal{N}/\mathcal{P} \).
• 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 $\ell$, $\neg \ell$ denotes the complementary literal of $\ell$. 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 complementary literals. A formula 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 $\phi$ if it is satisfied by every assignment. We say that a formula $\phi$ is consistent (resp. inconsistent) if $\phi$ (resp. its complement $\neg \phi$) does not appear in that formula.

An assignment is a function $\omega : X \rightarrow \{0, 1\}$. 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 $\psi$ is entailed by a formula $\varphi$, denoted by $\varphi \models \psi$, if every model of $\varphi$ is also a model of $\psi$. Two formulae $\varphi$ and $\psi$ 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 $\varphi$ be a formula, and let $\tau$ be a consistent term. The **conditioning** of $\varphi$ on $\tau$, denoted by $\varphi|\tau$, is the formula obtained from $\varphi$ by replacing every literal $\ell \in \tau$ by True and every literal $\ell$ such that $\neg \ell \in \tau$ by False.

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

**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 $\alpha$, we denote by $\text{Vars}(\alpha)$ the set of variables that appear in the subcircuit rooted at $\alpha$. An AND node $\alpha$ is decomposable (Darwiche 1999) if no variable is shared among its child nodes (that is, $\text{Vars}($child$_1) \cap \text{Vars}($child$_2) = \emptyset$ for any pair of different child nodes child$_1$ and child$_2$ of $\alpha$). 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 $\Delta$, denoted by $|\Delta|$, is measured by the number of edges in that circuit. Sometimes, we handle a node $\alpha$ of a circuit as the subcircuit rooted at $\alpha$.

**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 $E \subseteq P(V)$. A $(v_1, v_m)$-walk is a sequence $(v_1, e_2, v_2, e_3, \ldots, e_m, v_m)$ such that $v_i \in \{1, \ldots, m\}$ and $e_j \in E$ for all $i, j$. 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.

**Weak DNNF Circuits**

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

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

**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, $(x_1 \lor x_2) \land (x_2 \lor \neg x_3) \land (\neg x_3 \lor x_4)$ 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 \( \alpha \), the \( \text{CO}_p(\alpha) \) predicate is defined as follows:

(a) \( \text{CO}_p(\alpha = \text{True}) = \text{True} \).
(b) \( \text{CO}_p(\alpha = \text{False}) = \text{False} \).
(c) \( \text{CO}_p(\alpha) = \text{True} \) if \( \alpha \) is a literal.
(d) \( \text{CO}_p(\alpha = \bigvee_i \alpha_i) = \text{True} \iff \text{CO}_p(\alpha_i) \) is True for at least one i.
(e) \( \text{CO}_p(\alpha = \bigwedge_i \alpha_i) = \text{True} \iff \text{CO}_p(\alpha_i) \) is True for all i.

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

**Theorem 1. A wDNNF circuit with the root \( \alpha \) is consistent \iff \text{CO}_p(\alpha) = \text{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 \( \Delta \) and a consistent term \( \tau \), construct a circuit representing \( \Delta|\tau \).
- **Forgetting (FO):** Given a circuit \( \Delta \) and a set of variables \( Y \), construct a circuit equivalent to \( \exists Y.\Delta \).
- **Bounded conjunction (ABC):** Given \( \Delta_1 \) and \( \Delta_2 \) of the same type, construct a circuit representing \( \Delta_1 \land \Delta_2 \).
- **Bounded disjunction (\lor BC):** Given \( \Delta_1 \) and \( \Delta_2 \) of the same type, construct a circuit representing \( \Delta_1 \lor \Delta_2 \).
- **Negation (\neg C):** Given a circuit \( \Delta \), 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 \( \land\land BC \) and \( \neg C \) for wDNNF circuits can be shown using the same arguments as in the proof of Proposition 3.1 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 \( \alpha \) is the root of a wDNNF circuit \( \Delta \), and let \( Y \) be a set of variables. Then \( \text{FO}_p(\alpha, Y) \) denotes the wDNNF circuit that originates from \( \Delta \) by replacing every literal on a variable from \( Y \) by True.

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

**Theorem 2. Let \( \alpha \) be the root of a wDNNF circuit \( \Delta \), and let \( Y \) be a set of variables. Then \( \text{FO}_p(\alpha, Y) \) 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 \( \Delta_2 \) of type \( C_2 \), there exists an equivalent circuit \( \Delta_1 \) 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 \( C_2 \not< C_1 \).

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

Unless \( P = \overline{P} \), 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.**
The Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI-24)

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.

Dynamic Decomposition in D4

We describe how dynamic decomposition is performed in D4. Let \(\varphi\) 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 \(\varphi\), 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 \(\varphi\) 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 \(\varphi = (x_1 \lor \neg x_2 \lor x_4) \land (x_1 \lor x_3 \lor x_5) \land (\neg x_2 \lor \neg x_3 \lor \neg x_4) \land (x_1 \lor \neg x_2 \lor \neg x_5)\) be a residual CNF formula. First, we notice that \(\varphi\) cannot be split into multiple components. The dual hypergraph of \(\varphi\) 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 \(\varphi = (x_1 \lor x_3 \lor \neg x_4) \land (\neg x_2 \lor \neg x_3 \lor x_4) \land (x_1 \lor x_5 \lor \neg x_6) \land (\neg x_2 \lor \neg x_5 \lor x_6)\) be a CNF formula. \(\varphi\) has two components: \((x_1 \lor x_3 \lor \neg x_4) \land (\neg x_2 \lor \neg x_3 \lor x_4)\) and \((x_1 \lor x_5 \lor \neg x_6) \land (\neg x_2 \lor \neg x_5 \lor x_6)\), even though the variables \(x_1\) and \(x_2\) are shared since \(x_1\) (resp. \(x_2\)) is positive (resp. negative) in \(\varphi\).

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 \(\varphi\) be a residual CNF formula in Example 1. The variable \(x_1\) (resp. \(x_2\)) is positive (resp. negative) in \(\varphi\), meaning those variables will not occur in the dual hypergraph. The dual hypergraph of \(\varphi\) 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 \(x_3\).

---

\(^1\)https://github.com/Illner/BellaCompiler

\(^2\)https://github.com/crilab/d4

\(^3\)https://faculty.cc.gatech.edu/\~{}unit/software.html

\(^4\)http://gloes.dtic.umn.edu/gkhome/metis/hmetis/overview

\(^5\)https://kahypar.org/

\(^6\)http://reasoning.cs.ucla.edu/c2d/

---

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 \(P = NP\).
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 \( B \) 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 \( \phi \) is compiled by \( \text{Bella}(\phi, \emptyset) \).

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 \( \phi \) is satisfiable. If not, \( \phi \) can be represented by a \( False \) leaf returned at line 2. Otherwise, we continue to compile \( \phi \). At line 4, we get all the implied literals \( \ell_1, \ldots, \ell_m \) derived from \( \phi \) using unit propagation. And then, at line 5, an AND node containing the implied literals (that is, \( \ell_1 \land \cdots \land \ell_m \)) is created. At line 6, we remove the implied literals from \( \phi \) (that is, \( \phi[\ell_1, \ldots, \ell_m] \)). At lines 7 - 9, we check if the simplified formula \( \phi_i \) has any variable. If not, it is sufficient to return the AND node representing the implied literals (that is, \( \text{impliedLitsNode} \)). Otherwise, we continue to compile \( \phi_i \). Now comes the first difference between Bella and D4. At line 10, we compute the connected components of \( \phi_i \), and in addition, we get the set of pure clauses (a pure clause of \( \phi_i \) is a clause that contains only variables that are positive or negative in \( \phi_i \)). Now, we can trivially compile the pure clauses and remove them from \( \phi_i \), which is done at lines 11 and 12. Because all the clauses of \( \phi_i \) could be pure, we again check if the simplified formula \( \phi_{ip} \) has at least one variable. If not, a conjunction of the AN mechanisms representing the implied literals and the pure clauses (that is, \( \text{impliedLitsNode} \land \text{pureClsNode} \)) is returned. Otherwise, we continue to compile \( \phi_{ip} \). For each component, we do the following. First, we remove from \( \phi_{ip} \) all the clauses that do not contain variables from \( c \) (line 19). At lines 20 - 24, we use component caching to check if \( \phi_c \) has not already been compiled. If so, we add the corresponding cached node to the list \( \text{nodes} \) and continue to compile another component. Otherwise, the variables of \( \text{cut} \) are restricted to those in \( c \) at line 25. A new cut is computed (as described above using the equivalence simplification method) if \( \text{cut}_r \) 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 \( \text{cut}_r \) or \( c \) if \( \text{cut}_r \) 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.

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 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 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 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 (V oronov, 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 in-

---

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

**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 \( \Delta \) is an NNF circuit, by forgetting a literal \( \ell \) from \( \Delta \), we mean the operation \( \exists \ell \). \( \Delta \) \( \overset{df}{=} \) \( (\Delta | \ell) \lor (\neg \ell \land (\Delta | \neg \ell)) \). Darwiche and Marquis (2022) showed that literal forgetting can be performed efficiently on decision-DNNF circuits by simply replacing every occurrence of \( \ell \) by True. The same approach can be used for wDNNF circuits.
Lemma 1. If $\Delta$ is a wDNF circuit and $\Delta'$ originates from $\Delta$ by replacing every occurrence of $\ell$ by $\text{True}$, then $\Delta'$ is a wDNF circuit equivalent to $\exists \ell. \Delta$.

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

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

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

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

Proof of Theorem 3. Tardos (1988) described a class $C^+_n$ 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 $\Delta$ be a wDNF circuit representing a positive function from $C^+_n$, and let $\Delta'$ originate from $\Delta$ by forgetting all negative literals. By Lemma 2, it holds that $\Delta \equiv \Delta'$. By Lemma 1, we get that $|\Delta| \geq |\Delta'|$. Since $\Delta'$ 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 $\alpha$. Without loss of generality, let us assume that all variables from $x$ are positive in the subcircuit rooted at $\alpha$. It is clear that if $\alpha$ is satisfiable, then $\forall i : \alpha_i$ is satisfiable. Let $\omega_i$ be a model of $\alpha_i$, if every variable from $x$ is set to $\text{True}$ in $\omega_i$, then the changed assignment is also a model of $\alpha_i$. It follows that if $\forall i : \alpha_i$ is satisfiable, then $\alpha$ is satisfiable.

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

Proof of Proposition 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 $\text{True}$ or $\text{False}$, but this cannot violate weak decomposability. VBC: 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


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.


