Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning
Main Track

Leveraging Decision-DNNF Compilation for Enumerating Disjoint Partial Models

Jean-Marie Lagniez[1] , **Emmanuel Lonca**[1]
[1]CRIL, U. Artois & CNRS, F-62300 Lens, France
{lagniez,lonca}@cril.fr

Abstract

The All-Solution Satisfiability Problem (AllSAT) extends SAT by requiring the identification of all possible solutions for a propositional formula. In practice, enumerating all complete models is often infeasible, making the identification of partial models essential for generating a concise representation of the solution set. Deterministic Decomposable Negation Normal Form (d-DNNF) serves as a language for representation known to offer polynomial-time algorithms for model enumeration. Specifically, when a propositional formula is encoded in d-DNNF, it enables iterative model enumeration with polynomial delay between models. However, despite the existence of theoretical algorithms for this purpose, no available implementations are currently accessible. Furthermore, these theoretical approaches are nearly impractical as they solely yield complete models. We introduce a novel algorithm that maintains a polynomial delay between partial models while significantly enhancing efficiency compared to baseline approaches. Furthermore, through experimental validation, we demonstrate the superiority of compiling a CNF formula Σ into a d-DNNF formula Σ' and subsequently enumerating models of Σ' over existing state-of-the-art methodologies for CNF partial model enumeration.

1 Introduction

Propositional satisfiability (SAT) involves determining whether a Boolean formula is satisfiable. The All-Solution Satisfiability Problem (AllSAT) extends SAT by seeking to identify all possible solutions of a propositional formula. AllSAT finds applications across diverse domains, including hardware and software verification (Khurshid et al. 2003; Jin, Han, and Somenzi 2005), artificial intelligence (Lagniez, Lonca, and Mailly 2015; Spallitta et al. 2022), model-based diagnosis (Darwiche 1998), data mining (Han et al. 2007; Boudane et al. 2017), graph theory (Jabbour et al. 2022), biology (Trinh et al. 2022), and more.

There are several versions of AllSAT, and this paper addresses the enumeration problem, which involves listing the set of models of a propositional formula without redundancies (disjoint AllSAT). Considering the often vast number of complete models associated with a given formula, our focus lies in enumerating partial models. A partial model provides a compact representation of a model set by accommodating incomplete truth value assignments to certain variables. To qualify as a partial model, it must ensure that assigning any truth value to an unassigned variable does not affect the satisfiability of the assignment. As a result, a partial model with m variables encompasses 2^{n-m} complete models.

Given that allSAT extends SAT, numerous approaches leverage SAT solvers to enumerate the models of a propositional formula. These SAT-based propositional enumeration algorithms typically fall into two main categories: blocking solvers and non-blocking solvers. Blocking AllSAT solvers utilize Conflict-Driven Clause Learning (CDCL) to provide the complete set of satisfying assignments. They operate by iteratively introducing blocking clauses to the formula after each model is discovered until no further models are found (McMillan 2002; Jin, Han, and Somenzi 2005; Audemard, Lagniez, and Simon 2013; Yu et al. 2014). Conversely, non-blocking AllSAT solvers employ chronological backtracking. This technique ensures that upon encountering a conflict, the solver backtracks on the search tree by revising the most recently assigned variable. Chronological backtracking ensures that the same model of a formula is not covered multiple times (Li, Hsiao, and Sheng 2004; Grumberg, Schuster, and Yadgar 2004; Spallitta, Sebastiani, and Biere 2024).

Another approach to addressing the enumeration problem is through the lens of knowledge compilation (Selman and Kautz 1996), particularly utilizing the knowledge compilation map introduced by Darwiche and Marquis (2002). The objective is to target a language that enables more efficient queries, theoretically speaking. Choosing the appropriate target language can be guided by a knowledge compilation map, which evaluates languages based on multiple criteria. These criteria include the efficiency of queries and transformations supported by the languages in polynomial time, as well as their relative succinctness (i.e., their ability to represent information using minimal space). For instance, when the goal is to enumerate the (partial) models of a CNF formula, it is feasible to compile it into a language that efficiently satisfies this query, e.g. OBDD (Bryant 1986), DNNF (Darwiche 2001), d-DNNF (Darwiche 2002), EADT (Koriche et al. 2013), etc.

In (Toda and Soh 2016), the authors chose to utilize OBDD as the target language for implementing an allSAT solver. However, beyond this particular work, there have not been many attempts to leverage knowledge compilation for enumerating all solutions of CNF formulas. This lack of ex-

ploration is partly due to the fact that the knowledge compilation map is more of a theoretical tool than a practical one. Selecting a target language that satisfies the enumeration query is only one aspect of the challenge. It is equally important to have a compiler capable of translating a given CNF formula into this language. Moreover, while theoretically satisfying the enumeration query is significant, practical considerations come into play, especially when dealing with a large number of solutions. In such cases, it is essential to have a dedicated tool that can efficiently address this query.

In this study, we propose to use a compilation-based enumeration method with the targeted language being d−DNNF. This choice is motivated by the fact that the d−DNNF language satisfies the enumeration query, and notably, efficient knowledge compilers exist to translate CNF formulas into d−DNNF (Darwiche 2002; Muise et al. 2012; Lagniez and Marquis 2017a). However, to the best of our knowledge, there is a notable absence of tools designed to take a d−DNNF as input and enumerate its (partial) models. Moreover, aside from the mostly theoretical result presented in (Darwiche and Marquis 2002) (which serves as the baseline approach in our subsequent discussions), we encountered no algorithms detailing the enumeration of (partial) models of a d−DNNF. In light of this gap, we introduce an algorithm dedicated to the enumeration of (partial) models of a d−DNNF Σ. Our algorithm leverages the DAG structure of the d−DNNF to efficiently enumerate its models while ensuring that the memory requirement for model enumeration remains bounded by the size of Σ. Furthermore, we demonstrate that the models are enumerated in a disjoint manner, with a polynomial delay between two consecutive models (once the CNF formula has been compiled into a d−DNNF formula). Our experimental results underscore the superiority of our approach, dubbed d4+model−graph, over other solvers across a broad spectrum of benchmarks, showcasing the tangible benefits of our method.

The remainder of the paper unfolds as follows: we begin with formal preliminaries in Section 2, followed by the presentation of our enumerator for partial models of d−DNNF formulas in Section 3. Section 4 offers an empirical evaluation of our approach, leading to the concluding section (Section 5).

2 Formal Preliminaries

We consider a propositional language $PROP_{PS}$ in the standard manner, derived from a finite set PS of propositional symbols and the standard logical connectives ($\wedge$, $\vee$, $\neg$). $PROP_{PS}$ is interpreted classically. For any formula Σ in $PROP_{PS}$, $Var(\Sigma)$ denotes the set of propositional variables present in Σ. Given a finite set X of variables, $\{0,1\}^X$ represents the set of all possible Boolean assignments to the variables in X. Each propositional formula Σ in $PROP_{PS}$ defines a Boolean function over $Var(\Sigma)$, mapping Σ from $\{0,1\}^{|Var(\Sigma)|}$ to $\{0,1\}$. Assignments to $Var(\Sigma)$ that evaluate to 1 under Σ are termed *satisfying assignments* or *models* of Σ. $\bot$ represents the formula which is always falsified and $\top$ the formula which is always satisfied.

A *literal* is defined as either a Boolean variable or its negation. For any literal ℓ, $Var(\ell)$ represents the variable x of ℓ ($Var(x) = x$ and $Var(\neg x) = x$), and $\sim\ell$ denotes the complementary literal of ℓ. In other words, for every variable x, $\sim x = \neg x$ and $\sim\neg x = x$. The conditioning of a formula Σ by a literal $\ell = x$ (resp. $\ell = \neg x$) results in the formula $\Sigma[\ell]$, where each occurrence of x (resp. $\neg x$) in Σ is replaced by $\top$, and each occurrence of $\neg x$ (resp. x) is replaced by $\bot$. After such replacement, simplification is carried out using the semantics of the logical connectors (e.g., $\top \vee \Gamma = \top$, $\top \wedge \Gamma = \Gamma$, etc.) until a fixed point is reached. Each assignment μ is conceptualized as a (conjunctively interpreted) set of literals. We differentiate between total assignments and partial assignments based on whether all variables are assigned truth values or not, respectively. A partial assignment μ' is deemed a partial model of Σ if, for every total assignment μ from $Var(\Sigma)$ that extends μ' (i.e., for every literal ℓ in μ', ℓ is also in μ), μ is a model of Σ.

$Mod(\Sigma)$ represents the set of all models of Σ. Two formulas Σ_1 and Σ_2 are considered equivalent if their sets of models are identical, that is, if $Mod(\Sigma_1) = Mod(\Sigma_2)$. This equivalence is denoted as $\Sigma_1 \equiv \Sigma_2$. Partial models μ_1 and μ_2 are defined as disjoint if there exists a literal ℓ in μ_1 such that its negation $\sim\ell$ is present in μ_2. $Mod_p(\Sigma)$ is termed a shorten representation of the solution set $Mod(\Sigma)$ if, for every pair of partial models μ_1 and μ_2 in $Mod_p(\Sigma)$, they are disjoint. Additionally, for every model μ in $Mod(\Sigma)$, there exists a partial model μ' in $Mod_p(\Sigma)$ such that μ' is a subset of μ. Unlike $Mod(\Sigma)$, $Mod_p(\Sigma)$ is not necessarily unique. Additionally, considering $Mod_p(\Sigma) = Mod(\Sigma)$ can also be viewed as a concise representation of the solution set.

A CNF formula Σ is a conjunction of clauses, where a clause is a disjunction of literals. Every CNF is viewed as a set of clauses, and every clause is viewed as a set of literals.

Example 1. *Let* $\Psi = \{a \vee b, \neg a \vee \neg b, c \vee b\}$ *be a* CNF *formula.* $Var(\Psi) = \{a, b, c, d\}$ *and the complete models of* $Mod(\Psi)$ *are:*

$$\begin{array}{ccc} \{a, \neg b, c, d\} & \{\neg a, b, c, d\} & \{a, \neg b, c, \neg d\} \\ \{\neg a, b, c, \neg d\} & \{a, \neg b, \neg c, d\} & \{\neg a, b, \neg c, d\} \end{array}$$

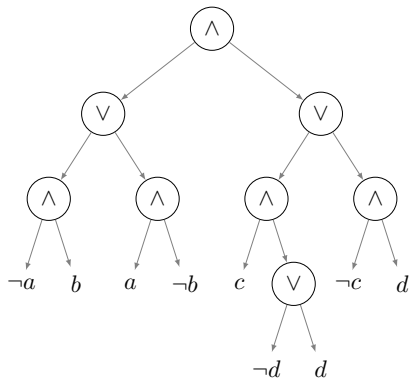
$Mod_p(\Sigma)$ *given by the following set of partial models:*

$$\{a, \neg b, \neg c, d\} \quad \{a, \neg b, c\} \quad \{\neg a, b, \neg c, d\} \quad \{\neg a, b, c\}$$

is a possible concise representation of $Mod(\Psi)$.

d−DNNF (deterministic Decomposable Negation Normal Form) consists of Boolean circuits (Vollmer 1999) with a single output, which serves as its root. It can be conceptualized as a rooted Directed Acyclic Graph (DAG), denoted as $\langle V, E\rangle$, 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 $N = \wedge(N_1, \ldots, N_k)$, no common variable is shared between the sub-circuits rooted at N_i and N_j for all $i \neq j$. In a deterministic gate of the form $N = \vee(N_1, \ldots, N_k)$, the sub-circuits rooted at N_i and N_j are jointly inconsistent for all $i \neq j$. The size of a d−DNNF $\Sigma = \langle V, E\rangle$, denoted by $|\Sigma|$ is its number of edges $|E|$. d−DNNF is universal, as it can accommodate every propositional theory (Darwiche 2002).

Example 2 (Example 1 cont'ed)**.** *Let us consider the* CNF *formula* Ψ *given in Example 1, the following* d-DNNF Σ *is equivalent to* Ψ*:*



decision-DNNF (decision Decomposable Negation Normal Form) is defined similarly, but with decision gates of the form $N = ite(x, N_1, N_2)$ replacing deterministic $\vee$ gates. Here, x is the decision variable at gate N, absent in the sub-circuits N_1 or N_2, and ite is a ternary connective denoting "if ...then ...else ...". decision-DNNF representations, also termed decomposable decision graphs (Fargier and Marquis 2006), can be converted into specific d-DNNF representations in linear time. By replacing a decision node of the form $N = ite(x, N_1, N_2)$ in a decision-DNNF representation with $N = (\neg x \wedge N_1) \vee (x \wedge N_2)$, the resulting d-DNNF representation maintains decomposable $\wedge$ nodes (as x appears neither in N_1 nor in N_2) and a deterministic $\vee$ node (since $(\neg x \wedge N_1) \wedge (x \wedge N_2)$ is inconsistent).

d-DNNF serves as a compelling language of representation due to its ability to efficiently handle various queries and transformations, such as satisfiability and conditioning in polynomial time. Notably, the models of a smooth d-DNNF can be enumerated in polynomial time in the number of its models (Darwiche and Marquis 2002). A d-DNNF satisfies the *smoothness* property if each disjunct of every disjunction node in the d-DNNF references the same variables. In other words, if $N_1, \ldots, N_m$ are the children of a disjunction node N, then $Var(N_i) = Var(N_j)$ for $i \neq j$. For example, in Example 2, the d-DNNF formula is smooth. Any d-DNNF Σ can be smoothed in $O(Var(\Sigma) \times |\Sigma|)$ time (Darwiche 2001).

It is noteworthy that the enumeration algorithm is characterized by polynomial delay, signifying that the time between the output of any two consecutive models is bounded by a polynomial function of the input size, in the worst-case scenario.

The baseline algorithm, which is described in Algorithm 1, leverages the polynomial-time satisfiability and conditioning properties of a d-DNNF Σ to enumerate its solutions. It takes a d-DNNF Σ and μ, a partial assignment representing the partial model under construction, as parameters. The algorithm proceeds recursively, beginning with a check for unsatisfiability of Σ. If Σ is unsatisfiable, the empty set is return (line 1). If Σ evaluates to $\top$, μ is considered a partial model of the initial formula and it is returned (line 2). Otherwise, a variable is selected from $Var(\Sigma)$ (line 4), and the algorithm is recursively called (line 5), considering the formula $\Sigma[x]$, where x is assigned true (expanding μ with x), and the formula $\Sigma[\neg x]$, where x is assigned false (expanding μ with $\neg x$).

Algorithm 1: baseline-enum

Input: a d-DNNF Σ, μ a partial assignment
Data: Δ the collected set of models.

1 **if** $\Sigma \equiv \bot$ **then return** $\emptyset$
2 **if** $\Sigma \equiv \top$ **then** $\{\mu\}$
3 **else**
4 Let $x \in Var(\Sigma)$
5 **return** baseline-enum$(\Sigma[x], \mu \cup \{x\}) \cup$ baseline-enum$(\Sigma[\neg x], \mu \cup \{\neg x\})$

This algorithm has a complexity of $O(|Var(\Sigma)| \times |\Sigma| \times |Mod(\Sigma)|)$, which is generally superior to the complexity of the algorithm proposed in (Darwiche 1998), which is $O(|\Sigma| \times |Mod(\Sigma)|^2)$. Unfortunately, it does not perform well in practice, as it requires frequent querying and transformation of the d-DNNF formula under consideration. Subsequently, we introduce an algorithm designed to enumerate the models of a d-DNNF in polynomial time with respect to the number of models and polynomial space with respect to the size of the d-DNNF. Importantly, this algorithm exhibits better practical performance as it operates directly on the circuit.

3 Enumerating Disjoint Partial Models using Decision-DNNF

In the subsequent discussion, we set aside trivial cases of d-DNNF ($\bot$ and $\top$), as the task of enumerating models poses no challenge for them. For simplicity, our enumeration algorithms assume that the d-DNNF circuits conform to certain conditions: all internal nodes have exactly two children, and no free variables are present (a variable x is considered free if $x \notin Var(\Sigma)$). Additionally, the children of internal nodes are ordered, and it is possible to retrieve the left (resp. right) child of a node n using the function left(n) (resp. right(n)). We further assume that our circuits have been simplified, meaning that all constants have been propagated in the gates ($\Delta \vee \bot = \Delta \wedge \top = \Delta$, $\Delta \vee \top = \top$, and $\Delta \wedge \bot = \bot$). This simplification process can be efficiently performed in a bottom-up manner. The presence of the constant $\top$ (resp. $\bot$) in the d-DNNF occurs only when the formula is equivalent to $\top$ (resp. $\bot$).

Before delving into general d-DNNFs, let us focus on smooth d-DNNFs, where the children of disjunction nodes share identical variables. This characteristic enables the computation of full models. We will relax this constraint in a second step. Let us start by noting that the set of models of a smooth d-DNNF can be simply constructed using a recursive bottom-up algorithm. For any given node, its set of models can be straightforwardly defined with respect to the variables involved in its descendants:

- If the node is labeled with a literal, that literal stands as its sole model (a trivial case);
- If the node is a conjunction node, its disjoint models encompass all possible combinations of a model from its left child and a model from its right child. This is facilitated by the decomposability property of the node, which guarantees the consistency of each such combination;
- If the node is a disjunction node, its set of disjoint models comprises the union of the disjoint models of its children. This is facilitated by the determinism property, which implies that combining models of the two children results in inconsistency, and by the smoothness property, which ensures that all variables in the descendants are involved in both children (thus, no free variables exist).

Leveraging this algorithm at the root of a formula yields the set of models for that formula. Remarkably, when seeking a single partial model, it suffices to set the algorithm associated with disjunction nodes such that it returns the model from one of its children:

- Leaves (literals) trivially define a single model;
- Conjunction nodes with children, each having a single model, will yield a single concatenation of models;
- Disjunction nodes return a single model, meaning only one descendant is considered for these nodes, as specified previously.

Obviously enough, the model that is returned depends on the choice of the partial models that are returned by the disjunction node.

While demonstrating such an algorithm in a bottom-up fashion is straightforward, it is impractical for real-world applications. This is because it necessitates memoizing all models, which cannot guarantee that the algorithm operates within space constraints bounded by the size of the d−DNNF circuit. Employing a top-down algorithm like Depth First Search (DFS) is also viable. Here is how it works: initiate the search from the root, propagate it to both children if the node is a conjunction, and to a single node if it is a disjunction. Every literal encountered during this traversal is a constitute of the model. The trace left by this DFS algorithm, which computes a model, can be viewed as a sub-graph of the d−DNNF, termed a *model graph*.

Definition 1 (model graph)**.** *Let $\Sigma = \langle V, E\rangle$ be a smooth* d−DNNF *rooted at $v_{root} \in V$. A model graph $\omega = \langle V', E'\rangle$ is a subgraph of Σ which is also a DAG, rooted at v_{root}, such that for each $v \in V'$:*

- *let $E_v \subseteq E$ be the set of edges which source is v: $E_v = \{e_i = (v, v_i)$ s.t. $(v, v_i) \in E\}$;*
- *if v is labelled with $\wedge$, then $E_v \subseteq E'$ and $\{v_i\} \subseteq V'$;*
- *if v is labelled with $\vee$, then there exists exactly one $(v, v_i) \in E_v$ such that $E_v \cap E' = \{(v, v_i)\}$, $v_i \in V'$ and for all $(v, v_j) \in (E_v \setminus \{(v, v_i)\})$, we have $v_j \notin V'$.*

Interestingly, the number of children the nodes of a model graph admits only depends on their labels. Nodes labelled with $\wedge$ have exactly two children, the ones labelled with $\vee$ exactly one, while the ones labelled with literals have none.

Algorithm 2: `build-model`

Input : n, the root node of the model graph $\omega = \langle V, E\rangle$
Output: the model associated with ω

```
   // leaf node
1  if {(n, n') ∈ E} = ∅ then return {label(n)}

   // internal node
2  Δ ← ∅
3  for (n, n') ∈ E do
4      Δ ← Δ ∪ build-model(n', ω)
5  return Δ
```

This is consistent with the notion of model: since disjunctions have a single child, they can be replaced by this child, letting the formula contain only conjunctions and literals, making it a term. This structure, in conjunction with the property of decomposition of the conjunction nodes implies that model graphs are not general graphs, but trees.

Proposition 1. *A model graph is a tree.*

Proof. Reductio ad absurdum. Let us suppose a model graph is not a tree. Then, there exists a node v such that this node has at least two parent nodes. Since the only nodes with at least two children are conjunction nodes, then there exists a $\wedge$-labelled node such that v is in descendants of both the children of the conjunction node. This implies that both children of the conjunction node share variables, which is incompatible with the decomposability property. □

This ensures the computation of a model does not need a number of steps larger than the size of the d−DNNF.

Corollary 1. *Let Σ be a smooth* d−DNNF*. Computing a model graph of Σ can be achieved in time linear to the size of the formula.*

Example 3 (Example 2 cont'ed)**.** *Let us examine the* d−DNNF *Σ constructed in Example 2, representing the CNF formula Ψ from Example 1. In Figure 1a, we present a model graph extracted from Σ, where bold edges directly correspond to the graph's representation within Σ.*

Algorithm 2 facilitates the retrieval of a model associated with a model graph ω rooted at node n, operating in a recursive manner. If n represents a literal, identified by being a leaf node with a label distinct from $\vee$ and $\wedge$, then the model containing this literal is returned (line 1). However, if n is an internal node, we iteratively explore its children to construct the associated models (lines 2–4). Initially, the variable Δ is initialized as an empty set (line 2). Subsequently, for each $(n, n') \in E$, we recursively call the `build-model` function, considering n' as the next node. It is important to note that if `label`(n) represents an or-node, only one such n' exists that satisfies the condition. Conversely, if `label`(n) signifies an and-node, exactly two such n' nodes exist that meet the condition. The results of these recursive calls are aggregated into Δ and returned (line 5).

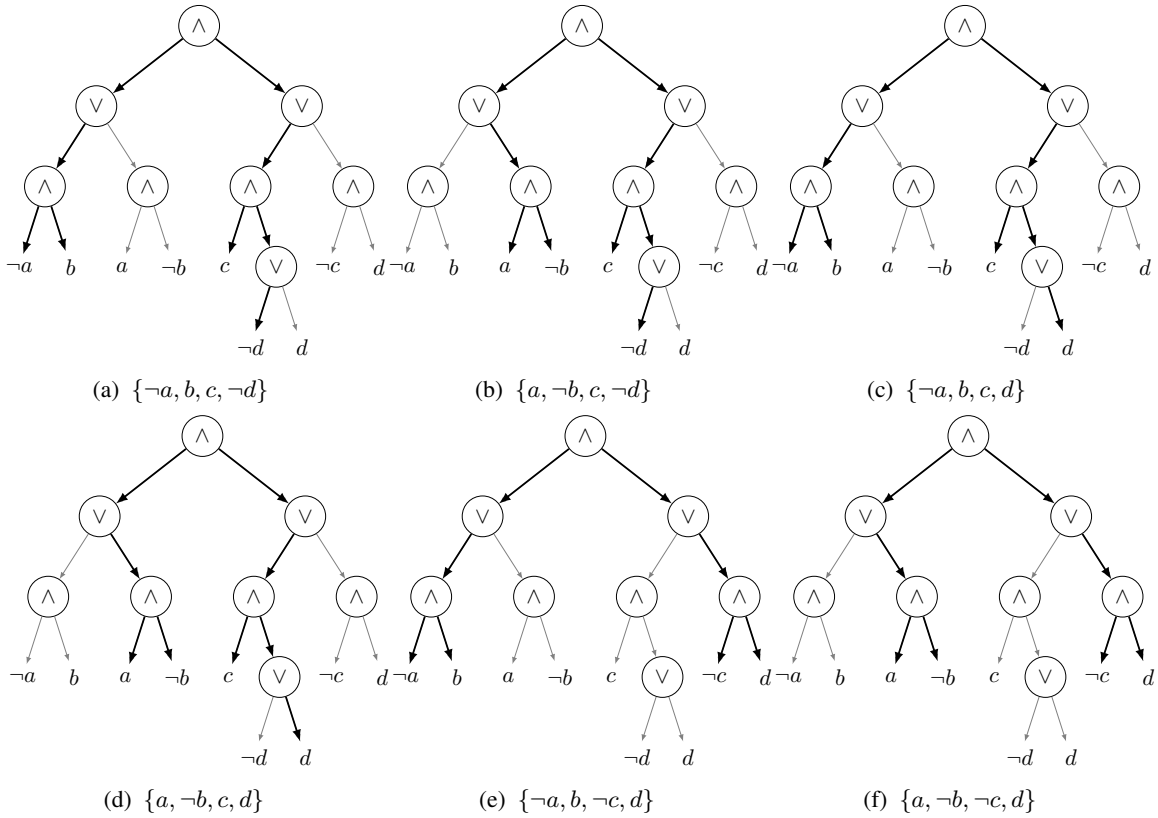


Figure 1: Enumeration of the (ordered) model graphs of a smooth d-DNNF, and their correspondence in term of models.

Example 4 (Example 2 cont'ed)**.** *Figure 1 depicts all the possible model graphs corresponding to* Σ *from Example 2. Each sub-figure corresponds to a model of* Σ*.*

Our goal is not solely to identify a single model of the formula, but rather to systematically enumerate all models in a disjoint manner. To accomplish this, we leverage the following proposition, which establishes a one-to-one correspondence between the model graphs of smooth d−DNNF formulas and their respective models.

Proposition 2. *Let* Σ *denote a smooth* d−DNNF*. There exists a one-to-one correspondence between the model graphs of* Σ *and its models.*

Proof. Every model of Σ can be computed using the DFS algorithm presented above by selecting the disjunction node children that are consistent with the model; thus each model has its associated model graph. Every model tree leads to a single model, since it gives the trace of the DFS algorithm, including which children have been selected; the model graphs are also mapped to a single model. Since the disjunction nodes are deterministic, we can be confident that two distinct model graphs correspond to two distinct models of Σ. □

Before proceeding further, let us define the function sub, which, given a graph $\mathcal{G} = \langle N, E\rangle$ and a node $n \in N$, returns the sub-graph $\mathcal{G}' = \langle N', E'\rangle$ of $\mathcal{G}$ such that $\mathcal{G}'$ contains all nodes reachable from n, denoted as $N' = \{n\} \cup \{n_p \in N \mid \text{there exists a directed path between } n \text{ and } n_p\}$, and includes all edges that connect nodes in N', denoted as $E' = \{(n_i, n_j) \in E \mid n_i \in N' \text{ and } n_j \in N'\}$. Given two graphs $\mathcal{G} = \langle N, E\rangle$ and $\mathcal{G}' = \langle N', E'\rangle$, we define the union of the two graphs as $\mathcal{G}'' = \langle N \cup N', E \cup E'\rangle$.

To enumerate all models in a disjoint manner, we introduce an approach that traverses through all model graphs of a d−DNNF formula Σ, represented as a DAG rooted at node n. The approach involves ordering the model graphs and then iterating through them according to this order. First, we define two functions, begin and end, which, given a d−DNNF rooted at node n, determine the first and last model graphs in this order, respectively. These functions are defined recursively based on the type of node under consideration. If n is a leaf node, both $\texttt{begin}(\Sigma)$ and $\texttt{end}(\Sigma)$ return $\langle n, \emptyset\rangle$, as a leaf node has only one model graph. If n is an or-node, we have $\texttt{begin}(\Sigma) = \langle\{n\}, \{(n, \texttt{left}(n))\}\rangle \cup \texttt{begin}(\texttt{sub}(\Sigma, \texttt{left}(n)))$ and $\texttt{end}(\Sigma) = \langle\{n\}, \{(n, \texttt{right}(n))\}\rangle \cup \texttt{end}(\texttt{sub}(\Sigma, \texttt{right}(n)))$. It is worth noting that the first model graph is arbitrarily chosen as the leftmost one anchored with respect to the or-node. Finally, if n is an and-node, $\texttt{f}(n)$ returns $\langle\{n\}, \{(n, \texttt{left}(n)), (n, \texttt{right}(n))\}\rangle \cup \texttt{f}(\texttt{sub}(\Sigma, \texttt{left}(n)) \cup \texttt{f}(\texttt{sub}(\Sigma, \texttt{right}(n))$, with $\texttt{f} \in \{\texttt{begin}, \texttt{end}\}$.

Example 5 (Example 2 cont'ed)**.** *Given the* d−DNNF Σ *constructed in Example 2, we observe that Figure 1a represents* $\texttt{begin}(\Sigma)$*, while Figure 1f represents* $\texttt{end}(\Sigma)$*.*

Since $\Sigma = \langle N, E \rangle$ is a DAG, computing $\texttt{begin}(n)$ and $\texttt{end}(n)$ for a node $n \in N$ can be done in linear time relative to the size of the sub-graph rooted at n.

Given the definitions of the first and last model graphs, we can now define the function `next`. This function, given a d-DNNF $\Sigma = \langle N, E \rangle$ and a model graph $\omega = \langle N_\omega, E_\omega \rangle$ of Σ rooted at n, returns the next model if it exists, or $\texttt{end}(n)$ if ω is the last model. Let us explore the different scenarios based on the type of node n. If n is a leaf node, the next model after ω is ω itself, as $\texttt{end}(n) = \langle \{n\}, \emptyset \rangle = \omega$.

If n is an or-node and $(n, m) \in E'$, we need to consider two scenarios: whether $\texttt{end}(\texttt{sub}(\Sigma, m))$ is equal to $\texttt{sub}(\omega, m)$ or not. If $\texttt{end}(\texttt{sub}(\Sigma, m)) \neq \texttt{sub}(\omega, m)$, there are still models to enumerate in the sub-graph associated with m, and we can find the next model by recursively calling the `next` function on $\texttt{sub}(\omega, m)$. Otherwise, if $\texttt{end}(\texttt{sub}(\Sigma, m)) = \texttt{sub}(\omega, m)$, we need to determine whether $m = \texttt{left}(n)$ or $m = \texttt{right}(n)$. If $m = \texttt{left}(n)$, all models in the left branch have been explored, and we can start enumerating models from the right branch, starting with $\texttt{begin}(\texttt{sub}(\Sigma, \texttt{right}(n)))$. Finally, if $m = \texttt{right}(n)$, we have reached the end, and $\texttt{next}(\omega)$ returns ω.

Let us examine the scenario where n is an and-node. If $\texttt{end}(\texttt{sub}(\Sigma, \texttt{left}(n))) \neq \texttt{sub}(\omega, \texttt{left}(n))$, there are still models to enumerate on the left side of the and-node. Therefore, we recursively call `next` on $\texttt{sub}(\omega, \texttt{left}(n))$ to find the next model. If $\texttt{end}(\texttt{sub}(\Sigma, \texttt{left}(n))) = \texttt{sub}(\omega, \texttt{left}(n))$, it means that all models of the sub-graph rooted at $\texttt{left}(n)$ have been enumerated. In this scenario, two possibilities arise depending on whether $\texttt{end}(\texttt{sub}(\Sigma, \texttt{right}(n)))$ is equal to $\texttt{sub}(\omega, \texttt{right}(n))$ or not. If $\texttt{end}(\texttt{sub}(\Sigma, \texttt{right}(n)))$ is not equal to $\texttt{sub}(\omega, \texttt{right}(n))$, there are still models to enumerate on the right side. In this case, the next model graph will consider the first model on the left, i.e., $\texttt{begin}(\texttt{left}(n))$, and the next model on the right side of the and-node, i.e., $\texttt{next}(\texttt{sub}(\omega, \texttt{right}(n)))$. Finally, if $\texttt{end}(\texttt{sub}(\Sigma, \texttt{right}(n))) = \texttt{sub}(\omega, \texttt{right}(n))$ then ω is the last model, and $\texttt{next}(\omega)$ returns ω.

Algorithm 3 delineates the function `next` as previously outlined. It is a recursive procedure that operates on a d-DNNF $\Sigma = \langle N, E \rangle$ and a model graph $\omega = \langle N_\omega, E_\omega \rangle$ rooted at n of Σ, aiming to determine the subsequent model graph following ω. The algorithm first handles the base case where $\omega = \texttt{end}(\Sigma)$, encompassing scenarios where n is a leaf. In such instances, the function returns ω.

Subsequently, it addresses the or-node case (lines 2–6). Initially, it evaluates whether there are remaining models to enumerate, signaled by $\texttt{end}(\texttt{sub}(\Sigma, m)) \neq \texttt{sub}(\omega, m)$ (line 4). If so, the function returns a model graph rooted at n, linked to the subsequent model graph of its child, i.e., $\texttt{next}(\texttt{sub}(\omega, m))$ with $(n, m) \in E_\omega$ (line 5). If $\texttt{end}(\texttt{sub}(\Sigma, m)) = \texttt{sub}(\omega, m)$, and $\texttt{left}(n)$ belongs to N_ω, it implies that the left branch has been exhausted, and the function returns the model graph rooted in n, linked to the first model graph of the sub-graph obtained from $\texttt{right}(n)$, i.e. $\texttt{begin}(\texttt{sub}(\Sigma, \texttt{right}(n)))$. The scenario where $\texttt{right}(n)$ belongs to N_ω and $\texttt{end}(\texttt{sub}(\Sigma, m)) = \texttt{sub}(\omega, m)$ cannot transpire. In such an event, $\omega = \texttt{end}(\Sigma)$, and this condition is already addressed in line 1.

Algorithm 3: `next`

Input : a d-DNNF $\Sigma = \langle N, E \rangle$ and a model graph $\omega = \langle N_\omega, E_\omega \rangle$ rooted at n of Σ
Output: the next model graph that comes just after ω

```
// leaf node and case where ω = end(Σ)
1  if ω = end(Σ) then return ω
2  if n is an or-node then
3      Let m s.t. (n, m) ∈ E_ω
4      if end(sub(Σ, m)) ≠ sub(ω, m) then
5          return ⟨{n}, {(n, m)}⟩ ∪ next(sub(ω, m))
6      return
           ⟨{n}, {(n, right(n))}⟩ ∪ begin(sub(Σ, right(n)))
7  if n is an and-node then
8      if end(sub(Σ, left(n))) ≠ sub(ω, left(n)) then
9          return ⟨{n}, {(n, left(n)), (n, right(n))}⟩ ∪
             next(sub(ω, left(n))) ∪ sub(ω, right(n))
10     return ⟨{n}, {(n, left(n)), (n, right(n))}⟩ ∪
         begin(sub(Σ, left(n))) ∪ next(sub(ω, right(n)))
```

Finally, the and-node case is handled (lines 7–10). It first determines if there are remaining models to enumerate in the left branch. If so (lines 8–9), the function returns the model graph, preserving everything except the segment of the sub-graph obtained from $\texttt{left}(n)$, replaced by the subsequent model graph, i.e., $\texttt{next}(\texttt{sub}(\omega, \texttt{left}(n))$. If there are no more models in the left part, the function returns the model graph rooted at n, linking to the initial model graph of the sub-graph from $\texttt{left}(n)$, i.e., $\texttt{begin}(\texttt{sub}(\Sigma, \texttt{left}(n)))$, and the subsequent model graph of the sub-graph from $\texttt{right}(n)$, i.e., $\texttt{next}(\omega, \texttt{right}(n))$. Similar to the or-node case, it is unnecessary to consider the scenario where $\texttt{sub}(\omega, \texttt{right}(n)) = \texttt{end}(\Sigma, \texttt{right}(n))$ as it is covered by the case addressed in line 1.

Example 6 (Example 2 cont'ed)**.** *With the smooth d-DNNF Σ constructed in Example 2, Figure 3 presents all the model graphs in the sequence they are generated using the function* `next`*, commencing with* $\texttt{begin}(\Sigma)$ *(1a) and concluding with* $\texttt{end}(\Sigma)$ *(1f).*

In this version, Algorithm 3 operates in quadratic time relative to $|\Sigma|$. To enhance efficiency, we can precompute a boolean flag for each node n in ω, updating them whenever ω changes. These flags indicate whether a descendant of n branches on a left node. With this preprocessing step, checking $\texttt{end}(\texttt{sub}(\Sigma, m)) \neq \texttt{sub}(\omega, m)$, with $(n, m) \in E_\omega$, becomes a constant-time operation. The subsequent proposition shows that the function `next` can run in linear time relative to the size of the circuit provided as a parameter.

Proposition 3. *When invoking* $\texttt{next}(\omega)$ *with a smooth* d-DNNF $\Sigma = \langle N, E \rangle$ *and a model graph* $\omega = \langle N_\omega, E_\omega \rangle$ *of* Σ *rooted at* n *the operation runs in linear time relative to* $|\Sigma|$.

Proof. First, let us recall that the function `begin` operates in linear time relative to the size of Σ. Now, suppose we have

Algorithm 4: `model-graph enumerator`

Input : Σ a smooth `d-DNNF`
Output: $Mod(\Sigma)$

1 $\Delta \leftarrow \emptyset$
2 $mg \leftarrow \texttt{begin}(\Sigma)$
3 **while** $mg \neq \texttt{end}(\Sigma)$ **do**
4 $\quad \Delta \leftarrow \Delta \cup \{\texttt{build-model}(mg)\}$
5 $\quad mg \leftarrow \texttt{next}(mg)$
6 **return** $\Delta \cup \{\texttt{build-model}(mg)\}$

a constant-time oracle for checking if a model graph has a successor, i.e., it contains at least one or-node branching on a left node. The number of operations needed to complete the call to the function `next` depends on the recursive calls made in lines 5, 9, and 10. Since in all cases `next` is called with a sub-graph of ω, we can be certain that it is not possible to visit more nodes than those in ω. □

Now, equipped with the means to enumerate the models of a `d-DNNF` by leveraging its representation as a graph, Algorithm 4 outlines the process for enumerating all disjoint models of a smooth `d-DNNF`. It initializes Δ to accumulate models (line 1) and mg to represent the first model graph of Σ (line 2). Then, as long as mg is not the last model graph of Σ, it is added to Δ (line 4), and the next model graph is obtained and stored in mg using the `next` function (line 5). Once all model graphs have been visited in the while loop, the last model is added to Δ and Δ is returned (line 6).

Now, let us establish that Algorithm 4 is sound, complete, and terminates within $O(|\Sigma| \times Mod(\Sigma))$ time.

Proposition 4. *Given a smooth* `d-DNNF` *formula Σ, Algorithm 4 enumerates all models of Σ in a disjoint manner. It concludes within $O(|\Sigma| \times Mod(\Sigma))$ time.*

Proof. First, let us show that Algorithm 4 enumerates all models of Σ in a disjoint manner. The proof is by induction on the number k of nodes of Σ. First assume $k = 1$, that is Σ is a literal ℓ (let us recall that we left aside the trivial cases where Σ is a constant). In this case, Algorithm 4 returns the sole model $\{\ell\}$, establishing the base case of the induction.

Let us consider $\Sigma = \Sigma_1 \; op \; \Sigma_2$, a `d-DNNF` rooted at n, where $op \in \{\wedge, \vee\}$ and both Σ_1 and Σ_2 are smooth `d-DNNF`s, with $|\Sigma_1| \leq k$ and $|\Sigma_2| \leq k$. When $op = \vee$, in Algorithm 4, mg initially corresponds to the model graph linking n with the first model graph of Σ_1 rooted at m_1, obtained with $\texttt{begin}(\Sigma_1)$. Then, while $mg \neq \texttt{end}(\Sigma)$, `next` is called on mg and it is updated accordingly. Now, let us delve into the `next` function outlined in Algorithm 3. As Σ is rooted at an or-node, we focus on lines 2–6. Here, as long as not all model graphs of Σ_1 have been considered (line 4), `next` is called on the model graph of Σ_1. By induction hypothesis, iterative calls to `next` on Σ_1 will visit all its model graphs disjointly, thereby adding all models of Σ_1 to Δ.

Once all models of Σ_1 have been considered, the condition in line 4 of Algorithm 3 becomes false, triggering the execution of line 6. This returns the model graph linking n with the first model graph of Σ_2 rooted at m_2. Similar to Σ_1, all models of Σ_2, except $\texttt{end}(\Sigma_2)$, which is added to Δ in line 6 of Algorithm 4, will be added to Δ. At the end of Algorithm 4 (line 7), Δ contains two sets of disjoint models, Δ_1 and Δ_2, representing respectively the models of Σ_1 and Σ_2. Or-nodes of Σ being deterministic, Δ_1 and Δ_2 are disjoint, and thus Δ consists of all disjoint models of Σ.

Now, let us explore the case where $op = \wedge$. Since Σ is a `d-DNNF`, and-nodes are decomposable. To enumerate models of Σ, it suffices to associate each model of Σ_2 with all models of Σ_1, i.e., $Mod(\Sigma) = \{m \times Mod(\Sigma_1) \mid m \in Mod(\Sigma_2)\}$. Let us demonstrate that iteratively calling `next` on mg achieves precisely this. Initially, mg links n with the first model graphs of both Σ_1 and Σ_2, obtained with `begin`. As n is an and-node, we focus on lines 7–10 in Algorithm 3. By calling `next` on $\texttt{sub}(\omega, m_1)$, all models Δ_1 of Σ_1 are enumerated due to the induction hypothesis. Then, lines 8–9 ensure that each model of Σ_1 is associated with a specific model of Σ_2. As Δ_1 consists of disjoint models, the resulting models, concatenated with a specific model of Σ_2 and added to Δ (line 4 in Algorithm 4), remain disjoint. Once this is done for a particular model of Σ_2, line 8 is falsified, and line 10 is executed. This assigns the next model graph linking n with the first model graph of Σ_1 and the next model graph of Σ_2, obtained with `next` on $\texttt{sub}(\omega, m_2)$. This process continues until $\texttt{sub}(\omega, m_2) = \texttt{end}(\texttt{sub}(\Sigma, m_2))$, i.e., until $\omega = \texttt{end}(\Sigma)$. By the induction hypothesis, all models Δ_2 of Σ_2 are covered. Moreover, as models of Δ_2 are disjoint, the newly added models in Δ will also remain disjoint from those added so far.

Finally, let us show that Algorithm 4 concludes within $O(|\Sigma| \times Mod(\Sigma))$ time. As each call of `next` takes time linearly bounded by the size of $|\Sigma|$, it follows directly that Algorithm 4 operates within $O(|\Sigma| \times Mod(\Sigma))$ time. □

So far, we have outlined an algorithm for enumerating complete models of a smooth `d-DNNF`. Interestingly, employing the same algorithm on general `d-DNNF` iterates over partial models of the formula. Following the same rationale as in the proof of Proposition 4, we can demonstrate that all partial models enumerated by Algorithm 4 are disjoint. Furthermore, any model of the formula can be constructed by extending one of these partial models.

Indeed, without the smoothness property, the children of disjunction nodes may involve different sets of variables. In other words, some variables may be missing in one child or in another; these are free variables, for which any literal can be added to form a model. Considering our model graphs, these variables cannot be brought by other nodes: if they were brought by an ancestor conjunction node (the only kind with an arity of more than one), then switching the child of the non-smooth disjunction node would break the decomposability property of this conjunction node.

To conclude, let us note that the worst-case time complexity of $O(|\Sigma| \times Mod(\Sigma))$ rarely occurs. Typically, the function `next` operates in a time complexity that is at worst proportional to the longest path in the DAG. Since `d-DNNF`s are predominantly comprised of or-nodes, traversal of the DAG tends to focus on a single path.

Additionally, it is worth mentioning that the binary node condition is chosen here for simplicity of the presentation. In practice, this constraint is not necessary. If we relax this condition and aggregate nodes of the same type, such as $\Sigma_1 \wedge (\Sigma_2 \wedge \Sigma_3)$ into $\Sigma_1 \wedge \Sigma_2 \wedge \Sigma_3$, while still ensuring constants to be absent in Σ, it is easy to demonstrate that Algorithm 4 operates in $O(2 \times |Var(\Sigma)| \times Mod(\Sigma))$. Indeed, if all these conditions are adhered to, the model graphs can only contain a number of nodes that is less than twice the number of propositional variables of Σ.

4 Empirical Evaluation

We have realized all the concepts elucidated in this paper into a software tool named `model-graph`. Developed in Rust, the tool's library and its documentation is accessible at https://crates.io/crates/decdnnf_rs. It accommodates various output modes: `quiet` (which solely touches discovered solutions without outputting them), `compact` (which provides partial model output), and `full` (which outputs complete models). Extensive testing has been conducted, achieving a code coverage of 96.9% as measured by kcov for the library. Additionally, fuzz testing has been performed for the binaries.

To systematically enumerate all partial solutions of a given `CNF` formula Σ, we employ a two-step procedure referred to as `d4+model-graph`. It first compiles Σ into a `decision-DNNF` Γ utilizing the knowledge compiler `d4` (available at https://github.com/crillab/d4v2), leveraging preprocessing techniques that preverse equivalence, including backbone simplification and CNF simplification (vivification and occurrence elimination)(Lagniez and Marquis 2014; Lagniez and Marquis 2017b). Subsequently, Γ is passed to `model-graph` to enumerate its partial models. To assess the efficacy of employing `model-graph` over the baseline approach for enumerating partial models from `decision-DNNF`, we conduct a comparative analysis with the baseline method termed `d4+baseline`. Our evaluation involves a comparison between `d4+model-graph` and state-of-the-art approaches such as `BDD` (https://www.disc.lab.uec.ac.jp/toda/code/cnf2obdd.html) and `TabularAllSAT` (https://zenodo.org/records/10397723), which have demonstrated superior performance over other methods in experiments detailed in (Spallitta, Sebastiani, and Biere 2024). It is worth noting that all solvers in our comparison solely identify discovered solutions without outputting them. Instead, each model is stored in an array, allowing users to retrieve and manipulate them as needed in their applications.

We examined a total of 2403 CNF instances, consisting of 1940 instances from previous enumeration studies (Spallitta, Sebastiani, and Biere 2024), 246 instances from the three most recent model counting competitions (https://mccompetition.org/), and 197 from the benchmark set used to evaluate the knowledge compiler d4 (Lagniez and Marquis 2017a). The first set is divided into four datasets: Binary Clauses (binary - 50 instances), random 3-SAT problems (rnd3sat - 410 instances), Random-3-SAT Instances with Controlled Backbone Size (CBS - 1000 instances), and Random-3-SAT Instances and Backbone-minimal Subinstances (BMS - 500 instances) (https://www.cs.ubc.ca/~hoos/SATLIB/). The instances from the model counting competitions, collected in the `competition` repository, were specifically chosen to be solvable within a 1-hour timeout and a 32GiB memory limit by `d4`. We also selected benchmarks with model counts requiring 20 digits or fewer for representation. From the `compilation` repository in (Lagniez and Marquis 2017a), we included 2 datasets: Planning (190 instances) and Qif (7 instances for Quantitative Information Flow analysis - security). All benchmarks are available at https://zenodo.org/records/11085774.

All experiments were conducted on a cluster equipped with quad-core bi-processors Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz and 128 GiB of memory, running CentOS 8 with Linux version 4.18.0-301.1.el8.x86_64 kernel. The compilers used was g++ version 13.2.0 and Rust 1.72.1. Hyperthreading was disabled, and no cache sharing between cores was permitted. To ensure a fair comparison, we set a timeout of 1200 seconds and a memory limit of 7.6 GiB, allowing other approaches the opportunity to manage benchmarks that exceeded these limits.

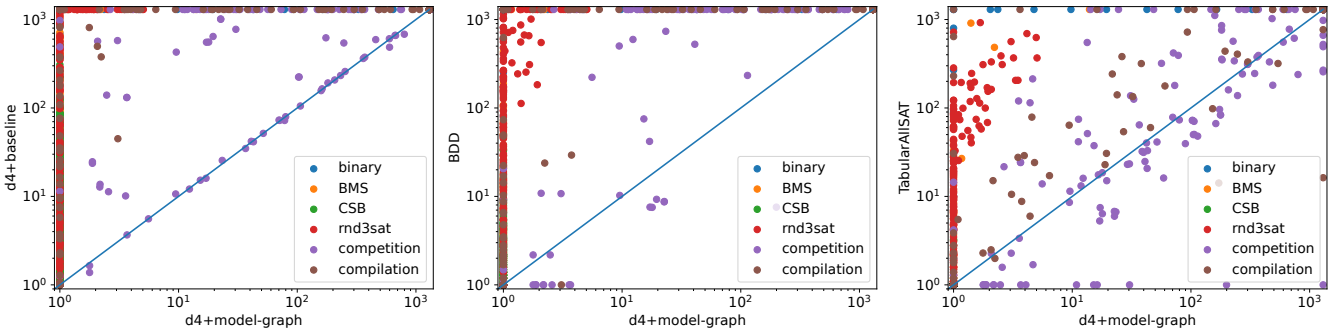
Table 1 presents the solver performance, indicating the number of instances solved by each solver within the specified timeout period. Here, 'solved' denotes the successful enumeration of a complete set of disjoint partial models covering all total models. Additionally, the table includes counts of timeouts and memory outs for each solver across different benchmark sets. Firstly, it is evident that `d4+model-graph` outperforms other solvers by solving the highest number of instances across all benchmark sets. Notably, it exhibits superior efficiency compared to the baseline approach, which proves to be the least effective in our experiments. Comparing against `BDD`, our approach significantly surpasses it, solving strictly more instances.

In comparison to `TabularAllSAT`, which closely resembles our approach in effectiveness, we observe a distinct advantage for `TabularAllSAT` that never experiences memory outs, making it favorable when memory resources are limited. Indeed, compiled-based approaches often need to store a representation of the input formula, which can grow exponentially in size compared to the original formula. In contrast, `TabularAllSAT` processes models individually, resulting in lower memory requirements. While it is feasible to design solvers that enumerate models with a memory footprint polynomial in relation to the input formula, `TabularAllSAT`'s clause learning feature may occasionally exceed this polynomial limit in practice, although this is rare.

For the benchmarks from (Spallitta, Sebastiani, and Biere 2024), `TabularAllSAT` and `d4+model-graph` demonstrate almost equivalent effectiveness. However, across the other benchmark sets, `d4+model-graph` exhibits slightly higher efficiency. The Virtual Best Solver (VBS), a hypothetical algorithm that always selects the best solver from the tested pool, solves 2271 instances. The primary contributors to the VBS are `TabularAllSAT` and our approach. Notably, our method independently solves an additional 29 instances.

	TabularAllSAT	BDD	d4+model-graph	d4+baseline
Dataset	#solve (#TO, #MO)	#solve (#TO, #MO)	#solve (#TO, #MO)	#solve (#TO, #MO)
binary (50)	21 (29, 0)	21 (3, 26)	**33** (17, 0)	18 (32, 0)
BMS (500)	499 (1, 0)	485 (0, 15)	**500** (0, 0)	470 (30, 0)
CSB (1000)	**1000** (0, 0)	999 (0, 1)	**1000** (0, 0)	**1000** (0, 0)
rnd3sat (410)	**410** (0, 0)	381 (22, 7)	**410** (0, 0)	308 (102, 0)
competition (246)	118 (128, 0)	37 (42, 167)	**124** (59, 63)	65 (123, 58)
compilation (197)	144 (53, 0)	85 (32, 80)	**155** (31, 11)	84 (102, 11)
Total (2403)	2192 (211, 0)	2008 (99, 296)	**2222** (107, 74)	1945 (389, 69)

Table 1: Table reporting the number of instances solved by each solver within the timeout time (1200 seconds).



(a) d4+baseline (b) BDD (c) TabularAllSAT

Figure 2: Scatter plots comparing CPU times of d4+model-graph with other solvers. Each dot represents an instance. The x-coordinate (resp. y-coordinate) shows the time (in seconds) for the solver on the x-axis (reps. y-axis). Both axes are log-scaled.

Figure 2 provides additional insights into the results presented in Table 1 by examining the runtime performance of the solvers compared to d4+model-graph. Each point represents an instance, with the time (in seconds) required to solve it using d4+model-graph (x-axis) and the time needed by other solvers (y-axis). To enhance readability, all instances solved in less than 1 second have been represented as solved within 1 second in the figures. Figures 2b and 2a clearly illustrate that d4+model-graph outperforms both BDD and d4+baseline in terms of runtime. In particular, the points on the left side of the graphs illustrate that d4+model-graph significantly outperforms these two competitors. Each dot signifies an instance that d4+model-graph solves in less than 1 second, while other approaches generally take much longer. The points at the top of the graphs clearly show that other methods frequently time out on the tested benchmarks.

When comparing with TabularAllSAT, Figure 2c illustrates that d4+model-graph demonstrates exceptional efficiency on the benchmarks from (Spallitta, Sebastiani, and Biere 2024). However, the results for the other two sets of benchmarks are somewhat mixed, particularly for the test set competition, where no single solver appears to consistently outperform the others. This observation can be attributed to the fact that, as noted in (Audemard, Lagniez, and Simon 2013), on certain instances containing few partial models, employing a SAT solver for model enumeration proves to be more efficient than using a compiler.

To compare the performance difference between d4+model-graph and TabularAllSAT, we present a scatter plot depicting the number of partial models computed by each approach. Each data point represents an instance, with the number of partial models enumerated by d4+model-graph plotted on the x-axis and the number enumerated by TabularAllSAT plotted on the y-axis. The plot clearly illustrates that d4+model-graph gener-

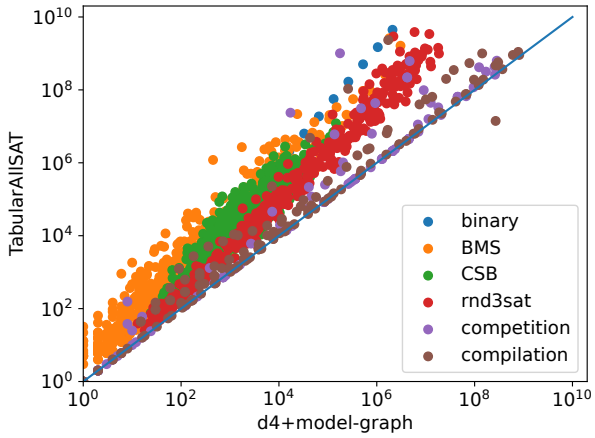


Figure 3: Scatter plot comparing number of partial models generated by d4+model-graph and TabularAllSAT.

ally requires fewer partial models to cover the entire search space of the formula. This observation can be attributed to the heuristic utilized by d4, which is specifically designed for knowledge compilation.

To conclude, it is worth noting that 99% of the benchmarks solved by d4+model-graph require a compilation time of less than one second. d4 successfully compiles 2386 benchmarks before the timeout, with 2300 taking less than 10 seconds. Memory outages do not always stem from d4, but can occur during the representation of the d-DNNF in the internal structure of model-graph. This issue could be mitigated by directly integrating our technique into d4, eliminating the need for disk I/O operations in the process.

5 Conclusion and Perspectives

In this paper, we introduce model-graph, an approach leveraging decision-DNNF compilation for enumerating disjoint partial models of propositional formulas. By harnessing the graph representation inherent in d-DNNFs, we devised a method capable of efficiently enumerating all models in a disjoint manner. Our experiments validate the efficacy of our two-step approach, which first compiles a CNF formula into d-DNNF and then enumerates partial models from the d-DNNF. Results demonstrate that our approach generally outperforms state-of-the-art methods, consistently delivering faster performance. Additionally, our approach typically generates a significantly smaller number of partial disjoint models compared to other competitors.

This work could be extended in several directions. Exploring parallelization techniques offers a promising avenue to accelerate the model enumeration process, especially for complex d-DNNFs. Utilizing multiple threads or processors can harness the power of modern computing architectures, potentially achieving a linear speedup. A notable drawback of our approach is that if the compilation phase fails, no partial models will be returned. To address this limitation, one interesting approch is to explore the possibility of enumerating partial models as they become available during the compilation phase. This proactive approach could mitigate the impact of compilation failures and enhance the robustness of the overall methodology. Finally, to further enhance our approach's performance, we plan to explore additional preprocessing techniques such as gate elimination (Lagniez and Marquis 2014) and definability (Lagniez, Lonca, and Marquis 2016; Lagniez, Lonca, and Marquis 2020; Lagniez and Marquis 2023). While preprocessing techniques that preserve equivalence can safely simplify the input formula, techniques like gate elimination or definability are more complex as they necessitate reconstructing each solution afterward. Solution reconstruction is polynomial for gate elimination but NP-hard for definability. Consequently, the feasibility of using such preprocessing techniques is uncertain and involves balancing the initial compilation time against the subsequent postprocessing required for solution reconstruction.

Acknowledgements

This work has benefited from the support of the AI Chair EXPEKCTATION (ANR-19-CHIA-0005-01) of the French National Research Agency.

References

Audemard, G.; Lagniez, J.; and Simon, L. 2013. Just-in-time compilation of knowledge bases. In Rossi, F., ed., *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 447–453. IJCAI/AAAI.

Boudane, A.; Jabbour, S.; Sais, L.; and Salhi, Y. 2017. Enumerating non-redundant association rules using satisfiability. In Kim, J.; Shim, K.; Cao, L.; Lee, J.; Lin, X.; and Moon, Y., eds., *Advances in Knowledge Discovery and Data Mining - 21st Pacific-Asia Conference, PAKDD 2017, Jeju, South Korea, May 23-26, 2017, Proceedings, Part I*, volume 10234 of *Lecture Notes in Computer Science*, 824–836.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers* 35(8):677–691.

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

Darwiche, A. 1998. Model-based diagnosis using structured system descriptions. *J. Artif. Intell. Res.* 8:165–222.

Darwiche, A. 2001. Decomposable negation normal form. *J. ACM* 48(4):608–647.

Darwiche, A. 2002. A compiler for deterministic, decomposable negation normal form. In Dechter, R.; Kearns, M. J.; and Sutton, R. S., eds., *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*, 627–634. AAAI Press / The MIT Press.

Fargier, H., and Marquis, P. 2006. On the use of partially ordered decision graphs in knowledge compilation and quantified boolean formulae. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, 42–47. AAAI Press.

Grumberg, O.; Schuster, A.; and Yadgar, A. 2004. Memory efficient all-solutions SAT solver and its application for reachability analysis. In Hu, A. J., and Martin, A. K., eds., *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*, 275–289. Springer.

Han, J.; Cheng, H.; Xin, D.; and Yan, X. 2007. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.* 15(1):55–86.

Jabbour, S.; Mhadhbi, N.; Raddaoui, B.; and Sais, L. 2022. A declarative framework for maximal k-plex enumeration problems. In Faliszewski, P.; Mascardi, V.; Pelachaud, C.; and Taylor, M. E., eds., *21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2022,*

Auckland, New Zealand, May 9-13, 2022, 660–668. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).

Jin, H.; Han, H.; and Somenzi, F. 2005. Efficient conflict analysis for finding all satisfying assignments of a boolean circuit. In Halbwachs, N., and Zuck, L. D., eds., *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, 287–300. Springer.

Khurshid, S.; Marinov, D.; Shlyakhter, I.; and Jackson, D. 2003. A case for efficient solution enumeration. In Giunchiglia, E., and Tacchella, A., eds., *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, 272–286. Springer.

Koriche, F.; Lagniez, J.; Marquis, P.; and Thomas, S. 2013. Knowledge compilation for model counting: Affine decision trees. In Rossi, F., ed., *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 947–953. IJCAI/AAAI.

Lagniez, J., and Marquis, P. 2014. Preprocessing for propositional model counting. In Brodley, C. E., and Stone, P., eds., *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, 2688–2694. AAAI Press.

Lagniez, J., and Marquis, P. 2017a. An improved decision-dnnf compiler. In Sierra, C., ed., *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 667–673. ijcai.org.

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

Lagniez, J., and Marquis, P. 2023. Boosting definability bipartition computation using SAT witnesses. In Gaggl, S. A.; Martinez, M. V.; and Ortiz, M., eds., *Logics in Artificial Intelligence - 18th European Conference, JELIA 2023, Dresden, Germany, September 20-22, 2023, Proceedings*, volume 14281 of *Lecture Notes in Computer Science*, 697–711. Springer.

Lagniez, J.; Lonca, E.; and Mailly, J. 2015. Coquiaas: A constraint-based quick abstract argumentation solver. In *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, November 9-11, 2015*, 928–935. IEEE Computer Society.

Lagniez, J.; Lonca, E.; and Marquis, P. 2016. Improving model counting by leveraging definability. In Kambhampati, S., ed., *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 751–757. IJCAI/AAAI Press.

Lagniez, J.; Lonca, E.; and Marquis, P. 2020. Definability for model counting. *Artif. Intell.* 281:103229.

Li, B.; Hsiao, M. S.; and Sheng, S. 2004. A novel SAT all-solutions solver for efficient preimage computation. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, 272–279. IEEE Computer Society.

McMillan, K. L. 2002. Applying SAT methods in unbounded symbolic model checking. In Brinksma, E., and Larsen, K. G., eds., *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, 250–264. Springer.

Muise, C. J.; McIlraith, S. A.; Beck, J. C.; and Hsu, E. I. 2012. Dsharp: Fast d-dnnf compilation with sharpsat. In Kosseim, L., and Inkpen, D., eds., *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, volume 7310 of *Lecture Notes in Computer Science*, 356–361. Springer.

Selman, B., and Kautz, H. A. 1996. Knowledge compilation and theory approximation. *J. ACM* 43(2):193–224.

Spallitta, G.; Masina, G.; Morettin, P.; Passerini, A.; and Sebastiani, R. 2022. Smt-based weighted model integration with structure awareness. In Cussens, J., and Zhang, K., eds., *Uncertainty in Artificial Intelligence, Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, UAI 2022, 1-5 August 2022, Eindhoven, The Netherlands*, volume 180 of *Proceedings of Machine Learning Research*, 1876–1885. PMLR.

Spallitta, G.; Sebastiani, R.; and Biere, A. 2024. Disjoint partial enumeration without blocking clauses. In Wooldridge, M. J.; Dy, J. G.; and Natarajan, S., eds., *Thirty-Eighth AAAI Conference on Artificial Intelligence, Vancouver, Canada*, 8126–8135. AAAI Press.

Toda, T., and Soh, T. 2016. Implementing efficient all solutions SAT solvers. *ACM J. Exp. Algorithmics* 21(1):1.12:1–1.12:44.

Trinh, V.; Benhamou, B.; Hiraishi, K.; and Soliman, S. 2022. Minimal trap spaces of logical models are maximal siphons of their petri net encoding. In Petre, I., and Paun, A., eds., *Computational Methods in Systems Biology - 20th International Conference, CMSB 2022, Bucharest, Romania, September 14-16, 2022, Proceedings*, volume 13447 of *Lecture Notes in Computer Science*, 158–176. Springer.

Vollmer, H. 1999. *Introduction to Circuit Complexity - A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Yu, Y.; Subramanyan, P.; Tsiskaridze, N.; and Malik, S. 2014. All-sat using minimal blocking clauses. In *2014 27th International Conference on VLSI Design, VLSID 2014, and 2014 13th International Conference on Embedded Systems, Mumbai, India, January 5-9, 2014*, 86–91. IEEE Computer Society.