Circuit-Aware d-DNNF Compilation

Vincent Derkinderen[1] , **Jean-Marie Lagniez**[2]
[1]KU Leuven, B-3000 Leuven, Belgium
[2]CRIL, U. Artois & CNRS, F-62300 Lens, France
vincent.derkinderen@kuleuven.be, lagniez@cril.fr

Abstract

Boolean circuits in d-DNNF (determinstic Decomposable Negation Normal Form) enable tractable probabilistic inference, motivating research into compilers that transform arbitrary Boolean circuit into this form. However, d-DNNF compilers commonly require the input to be in conjunctive normal form (CNF), which means that a user must first convert their Boolean circuit into CNF. In this work, we argue that d-DNNF compilation would substantially benefit from reasoning over the original input circuit's structure, rather than solely relying on its CNF representation. To this end, we adapt an existing compiler and implement an optimisation that becomes more readily available once we reason over the input circuit: the identification and elimination of don't care variables. We empirically demonstrate the effectiveness of this approach, achieving a significant improvement in both the number of solved instances and the size of the resulting circuits.[1]

1 Introduction

Boolean functions are compactly represented by Boolean circuits. Beyond size, their practical utility also depends on structural properties [Darwiche and Marquis, 2002]. In this work, we focus specifically on the class of d-DNNF (deterministic Decomposable Negation Normal Form) circuits, as they facilitate tractable probabilistic inference [Chavira and Darwiche, 2008; Derkinderen *et al.*, 2024].

Existing d-DNNF compilers, which compile arbitrary input to d-DNNF, can be categorized as either bottom-up or top-down. The former rely on an apply-operation that allows the circuit to be built incrementally but also restricts the class of produced d-DNNF circuits [Darwiche, 2011]. The latter approaches do not rely on an apply operator and instead use the traces of a CDCL-based model counting algorithm [Huang and Darwiche, 2005; Lagniez and Marquis, 2017a; Korhonen and Järvisalo, 2023]. We will focus on the latter, top-down compilers.

Top-down compilers require input circuits in conjunctive normal form (CNF), a standard assumption they inherit from the CDCL algorithm on which they are based. This assumption is made possible by the Tseitin transformation [Tseitin, 1983], which transforms any circuit into an equisatisfiable CNF prior to compiling. As a downside, however, structural information of the input circuit is lost, or becomes less apparent, exactly due to this transformation [Thiffault *et al.*, 2004]. Retaining such information can enhance preprocessing [Lagniez *et al.*, 2020], and mitigate inefficiencies that arise during compilation. For example, [Derkinderen, 2024] demonstrated that parts of the compiled circuit are not pertinent, needlessly increasing the circuit size.

Our primary contribution is a novel compiler that leverages information from the original input circuit. We argue that d-DNNF compilers are more effective when they reason about this circuit rather than solely its CNF representation, and we consequently advocate for directing compiler development towards this approach. To support this, we adapt a compiler to retain the original circuit and study an optimisation that, as a consequence, becomes more readily available. This optimisation builds on the observation that don't care variables—circuit gates that become disconnected from the output during compilation—can be eliminated. While this idea has previously been explored in the context of circuit satisfiability solving [Thiffault *et al.*, 2004; Drechsler *et al.*, 2009], we are the first to show its efficacy in the domain of knowledge compilation for d-DNNF circuits. Importantly, our implementation maintains both the original input circuit and its CNF representation. This allows our method to benefit from existing data structures and algorithms, while enabling seamless integration into other state-of-the-art CDCL-based d-DNNF compilers.

To evaluate our approach, we modified the compiler `d4` [Lagniez and Marquis, 2017a]. Experiments on a variety of benchmarks involving probabilistic inference problems modeled as circuits reveal significant computational benefits. In many cases, the computational overhead and size of the resulting d-DNNF circuits are reduced by up to an order of magnitude compared to the baseline `d4`.

The remainder of the paper is organized as follows. First, we present the formal preliminaries. Then, we describe the theoretical basis and provide implementation details. Next, we describe the experimental protocol used in our empirical

[1]This article has been accepted for publication at the International Joint Conference on Artificial Intelligence (IJCAI2025).

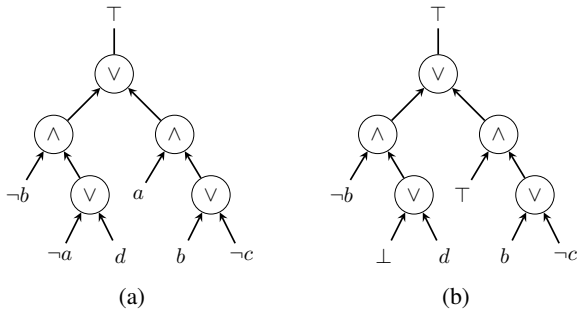


Figure 1: Example of circuits.

evaluations and discuss the results. Finally, we conclude the paper by highlighting potential directions for future research.

2 Background

A *Boolean function* $f(X)$ over a set of n Boolean variables X maps each instantiation of X to a Boolean value $\{\top, \bot\}$. An instantiation that f maps to $\top$ is called a *model* of f. We use $Mods(f)$ to refer to the set of models of f. The computational problem of determining the number of models of f, is called *model counting*. A *literal* l is an instantiated Boolean variable x, which we denote using x or its negation $\neg x$.

A *circuit* Σ represents a Boolean function as a rooted directed acyclic graph, denoted as a tuple of nodes and edges $\langle N, E\rangle$, wherein each leaf node is a literal or a Boolean constant ($\top$ or $\bot$), and each internal node is a *gate* with type negation ($\neg$), conjunction ($\wedge$) or disjunction ($\vee$), each with the usual semantics. We define conjunction and disjunction gates to have at least two inputs (incoming edges), and a negation gate to have exactly one input. When each gate of Σ only has one output (i.e., one outgoing edge), we call Σ a formula. Without loss of generality, we assume that each circuit that has to be compiled has a single sink, denoted as $sink(\Sigma)$. We use $Vars(\Sigma)$ to denote the set of Boolean variables that Σ uses in its leaf nodes. For convenience, we may additionally use the equivalence symbol $\Leftrightarrow$ with the usual semantics, and may use Σ to refer to the Boolean function f that it represents. For instance, we may use $Mods(\Sigma)$ in place of $Mods(f)$.

Conditioning circuit Σ on literal ℓ, denoted as $\Sigma_{|\ell}$, is equivalent to replacing each occurrence of ℓ in Σ with $\top$, and $\neg\ell$ with $\bot$. We use $\exists x.\Sigma$ to denote existential quantification of x, which is semantically equivalent to $\Sigma|_x \vee \Sigma|_{\neg x}$. We extend this to a set operation $\exists X.\Sigma$.

Example 1. *Figure 1 presents two circuits. Figure 1a depicts the formula* $\Sigma = ((\neg a \vee d) \wedge \neg b) \vee (a \wedge (b \vee \neg c))$ *over* $Vars(\Sigma) = \{a, b, c, d\}$*, while Figure 1b illustrates* $\Sigma_{|a}$.

By imposing constraints on the circuit structure, we define specific families of representation languages. In this work, we focus on CNF and d-DNNF circuits. CNF (conjunctive normal form) is a particular type of circuit that represents a conjunction of clauses, where each *clause* is a disjunction of literals. CNF is an appealing language because any circuit can be transformed into a CNF formula in polynomial time and size using the *Tseitin transformation* [Tseitin, 1983]. This transformation introduces a new auxiliary variable x for each $\vee$- and $\wedge$-gate, making the variable equivalent to the gate's output. Consequently, any other gate that refers to this output can instead refer to x. Because of this, we can assume that there are no explicit negation gates, as $\wedge$- and $\vee$-gates can simply refer to input literal $\neg x$ instead of the gate that otherwise negates x explicitly. The Tseitin transformation results in a set of equivalences of the form $\ell \Leftrightarrow \bigwedge_i \ell_i$ or $\ell \Leftrightarrow \bigvee_i \ell_i$, which can easily be transformed into CNF.

Example 2 (Example 1 cont'd)**.** *Consider the circuit* Σ *from Example 1. The Tseitin transformation* $\mathcal{T}$ *introduces five fresh variables* $\{x_1, \ldots, x_5\}$ *for* Σ*, each defined as follows:*

$x_1 \Leftrightarrow \neg a \vee d \quad x_2 \Leftrightarrow b \vee \neg c \quad x_3 \Leftrightarrow \neg b \wedge x_1$

$x_4 \Leftrightarrow a \wedge x_2 \quad x_5 \Leftrightarrow x_3 \vee x_4$

Since Σ *must be satisfied, we set* $x_5 \Leftrightarrow \top$.

Unfortunately, CNF does not allow for a polynomial-time consistency test (unless P=NP), making it unsuitable as a target representation language. In contrast, d-DNNF (deterministic Decomposable Negation Normal Form) is a compelling choice due to its support for tractable probabilistic inference [Darwiche and Marquis, 2002]. d-DNNF consists of Boolean circuits where each input is either a literal or a Boolean constant ($\bot$ or $\top$), and each internal gate is either a decomposable $\wedge$-gate or a deterministic $\vee$-gate. In a decomposable gate of the form $\wedge(N_1, \ldots, N_k)$, no common variable is shared between the subcircuits rooted at N_i and N_j for all $i \neq j$. In a deterministic gate of the form $\vee(N_1, \ldots, N_k)$, the subcircuits rooted at N_i and N_j are jointly inconsistent for all $i \neq j$. The size of a circuit Σ, denoted by $|\Sigma|$, is its number of gates. d-DNNF is universal, as it can accommodate every propositional theory [Darwiche, 2002].

In this paper, we focus on compilers that take CNF formulas as input and compile them into d-DNNF circuits. In state-of-the-art d-DNNF compilers, the circuit is derived from the trace of an exhaustive search conducted by a CDCL SAT solver. The *Conflict-Driven Clause Learning* (CDCL) algorithm was originally designed to solve satisfiability problems, determining whether a model exists [Silva and Sakallah, 1996]. However, it can be adapted for model counting [Birnbaum and Lozinskii, 1999] and, by leveraging the trace of the procedure [Huang and Darwiche, 2005], it can function as a d-DNNF compiler.

The algorithm operates by iteratively branching on literals (making decisions) until either all clauses are satisfied or a conflict arises. When a conflict occurs, the algorithm identifies its root cause and generates a conflict clause, which is added to the search process. It then backtracks to an earlier decision point and continues. A particularly useful optimization for model counting and d-DNNF compilation is component decomposition and component caching [Bacchus *et al.*, 2003]. A *component* is a subset of clauses that does not share any variables with clauses outside the component, allowing it to be solved independently. Component decomposition involves regularly partitioning the remaining clauses into independent components, solving each separately, and combining the results using a $\wedge$-gate. Additionally, by incorporating a caching mechanism, the algorithm can reuse results from previously solved components. In the context of compilation, this leads to more succinct d-DNNF circuits with a non-tree

structure.

3 Circuit Aware Compilation

The assumption that the input circuit is a CNF formula is ingrained in several aspects of CDCL-based compilers. For example, most variable ordering heuristics and component representations have been designed specifically for CNF. In this work, we argue for the development of compilers capable of reasoning about more general input circuits. To support this, we adapt an existing compiler to preserve the original circuit Σ and reason about it. Importantly, to leverage the efficient implementations already proposed for CNF formulas, we maintain both the circuit and its CNF version. This way, each part of the compiler can reason using the most suitable representation.

This approach allows us to dynamically eliminate gates from Σ that have become irrelevant. The key challenge lies in determining the conditions under which this elimination is possible. In the following sections, we present theoretical insights that define these conditions, followed by the introduction of the algorithms designed to efficiently identify gates for removal. Such algorithms have previously been used in the context of solving circuit satisfiability problems [Thiffault *et al.*, 2004], but have not yet been applied to model counting and knowledge compilation prior to this work.

3.1 Theoritical Insights

Currently, compiling a circuit Σ into a d-DNNF involves translating it to a CNF formula Φ using the Tseitin encoding and then running a d-DNNF compiler on Φ. Due to the Tseitin encoding, the formula Φ introduces additional variables X that were not present in the original circuit Σ.

Two strategies can be envisioned to obtain the d-DNNF circuit representing Σ from Φ. The first strategy involves compiling the formula $\exists X.\Phi$, treating the variables in $Vars(\Sigma)$ as the set of projected variables in Φ. Although this approach is appealing because the resulting d-DNNF would be constructed using only the variables in $Vars(\Sigma)$, it may be less efficient in practice. This inefficiency arises from the added constraints on the branching heuristic, which dictates the order in which variables can be selected as decisions, thereby making the problem more challenging. This challenge is also reflected in the literature, where upper bounds on the time complexity of model counting for formulas with fixed treewidth k are significantly higher for projected model counting (e.g., $\mathcal{O}(2^k)$ for standard versus $\mathcal{O}(2^{2^k})$ for projected model counting) [Fichte *et al.*, 2023].

The second strategy bypasses this branching heuristic limitation by compiling the formula Φ directly. This approach is feasible because the variables in X are derived from the variables in $Vars(\Sigma)$. The key advantage of this strategy is that it allows for direct 'branching' on gates. As a side effect, the resulting d-DNNF Δ is not structurally equivalent to Σ but is query-equivalent. This means that while queries are formulated using the variables in $Vars(\Sigma)$, the answers will remain the same. Specifically, there is a one-to-one mapping between the models of Δ and the models of Σ.

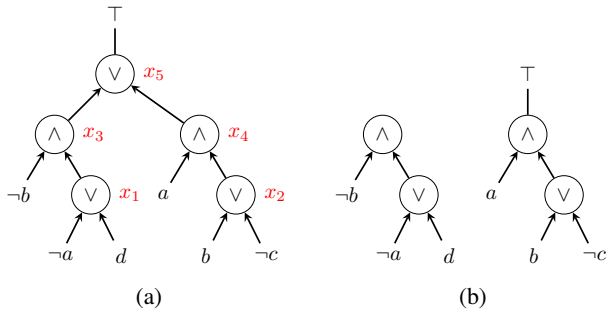


Figure 2: Figure 2a illustrates the circuit from Figure 1a, with gates assigned to Boolean variables via the Tseitin translation. Figure 2b shows the residual circuit obtained after conditioning on $\{x_5, x_4\}$.

Both strategies overlook the underlying circuit structure they represent, meaning that certain information is not directly accessible to the d-DNNF compiler. For instance, the compiler may not easily recognize that some gates do not affect the circuit's overall consistency. To illustrate this situation, consider the following example:

Example 3 (Example 2 cont'd)**.** *Consider the circuit in Figure 2a with Boolean variables from the Tseitin translation in Example 2. If variables x_5 and x_4 are true in the CNF formula, the subcircuit $\neg b \wedge (\neg a \vee d)$ becomes irrelevant, as shown in Figure 2b.*

Even though it is clear from inspecting the circuit that the valuation of x_3 is irrelevant, this insight is not as easily derived when dealing with the set of equivalences generated by the Tseitin encoding. The Boolean function encoded by the set of equivalences $\Gamma_1 = \{x_5 \Leftrightarrow \top, x_4 \Leftrightarrow \top, x_1 \Leftrightarrow \neg a \vee d, x_2 \Leftrightarrow b \vee \neg c, x_3 \Leftrightarrow \neg b \wedge x_1, x_4 \Leftrightarrow a \wedge x_2\}$ differs from that of $\Gamma_2 = \{x_5 \Leftrightarrow \top, x_4 \Leftrightarrow \top, x_2 \Leftrightarrow b \vee \neg c, x_4 \Leftrightarrow a \wedge x_2\}$, with $Mods(\Gamma_1) \neq Mods(\Gamma_2)$. However, since our goal is to compile a d-DNNF that is query-equivalent, the following theorem demonstrates that it is possible to remove disconnected gates while preserving the set of projected models. We define a *disconnected gate* as a gate whose output neither serves as the input to another gate, nor is fixed to a Boolean constant. The variable associated with such a gate is called a *don't care* variable [Thiffault *et al.*, 2004].

Proposition 1. *Let Σ be a Boolean circuit and $\Gamma = \mathcal{T}(\Sigma)$ the set of equivalences representing Σ through the Tseitin encoding, where X denotes the set of Tseitin variables. If the output of a gate g in Σ neither serves as the input to another gate nor is fixed to a Boolean constant, then there exists a one-to-one mapping between $Mods(\Sigma)$ and $Mods(\Gamma \setminus \{x_g \Leftrightarrow \oplus_i \ell_i\})$, where $\oplus \in \{\vee, \wedge\}$ and $x_g \in X$ is the Tseitin variable representing gate g in Γ.*

Proof. Without loss of generality, assume $\oplus = \vee$ (the proof for $\oplus = \wedge$ follows similarly). First, note that if gate g is disconnected in Σ, removing it does not alter the models of Σ. This is because g does not impose any constraints on the valuation of its input variables, given that there are no constraints on the valuation of g's output. Consequently, the circuit Σ', obtained by removing g, has the same models as

Σ, i.e., $Mods(\Sigma) = Mods(\Sigma')$. By using the same Tseitin variables to identify common gates in Σ and Σ', we have $\Gamma \setminus \{x_g \Leftrightarrow \vee_i \ell_i\} = \mathcal{T}(\Sigma')$. Due to the properties of the Tseitin encoding, there exists a one-to-one mapping between $Mods(\Sigma')$ and $Mods(\Gamma \setminus \{x_g \Leftrightarrow \vee_i \ell_i\})$, and therefore, also between $Mods(\Sigma)$ and $Mods(\Gamma \setminus \{x_g \Leftrightarrow \vee_i \ell_i\})$. □

The gate elimination rule can be applied iteratively until no disconnected gates remain in the circuit. However, detecting whether a disconnected gate exists requires scanning the entire circuit, potentially necessitating a quadratic number of steps in the worst case to ensure that no gates are disconnected. This process can be particularly time-consuming for large circuits.

3.2 Implementation Details

To improve the efficiency of identifying gates that can be removed using the gate elimination rule, we introduce the `CircuitManager` object. This specialized utility integrates efficient data structures and algorithms tailored for this purpose, allowing seamless incorporation into state-of-the-art d-DNNF compilers.

To integrate our approach into compilers that only handle CNF formulas, we must efficiently coordinate the two engines: the CNF engine and the circuit engine, modelled by the `CircuitManager`. The CNF engine primarily facilitates the efficient retrieval of information for the compilation process, such as identifying connected components, managing the current formula, and computing heuristics. This engine is continuously updated at each decision node by pushing the literals assigned as true by the compiler. Additionally, when the compiler backtracks, the CNF engine is updated by restoring the formula to its previous state.

To synchronize the two engines, the circuit engine must be updated in tandem with the CNF engine. When a set of literals L is assigned to true in the CNF engine, the circuit engine must determine which gates in the circuit become *deactivated*. We distinguish between two types of such gates: the set R of gates that become resolved due to literal assignment L (i.e., the constraint represented by the gate has become satisfied), and the set D that subsequently become disconnected. This distinction aids in identifying disconnected gates, as the disconnection is triggered by resolving an ancestor gate.

Example 4 (Example 3 cont'd)**.** *When assigning $\{x_5, x_4\}$ to true in Example 3, the gate associated with x_5 is resolved ($x_5 \in R$), such that x_3 and subsequently x_1 both become disconnected $\{x_3, x_1\} \in D$.*

In the following discussion, on how to determine R and D, we assume that the circuit $\Sigma = \langle N, E \rangle$ has been translated into a CNF formula using the Tseitin encoding $\mathcal{T}$. This translation allows us to uniquely identify each gate in the circuit by the literal associated with it.

Example 5 (Example 2 cont'd)**.** *The circuit $\Sigma = \langle N, E \rangle$ labeled by $\mathcal{T}$ in Example 2 consists of the following elements: $N = \{a, b, c, d, \neg a, \neg b, \neg c, \neg d, x_1, x_2, x_3, x_4, x_5\}$ and $E = \{(x_2, x_4), (x_3, x_5), (x_4, x_5), (\neg a, x_1), (\neg c, x_2), (\neg b, x_3), (d, x_1), (b, x_2), (x_1, x_3), (a, x_4)\}$.*

Algorithm 1: `initWatches`

Input: a circuit $\Sigma = \langle N, E \rangle$.
Output: the set of gates that are not watched.

1 $G \leftarrow \{g \in N \mid \nexists (g', g) \in E\}$
2 Let `watches` be an empty map
3 **for** $g \in N \setminus \{g \in N \mid \nexists (g', g) \in E\}$ **do**
4 $\quad$ `watches`$[g] = \{g' \in N \setminus G \mid (g', g) \in E\}$
5 $\quad$ $G \leftarrow G \cup$ `watches`$[g]$
6 **return** $N \setminus \{G \cup sink(\Sigma)\}$

To determine R, we consider for each literal $l \in L$ the active gates it is involved in (either as output or input) and apply the following rules. A $\wedge$-gate g is resolved if its Tseitin variable x_g is assigned to true, or x_g is assigned to false and at least one of the input literals to g is also assigned to false. In case of the former, the CNF engine will propagate the input literals of g to also become true, so these must not be checked anymore. Similarly, a $\vee$-gate g is resolved if its Tseitin variable x_g is assigned to false, or if x_g is assigned to true and at least one of the input literals to g is also assigned to true. To efficiently apply these rules, we can reuse data structures already present in the CNF engine.

As the gates in R are resolved, they become inactive and may cause a child gate to become disconnected. Rather than verifying for each child whether any of its outputs are still active to determine disconnection, we propose a more efficient mechanism akin to the concept of watched literals [Moskewicz *et al.*, 2001]. Thus, for each circuit gate g', we designate exactly one parent gate g to watch it, called the sentinel of g'.

To achieve this, `CircuitManager` includes a mapping of watch lists, denoted as `watches`. This structure links each gate $g \in N$ to a list `watches`$[g]$ that contains all gates monitored by g. We initialize this `watches` data structure by invoking the method `initWatches`, depicted in Algorithm 1. This function begins by initializing the set G as all nodes without incoming edges (i.e., non-gate nodes), as these nodes cannot function as a sentinel and must also not be watched (line 1). Then, the function iterates over all gates g (line 3), populating `watches`$[g]$ with the child gates that g will be the sentinel of (line 4). Importantly, each gate is only watched by one sentinel, so we also use G to track the gates that are already being watched (line 5). Finally, this function returns the set of gates that are not being watched (line 6). If we assume that the initial input circuit has no disconnected gates, then this will always return the empty set.

Example 6 (Example 2 cont'd)**.** *Consider the circuit Σ described in Example 2 with gates $\{x_1, x_2, \ldots, x_5\}$. The* `watches` *data structure is then initialized as follows.*

`watches`$[x_1] =$ `watches`$[x_2] = \emptyset$ `watches`$[x_3] = \{x_1\}$
`watches`$[x_5] = \{x_3, x_4\}$ `watches`$[x_4] = \{x_2\}$

To complete the initialization of the `CircuitManager` object, we introduce an array named `isActiveGate` to track the status of each gate. This array maps each gate in Σ (identified by its corresponding literal) to a Boolean value, initially set to `true` for all gates, indicating that they are ac-

Algorithm 2: `propagate`

Input: a circuit $\Sigma = \langle N, E \rangle$, and a set of deactivated gates R.
Output: the set of gates that are deconnected.

```
1  D ← ∅
2  for g ∈ R do isActiveGate[g] = false
3  while ∃g ∈ R do
4      R ← R \ {g}
5      foreach g′ ∈ watches[g] do
6          if isActiveGate[g′] = false or x_{g′} is assigned then continue
7          if ∃g″ ∈ N s.t. (g′, g″) ∈ E and isActiveGate[g″] = true then
8              watches[g″] ← watches[g″] ∪ {g′}
           else
9              Add {g′} into R and D
10             isActiveGate[g′] = false
11     Remove all active gates from watches[g]
12 return D
```

tive. If a gate becomes inactive, its corresponding value in `isActiveGate` is set to `false`. Additionally, we maintain a stack S to monitor changes made to `isActiveGate` during each invocation of the `propagate` function.

We use the `propagate` function outlined in Algorithm 2 to effectively determine D using the `watches` data structure. The primary goal of `propagate` is to maintain an invariant: each gate not yet assigned to `true` or `false` must be monitored by one active gate. It takes as input the circuit Σ and the set R of gates that have been deactivated. The function returns the set D of gates identified as deactivated after considering the changes in R. After invoking this function, the CNF engine can remove all clauses corresponding to the gates $g \in D$.

The function begins by initializing D as $\emptyset$ (line 1) and marking all gates in R as deactivated (line 2). Since these are deactivated, we must reconsider the sentinel of all active gates g' they watch, i.e., all active gates in `watches`$[g]$ (line 5 and 6). We can either assign these gates a new sentinel that is still active (line 7 and 8), or, if none is available anymore, add them to D to indicate that they have become disconnected (line 9 and 10). In the latter case, we also add them to R such that their watched gates are re-examined (line 3 and 9). An exception to this is when the watched gate g' is not disconnected but connected to a constant (i.e., when the compilation process already assigned $x_{g'}$ to true or false; line 6). After processing, all active gates are removed from `watches`$[g]$ (line 11) since the responsibility of watching them has been transferred to other gates that are still active. Finally, the function returns the set D of newly deactivated gates.

Synchronizing both engines during the backtrack phase only requires reactivating gates that were deactivated due to the assignment of literals in L. This process is efficiently managed using a stack S. Each time literals are pushed onto the CNF engine, the corresponding elements from $R \cup D$ are also pushed onto S. During backtracking, gates can be reactivated by processing S in reverse order and by updating the data structure `isActiveGate` accordingly.

4 Related Work

We are not the first to recognize the advantages of reasoning over the original input circuit. This insight has been previously explored in the context of related problems, such as determining circuit satisfiability [Silva and Guerra e Silva, 1999; Ganai *et al.*, 2002; Lu *et al.*, 2003; Wu *et al.*, 2007; Drechsler *et al.*, 2009; Zhang *et al.*, 2021; Hu and Chu, 2023]. We specifically point out the work of [Thiffault *et al.*, 2004], which has similarly investigated the elimination of don't care variables in the context of circuit satisfiability. In the domain of model counting and d-DNNF compilation, however, there exists little work that leverages the input circuit during the compilation process itself, aside from preprocessing steps.

For example, [Lagniez *et al.*, 2020] introduced a preprocessing technique that uses gate information to optimize the input CNF formula by eliminating variables that are functionally dependent on others. Their approach starts with an input CNF formula and then extracts the original gate information to apply their optimization. This underscores the value of retaining the original input circuit structure. While their work primarily uses structural information in preprocessing, our approach innovatively eliminates gates dynamically during the compilation process itself.

Regarding the gate elimination, [Derkinderen, 2024] presents related work on eliminating tautological subcircuits in a compiled d-DNNF. Such subcircuits may emerge when performing existential quantification of the Tseitin variables after compilation. In contrast to our approach, which detects and eliminates subcircuits during compilation, their method operates post-compilation through a bottom-up evaluation of the d-DNNF circuit. By detecting irrelevant gates during compilation, our approach reduces compilation time. Additionally, our method targets a slightly different type of irrelevant subcircuit.

Information on which variables are Tseitin variables can be exploited. For instance, to compile via projected knowledge compilation [Lagniez and Marquis, 2019], instead of via the regular compilation task. However, as mentioned in the previous section, in this setting the compilers are more limited as they cannot branch on every variable. Similarly, blocking clause elimination (BCE) [Lagniez *et al.*, 2024] could be used in this setting to eliminate clauses that do not affect the result, but this again restricts the variables that can be branched on. In contrast, our approach allows the compiler to branch on both input and Tseitin variables, a benefit we empirically demonstrate through our comparison in the next section.

[Dubray *et al.*, 2023] explores projected model counting in the context of Horn clauses—clauses with at most one positive literal. Because of this restriction they can easily detect don't care variables as part of their propagation process. However, their work does not address compilation and focuses solely on Horn clauses, thus limiting their reasoning to CNF formulas.

5 Experiments

To empirically assess the benefits of eliminating don't care variables, we conducted experiments using 549 instances of probabilistic inference problems modeled as circuits. These instances, as well as the code, are available at [Derkinderen and Lagniez, 2025]. The instances are grouped into eight datasets:

- `dnf`: 4 instances of formulas in disjunctive normal form (DNF), that originate from the domain of neuro-symbolic AI [Maene and De Raedt, 2023].
- `noisy`: 7 instances of a noisy-OR probabilistic query, with an increasing number of parents.
- `bnkr`: 12 instances of a Bayesian network [Scutari, 2023] query, encoded as a ProbLog program.
- `games`: 3 instances of a probabilistic query in a ProbLog program that models a game of chance.
- `grid`: 28 instances of a probabilistic query in a ProbLog model of a power grid network [Wiegmans, 2016], adapted from [Latour *et al.*, 2019].
- `raki` 142 instances from [Kiesel and Eiter, 2023]: "a new set of benchmarks using two tools to translate (probabilistic) logic programs to CNFs [Janhunen and Niemelä, 2011; Eiter *et al.*, 2021] on standard benchmarks from probabilistic logic programming".
- `smokers` includes two benchmark sets for probabilistic queries in an influence network, modelled as a ProbLog program. The first set contains 109 instances with networks provided by the networkx package [Hagberg *et al.*, 2008], which were converted into directed influence networks by randomly assigning directions. The second set contains 220 instances of an influence network randomly generated using the Extended Barabasi-Albert Graph algorithm from the networkx package.
- `verilog`: 24 instances of the ISCAS85 and the EPFL Combinational Benchmark Suite [Sweeney, 2020].

Our experiments ran on Intel Xeon E5-2643 processors at 3.30 GHz with 32 GiB of RAM and Linux CentOS. Each instance had a time-out of 3600 seconds and a 32 GiB memory limit. We evaluated the runtime performance of the `d4` compiler (available at https://github.com/crillab/d4v2) with three input types:

- `cnf`: `d4` receives as input a CNF formula derived from the circuit translation using Tseitin encoding.
- `pcnf`: `d4` receives an existentially quantified formula from the circuit's Tseitin encoding, treating non-existential variables as projected. `d4` compiles the projected CNF into `d-DNNF`, branching only on projected variables. We used `d4` with the BCE rule.
- `circuit`: `d4` receives the circuit. As `d4` cannot natively process circuits, it is translated into CNF to maintain all `d4` features. Thus, our method operates with two engines: a circuit engine that processes circuits directly and a CNF engine that works with the Tseitin transformations of circuits.

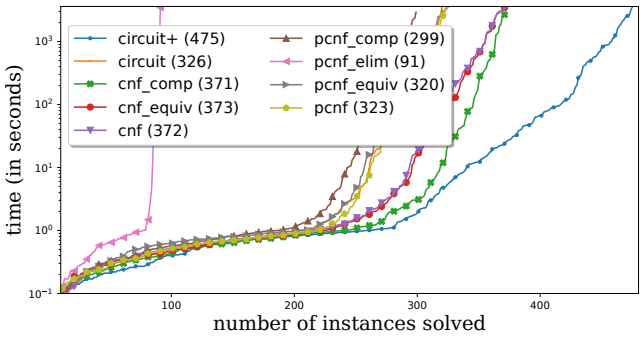


Figure 3: Cactus plot showing the run time of with various settings.

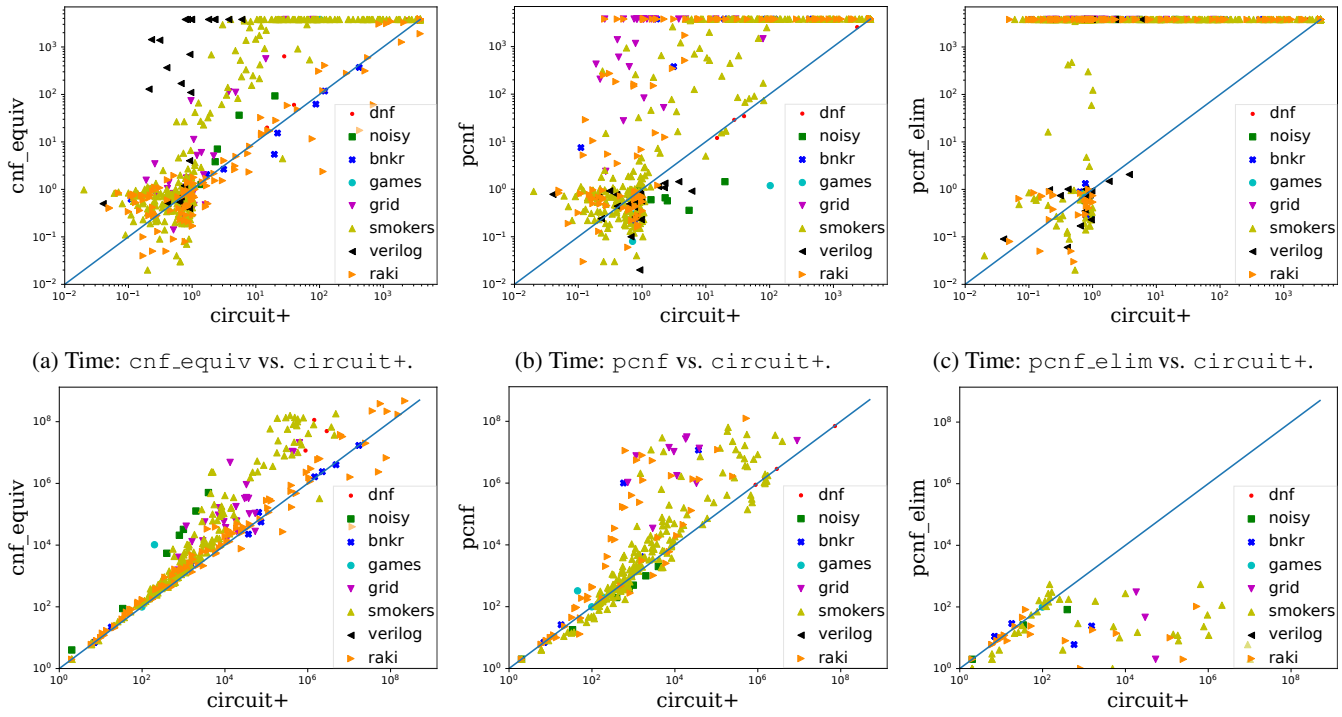
This leads to three primary versions to be tested: `cnf`, `pcnf`, and `circuit`. For CNF-based transformations, we apply three preprocessing methods. `equiv` involves running `d4` on instances that have been simplified using vivification, backbone detection, and occurrence elimination [Lagniez and Marquis, 2017b]. We also consider preprocessing methods that extend the `equiv` approach by allowing for the elimination of existentially quantified variables. This yields two additional methods: `comp` and `elim`. `comp` eliminates variables if it does not increase CNF size [Lagniez *et al.*, 2020], while `elim` aims to remove all existentially quantified variables. These preprocessing methods can be applied to `cnf` and `pcnf` transformations, resulting in six versions: `cnf_equiv`, `cnf_comp`, `cnf_elim`, `pcnf_equiv`, `pcnf_comp`, and `pcnf_elim`. Since `cnf_elim` and `pcnf_elim` are equivalent, only `pcnf_elim` is used in our experiments.

The final version we considered, named `circuit+`, extends the `circuit` version by incorporating the previously presented gate elimination procedure. Deactivated gates in the circuit engine prompt the CNF engine to remove the corresponding clauses.

In summary, our experimental analysis evaluated nine versions of the `d4` compiler. Figure 3 presents a cactus plot illustrating the effectiveness of these different versions. Each line in the plot represents a different method, with the number of instances solved indicated in parentheses in the legend. The plot displays the number of instances completed within a given CPU time limit, measured in seconds.

The `circuit` version, which theoretically matches the performance of `cnf`, is less efficient in practice, solving 326 instances versus 372 for `cnf`. This inefficiency is partly due to managing two engines and the impact of SAT solver simplifications on `d4`'s heuristics. Eliminating all existentially quantified variables is not advisable for `cnf` and `pcnf` transformations due to excessive memory requirements, leading to 427 memory overflows. The other preprocessing techniques also prove ineffective, as they do not enhance `d4`'s performance. In fact, applying these preprocessing steps typically reduces the number of instances that `d4` can solve, making their use generally inadvisable.

Regarding the impact of transformations on compilation time, communicating the clauses to be removed from the CNF engine positively affects `d4`'s performance. Specifically, the



(a) Time: `cnf_equiv` vs. `circuit+`. (b) Time: `pcnf` vs. `circuit+`. (c) Time: `pcnf_elim` vs. `circuit+`.

(d) Size: `cnf_equiv` vs. `circuit+`. (e) Size: `pcnf` vs. `circuit+`. (f) Size: `pcnf_elim` vs. `circuit+`.

Figure 4: Pairwise comparison of the run time and d-DNNF circuit size of the compilation process.

`circuit+` version solves 475 instances, which is 102 (30%) more than `cnf_equiv` (the most effective `cnf` transformation version) and 152 (45%) more than `pcnf`, the most effective `pcnf` transformation version. Additionally, it is noteworthy that versions based on the `pcnf` transformation are generally less efficient. This inefficiency may stem from the constraints imposed by the branching heuristic, which forces branching on non-existentially quantified variables, potentially diminishing the compiler's performance.

To gain further insights into the impact of the transformations, we examined the compilation times and sizes of the d-DNNF compiled forms generated by `circuit+`, `cnf_equiv`, and `pcnf`. The results are depicted in the scatter plots shown in Figure 4. Each dot represents an instance, showing the time (in seconds) needed to solve it or the size (in number of gates) of the resulting compiled form. Logarithmic scales are used for both axes. Any instance above the diagonal is favorable for `circuit+`. As illustrated in Figures 4a and 4b, `circuit+` is generally faster than both `cnf_equiv` and `pcnf`. Additionally, Figures 4d and 4e demonstrate that the size of the compiled d-DNNF formulas is also generally smaller for `circuit+`.

Figure 4 also presents a pairwise comparison between `circuit+` and `pcnf_elim`. Although `pcnf_elim` is significantly slower than `circuit+` (see Figure 4c) and reaches the time-out on many instances, Figure 4f reveals that the size of the d-DNNF circuits that `pcnf_elim` did produce are substantially smaller. This reduction in size can be attributed to the fact that the Tseitin encoding, which replaces gates with clauses and adds variables, may diminish the effectiveness of syntactical caching in `d4`. While investigating this phenomenon is beyond the scope of this paper, it suggests promising potential for leveraging circuit-based information to reduce the size of compiled d-DNNF circuits.

6 Conclusion

We have advocated for the development of d-DNNF compilers that reason directly over the input circuit, as opposed to solely over a CNF version of it. We have taken a first step in this direction by adapting the d-DNNF compiler `d4` to maintain both the original input circuit and its CNF version. This adaptation enabled us to implement an optimization novel to compilation: identifying and eliminating don't care variables, i.e., gates whose output has become disconnected from the rest of the circuit. Our empirical evaluation demonstrates the advantages of this optimization, as it results in solving more instances with smaller d-DNNF circuits. This suggests that reasoning directly over the input circuit can enhance compiler performance.

As a future direction, we plan to further explore the use of circuit-based information. This includes developing branching heuristics that specifically prioritize gate elimination. Additionally, to reduce the overhead associated with managing two representation engines, we aim to develop a version of `d4` that eliminates the need for the CNF representation engine entirely. This would remove the constraint of associating a name with each gate, potentially increasing the number of cache hits and improving overall efficiency.

Acknowledgments

This work has benefited from the support of the AI Chair EXPEKCTATION (ANR-19-CHIA-0005-01) of the French National Research Agency (ANR). It was also partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215, by the KU Leuven Research Fund (C14/18/062), by iBOF/21/075, and by the FWO research foundation - Flanders under project No G097720N. The authors thank Pierre Marquis for the insightful discussions.

References

[Bacchus *et al.*, 2003] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. DPLL with caching: A new algorithm for #sat and Bayesian inference. *Electron. Colloquium Comput. Complex.*, TR03-003, 2003.

[Birnbaum and Lozinskii, 1999] Elazar Birnbaum and Eliezer L. Lozinskii. The good old Davis-Putnam procedure helps counting models. *J. Artif. Intell. Res.*, 10:457–477, 1999.

[Chavira and Darwiche, 2008] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.

[Darwiche, 2002] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In Rina Dechter, Michael J. Kearns, and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 627–634. AAAI Press / The MIT Press, 2002.

[Darwiche, 2011] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, pages 819–826. IJCAI/AAAI, 2011.

[Derkinderen and Lagniez, 2025] Vincent Derkinderen and Jean-Marie Lagniez. Source data and code for 'circuit-aware d-dnnf compilation', 2025.

[Derkinderen *et al.*, 2024] Vincent Derkinderen, Robin Manhaeve, Pedro Zuidberg Dos Martires, and Luc De Raedt. Semirings for probabilistic and neuro-symbolic logic programming. *Int. J. Approx. Reason.*, 171:109130, 2024.

[Derkinderen, 2024] Vincent Derkinderen. Pruning Boolean d-DNNF circuits through Tseitin-awareness. In *ICTAI*, pages 663–670. IEEE, 2024.

[Drechsler *et al.*, 2009] Rolf Drechsler, Tommi A. Junttila, and Ilkka Niemelä. Non-clausal SAT and ATPG. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 655–693. IOS Press, 2009.

[Dubray *et al.*, 2023] Alexandre Dubray, Pierre Schaus, and Siegfried Nijssen. Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses. In *CP*, volume 280 of *LIPIcs*, pages 15:1–15:17, 2023.

[Eiter *et al.*, 2021] Thomas Eiter, Markus Hecher, and Rafael Kiesel. Treewidth-Aware Cycle Breaking for Algebraic Answer Set Counting. In *KR*, pages 269–279, 2021.

[Fichte *et al.*, 2023] Johannes Klaus Fichte, Markus Hecher, Michael Morak, Patrick Thier, and Stefan Woltran. Solving projected model counting by utilizing treewidth and its limits. *Artif. Intell.*, 314:103810, 2023.

[Ganai *et al.*, 2002] Malay K. Ganai, Pranav Ashar, Aarti Gupta, Lintao Zhang, and Sharad Malik. Combining strengths of circuit-based and cnf-based algorithms for a high-performance SAT solver. In *DAC*, pages 747–750. ACM, 2002.

[Hagberg *et al.*, 2008] Aric Hagberg, Pieter J Swart, and Daniel A Schult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.

[Hu and Chu, 2023] Kunmei Hu and Zhufei Chu. An efficient circuit-based SAT solver and its application in logic equivalence checking. *Microelectron. J.*, 142:106005, 2023.

[Huang and Darwiche, 2005] Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI*, pages 156–162. Professional Book Center, 2005.

[Janhunen and Niemelä, 2011] Tomi Janhunen and Ilkka Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2011.

[Kiesel and Eiter, 2023] Rafael Kiesel and Thomas Eiter. Knowledge Compilation and More with SharpSAT-TD. In *KR*, pages 406–416, 2023.

[Korhonen and Järvisalo, 2023] Tuukka Korhonen and Matti Järvisalo. SharpSAT-TD in Model Counting Competitions 2021-2023. *CoRR*, abs/2308.15819, 2023.

[Lagniez and Marquis, 2017a] Jean-Marie Lagniez and Pierre Marquis. An Improved Decision-DNNF Compiler. In Carles Sierra, editor, *IJCAI*, pages 667–673, 2017.

[Lagniez and Marquis, 2017b] Jean-Marie Lagniez and Pierre Marquis. On preprocessing techniques and their impact on propositional model counting. *J. Autom. Reason.*, 58(4):413–481, 2017.

[Lagniez and Marquis, 2019] Jean-Marie Lagniez and Pierre Marquis. A Recursive Algorithm for Projected Model Counting. In *AAAI*, pages 1536–1543. AAAI Press, 2019.

[Lagniez *et al.*, 2020] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Definability for model counting. *Artif. Intell.*, 281:103229, 2020.

[Lagniez *et al.*, 2024] Jean-Marie Lagniez, Pierre Marquis, and Armin Biere. Dynamic blocked clause elimination for projected model counting, 2024.

[Latour *et al.*, 2019] Anna Louise D. Latour, Behrouz Babaki, and Siegfried Nijssen. Stochastic constraint propagation for mining probabilistic networks. In *IJCAI*, 2019.

[Lu *et al.*, 2003] Feng Lu, Li-C. Wang, Kwang-Ting Cheng, and Ric C.-Y. Huang. A circuit SAT solver with signal correlation guided learning. In *DATE*, pages 10892–10897. IEEE Computer Society, 2003.

[Maene and De Raedt, 2023] Jaron Maene and Luc De Raedt. Soft-unification in deep probabilistic logic. In *NeurIPS*, 2023.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[Scutari, 2023] Marco Scutari. The bnlearn Bayesian network repository. https://www.bnlearn.com/bnrepository/, 2023.

[Silva and Guerra e Silva, 1999] João P. Marques Silva and L. Guerra e Silva. Solving satisfiability in combinational circuits with backtrack search and recursive learning. In *Proceedings. XII Symposium on Integrated Circuits and Systems Design (Cat. No.PR00387)*, pages 192–195, 1999.

[Silva and Sakallah, 1996] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227. IEEE Computer Society / ACM, 1996.

[Sweeney, 2020] Joseph Sweeney. Github verilog_benchmark_circuits. https://github.com/jpsety/verilog_benchmark_circuits Accessed: 2024-07-15., 2020.

[Thiffault *et al.*, 2004] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with DPLL search. In *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer, 2004.

[Tseitin, 1983] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[Wiegmans, 2016] Bart Wiegmans. GridKit: European and North-American extracts. doi: 10.5281/zenodo.47317, 2016.

[Wu *et al.*, 2007] Chi-An Wu, Ting-Hao Lin, Chih-Chun Lee, and Chung-Yang Huang. Qutesat: a robust circuit-based SAT solver for complex circuit structure. In *DATE*, pages 1313–1318. EDA Consortium, San Jose, CA, USA, 2007.

[Zhang *et al.*, 2021] He-Teng Zhang, Jie-Hong R. Jiang, and Alan Mishchenko. A circuit-based SAT solver for logic synthesis. In *ICCAD*, pages 1–6. IEEE, 2021.