Specialising Model Counting for Probabilistic Inference

Alexandre Dubray

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

June 2025

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Pierre **Schaus** (Co-Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Siegfried **Nijssen** (Co-Advisor)	UCLouvain/ICTEAM and KU Leuven, Belgium
Pr. Charles **Pêcheur**	UCLouvain/ICTEAM, Belgium
Pr. Éric **Piette**	UCLouvain/ICTEAM, Belgium
Pr. Jean-Marie **Lagniez**	Université d'Artois/CRIL, France
Dr. Thomas **Schiex**	INRIA, France

Victoriae mundis et mundis lacrima

Acknowledgments

Firstly, I would like to thank my advisors, Pierre and Siegfried. Throughout these six years, we worked on many subjects, including data mining, visualisation, optimisation, and model counting. They allowed me to work on topics that interested me, even if it meant a total shift in my PhD content halfway through it. I learned a great deal from both of them and truly enjoyed our work over the past few years.

During my time in the department, I met many people, but some of them deserve personal words. My first thought goes to Guillaume D., who introduced me to the research world during my master's thesis. You were always present for a good discussion (technical or not), and you taught me the ways of the cards, for better or worse, depending on who you ask. Then, I want to express my gratitude to Vanessa. We had many enjoyable times together during these six years, and I always appreciated your company (which I hope to continue in the coming years). Harold, who shared my office this whole time, is one of the people with whom I've spent most of my time during my thesis. We had a great time together, discussing technical problems, life, or anything else. I would also like to thank the current and past members of the AIA team (Augustin, Emma, Hélène, Guillaume, Xavier, Vianney, and all the others) for all the fun we had both at work and at conferences.

Finally, I would like to express my gratitude to my friends and family, particularly to my parents, who have supported me throughout these years. At the end of my time at UCLouvain, I have nostalgic thoughts for Thomas and Florent about the time we spent together during and after our studies. I also have a special thank you to Emma; although our time in the department together was relatively short, I appreciated our discussions during our (sometimes long) rides south, and hopefully, we have many more to come. Then, I want to thank Mélanie; even though I mainly see you when I'm not working, we always have such a good time together, and you always encourage me. I'm also grateful to the members of the TTST club (Marion, Carline, Julien, Mathilde, Fred, Céline, and many others) for the time we spent together training. We felt true pain during some sessions, which significantly reduced the perceived effort of writing this thesis. Lastly, I want to thank Michel and Pouette; although your conversation skills could be improved, you were always great listeners.

Contents

Introduction 1

Reasoning under uncertainty is the task of reasoning about one or more uncertain events. Probability theory is the most common method for quantifying uncertainty. In this method, each event is represented by a random variable associated with a specific probability distribution. A *probabilistic model* represents the joint distribution of all the random variables, and *probabilistic inference* is the process of querying this representation. Probabilistic reasoning has applications in many real-world domains. For example, Bayesian networks, one of the most popular probabilistic models, have been used in medicine [And+91; Oni03], biology [JK99], weather prediction [Abr+96], or wildlife monitoring [AMD08; For+16].

Another popular reasoning system is based on *symbolic* (i.e., human-readable) constraints. For example, propositional logic is one of the most popular languages to express such constraints (e.g., the variable X and Y must be true, at least one variable between X and Y must be true) and has many applications such as planning [KS92], biology [LM06], software testing [KM04], and many more [Mar08]. In particular, over the last two decades, propositional logic has emerged as a method for solving probabilistic inference problems [CD08]. The idea is that a probabilistic model can be encoded as a weighted boolean formula, and the inference task is reduced to a propositional logic problem. For example, it is known that computing the probability of some observation can be reduced to counting the number of satisfying assignments of a weighted boolean formula; this is the *weighted model counting* (WMC) problem.

In their general form, the probabilistic inference tasks studied in this work and WMC are #*P*-Complete.Designing algorithms that solve such problems is challenging, but modern model counters efficiently solve a wide range of problems. However, such efficiency comes at the cost of complex solvers that are difficult to extend and modify. Nowadays, probabilistic logical reasoning (combining probabilistic and symbolic reasoning) solvers are used as a subtask in other AI domains; hence, it becomes crucial to have solvers that can be adapted to various contexts and requirements.

One domain that has gained significant interest in recent years is *neuro-symbolic* (NeSy) AI, which combines neural networks with symbolic reasoning. NeSy AI aims to combine the strength of neural networks (NN), which

have demonstrated state-of-the-art performance in various AI domains, with symbolic reasoning. To illustrate the need for such mixed systems, let us take the example of autonomous driving [Sin+22; Giu+23]. Autonomous driving requires a system for identifying objects (e.g., road lines, stop signs, other vehicles, pedestrians) and making appropriate decisions according to their environment. However, pure neural networks have difficulties complying with background symbolic constraints (e.g., a traffic light cannot be red and green simultaneously; a person is different from a car). On the other hand, such symbolic constraints are easily expressed using propositional logic. Hence, in the context of autonomous driving, combining neural and symbolic reasoning allows the power of neural networks to be leveraged for object identification while ensuring that their output respects a set of predefined constraints.

This work explores how WMC can be specialised for probabilistic inference in a simple and easily adaptable manner. More specifically, we propose a new modelling language called Schlandals, based on propositional logic, which incorporates the structure of probabilistic models. We associate a new solver of the same name with this language, which includes algorithms to compute exact and approximate weighted model counts on its formulas. To be easily extensible and adaptable, the Schlandals solver implements the most basic elements needed to solve the WMC problem. Despite its simplicity, we demonstrate that this solver is competitive with state-of-the-art model counters and, in some cases, outperforms them. Moreover, when the count cannot be computed exactly, Schlandals is the first solver to return a good lower and upper bound on the weighted model count. On the other hand, most classical approximation algorithms for the WMC provide approximations or lower bounds with statistical guarantees, meaning they are probabilistically valid.

The rest of this thesis is organised as follows. Chapter 2 introduces the necessary knowledge to understand this manuscript and related work. It defines the probabilistic models, inference tasks studied, and model counting problems. Then, the Schlandals modelling language is defined in Chapter 3, and the solver, as well as the exact counting algorithms, are described in Chapter 4. The approximate counting algorithms are developed in Chapter 5. We conclude and give further research directions in Chapter 6.

This manuscript is mainly based on the two following papers:

- A. Dubray, P. Schaus, and S. Nijssen. "Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses". In: *LIPIcs, Volume 280, CP 2023* 280 (2023). Ed. by R. H. C. Yap. ISSN: 1868-8969. DOI: 10.4230/LIPICS.CP.2023.15. (Visited on 03/07/2025)

- A. Dubray, P. Schaus, and S. Nijssen. "Anytime Weighted Model Counting with Approximation Guarantees for Probabilistic Inference". In:

LIPIcs, Volume 307, CP 2024 307 (2024). Ed. by P. Shaw. ISSN: 1868-8969. DOI: 10.4230/LIPICS.CP.2024.10. (Visited on 03/07/2025)

Moreover, our work has led to the following collaboration:

- L. Dierckx, A. Dubray, and S. Nijssen. "Learning from Logical Constraints with Lower- and Upper-Bound Arithmetic Circuits" International Joint Conference on Artificial Intelligence (IJCAI) 2025.

The parts of this paper related to this thesis' scope are also presented in this manuscript. This manuscript also contains new, unpublished content. The introduction to each chapter clearly defines the content that has not been published before.

Finally, our work has also resulted in the following publications, which are not directly related to the content of this manuscript and are therefore not included herein.

- A. Dubray, G. Derval, S. Nijssen, and P. Schaus. "Mining Constrained Regions of Interest: An Optimization Approach". In: *Discovery Science*. Ed. by A. Appice, G. Tsoumakas, Y. Manolopoulos, and S. Matwin. Vol. 12323. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-61526-0 978-3-030-61527-7. DOI: 10.1007/978-3-030-61527-7_41. (Visited on 03/07/2025)

- A. Dubray, S. Nijssen, I. Thomas, and P. Schaus. "A Seriation Based Framework to Visualize Multiple Aspects of Road Transport from GPS Trajectories". In: *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021. (Visited on 03/07/2025)

- A. Dubray, G. Derval, S. Nijssen, and P. Schaus. "Optimal Decoding of Hidden Markov Models with Consistency Constraints". In: *Discovery Science*. Ed. by P. Pascal and D. Ienco. Vol. 13601. Cham: Springer Nature Switzerland, 2022. ISBN: 978-3-031-18839-8 978-3-031-18840-4. DOI: 10.1007/978-3-031-18840-4_29. (Visited on 03/07/2025)

Background Knowledge | 2

This chapter introduces the basic concepts of propositional model counting and probabilistic inference. Moreover, the three probabilistic models and inference tasks used in our experiments are presented.

2.1 Propositional Logic and Model Counting

This section introduces the notations related to propositional logic and the model counting problem: we define the counting problems and briefly outline how to solve them using search-based and compilation algorithms.

2.1.1 From Satisfiability to Counting Problems

In the rest of this work, bold letters denote vectors or sets of elements; capital letters are used for variables. True is denoted by $\top$ and false by $\bot$. Let $\boldsymbol{B} = \{B_1, \ldots, B_n\}$ be a set of boolean variables ($B \in \{\top, \bot\}\ \forall B \in \boldsymbol{B}$). A literal l denotes either the variable B or its negation $\neg B$. This work focuses on propositional formulas in *Conjunctive Normal Form* (CNF); hence, a clause is a disjunction of literals $C = l_1 \vee \ldots \vee l_k$ and a formula F is a conjunction of clauses $F = C_1 \wedge \ldots \wedge C_m$. An *interpretation* I is an assignment to a subset $\boldsymbol{X} \subseteq \boldsymbol{B}$ of the variables and is called *partial* if $\boldsymbol{X} \neq \boldsymbol{B}$. We define an interpretation I as a function $I : \boldsymbol{X} \mapsto \{\top, \bot\}$ mapping elements of $\boldsymbol{X}$ to a truth value.

Let $I_1 : \boldsymbol{X} \mapsto \{\top, \bot\}$ and $I_2 : \boldsymbol{Y} \mapsto \{\top, \bot\}$ be two interpretation such that$\boldsymbol{X} \cap \boldsymbol{Y} = \emptyset$. We denote $I = I_1 \cup I_2$ the assignment on $\boldsymbol{Z} = \boldsymbol{X} \cup \boldsymbol{Y}$ such that $I : \boldsymbol{Z} \mapsto \{\top, \bot\}$ is defined as

$$I(Z) = \begin{cases} I_1(Z) & \text{if } Z \in \boldsymbol{X} \\ I_2(Z) & \text{otherwise} \end{cases}$$

We denote by $F[I]$ the evaluation of F when the variables in I are replaced by their assignments, and the formula is reduced using the classical rules of propositional logic. A *model* of F is an interpretation $I : \boldsymbol{B} \mapsto \{\top, \bot\}$ such that $F[I] = \top$.

Problem 1 (Boolean Satisfiability (SAT))**.** *Let F be a boolean formula, in CNF, over variables $\boldsymbol{B}$. The satisfiability problem is to decide if there exists an interpretation $I : \boldsymbol{B} \mapsto \{\top, \bot\}$ such that $F[I] = \top$.*

Example 1: Satisfiability

Consider the following boolean formula in CNF, used as a running example:

$$F = (A \vee \neg B) \wedge (\neg A \vee C \vee D) \wedge (\neg A \vee \neg C).$$

To be satisfied by an interpretation I, each clause of F must be satisfied. If we define I such that $I(A) = \top$, $I(C) = \bot$ and $I(D) = \top$, it can be seen that each clause is satisfied. Hence, F is satisfiable.

Problem 1 is the classical NP-Complete decision problem [Coo23]. Multiple variants exist of the SAT problem, but this work focuses on a particular type of variation: the model *counting* problem.

Problem 2 (Model Counting (#SAT))**.** *Let F be a boolean formula, in CNF, over variables $\boldsymbol{B}$. The set of models of F is defined as follows.*

$$\mathcal{S}(F) = \{I : \boldsymbol{B} \mapsto \{\top, \bot\} \mid F[I] = \top\}$$

The model counting problem is to compute $\#SAT(F) = |\mathcal{S}(F)|$

Example 2: Model Counting

Let us demonstrate the counting problem with our running example. One way to compute the number of models of F is to create its truth table and count the entries that evaluate $\top$. From F's truth table below, it can be seen that $\#SAT(F) = 6$.

A	B	C	D	F(I)	A	B	C	D	F(I)
⊤	⊤	⊤	⊤	⊥	⊤	⊤	⊥	⊥	⊥
⊤	⊤	⊤	⊥	⊥	⊤	⊤	⊥	⊥	⊤
⊤	⊥	⊤	⊤	⊥	⊤	⊥	⊥	⊥	⊥
⊤	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊤
⊥	⊤	⊤	⊤	⊥	⊥	⊤	⊥	⊥	⊥
⊥	⊤	⊤	⊥	⊥	⊥	⊤	⊥	⊥	⊥
⊥	⊥	⊤	⊤	⊤	⊥	⊥	⊥	⊥	⊤
⊥	⊥	⊤	⊥	⊤	⊥	⊥	⊥	⊥	⊤

Notice that Problem 2 is the most straightforward counting extension of Problem 1: instead of searching for one model, one must count all of them.

Problem 2 is then structurally harder than Problem 1; it is #*P*-Complete [Val79b]. To define this complexity class, we first define *counting Turing mahcine.*

Definition 1 (Counting Turing machine [Val79a])**.** *A* counting *Turing machine is a standard non-deterministic Turing machine (i.e., a Turing machine with mutltiple possible actions per state) with an auxiliary output device that prints, on a special tape, the number of accepting computations induced by the input. It has a worst-case time complexity of $f(n)$ if the longest accepting computation induced by the set of all inputs of size n takes $f(n)$ steps.*

We can now define the #*P* complexity class.

Definition 2 (#*P* complexity class [Val79a])**.** *#P is the class of all function that can be computed by counting Turing machines of polynomial time complexity.*

It is known that the satisfiability problem can be solved in polynomial time on a non-deterministic TM; hence, it follows from Definition 2 that the model counting problem is in #*P*. In particular, Valiant showed that it was #*P*-Complete [Val79b].

A key observation when modelling a problem as a propositional formula is that, sometimes, additional boolean variables must be introduced to encode the constraints of the initial problem. These additional variables might introduce unwanted models. To solve this issue, the problem of *projected* model counting has been defined [Azi+15].

Problem 3 (Projected Model Counting (#∃SAT))**.** *Let F be a boolean formula, in CNF, over variables $\boldsymbol{B}$. Let $\boldsymbol{P}, \boldsymbol{X} \subseteq \boldsymbol{B}$ be a partitioning of the variables. The set of models of F projected on the variables in $\boldsymbol{P}$ is defined as follows.*

$$\mathcal{S}_{\boldsymbol{P}}(F) = \{I : \boldsymbol{P} \mapsto \{\top, \bot\} \mid \exists\, I' : \boldsymbol{X} \mapsto \{\top, \bot\} \text{ such that } F[I \cup I'] = \top\}$$

The projected model counting problem is to compute $\#\exists SAT(F, \boldsymbol{P}) = |\mathcal{S}_{\boldsymbol{P}}(F)|$

The projected version of the model counting problem only considers models for a subset of the variables called the *projected variables.* The interpretations in the set $\mathcal{S}_{\boldsymbol{P}}(F)$ can be viewed as a grouping of F's models based on their assignments to the projected variables. Hence, two different models of F can lead to the same element in $\mathcal{S}_{\boldsymbol{P}}(F)$. We call the elements in $\mathcal{S}_{\boldsymbol{P}}(F)$ *models projected* on $\boldsymbol{P}$, or projected models when $\boldsymbol{P}$ is evident from the context. Problem 3 is more generic than Problem 2: the model counting problem can be viewed as a projected model counting problem with $\boldsymbol{P} = \boldsymbol{B}$.

Example 3: Projected Model Counting

Let us compute the projected model count of our small example formula

$$F = (A \vee \neg B) \wedge (\neg A \vee C \vee D) \wedge (\neg A \vee \neg C)$$

with $\boldsymbol{P} = \{A, B\}$. Projected model counting can be seen as a succession of two steps: a counting step on the projected variables and a satisfiability check on the non-projected variables. Below is the truth table for F with coloured boxes highlighting interpretations that share the same assignment for A and B. There are only four possible interpretations on $\{A, B\}$; hence, the four boxes represent the counting part of the problem. If an interpretation on $\{A, B, C, D\}$ that evaluates to $\top$ exists in a box, then the corresponding interpretation on $\{A, B\}$ is a projected model. It can be seen that the green box is the only one not containing a model on F; hence, we have that $\#\exists SAT(F, \boldsymbol{P}) = 3$.

A	B	C	D	F(I)	A	B	C	D	F(I)
$\top$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$

Finally, we define the problem of projected *weighted* model counting; in this variation, every interpretation has a weight, and the weighted sum of the models must be computed. In this work, we consider the commonly used framework in which a weight is assigned to each literal, and the weight of an interpretation is the product of the weights of its literals.

Problem 4 (weighted#$\exists SAT$)**.** *Let F be a boolean formula, in CNF, over variables $\boldsymbol{B} = \boldsymbol{P} \cup \boldsymbol{X}$, with $\boldsymbol{P}$ the projected variables. Let $\boldsymbol{L} = \cup_{P \in \boldsymbol{P}} \{P, \neg P\}$ be the set of possible literals of the projected variables of F and $\omega : \boldsymbol{L} \mapsto \mathbb{R}$ a weighting function. The weighted $\#\exists SAT$ problem is to compute the weighted sums of the projected models of F defined as follows.*

$$weighted\#\exists SAT(F, \boldsymbol{P}) = \sum_{I \in \boldsymbol{S_P}(F)} \left(\prod_{\substack{P \in \boldsymbol{P} \\ I(P) = \top}} \omega(P) \times \prod_{\substack{P \in \boldsymbol{P} \\ I(P) = \bot}} \omega(\neg P) \right)$$

This last problem is the most generic version of the counting problems considered; we can compute the (projected) model count of F using a unit weight for all literals.

Example 4: Weighted Projected Model Counting
We showed above that if $F = (A \vee \neg B) \wedge (\neg A \vee C \vee D) \wedge (\neg A \vee \neg C)$ and $\boldsymbol{P} = \{A, B\}$, then $\#\exists SAT(F, \boldsymbol{P}) = 3$. In particular, the only partial interpretation I not being a projected model of F on $\boldsymbol{P}$ is such that $I(A) = \bot$ and $I(B) = \top$. Let us define the weight function ω as follows: $\omega(A) = 0.6, \omega(\neg A) = 0.4, \omega(B) = 0.2, \omega(\neg B) = 0.8$. Then we have

$$\text{weighted}\#\exists SAT(F, \{A, B\}) = \underbrace{0.6 \times 0.2}_{A=\top, B=\top} + \underbrace{0.6 \times 0.8}_{A=\top, B=\bot} + \underbrace{0.4 \times 0.8}_{A=\bot, B=\bot} = 0.92$$

The last example highlights a standard setting in weighted model counting in which $\omega(P) + \omega(\neg P) = 1$ ($P \in \boldsymbol{P}, \omega(P), \omega(\neg P) > 0$). Such a setting is typical when modelling probabilistic inference problems, as the weights represent probabilities. We assume this setting unless stated otherwise. This work focuses on projected weighted model counting; for ease of notation, we refer to this problem as model counting and denote the weighted model count of a formula by count. We refer to Problem 2 as the *unweighted* model counting problem to avoid confusion when necessary. Moreover, we use the notation $\text{pwmc}(F, \boldsymbol{P})$ as a shorthand way of writing $\text{weighted}\#\exists SAT(F, \boldsymbol{P})$.

2.1.2 Counting Models with a Search-based Algorithm

There are two main approaches for counting the assignments of a boolean formula, and the first one is a variation of the well-known DPLL search [DP60]. Such methods do an exhaustive depth-first search over the formula's assignments, branching over the boolean variables and applying *Boolean Unit Propagation* (BUP) at each search tree node. Let us first detail this propagation before explaining the search procedure.

Algorithm 1 details how BUP works when a choice (e.g., $B = \top$) is made. It is a fixed-point algorithm that assigns truth values to variables. The formula is unsatisfiable if a variable is assigned to contradictory values, and $\bot$ is returned (line 6). Otherwise, the value is assigned to the variable (line 7), and clauses that are respected (i.e., that evaluates to $\top$) are removed from the residual formula (lines 8-9). Then, literals evaluated to $\bot$ are removed from the remaining clauses (lines 10-16). During this process, a new assignment is added to the queue if a clause is reduced to a single literal. The residual formula F' is returned when the assignment queue is empty (line 19).

Search-based model counters require the implementation of two key techniques for efficiency. First, they must implement *decomposition into independent components*. When a formula F can be decomposed into sub-formulas that do not share variables, the count of each sub-formula can be counted independently. Then, F's count is the multiplication of its sub-formulas' count.

Algorithm 1: Boolean Unit Propagation

```
1  Function BUP(F, B, b)
       input  : F a boolean formula in CNF over variables B
       input  : B ∈ B a boolean variable to set to value b ∈ {⊤, ⊥}
       output: F′ the residual formula
2      F′ ← F
3      Q ← Queue(); Q.push((B, b))
4      while |Q| > 0 do
5          (V, v) ← Q.pop()
6          if V has been previously assigned to ¬v then return ⊥
7          assign v to V
8          if v = ⊤ then F′ ← F′ \ {C | C ∈ F′ ∧ V ∈ C}
9          if v = ⊥ then F′ ← F′ \ {C | C ∈ F′ ∧ ¬V ∈ C}
10         foreach C ∈ F′ do
11             if v = ⊤ and ¬V ∈ C then C = C \ {¬V}
12             if v = ⊥ and V ∈ C then C = C \ {V}
               /* C is a clause with only one literal */
13             if |C| = 1 then
14                 if C = V′ then Q.push((V′, ⊤))
15                 if C = ¬V′ then Q.push((V′, ⊥))
16             end
17         end
18     end
19     return F′
```

Example 5: Independent Components

Let $F = (A \vee \neg B) \wedge (\neg A \vee C \vee D) \wedge (\neg A \vee \neg C) \wedge (E \vee \neg G)$ and $\boldsymbol{P} = \{A, B, E, G\}$. This formula can be decomposed into two sub-formulas that do not share variables: $F_1 = (A \vee \neg B) \wedge (\neg A \vee C \vee D) \wedge (\neg A \vee \neg C)$ and $F_2 = (E \vee \neg G)$. An interpretation of F can also be decomposed into interpretations of F_1 and F_2. Moreover, when computing the projected model count on each component, the projected variables can be reduced to those in the component. We already saw that $\#\exists SAT(F_1, \{A, B\}) = 3$, and $\#\exists SAT(F_2, \{E, G\}) = 3$ as only the assignment $E = \bot, G = \top$ leads to $\bot$.

Hence, any combination of a model of F_1 with a model of F_2 forms a model of F since all clauses are respected. For example, $A = \top, B = \top$ is a projected model of F_1 and $E = \top, G = \bot$ is a model of F_2. Hence, the assignment $A = \top, B = \top, E = \top, G = \bot$ is a projected model of F.

Let us define the weights of the subformula's projected models as fol-

lows.

$$\omega(A = \top, B = \top) = 0.18 \quad \omega(E = \top, G = \top) = 0.08$$
$$\omega(A = \top, B = \bot) = 0.12 \quad \omega(E = \top, G = \bot) = 0.02$$
$$\omega(A = \bot, B = \bot) = 0.28 \quad \omega(E = \bot, G = \bot) = 0.18$$

The weighted model count of F_1 is 0.58, the weighted model count of F_2 is 0.28, and the weighted model count of F is $0.58 \times 0.28 = 0.1624$. The computation of F's weighted model count is as follows. Every combination of the subformula's projected models is a projected model of F, and its weight is given by multiplying the weight of the subformula's models. For example, the models' weights based on $A = \top$ and $B = \top$ are as follows.

$$\begin{aligned} 0.18 \times 0.08 + 0.18 \times 0.02 + 0.18 \times 0.18 &= 0.18 \times (0.08 + 0.02 + 0.18) \\ &= 0.18 \times \text{pwmc}(F_2, \{E, G\}) \end{aligned}$$

If c_2 represents the weighted model count of F_2, then the weighted model count of F is given as follows.

$$\begin{aligned} 0.18 \times c_2 + 0.12 \times c_2 + 0.28 \times c_2 &= c_2 \times (0.18 + 0.12 + 0.28) \\ &= c_2 \times \text{pwmc}(F_1, \{A, B\}) \end{aligned}$$

This decomposition is not only applied at the root node; after each decision in the search tree, after the BUP algorithm, it is possible to detect independent components. Hence, this is a powerful tool for search-based model counters as it can significantly reduce the size of the formulas.

The second key aspect of search-based model counters is a *caching system*. Search-based model counters work by assigning a variable to either $\top$ or $\bot$, applying the BUP algorithm, and recursively computing the count of the residual formula. The same residual formula can appear multiple times in the search space; hence, recent model counters store the count of each solved sub-formula in a cache. Such a caching system is typically implemented using a HashMap: a hashable key (e.g., a string representation) is computed for each sub-formula, and its count is stored as the corresponding value in the HashMap. The most basic representation for a formula directly encodes each clause with integers.

Example: Caching System
Let $F = (X_1 \vee \neg X_3 \vee X_4) \wedge (X_2 \vee \neg X_3 \vee X_4)$. Let us assume that a search-based solver has the partial assignment $X_1 = \top, X_2 = \bot$; then, the residual formula is $F_1 = \neg X_3 \vee X_4$, whose model count is 3. It can be represented

using the sequence $-3, 4, 0$: each integer represents a variable, the sign corresponds to their polarity, and 0 indicates the end of a clause. Let us denote C as the cache. Then, the solver computes the model count of F_1 and stores the mapping $C[-3, 4, 0] \mapsto 3$.

Then, let us assume that in another part of the search space, the solver has the partial assignment $X_1 = \bot, X_2 = \top$. Such an assignment results in a residual formula $F_2 = \neg X_3 \vee X_4$, which is equivalent to F_1 and, hence, has the same representation. The solver can then directly query the cache and return $C[-3, 4, 0]$.

Algorithm 2: DPLL-based algorithm for solving Problem 4

```
1  Function DPLL-PWMC(F, P, ω, C)
       input  : F a boolean formula, over variables B, in CNF
       input  : P ⊆ B a set of projected variables
       input  : ω a literal-weight function
       input  : C a cache of sub-results
       output: pwmc(F, P)
2      if F ∈ C then return C[F]
3      if F = ⊤ then return ∑_{I∈S_P(F)} ω(I)
4      if P = ∅ then return 1 if F is SAT else 0
5      P ← variableSelection(F, P)
6      foreach v ∈ {⊤, ⊥} do
7          F' ← BUP(F, P, v)
8          if F' = ⊥ then count_v ← 0
9          else
10             fixed ← {P' ∈ P | P' is forced to ⊤ or ⊥ during BUP }
11             count_v ←
                   (∏_{P'∈fixed|P'=⊤} ω(P')) × (∏_{P'∈fixed|P'=⊥} ω(¬P'))
12             Components ← all independent components of F'
13             foreach Comp ∈ Components do
14                 P' ← P reduced to the variables in Comp
15                 count_v ← count_v × DPLL-PWMC(Comp, P', ω, C)
16             end
17         end
18     end
19     C[F] ← count_⊤ + count_⊥
20     return C[F]
```

Algorithm 2 shows the general search procedure to compute pwmc(F, $\boldsymbol{P}$). Its structure resembles the one of a classical DPLL algorithm: it heuristically

selects one variable in $\boldsymbol{P}$ (line 5), assigns it to true or false, and propagates its choice using BUP, reducing F (line 7), and recursively explore the residual formula (line 15). When a formula is reduced by the propagation (line 7) and is not $\bot$, it is decomposed into independent components (line 12) that are solved independently. A model has been found when the residual formula becomes $\top$ (line 3). However, not all projected variables may have been assigned. In such a case, all assignments to the remaining projected variables are models of F, and the weighted model count is the weighted sum of all remaining interpretations. On the other hand, when $\boldsymbol{P} = \emptyset$, there are no more variables to branch on, but F might still contain clauses; hence, it must be checked that F is satisfiable (line 4). Note that this problem is NP-complete in the general case; hence, depending on the structure of the residual formula, this check can take a long time.

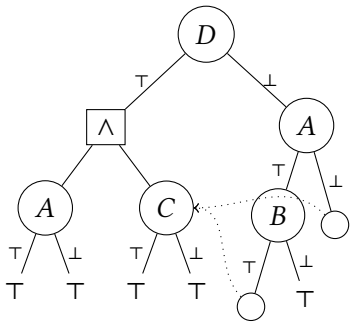
The intuition behind the computation of a branch's count (count_v) is as follows. When branching on a value for P, the algorithm builds an interpretation by setting P and possibly other variables to a truth value during the propagation. Hence, all variables $P' \in \boldsymbol{P}$ assigned during the propagation (line 10) are part of the interpretation, and the product of their weights is computed (line 11). Then, in the recursive calls, the algorithm builds the rest of the interpretations, and their weights (i.e., the value returned by the `DPLL-PWMC` function) are multiplied by the branch count. As all interpretations share the same assignment for the variables assigned during the propagation, the first term of count_v can be factorised and added at the beginning. Finally, the weighted count of F is computed by summing the counts of the two branches.

Note that Algorithm 2 is general enough to solve the non-projected and non-weighted variation of the counting problem. If $\boldsymbol{P} = \boldsymbol{B}$, the condition at line 3 is redundant with the condition at line 4. On the other hand, for the unweighted case, using $\omega(P) = 1$ effectively gives the model count of F.

One of the strengths of depth-first search is its memory efficiency; however, in the case of model counting, a cache is used to store the result of sub-formulas encountered during the search. Hence, the memory used by Algorithm 2 grows linearly with the size of the search space, which can be exponentially large. Fortunately, it is possible to limit the size of the cache to reduce the memory used by the solvers. When the cache reaches its limit, some entries are heuristically removed (e.g., [Thu06; LM20]). For example, a score-based cache system has been proposed in the literature: when the cache reaches a specific limit, the entries that are less frequently queried are removed [Thu06].

Example 7: DPLL-based Search Algorithm

As an example for the execution of Algorithm 2, let us show how it computes the weighted model count of the formula $F = (A \vee B) \wedge (B \vee D \vee \neg E) \wedge (C \vee \neg E)$. Below is a possible search tree when applying the Algorithm 2 on F with $\boldsymbol{P} = \{A, B, C, D\}$. For simplicity, we assume that each literal has a unit weight. Hence, in this example, we solve the classical model counting problem.



Let us assume the algorithm starts by branching on D and first explores the branch $D = \top$. Applying BUP with $D = \top$ only removes F's second clause, and no unit propagation is done. Hence, the formula becomes $F' = (A \vee B) \wedge (C \vee \neg E)$, and it can be seen that the two clauses of F' do not share any variables. Algorithm 2 decomposes F' into $F'_1 = A \vee B$ and $F'_2 = C \vee \neg E$.

For F'_1, let us assume that the algorithm branches on the variable A. When applying the propagation with $A = \top$, the formula is satisfied, and no clause remains in the residual formula. Subsequently, a leaf is found, but B is not assigned. Hence, the algorithm returns 2 ($= \omega(B) + \omega(\neg B)$) as count. In the other branch, setting $A = \bot$ forces, during propagation, $B = \top$, and the formula is satisfied, leading to a leaf with a count of 1. Both branches have been explored, and the model count of $A \vee B$ is 3. The same reasoning can be made for F'_2, and its model count is also 3. Both decompositions have been solved, and the model count of the decomposition is $3 \times 3 = 9$.

The whole left sub-tree has been explored at the root, and only the case $D = \bot$ remains. After the BUP, the residual formula is $F' = (A \vee B) \wedge (B \vee \neg E) \wedge (C \vee \neg E)$. When branching on $A = \bot$, the rightmost branch, the problem is solved after the propagation. Indeed, after this assignment, the first clause becomes a unit one, forcing $B = \top$ and removing the second clause from F'. Hence, the formula becomes $C \vee \neg E$, which has already been solved and stored in the cache and can be retrieved (as shown by the dotted line leading to the corresponding sub-tree). The same behaviour occurs when assigning $A = \top$, which removes the first clause from F'. When branching on B, the assignment either removes the second clause or forces the assignment on E.

2.1.3 Literature Review of Search-Based Model Counters

Cachet [San+04; SBK05a] is one of the first successful search-based model counters which incorporates the components described above: formula caching [BDP03; Bea+03; ML98] and dynamic component analysis [BP00; BDP03]. In addition, it also makes use of *clause learning* [SS96; Zha97; Zha+01]. Clause learning is a technique initially developed for SAT solvers: when a conflict is encountered (i.e., forcing opposite literals during propagation), the cause (i.e., a subset of assignments made on the current branch) is detected, and a new clause is added to the formula forbidding such assignments. For example, if the cause for a conflict is the assignment $x = \top$, $y = \bot$, and $z = \top$, the learned clause is $\neg x \vee y \vee \neg z$. Although clause learning and component caching seem to be orthogonal techniques, the author of Cachet showed that combining these techniques must be done carefully as it can lead to cache entries with incorrect model counts.

Finally, Cachet also analyses various branching heuristics for model counting and proposes a new heuristic called *Variable State Aware Decaying Sum* (VSADS) [SBK05a]. VSADS is a mix of literal counting heuristic [Mar99] and *Variable State Independent Decaying Sum* (VSIDS) [Mad+01; Zha+01]. It scores each variable based on its number of occurrences in the residual formula (literal counting) and the number of times it appears in the learned clauses (VSIDS). Overall, the VSADS heuristic performs better than the other heuristics considered in the Cachet's benchmarks.

The sharpSAT solver serves as the basis of many modern model counters [Thu06]. One of the specificities of sharpSAT is that it explores multiple caching schemes. In the original sharpSAT paper, multiple representations for the components are analysed, and a bounded-caching system is developed to avoid memory overhead. A score is assigned to each component based on how many times they get requested, and when the cache becomes full, all entries with a score lower than a threshold are removed.

Ganak is built upon sharpSAT; hence, it incorporates all its components and adds several new features [Sha+19]. One of Ganak's main features is probabilistic component caching: it uses a special kind of hash function that takes less memory but has a small (bounded) probability of producing false positive cache hits. Ganak also enhances the VSADS branching heuristics by adding the cache state into the scoring of each variable. This new heuristic aims to improve the cache-hit ratio of Ganak by branching on variables that have not appeared in recently added cache components. Another key aspect of Ganak is the computation of *minimum independent support* for hard instances. An independent support is a subset of variables that, when assigned, uniquely determines the value of the remaining variables. Hence, when Ganak estimates that an instance is complex, it computes independent

support as small as possible and branches on the variables it contains, which can drastically reduce the size of the search space.

sharpSAT-TD is another model counter built upon sharpSAT; its main addition is the integration of *tree decomposition* into the branching heuristic. A tree decomposition of a graph is a tree whose nodes are sub-sets of the graph's nodes. For clarity, we denote by *bag* the nodes of the tree decomposition. A tree decomposition must also respect the three following conditions: 1. Each node of the graph must appear in a bag, 2. There must be a bag containing the source and target nodes of each edge in the graph, and 3. For each node of the graph, the sub-tree composed of all bags containing it must be connected. sharpSAT-TD leverages the tree decomposition of a formula's primal graph (i.e., a graph with nodes representing boolean variables and there is an edge between two variables if they appear in a clause together) to help decompose the formula into independent components. The idea is that branching on variables appearing near the root of the decomposition leads to components with variables either in the left or right subtree of the decomposition. GPMC is another solver based on the same ideas as sharpSAT-TD [SHS17].

The projMC solver [LM19] uses another approach to solve the projected model counting problem. It is based on *disjunctive decomposition*: To compute the projected count of a formula F (over projected variables $\boldsymbol{P}$), projMC first computes a disjunctive deterministic form for F with respect to $\boldsymbol{P}$. Such a form is a set of boolean formulas $\varphi^1, \ldots, \varphi^k$ that are mutually exclusive ($\varphi^i \wedge \varphi^j = \bot$ for $i \neq j$) but disjunctively valid ($\vee_{i=1}^{k} \varphi = \top$). These two properties imply that the projected model count of F can be computed as $\sum_{i=1}^{k} F \wedge \varphi^i$. By carefully selecting the disjunctive deterministic form, each formula $F \wedge \varphi^i$ is easier to count than F.

2.1.4 Model Counting by Knowledge Compilation

All search-based model counters presented take a literal-weighted boolean formula as input and return its model count. However, some settings, such as NeSy AI, require computing many times the weighted model count of the same formula. Let us consider the Sudoku classifier example from the NeSy literature [Aug+22]: a convolutional neural network is employed to identify handwritten digits within a Sudoku grid, and the goal is to verify the validity of the grid. One way of modelling our NeSy system is as follows. The CNN outputs, for each cell, a probability distribution representing the probability that the digit in the cell corresponds to each of the nine possibilities. There are nine variables $B_{ij}^1, \ldots, B_{ij}^9$ per cell (i, j), such that $B_{ij}^k = \top$ means that the digit in cell (i, j) is k. The weight of each variable B_{ij}^k ($1 \leq k \leq 9$) is given by the output of the CNN. Let $\boldsymbol{B}$ be the set of boolean variables for every cell

of the Sudoku grid; the boolean formula F encoding the Sudoku constraints is constructed from pairwise binary inequality (i.e., $\neg B \vee \neg B'$ for $B, B' \in \boldsymbol{B}$) for the variables on the same row, the same column, and in the same square. In this setting, the weighted model count of F represents the probability that the Sudoku grid is valid, given the CNN's predictions.

Let us assume a typical setting in which the CNN is learned using gradient descent. A training set with filled Sudoku grids, labelled as valid or not, is available in such a setting. The goal is then to have a weighted model count of 1 for the valid Sudoku grids and 0 for the others. Computing the count with search-based model counters has two significant limitations. First, computing the weighted model count each time from scratch is inefficient and time-consuming. Indeed, computing the count with a search-based model counter entails numerous operations (e.g., constraint propagation, component detection) and exploring unsatisfiable parts of the search space. However, the formula's set of models remains unchanged during the learning process, and in our example, the models are the same for every sample in the training set. Hence, it is inefficient to recompute the models each time.

In addition, the loss used for gradient descent is computed on the output of the counting algorithm; hence, backpropagation must be performed through the logical constraints and then only through the CNN. However, search-based model counters only return a count; it is not possible to do such backpropagation.

A solution to these problems is to solve the model counting problem with *Knowledge Compilation* (KC) techniques. In model counting, *knowledge compilers* (or simply compilers) are algorithms that take a CNF formula as input and translate it to a target language. In the next section, we explain what a language is; the intuition is that compilers represent all the models of a CNF formula in a compact form, often as a diagram. Moreover, this representation can be used to compute the formula's model count; hence, when the weights change, it can still be used to calculate the new count. Finally, the representations considered in this work are differentiable; hence, the whole NeSy system can be learned end-to-end using automatic differentiation tools. Although we showcased the utility of knowledge compilers on NeSy systems, modern compilers are competitive with state-of-the-art search-based model counters. They can be used as standalone tools to compute a formula's count only once.

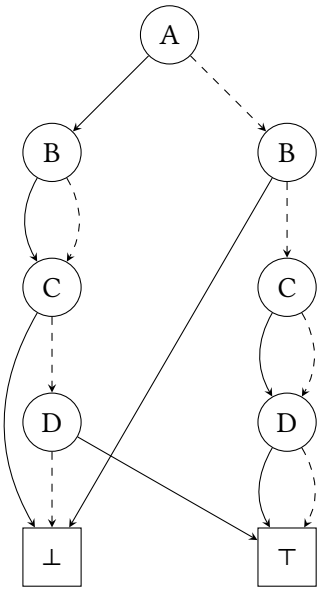
2.1.4.1 Existing Target Languages for Knowledge Compilation

In the context of KC, a language is a means of representing knowledge that supports operations within a given level of complexity. For example, the input language in our work is the language of propositional formulas in CNF,

and the operation of computing their weighted model count is #P-Complete. A review of popular target languages has been done by Darwiche and Marquis, analysing them on two axes: their succinctness and the queries they support in polynomial time[DM02]. This work focuses on model counting problems, so we limit our description to languages that support such queries in linear time. To give an idea of the compilation process, let us show how to encode the models of a propositional formula as a Binary Decision Diagram (BDD) [Bry86].

Example: Model Counting Using Binary Decision Diagram

Let $F = (A \vee \neg B) \wedge (\neg A \vee C \vee D) \wedge (\neg A \vee \neg C)$ for which we know, from previous examples, that $\#SAT(F) = 6$. A BDD is a rooted, directed, acyclic graph in which each node represents a decision variable (i.e., F's variables), and edges represent a choice (i.e., assigning $\top$ or $\bot$). Below is a BDD encoding F's interpretations: each node is labelled with a boolean variable, solid outgoing edges represent an assignment to $\top$, dashed edges an assignment to $\bot$, and leaves are labelled with $\top$ or $\bot$.



Each path in such a diagram represents a possibly partial assignment to F's variables that is satisfiable if it leads to the $\top$ leaf. For example, the path $A = \bot$ (dashed edge outgoing from the root) and $B = \top$ (solid edge) leads to $\bot$, which is inconsistent with F's clauses. Indeed, the first clause forbids such an assignment. Hence, the model count of F is the number of paths from the root to the $\top$ leaf, which is 6 as expected.

This example illustrates how knowledge compilation works: by repre-

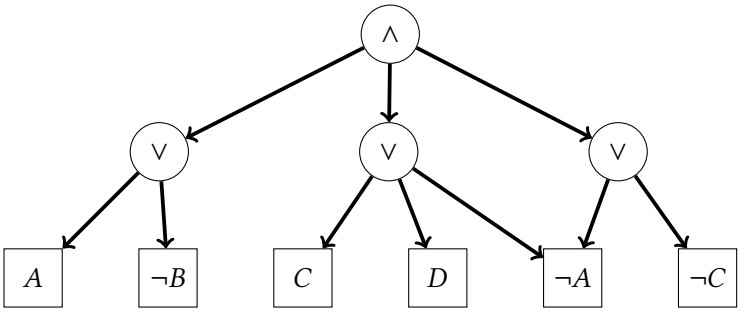
senting all assignments and, hence, models of a boolean formula, it is possible to compute its model count in time *linear in the size of the new representation.* Representing all models of a formula can, in practice, require exponentially more space than the initial CNF formula; hence, it does not reduce the complexity of model counting.

We have presented a conversion from a propositional formula to a binary decision diagram; however, knowledge compilation is designed over more generic languages. One of the most popular target languages for knowledge compilation is the *Negation Normal Form* (NNF) language, defined as follows [Bar82].

Definition 3 (NNF Language)**.** *Let $\boldsymbol{B}$ be a set of propositional variables. A sentence in NNF is a rooted, directed, acyclic graph where each leaf is labelled as $\top$, $\bot$, B, or $\neg B$ for $B \in \boldsymbol{B}$. Each internal node is labelled with $\wedge$ or $\vee$, which are interpreted as the logical operators in propositional logic.*

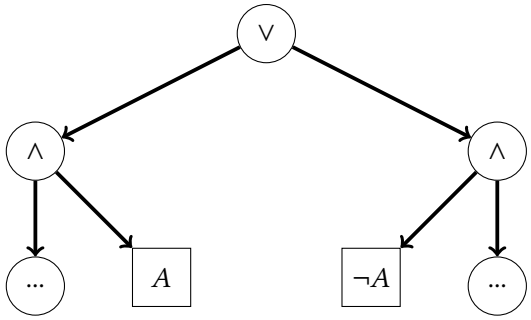
Example: Representing Knowledge in NNF

The formula $F = (A \vee \neg B) \wedge (\neg A \vee C \vee D) \wedge (\neg A \vee \neg C)$ can be represented in NNF as follows.



Hence, the CNF language is a subset of the NNF language, such that each sentence in CNF has a depth of 2, the root is a conjunction node, and its children are disjunction nodes.

The BDD language defines a sub-set of NNF sentences corresponding to binary decision diagrams. The idea is that each decision node can be transformed into a partial NNF diagram and combined. The following partial NNF sentence shows how the root of our previously defined BDD is transformed into NNF.



When encountering a decision node in a decision diagram, both outgoing edges can lead to models (i.e., a $\top$ leaf); hence, models are in the left *or* right child. Decision nodes in binary decision diagrams correspond to $\vee$-nodes in NNF. Then, when a choice is made for a variable (i.e., following an edge in the decision diagram), the choice must be enforced in the NNF. Hence, a $\wedge$-node is used. The BDD language is a subset of the NNF language in which each sentence's root is structured, as shown above.

Target languages for compilers are sub-sets of the NNF language (i.e., they respect the conditions in Definition 3) with additional structural constraints. Formally defining each language derived from NNF is out of the scope of this work; for a comprehensive review of such languages, see [DM02].

Two structural properties of NNF sentences are of particular interest for model counting: decomposability [Dar01a] and determinism [Dar01b]. A $\wedge$-node of an NNF sentence is decomposable if its sub-sentences do not share any variable. An NNF sentence is decomposable if each of its conjunction nodes is decomposable. A $\vee$-node is deterministic if its sub-sentences are contradictory, and an NNF sentence is deterministic if all its disjunction nodes are deterministic. The d-DNNF language is a subset of NNF that satisfies both the decomposability and determinism properties.

These two structural properties lead to two subsets of the NNF language that are particularly interesting for model counting. The DNNF language is the subset of NNF satisfying the decomposability property, the d-NNF language is the subset of NNF satisfying the determinism property, and the d-DNNF language is the subset of NNF satisfying both properties. It is known that a sentence in d-DNNF supports the model counting operation in linear time [DM02].

We have already seen that the BDD language imposes a particular structure on its nodes; the language of *Ordered* Binary Decision Diagram (OBDD) further imposes an order on the variables. If a variable has a smaller order than another, it must appear before it in the OBDD [Bry86]. OBDDs respect the determinism and decomposability properties; hence, OBDD is a sub-set

of d-DNNF. One of the main advantages of OBDD, compared to d-DNNF, is that they are canonical. Given an ordering, exactly one OBDD representing the initial boolean formula exists using this ordering. Hence, from a practical point of view, computing an optimal OBDD can be reduced to finding an optimal ordering for the variables. Moreover, OBDD can be combined using any boolean operator in polynomial time, leading to straightforward compilation algorithms. Other types of languages based on decision diagrams have been proposed in the literature (e.g., [Bar+14; DPV20]). Interestingly, knowledge compilation to decision diagrams extends beyond propositional logic and model counting; it has been extensively used to solve combinatorial optimisation problems (e.g., [Ber+16; Kor+13]).

More recently, a new representation, *Sentential Decision Diagrams*, has been proposed as a compromise between d-DNNF and Ordered Binary Decision Diagrams [Dar11]. This representation is based on two structural properties of NNF: structural decomposability [PD08] and strongly deterministic decompositions [PD10]. Structural decomposability imposes a stronger requirement than decomposability on which variable can appear in sub-sentences of $\wedge$-nodes. In addition to not sharing any variables, the variables must be partitioned according to the structure of a predefined *variable tree* (vtree). A DNNF respecting a vtree is called a *structured DNNF*. Structured DNNF induces a decomposition of the boolean function they represent as n pairs of sub-functions $g^i \wedge h^i$ $(1 \leq I \leq n)$. Such a decomposition is strongly deterministic if and only if $g^i \wedge g^j$ is inconsistent for all $i \neq j$. A key aspect of SDDs is that, similarly to OBDDs, they are canonical and can be combined in polynomial time. Hence, similarly to OBDDs, the main challenge in compiling SDDs is to find an optimal vtree. Probabilistic SDDs (PSDDs) have been proposed as a probabilistic extension of SDDs [Kis+14]; instead of logical decision nodes, a PSDD has probabilistic decision nodes, where the possible choices define a probability distribution.

Other languages than NNF exist to perform tractable model counting. For example, it has been proposed to use affine clauses (i.e., XOR constraints) in diagram-based languages to perform model counting [Kor+13]. More precisely, Koriche et al. defined multiple languages based on *Affine Decision Trees* (ADT). ADT are directed acyclic graphs in which leaves are labelled with boolean constants (i.e., $\top$ or $\bot$) and internal nodes are either conjunctions, disjunctions, or affine clauses. Similarly to sub-languages of NNF; imposing structural constraints on ADT yield sub-languages on which model counting can be performed in linear time (over the trees size).

2.1.5 Literature Review of Knowledge Compilers

As for search-based model counters, several compilation algorithms have been developed over the years. A first key observation made by Huang and Darwiche is that the trace of DPLL-style search-based model counters corresponds to a d-DNNF sentence [HD07]. Hence, any such model counter can be converted to a knowledge compiler. An example of such a compiler is `Dsharp`, which leverages the `sharpSAT` search-based model counter [Mui+10]. `D4` is another tree-search-based compiler and shares many similarities with `Dsharp` [LM17a]. Its main improvement consists of smarter decomposition schemes: independent components are not computed at each decision node, and the solver focuses on the decomposition quality, minimising the time spent decomposing the formula.

As explained in the previous section, SDDs have been designed to offer an alternative to OBDDs. In particular, an `apply` operator can be defined for SDDs that work similarly to the operator for decision diagram [Dar11]. Given two SDDs s_1 and s_2, the `apply` operator can create an SDD s representing either the conjunction or disjunction of s_1 and s_2. Hence, creating an SDD from a CNF formula is possible by first converting each clause to an SDD and combining them using the `apply` operator. Such compilation is called *bottom-up* since it starts from the clauses to construct a diagram for the whole formula.

Later, an improved bottom-up compilation algorithm has been proposed for dynamically minimising SDDs [CD13]. Since SDDs are canonical, given a vtree, this new method proposes an alternative way of exploring the space of vtrees. Choi and Darwiche propose two operations for exploring such space: rotating a vtree and swapping nodes. Then, their algorithm greedily searches for a good vtree using these operations. There also exist *top-down* compilers for SDDs; the idea is to start from the whole formula, compile parts of it and then combine them together [OD15].

Recently, a new representation, called CCDD, has been designed based on the notion of *literal equivalence* [LMY21]. Two literals are equivalent if interverting them results in a logically equivalent formula. Lai, Meel, and Yap show that CCDD is a generalisation of d-DNNF which supports linear time model counting and proposes `ExactMC`, an exact compiler to this new language.

2.1.6 Approximate Model Counting

Given the intrinsic complexity of model counting, several methods have been developed to avoid exploring the whole search space. In particular, a popular approach is to compute an *approximate count*; that is, a count that may differ from the true count but with an error bounded in some way. This section

reviews some of the most popular approaches.

ApproxCount is an approximate model counter based on sampling models of the boolean formula to estimate its count [WS05]. However, this early approach requires a (near-)uniform sampling to be effective, which is known to be a complex problem. Moreover, ApproxCount does not guarantee the quality of its estimation. SampleCount is another approximate model counter based on sampling solutions from a boolean formula [Gom+07]. However, it does not directly use the samples to estimate the model count. Instead, the samples are used to select an assignment of a variable that divides the solution space evenly (i.e., a variable that appears as positively as it does negatively in the models). The authors demonstrate that it is possible to provide a statistical guarantee on the lower bound returned by SampleCount and that it does not require (near-)uniform sampling to function effectively. PartialKC is an approximate model counter based on partial compilation to CCDD [LMY22]. It iteratively produces partial diagrams that are increasingly larger; hence, a key advantage of PartialKC is that it detects when the CCDD is complete and returns the exact count.

Another popular type of approximate algorithm produces (ε, δ)-approximations, also called *Probably Approximately Correct* (PAC) approximations in the literature [Val84], defined as follows.

Definition 4 ((ε, δ)-approximation)**.** *Let c be the true (weighted) model count of a boolean formula, $\hat{c}$ an approximation of c, and $\varepsilon > 0$ an error factor. $\hat{c}$ is an ε-approximation of c if and only if the following inequalities hold:*

$$\frac{c}{1+\varepsilon} \leq \hat{c} \leq c(1+\varepsilon).$$

Moreover, for $0 \leq \delta \leq 1$, $\hat{c}$ is an (ε, δ)-approximation if the following inequality holds:

$$P\left[\frac{c}{1+\varepsilon} \leq \hat{c} \leq c(1+\varepsilon)\right] \geq 1-\delta$$

Ganak is designed to compute $(0, \delta)$-approximations due to its probabilistic cache. However, it has been observed that it often returns the actual model count. One of the most popular approximate model counters computing (ε, δ)-approximations is ApproxMC [CMV16; SM19; SGM]. The main idea behind ApproxMC is to use hash functions to divide the solution space into cells containing the same number of solutions. Then, when the cells are small enough, an exact model counter is used to calculate the model count of a single cell. Then, this count is multiplied by the number of cells to produce an (ε, δ)-approximation of the true model count. The primary challenge in ApproxMC is to ensure that the correct hash function is employed, thereby ensuring the final count meets the required guarantees. WeightMC is based on the same ideas as ApproxMC but targets weighted model counting [Cha+14].

2.2 Probabilistic Inference

A *probabilistic model* can be seen as modelling a joint probability distribution over a set of random variables. Several tasks can be defined for such models, such as computing the probability of a partial observation of the variables or determining their most likely assignment. Finding the solution to such tasks is called *probabilistic inference*, and many model- and task-dependent algorithms exist to perform it. However, over the last two decades, SAT-based methods have emerged as a unified approach to addressing various probabilistic inference tasks. This section shows how some probabilistic inference tasks can be reduced to weighted model counting. We first explain the intuition behind this reduction before detailing how it is implemented for the probabilistic inference tasks used in our experiments.

2.2.1 Reducing Probabilistic Inference to Model Counting

To explain the similarity between probabilistic inference and model counting, let us consider Example 2.2.1 in which we construct a small probabilistic model and perform inference with a straightforward approach.

Example 10: Probabilistic Inference by World Enumeration
Let us consider a small probabilistic model containing the following variables: `smokes`, `bronchitis`, and `dyspnoea`. We assume that having dyspnoea depends on both smoking and having bronchitis, which we assume are independent events. A probability distribution is defined as follows. First, we define the distribution of the two independent variables.

- $P(\text{smokes} = \text{yes}) = 0.2, P(\text{smokes}) = \text{sometimes}) = 0.1, P(\text{smoking} = \text{no}) = 0.7$
- $P(\text{bronchitis} = \text{yes}) = 0.1, P(\text{bronchitis} = \text{no}) = 0.9$

Now, we can define the distribution for the variable `bronchitis`, which depends on the two other variables.

$$P(\text{dyspnoea} = \text{yes}) = \begin{cases} 1.0 & \text{if smokes = yes and bronchitis = yes} \\ 0.5 & \text{if smokes = sometimes and bronchitis = yes} \\ 0.0 & \text{otherwise} \end{cases}$$

$$P(\text{dyspnoea} = \text{no}) = \begin{cases} 0.0 & \text{if smokes = yes and bronchitis = yes} \\ 0.5 & \text{if smokes = sometimes and bronchitis = yes} \\ 1.0 & \text{otherwise} \end{cases}$$

The joint distribution over all three variables can be represented by an array that contains all possible combinations of the variables. Each possible world has an associated probability computed by multiplying the result of each probability distribution.

Let us assume the following query: What is the probability that a person has dyspnoea and smokes? The simplest way to answer this query is to look into the joint probability table, select all entries corresponding to "smokes = yes, Dyspnoea = yes," and sum their probabilities. The entries corresponding to the query are highlighted in green, and the query has a probability of 0.02. The entries for another query, "What is the probability of not having dyspnoea?" are highlighted in pink.

Smokes	Bronchitis	Dyspnoea	Proba.
yes	yes	yes	0.02
yes	yes	no	0.0
yes	no	yes	0.0
yes	no	no	0.18
Sometimes	yes	yes	0.005
Sometimes	yes	no	0.005
Sometimes	no	yes	0.0
Sometimes	no	no	0.09
no	yes	yes	0.0
no	yes	no	0.07
no	no	yes	0.0
no	no	no	0.63

This example illustrates that weighted model counting and probabilistic inference share a common structure. Both approaches enumerate possible worlds (i.e., possible assignments to variables, either binary or random), select the ones satisfying a set of constraints (i.e., a boolean formula or a query), and sum their weights. Hence, it has been proposed in the literature to reduce probabilistic inference to (projected) weighted model counting using the process illustrated in Figure 2.1. First, the probabilistic model and the query are transformed into a literal-weighted CNF formula F using an *encoding algorithm*. This encoding part is specific for each model and not unique; a given probabilistic model can be encoded in multiple ways using propositional logic. Moreover, the encoding must ensure that the weighted model count of F corresponds to the query's probability given the model. Such a guarantee is obtained by weighting the literals using the model's distribution and transforming its constraints (e.g., smoking and having bronchitis both influence having dyspnoea) into clauses. Then, any weighted model counter

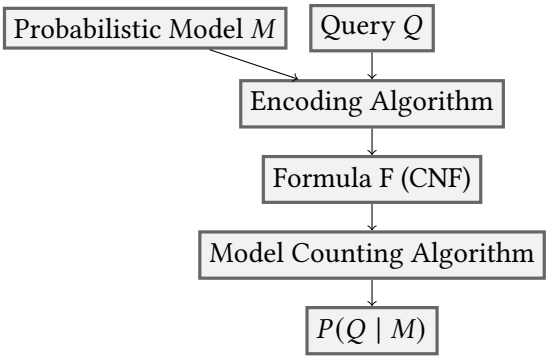


Figure 2.1: Process for solving probabilistic inference task using Weighted #∃SAT.

can be used on F to compute the probability of the initial query.

Using weighted model counting instead of specialised algorithms presents some benefits. First, decomposing the inference process into two steps — encoding and counting — allows for a unified approach to multiple probabilistic models and queries. Moreover, the encoding and counting algorithms are independent; hence, improving one of these two phases enhances the quality of the inference process. For example, modifying the pre-processing of the counting algorithm may improve performance without altering the encoding. Finally, using Knowledge Compilation, the inference task can be represented as diagrams that can be used in other pipelines, such as NeSy frameworks.

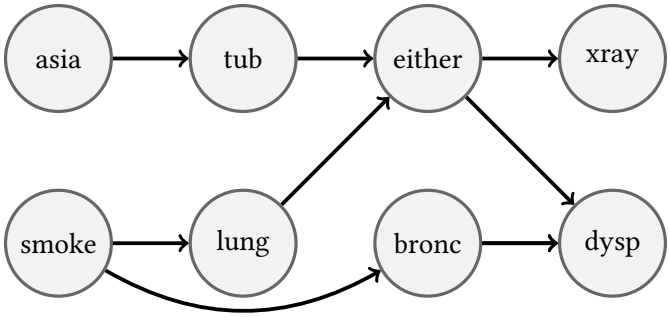
We now define the inference tasks used in our experiments. We first formally define the probabilistic models and the inference tasks considered. Then, we show how they can be encoded into a literal-weighted CNF formula.

2.2.2 Bayesian Networks

Bayesian networks (BNs) are models that belong to the category of probabilistic graphical models. A BN uses a graph structure to represent dependencies between random variables. In particular, a BN is a directed acyclic graph, and each node is associated with a conditional probability distribution.

Example: Bayesian Networks

A Bayesian network can be represented as a graph. Each node represents a random variable, and the edges represent the dependencies (or lack thereof) of one variable on another. For example, the following BN reproduced from [LS88] has eight variables.



Each node is associated with a *Conditional Probability Table* (CPT) that defines the node's probability distribution. Such distributions are conditional on the node's parents (e.g., $P(\text{bronc} \mid \text{smoke})$) if they exist. Setting probabilities to 1.0 and 0.0 in a CPT makes encoding determinism in a Bayesian network possible. For example, the CPTs for nodes `asia`, `bronc`, and `either` are listed below.

asia	
$\top$	$\bot$
0.1	0.99

	bronc	
smoke	$\top$	$\bot$
$\top$	0.6	0.4
$\bot$	0.3	0.7

		either	
tub	lung	$\top$	$\bot$
$\top$	$\top$	1	0
$\top$	$\bot$	1	0
$\bot$	$\top$	1	0
$\bot$	$\bot$	0	1

Let us now formally define a Bayesian Network. Let $(\boldsymbol{V}, \boldsymbol{\Phi})$ be a Bayesian network over random variables $\boldsymbol{V}$. The probability tables are represented using the weight functions $\boldsymbol{\Phi}$ and $\Phi_V \in \boldsymbol{\Phi}$ is the table associated with node $V \in \boldsymbol{V}$. The scope of Φ_V is the set of variables used in V's CPT and is defined by $\text{scope}(\Phi_V) = \{V, Y_1, \ldots, Y_p\}$ where $Y_1, \ldots, Y_p$ are the parents of V. It follows from our definition of scope that a Bayesian network can be seen as a directed graph. To be valid, the variables' scope must define an acyclic graph.

Using these notations, the weight function of $V \in \boldsymbol{V}$ can be defined as $\Phi_V : \bigtimes_{Z \in \text{scope}(\Phi_V)} \text{dom}(Z) \mapsto [0, 1]$. The two following properties hold since the weight functions must define valid probability distributions:

1. $\Phi_V(z) \in [0, 1]\ \forall z \in \bigtimes Z \in \text{scope}(\Phi_V)\ \text{dom}(Z)$,
2. $\sum_{v \in \text{dom}(V)} \Phi(v, y_1, \ldots, y_n) = 1\ \forall (y_1, \ldots, y_p) \in \text{dom}(Y_1) \times \cdots \times \text{dom}(Y_p)$.

Let $\boldsymbol{v}$ be an assignment to the network's variable and

$$\pi_V : \bigtimes_{Y \in \boldsymbol{V}} \text{dom}(Y) \mapsto \bigtimes_{Z \in \text{scope}(\Phi_V)} \text{dom}(Z)$$

be a projection operator from the whole set of variables to the subset of variables in scope(Φ_V). Then, the probability of the assignment is computed as follows.

$$P(\boldsymbol{V} = \boldsymbol{v}) = \prod_{V \in \boldsymbol{V}} \Phi_V(\pi_V(\boldsymbol{v}))$$

In other words, in Bayesian networks, the joint probability of the variables is factorised as a product of conditional probabilities. A natural task in such networks is to compute the probability of a partial observation of the networks' variables, called *evidence*.

Inference Task 1 (Probability of Evidence (PR))**.** *Let $(\boldsymbol{V}, \boldsymbol{\Phi})$ be a Bayesian network, $\boldsymbol{E} \subseteq \boldsymbol{V}$ a set of observed variables with value $\boldsymbol{e}$, and $\boldsymbol{X} = \boldsymbol{V} \setminus \boldsymbol{E}$ the remaining variables. The PR task is to compute $P(\boldsymbol{E} = \boldsymbol{e})$, which is given by the following formula:*

$$\begin{aligned} P(\boldsymbol{E} = \boldsymbol{e}) &= \sum_{\boldsymbol{x} \in \bigtimes_{X \in \boldsymbol{X}} \mathrm{dom}(X)} P(\boldsymbol{E} = \boldsymbol{e}, \boldsymbol{X} = \boldsymbol{x}) \\ &= \sum_{\boldsymbol{x} \in \bigtimes_{X \in \boldsymbol{X}} \mathrm{dom}(X)} \prod_{V \in \boldsymbol{V}} \Phi_V(\pi_V(\boldsymbol{e} \cup \boldsymbol{x})) \end{aligned}$$

In the above formula, the $\cup$ operator denotes the concatenation of the assignment.

Notice how Task 1 fits our intuitive explanation of probabilistic inference. Remember that a possible world for a Bayesian network is an assignment to all its variables. Hence, by summing all the possible assignments for $\boldsymbol{X} = \boldsymbol{V} \setminus \boldsymbol{E}$, all partial worlds on $\boldsymbol{X}$ are considered and then extended into a complete possible world by the imposed assignment $\boldsymbol{E} = \boldsymbol{e}$.

2.2.2.1 Solving the PR Task with `Toulbar2`

Before explaining how Bayesian networks can be encoded in CNF, let us briefly describe the `Toulbar2` solver and explain its connection to Bayesian networks as we utilise it in our experiments. `Toulbar2` is a solver for *Cost Function Networks* (CFN). Let $\boldsymbol{X}$ be a set of discrete variables; in a CFN, the constraints are expressed as a set of function $\boldsymbol{F}$ over variables. A function $F_{\boldsymbol{Y}} \in \boldsymbol{F}$ is associated with a subset $\boldsymbol{Y} \subseteq \boldsymbol{X}$ of the variables and assigns a weight to each possible assignment $\boldsymbol{y}$ to them. The weight of an assignment $\boldsymbol{v}$ of the variable is defined as the sum of each function evaluation with $\boldsymbol{v}$ (projected on the variables in the function's scope).

The similarity with Bayesian networks is evident: BNs can be encoded as CFN by taking $\boldsymbol{V} = \boldsymbol{X}$, $\boldsymbol{\Phi} = \boldsymbol{F}$, and using log-probabilities. Hence, solving Inference task 1 with `Toulbar2` is straightforward: the solver finds all

assignments respecting the constraints and sums their weight. Many algorithms are implemented in `Toulbar2`; detailing all of them is outside the scope of this work. However, at the core, `Toulbar2` is a tree-search-based solver: it recursively branches on a variable $X \in \boldsymbol{X}$, heuristically selects one value x of its domain, and asserts $X = x$. Then, it conditions the functions based on this assignment and continues until a complete assignment is found.

2.2.2.2 CNF Encoding

Various encoding algorithms exist to transform a Bayesian network into a literal-weighted boolean formula (e.g.,[Dar02; CD05; CD06; CD08; Bar+16]). This section reviews the first encoding proposed by Darwiche [Dar02] and then explores some of its extensions.

Variables Two types of variables are used in the encoding: *indicator* variables are used to indicate which values a node takes, while *parameter* variables indicate which entry in the CPT is active. More formally, given a BN $(\boldsymbol{V}, \boldsymbol{\Phi})$, the following boolean variables are created.

$$\lambda_v \in \{\top, \bot\} \qquad \forall v \in \mathrm{dom}(V), \quad \forall V \in \boldsymbol{V} \tag{2.1}$$

$$\theta_{\boldsymbol{z}} \in \{\top, \bot\} \qquad \forall \boldsymbol{z} \in \bigtimes_{Z \in \mathrm{scope}(\Phi_V)} \mathrm{dom}(Z), \quad \forall V \in \boldsymbol{V} \tag{2.2}$$

Since these encodings have been designed for non-projected weighted model counting, each literal must be assigned a weight. Intuitively, the encoding is such that the weight of an interpretation of the boolean formula corresponds to the weight of the associated possible world of the Bayesian network. Remember that the weight of a complete assignment of a BN's variables is given by multiplying all CPTs' entries corresponding to the assignment. Hence, the probabilities defined in the CPTs must be given to the parameter variables. The following weighting scheme ensures that, given the correct set of clauses, each interpretation has the same weight as its corresponding possible world. Note that indicator variables and negative literals are assigned a unit weight; this ensures that they do not impact the probability, as 1 is neutral for multiplication.

$$\omega(\lambda_v) = \omega(\neg\lambda_v) = 1 \quad \forall v \in \mathrm{dom}(V), \qquad \forall V \in \boldsymbol{V} \tag{2.3}$$

$$\omega(\theta_{\boldsymbol{z}}) = \Phi_V(\boldsymbol{z}) \quad \forall \boldsymbol{z} \in \bigtimes_{Z \in \mathrm{scope}(\Phi_V)} \mathrm{dom}(Z), \qquad \forall V \in \boldsymbol{V} \tag{2.4}$$

$$\omega(\neg\theta_{\boldsymbol{z}}) = 1 \quad \forall \boldsymbol{z} \in \bigtimes_{Z \in \mathrm{scope}(\Phi_V)} \mathrm{dom}(Z), \qquad \forall V \in \boldsymbol{V} \tag{2.5}$$

Clauses The clauses ensure that any interpretation of F corresponds to a possible world of the Bayesian network. The following clauses impose that exactly one value must be selected for each node $V \in \boldsymbol{V}$.

$$\vee_{v \in \mathrm{dom}(V)} \lambda_v \quad \forall V \in \boldsymbol{V} \tag{2.6}$$

$$\neg\lambda_{v_i} \vee \neg\lambda_{v_j} \quad \forall v_i, v_j \in \mathrm{dom}(V)(v_i \neq v_j),\ \forall V \in \boldsymbol{V} \tag{2.7}$$

Clause (2.6) corresponds to a *at least one constraint* and Clauses (2.7) encodes an *at most one* constraint.

Finally, the last set of clauses encodes the BN's constraints; when all nodes in a CPT's scope have been assigned (i.e., their indicator variables are fixed), then the corresponding parameter variable must be set to $\top$.

$$\lambda_v \wedge \lambda_{y_1} \wedge \ldots \wedge \lambda_{y_p} \Leftrightarrow \theta_{v y_1 \ldots y_p} \quad \forall (v, y_1, \ldots, y_p) \in \bigtimes_{Z \in \mathrm{scope}(\Phi_V)} dom(Z)$$

$$\forall \Phi_V \in \boldsymbol{\Phi} \tag{2.8}$$

Evidence The evidences are encoded using the indicator variables with the following clauses:

$$\lambda_{x_i} \quad \forall x_i \in \boldsymbol{e} \tag{2.9}$$

It has been proved that with such an encoding, it is possible to solve Inference task 1 using weighted model counting [Dar02].

Theorem 1. *Let $(\boldsymbol{V}, \boldsymbol{\Phi})$ be a Bayesian network, $\boldsymbol{E} \subseteq \boldsymbol{V}$ a set of evidence, and $\boldsymbol{e}$ an assignment to $\boldsymbol{E}$. Let F be a boolean formula defined over the set of variables defined by Equations* (2.1)-(2.2)*, using the weighting scheme defined by Equations* (2.3)-(2.5)*, and the clauses defined by Equations* (2.6)-(2.9)*. Then, the following equality holds.*

$$P(\boldsymbol{E} = \boldsymbol{e}) = \textit{weighted \#SAT}(F)$$

Example: Bayesian Networks (cont.)

As an example, let us show how a small part of our example network can be encoded as a boolean formula. We will define the variables and clauses for the node `asia` and `bronc`. The latter requires we define some variables for the `smoke` node. For conciseness, we denote the network's nodes by the first letter of the variables they represent.

First, we have the following indicator variables, one for each node's value. For example, if the variable $\lambda_{A_\top}$ is true, it indicates that the person has been to Asia.

$$\lambda_{A_\top}, \lambda_{A_\bot}, \lambda_{B_\top}, \lambda_{B_\bot}, \lambda_{S_\top}, \lambda_{S_\bot}$$

Moreover, the following six parameter variables (one for each CPT entry) are required.

$$\theta_{A_\top}, \theta_{A_\bot}, \theta_{B_\top S_\top}, \theta_{B_\bot S_\top}, \theta_{B_\top S_\bot}, \theta_{B_\bot S_\bot}$$

Their weights are defined following the CPTs' entries.

$$\omega(\theta_{A_\top}) = 0.01 \qquad \omega(\theta_{B_\top S_\top}) = 0.6 \qquad \omega(\theta_{B_\top S_\bot}) = 0.3$$
$$\omega(\theta_{A_\bot}) = 0.99 \qquad \omega(\theta_{B_\bot S_\top}) = 0.4 \qquad \omega(\theta_{B_\bot S_\bot}) = 0.7$$

Finally, one clause is defined per CPT entry.

$$\lambda_{A_\top} \Leftrightarrow \theta_{A_\top} \qquad \lambda_{B_\top} \wedge \lambda_{S_\top} \Leftrightarrow \theta_{B_\top S_\top} \qquad \lambda_{B_\bot} \wedge \lambda_{S_\top} \Leftrightarrow \theta_{B_\bot S_\top}$$
$$\lambda_{A_\bot} \Leftrightarrow \theta_{A_\bot} \qquad \lambda_{B_\top} \wedge \lambda_{S_\bot} \Leftrightarrow \theta_{B_\top S_\bot} \qquad \lambda_{B_\bot} \wedge \lambda_{S_\bot} \Leftrightarrow \theta_{B_\bot S_\bot}$$

Assuming that all CPTs of the network have been encoded that way, the query $P(\text{dysp} = \top)$ can be encoded by adding the following unit clause $\lambda_{D_\top}$.

The encoding presented above serves as the basis for various other encodings, and, using the same naming convention as in [CD08], will be denoted ENC1. One of the crucial optimisations for scalable model counting is encoding the BN's local structure into the CNF formula [CD05]. In particular, the encoding ENC3 uses only one parameter variable to represent multiple entries of *the same CPT* with the same weights, which can drastically reduce the number of variables in the final formula. Decomposing a problem into independent sub-problems is a crucial aspect of model counting. However, it has been shown that classical counting algorithms sometimes fail to detect these sub-problems due to the encoding [CD06]. The encoding ENC4 is designed to enhance decomposability (while conserving the advantages of ENC3) by reducing the clauses using prime applicants. Finally, a recent encoding, called ENC4LINP, proposed to encode implicitly the most frequent weights in each CPT as well as log-encodings to reduce the size of the resulting formula [Bar+16].

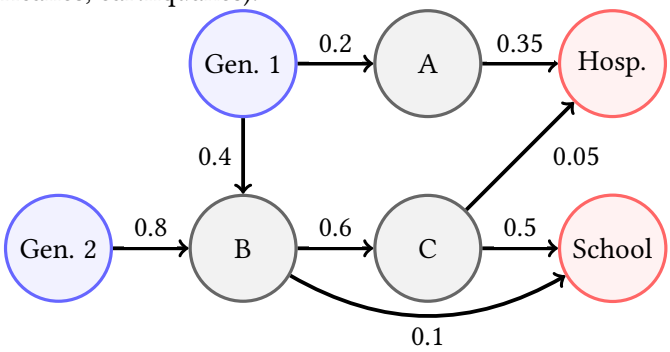
2.2.3 Probabilistic Graphs

Graph structures can model many real-world situations, such as road networks, social interactions, or electrical grid installations. In such contexts, computing the reliability of these systems (e.g., the probability that a power plant remains connected to a hospital or the spread of misinformation in social networks) can be crucial. In this section, we explore how probabilistic graphs can be used to model such situations and how to encode reliability queries using propositional formulas. Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a graph (directed or

undirected) with nodes $\boldsymbol{V} = \{V_1, \ldots, V_n\}$ and edges $\boldsymbol{E} = \{E_1, \ldots, E_m\}$. Moreover, each edge $E \in \boldsymbol{E}$ has an associated probability $P(E)$ of being active.

Example: Probabilistic Graph: Electrical network

As a running example, consider the following graph, which represents a simplified electrical network for a small city. In this example, two nodes (in blue) generate the electricity and send it to target nodes (in red) through the network, passing through relays (grey nodes). Each edge is annotated with its probability of being functional after some natural event (e.g., flooding, hurricanes, earthquakes).



This work focuses on a reliability problem called $(S-T)$-reliability. In such a problem, the goal is to compute the probability that a source node $S \in \boldsymbol{V}$ is connected to a target node $T \in \boldsymbol{V}$. The probability that S and T are connected can be computed from the graph's possible worlds. Let $\boldsymbol{X} \subseteq \boldsymbol{E}$ be the subset of edges that are still functional, then the probability of this world is given by

$$P(\boldsymbol{X}) = \left(\prod_{E \in \boldsymbol{X}} P(E)\right) \times \left(\prod_{E \notin \boldsymbol{X}} (1 - P(E))\right).$$

A probabilistic graph defines a distribution over non-probabilistic graphs. Verifying whether each possible world contains an (S-T) path is easy with classical graph algorithms. Let us denote $\boldsymbol{X} \models \text{path}(S, T)$ the fact that there exists an (S-T)-path in the graph $(\boldsymbol{V}, \boldsymbol{X})$, then the problem of reliability estimation is as follows.

Inference Task 2 (Reliability Estimation)**.** *Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a weighted graph, $S \in \boldsymbol{V}$ a source node, $T \in \boldsymbol{V}$ a target node, and $P(E)$ the weight of each edge E. The task of reliability estimation is to compute the probability that an (S-T) path exists in G. It is defined as follows:*

$$P(path(S, T)) = \sum_{\boldsymbol{X} \subseteq \boldsymbol{E} | \boldsymbol{X} \models path(S,T)} P(\boldsymbol{X}).$$

2.2.3.1 CNF Encoding

Contrary to Bayesian networks, the encoding presented in this section relies on using *projected* weighted model counting [Due+17]. This encoding is special because it is not used to compute the required probability but the complementary probability (i.e., the probability that the source and the target are not connected). We first present the encoding, and then show that using *unprojected* weighted model counting or encoding directly the query do not work.

Variables The set of projected variables used to compute the probability of an interpretation are linked to the edges and represent the fact that an edge is active. On the other hand, to help formalise the problem constraints, one boolean variable is used per node. They are used to indicate if a node is connected to the source. The following variables are defined:

$$\lambda_V \in \{\top, \bot\} \qquad \forall V \in \boldsymbol{V} \tag{2.10}$$

$$\theta_E \in \{\top, \bot\} \qquad \forall E \in \boldsymbol{E} \tag{2.11}$$

The set of projected variables is defined as $\boldsymbol{P} = \{\theta_E \mid E \in \boldsymbol{E}\}$. The weight of the projected variables is given by the edges' probabilities in the graph; hence, we have $\forall \theta_E \in \boldsymbol{P} : \ \omega(\theta_E) = P(E)$ and $\omega(\neg\theta_E) = 1 - P(E)$.

Clauses The clauses must encode the constraint that, given a choice for the edges, there is no path between S and T. The encoding relies on the transitive property of the graph's paths. That is, if the source is linked to a node $V \in \boldsymbol{V}$ and there is an edge between V and $V' \in \boldsymbol{V}$, then there is a path from the source to V'. The following clauses encode that behaviour.

$$\lambda_{V_1} \wedge \theta_E \Rightarrow \lambda_{V_2} \quad \forall E = (V_1, V_2) \in \boldsymbol{E} \tag{2.12}$$

The query Finally, given a source S and a target T, the query on the probabilistic graph can be encoded using the two following unit clauses.

$$\lambda_S \quad \text{for a source node } S \in \boldsymbol{V} \tag{2.13}$$

$$\neg\lambda_T \quad \text{for a target node } T \in \boldsymbol{V} \tag{2.14}$$

Example: Probabilistic Graph: Electrical Network (cont.)
Let us encode our small example to compute the following query: What is the probability that the first generator and the school are connected? First, the following boolean variables are required for the nodes (we use G, H,

and S to refer to the generators, the hospital, and the school, respectively).

$$\lambda_{G_1}, \lambda_{G_2}, \lambda_A, \lambda_B, \lambda_C, \lambda_H, \lambda_S \in \{\top, \bot\}$$

Next, we identify each edge with a boolean variable using the nomenclature $\theta_{\text{from to}}$. Hence, the following variables are defined.

$$\theta_{G_1A}, \theta_{G_1B}, \theta_{G_2B}, \theta_{AH}, \theta_{AC}, \theta_{CA}, \theta_{BC}, \theta_{BS}, \theta_{CH}, \theta_{CS} \in \{\top, \bot\}$$

Finally, the following clauses encode the structure of the graph.

$$\begin{array}{lll} \lambda_{G_1} \wedge \theta_{G_1A} \Rightarrow \lambda_A & \lambda_B \wedge \theta_{BS} \Rightarrow \lambda_S & \lambda_C \wedge \theta_{CH} \Rightarrow \lambda_H \\ \lambda_{G_1} \wedge \theta_{G_1B} \Rightarrow \lambda_B & \lambda_B \wedge \theta_{BC} \Rightarrow \lambda_C & \lambda_C \wedge \theta_{CS} \Rightarrow \lambda_S \\ \lambda_{G_2} \wedge \theta_{G_2B} \Rightarrow \lambda_B & \lambda_A \wedge \theta_{AH} \Rightarrow \lambda_H & \end{array}$$

The query is encoded using the following two unit clauses: λ_{G_1} and $\neg\lambda_S$.

This encoding is easily extended to undirected graphs. Indeed, if the edges are undirected, the transitivity property holds in both directions. That is, if there is an active edge between a node X and a node Y (i.e., $\theta_{XY} = \top$), and either X or Y is reachable from the source, then the other node is also reachable from the source. For example, the following clauses encode the edge between X and Y.

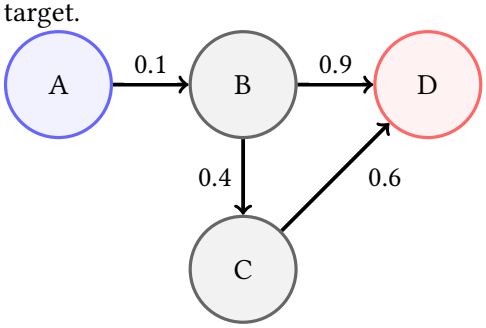
$$\lambda_X \wedge \theta_{XY} \Rightarrow \lambda_Y \qquad\qquad \lambda_Y \wedge \theta_{XY} \Rightarrow \lambda_X$$

The following theorem states that the encoding presented above is correct, and proof can be found in [Due+17].

Theorem 2. *Let $G = (\boldsymbol{V}, \boldsymbol{E})$ be a probabilistic graph such that each edge has an associated probability $P(E)$ of being present. Moreover, let $S \in \boldsymbol{V}$ be a source node and $T \in \boldsymbol{V}$ be a target node. Let F be a boolean formula defined over variables $\boldsymbol{\lambda}$ defined by Equation* (2.10)*, $\boldsymbol{\theta}$ defined by Equation* (2.11) *and the clauses defined by Equations* (2.12)-(2.14)*. Then, the probability of S and T being connected in G is given by the following value:* $1 - \text{pwmc}(F, \boldsymbol{\theta})$.

Two particularities of the encoding presented above are necessary: using projected variables and solving the complementary connectivity problem. Let us demonstrate how, without these two elements, the encoding presented above would not work.

Example 15: Non-working encodings for ($S-T$)-reliability
Let us consider the small network below, where A is the source and D is the target.



The following clauses encode the problem of computing the probability that A and D are disconnected.

$$\lambda_A \wedge \theta_{AB} \Rightarrow \lambda_B \qquad \lambda_B \wedge \theta_{BC} \Rightarrow \lambda_C \qquad \lambda_B \wedge \theta_{BD} \Rightarrow \lambda_D$$
$$\lambda_C \wedge \theta_{CD} \Rightarrow \lambda_D \qquad \lambda_A \qquad \neg\lambda_D$$

Case 1: unprojected weighted model counting) Let the graph's probabilities assign weights to each θ variable, and let the weight of each λ variable be 1. Now, let us assume a DPLL-style solver which sets, in order, $\theta_{AB} = \top, \theta_{BC} = \bot$, and $\theta_{BD} = \bot$. The first assignment forces $\lambda_B = \top$ while the others only remove clauses from the formula. Hence, the residual formula obtained after the third assignment is

$$\lambda_C \wedge \theta_{CD} \Rightarrow \bot \Leftrightarrow \neg\lambda_C \vee \neg\theta_{CD}$$

The truth table for that formula and the associated weights are shown below.

θ_{CD}	λ_C	weight
$\top$	$\top$	0.6
$\top$	$\bot$	0.6
$\bot$	$\top$	0.4
$\bot$	$\bot$	0.4

At that point, the assignments correspond to a world where A and D are disconnected; hence, 1 *must* be returned. However, the weighted model count of the residual formula is 1.4 since only the assignment $\theta_{CD} = \top, \lambda_C = \top$ is not a model of the residual formula. Such a problem is avoided when projected variables are used. The residual formula only has two projected models, $\theta_{CD} = \top$ and $\theta_{CD} = \bot$, whose weighted sum is 1.

Case 2: not encoding the complementary problem) Now let us consider the impact of encoding the query using the clause λ_D instead of $\neg\lambda_D$ but with the projected variables as defined in the encoding (i.e., the $\boldsymbol{\theta}$ variables). In such a case, during the first propagation, the two clauses implying λ_D are removed because an implication implying $\top$ is always respected. The residual formula is as follows:

$$\theta_{AB} \Rightarrow \lambda_B \qquad\qquad \lambda_B \wedge \theta_{BC} \Rightarrow \lambda_C$$

Using the same reasoning as above, it is clear that any assignment on θ_{AB} and θ_{BC} can be extended by the assignment $\lambda_B = \lambda_C = \top$ to form a model of this residual formula. Hence, we have $\text{pwmc}(F, \boldsymbol{\theta}) = 1$, which is incorrect. Using projected variables means we cannot directly encode the query, as it does not constrain the problem.

2.2.4 Probabilistic Logic Programming with ProbLog

ProbLog is a probabilistic extension of the Prolog language; it consists of a (deterministic) logic program (LP) and probabilistic facts [DKT07]. In that sense, a ProbLog program defines a distribution over logic programs: an element of the distribution is the logic program LP extended with a subset of the probabilistic facts. Since ProbLog is an extension of Prolog, their syntaxes are similar. A ProbLog program consists of a set of rules, defined using the `:-` operator, which are logical implications. Like Prolog, the implications are written from right to left; hence, the logical implication $A \wedge B \wedge C \Rightarrow D$ is written as follows.

```
D :- A,B,C.
```

Notice that commas replace the logical $\wedge$ operator, and the dot at the end finishes the clauses. Although negations can be added to the right-hand side of the rules, this work only considers the setting in which each rule variable is positive. We denote the left-hand side of the rule the *head* and its right-hand side the *implicant*. In ProbLog, the head of a rule might contain multiple variables annotated with probabilities. Such rules are called *annotated disjunctions* (AD) [VVB04]. For example, the following rule defines, in practice, a probability distribution over three rules.

```
0.3::A; 0.5::B; 0.2::C :- D.
```

Such a rule can be understood as follows: if D is true, then A is true with a probability of 0.3, B is true with a probability of 0.5, or C is true with a probability of 0.2. In practice, this can be seen as encoding the tree following rules: `A :- D, P1.`, `B :- D, P2.`, and `C :- D, P3.` with $P(P1) = 0.3$, $P(P2) =$

0.5, and $P(P3) = 0.2$. The additional variables form a distribution, where exactly one must be true, and their weights sum to 1. When one variable is set to $\top$, the associated rule is active, and the other variables are set to $\bot$, deactivating their respective rules. Hence, an annotated disjunction can be seen as defining a distribution over (deterministic) Prolog programs.

Finally, a ProbLog program can be queried using the `query` operator. Let us show how Bayesian networks and probabilistic graphs can be encoded as a ProbLog program.

Example 16: Bayesian Network and Reliability Estimation problems in ProbLog

The ProbLog program encoding the Bayesian network used in our previous examples is shown below. Each probability table is encoded as a set of rules (lines 2-26) followed by a query (line 28). The network's distributions are encoded using annotated disjunctions. Each rule of the program encodes a row in a given CPT. Let us look more closely at some rules.

First, it can be seen that there is no implicant for `asia` (line 2) since it has no parent in the network. Hence, these rules define *probabilistic facts*. Next, the rules for the `bronch` variables (lines 12-13) are classical annotated disjunctions. There is one AD per CPT row. The implicant of each rule is the choice for the `smoke` variable (the only parent of `bronch`), and the heads are annotated with the CPT probabilities. Finally, when a CPT contains determinism, such as for `either`, the CPT can be encoded using non-probabilistic rules (lines 15-18).

```
1  // CPT for Asia
2  0.01::asia_yes; 0.99::asia_no.
3  // CPT for tub
4  0.05::tub_yes; 0.95::tub_no :- asia_yes.
5  0.01::tub_yes; 0.99::tub_no :- asia_no.
6  // CPT for Smoke
7  0.5::smoke_yes; 0.5::smoke_no.
8  // CPT for lung
9  0.1::lung_yes; 0.9::lung_no :- smoke_yes;
10 0.01::lung_yes; 0.99::lung_no :- smoke_no;
11 // CPT for Bronch
12 0.6::bronch_yes_; 0.4::bronch_no :- smoke_yes.
13 0.3::bronch_yes; 0.7::bronch_no :- smoke_no.
14 // CPT for either
15 either_yes :- lung_yes, tub_yes.
16 either_yes :- lung_no, tub_yes.
17 either_yes :- lung_yes, tub_no.
18 either_no :- lung_no, tub_no.
19 // CPT for dysp
20 0.9::dysp_yes; 0.1::dysp_no :- bronch_yes, either_yes.
21 0.7::dysp_yes; 0.3::dysp_no :- bronch_no, either_yes.
22 0.8::dysp_yes; 0.2::dysp_no :- bronch_yes, either_no.
```

```
23 0.1::dysp_yes; 0.9::dysp_no :- bronch_no, either_no.
24 // CPT for xray
25 0.98::xray_yes; 0.02::xray_no :- either_yes.
26 0.05::xray_yes; 0.95::xray_no :- either_no.
27 // Query
28 query(dysp_yes).
```

As for Bayesian networks, let us examine the graph used previously to illustrate how reliability estimation problems can be described in ProbLog. The ProbLog program for this network, assuming a query between the first generator and the hospital, is below. Notice that the structure of this program slightly differs from the one for Bayesian networks. The first part of the program defines a set of probabilistic facts (lines 1-8), defining the binary distributions of the edges, followed by a (deterministic) logic program encodings the reachability property (lines 9-10). Notice that, in this latter part, `X` and `Y` are not atoms; they are variables that can be replaced by the program's atoms (e.g., `gen1`, `a`). Hence, this program states that there is an path from a variable `X` to a variable `Y` if either: i) There is an edge from `X` to `Y` or, ii) There is a edge from `X` to another variable `Z` and a path from `Z` to `Y`.

```
1 0.2::edge(gen1,a).
2 0.4::edge(gen1,b).
3 0.8::edge(gen2, b).
4 0.6::edge(b,c).
5 0.1::edge(b,school).
6 0.35::edge(a,hosp).
7 0.05::edge(c,hosp).
8 0.5::edge(c,school).
9 path(X, Y) :- edge(X, Y).
10 path(X, Y) :- edge(X, Z), path(Z, Y).
11 query(path(gen1, hosp)).
```

Before formalising a ProbLog program and the distribution it defines, let us explain its solving process. The goal of a ProbLog program is to compute the probability of the query (e.g. `query(path(gen1, hosp)).`); hence, it must find in which cases the query is respected (e.g., when is there a path from `gen1` to `hosp`). To do so, it leverages the Prolog proof system, which tries to prove the query by contradiction: it asserts that the query is falsified and tries to find a contradiction (i.e., $\top \implies \bot$).

The base of Prolog proof system is the *resolution rule*, an inference rule for propositional logic. Let $C_1 = \neg A_1 \vee \neg A_2 \vee D$ and $C_2 = B \vee \neg D$ be two clauses, then the resolution rule states that it is possible to infer a new clause $C = \neg A_1 \vee \neg A_2 \vee B$. Another way of understanding resolution is to rewrite the clauses as an implication: $C_1 = (A_1 \wedge A_2) \Rightarrow D$ and $C_2 = D \implies B$. The first clause states that if A_1 and A_2 are true, then D must be true. Similarly,

the second clause states that if D is true, B must be true. Hence, if both A_1 and A_2 are true, B must be true, as stated by the resolution rule.

This latter interpretation of the resolution rules makes it easier to understand Prolog's proof system. Each rule in a Prolog program is an implication, and the goal is to find a chain of implication, starting from the negated query, such that $\top \Rightarrow \bot$ is found. If $\top \Rightarrow \bot$ is found, the query is said to be *proven*.

Example: Prolog Proof System

Let us consider, as an example, our small reliability estimation program and show how Prolog proves it. For now, let us assume that it is a deterministic program (i.e., the probabilities are not taken into account)

```
1  0.2::edge(gen1,a).
2  0.4::edge(gen1,b).
3  0.8::edge(gen2, b).
4  0.6::edge(b,c).
5  0.1::edge(b,school).
6  0.35::edge(a,hosp).
7  0.05::edge(c,hosp).
8  0.5::edge(c,school).
9  path(X, Y) :- edge(X, Y).
10 path(X, Y) :- edge(X, Z), path(Z, Y).
11 query(path(gen1, hosp)).
```

Prolog starts with a single rule whose left-hand side is empty ($\bot$ by convention), and the right-hand side is the query, forcing it to be false. Then, it uses resolution to update the implicant of the clause until the right-hand side becomes empty ($\top$ by convention), which is a contradiction. Hence, in our example, the rules start as follows.

```
:- path(gen1, hosp).
```

Next, Prolog searches for a rule in the program that the resolution inference can use. It can be seen that there are no rules implying `path(gen1, hosp)` in the program; resolution can not be applied directly. However, two rules (lines 9, 10) have the `path` term on the left-hand side of the rule, but with variables. Let us assume that ProbLog decides to apply the resolution rule with line 9, which implies `path(X, Y)`. To be valid, the mappings `X = gen1` and `Y = hosp` must be applied first; then, resolution can be used. This process of renaming variables is called *unification*. Hence, after unification and resolution, the rule becomes

```
:- edge(gen1, hosp).
```

Then, no rule can be used by resolution (with or without unification); hence, Prolog can not prove a contradiction. Intuitively, it means there is no way of having `edge(gen1, hosp)` set to true. Hence, it must be false, leading to $\bot \Rightarrow \bot$.

However, there are still possibilities of finding a contradiction: previously, we decided to use resolution on our initial rule with the rule at line 9, but the rule at line 10 could also be used. Hence, in such a case, Prolog backtracks to such a decision point and explores other possibilities to prove the query. For example, a possible way of finding a contradiction is as follows (the rules on the left of the arrows are the ones used for the resolution):

```
                                                    :- path(gen1, hosp).
path(gen1,hosp):-edge(gen1,Z),path(Z,hosp). ↓
                                                    :-edge(gen1,Z),path(Z,hosp).
                                  edge(gen1,a). ↓
                                                    :-path(a,hosp).
                       path(a,hosp):-edge(a,hosp). ↓
                                                    :-edge(a,hosp).
                                    edge(a,hosp). ↓
                                                    :- .
```

The proof can be understood as follows. There is a path from `gen1` to `hops` if there is an edge from `gen1` to another node from which a path to `hosp` can be found (first resolution). It is known (from the facts) that there is an edge from `gen1` to `a`; hence, Prolog proves that there is a path from `a` to `hosp` (second resolution). If there is an edge from `a` to `hosp`, then there is a path between these nodes (third resolution). The facts of the program state that there is such an edge (last resolution); hence there is a path from `gen1` to `hosp` (the last rule is empty), which is a contradiction since Prolog assumed that no such path exists.

In practice, Prolog's resolution procedure can be seen as a tree of proof; it explores the possible proofs until a contradiction is found. If no such proof can be found, then the query cannot be proven from the program. Let Q be a query, and $\boldsymbol{L}$ be a Prolog program; we denote $\boldsymbol{L} \models Q$ the fact that Prolog's resolution procedure finds a proof for Q given $\boldsymbol{L}$.

Let us now formally define a ProbLog program, the probability distribution it defines, and the inference task it supports. Let $\boldsymbol{A}$ be a set of atoms, $\boldsymbol{R}$ rules (including facts) over $\boldsymbol{A}$, and $Q \in \boldsymbol{A}$ a query. The rules are partitioned into probabilistic rules $\boldsymbol{R}^P$ and a set of deterministic rules $\boldsymbol{R}^D$. We denote $P(R)$ the probability of a probabilistic rule $R \in \boldsymbol{R}^P$. Moreover, the set of probabilistic rules is further partitioned into n distributions (i.e., the rules defined by the annotated disjunctions) $\boldsymbol{R}^P_1, \ldots, \boldsymbol{R}^P_n$.

Such a program defines a distribution over (deterministic) logic programs (i.e., Prolog programs) as follows. A logic program is formed by taking a subset $\boldsymbol{X} \subseteq \boldsymbol{R}^P$ of the probabilistic rules. The probability of such a subset is

defined as follows.

$$P(\boldsymbol{X}) = \begin{cases} \prod_{R \in \boldsymbol{X}} P(R) & \text{if } |\boldsymbol{X} \cap \boldsymbol{R}_i^P| = 1 \; \forall 1 \leq i \leq n \\ 0 & \text{otherwise} \end{cases}$$

A logic program in the distribution defined by a ProbLog program is valid if it selects exactly one probabilistic rule per annotated disjunction. Given these definitions, we can now define ProbLog's inference task.

Inference Task 3. *Let $\boldsymbol{R}^P \cup \boldsymbol{R}^D$ be a ProbLog program and Q a query on this program. The inference task of ProbLog is to compute the probability that the query succeeds, which is calculated as follows.*

$$P(Q \mid \boldsymbol{R}) = \sum_{\boldsymbol{X} \subseteq \boldsymbol{R}^P \mid \boldsymbol{X} \cup \boldsymbol{R}^D \models Q} P(\boldsymbol{X})$$

In other words, the probability of a query is the weighted sum of all Prolog programs in which a proof for the query can be found.

2.2.5 Reducing ProbLog Inference to Weighted Model Counting

Recent implementations of ProbLog solve Inference task 3 by first transforming the ProbLog program into a weighted boolean formula and then computing its *unprojected* weighted model count [Fie+15; Vla+16]. This section provides a brief overview of the key aspects of such a reduction.

ProbLog relies on the Prolog proving system, which can be seen as a tree exploration of the possible proofs for the query. Moreover, all proofs must be considered to compute the query's probability of being respected. Hence, ProbLog's first step is to apply a modified version of Prolog's resolution algorithm to find all proofs for the query. During this first step, several *grounded terms* (i.e., terms without variables) are generated by the unification process; all such terms are used to create a *grounded program*.

This grounding procedure is necessary to eliminate the variables; creating a boolean formula with such variables would result in a formula in first-order logic. A grounded program, however, is similar to a propositional formula: each fact is either true or false. Moreover, the grounded program only contains terms necessary for the query computation. For example, the grounded program for a Bayesian network only contains terms related to the ancestors of the node queried.

Then, the grounded program can be transformed into a boolean formula. For a program without positive cycles (i.e., a set of rules depending positively on each other), this can be done using *Clark's completion* algorithm [Llo12; Jan04]. However, this algorithm does not work with positive cycles. Techniques to remove such loops are outside the scope of this work, but removing

such loops can produce CNF formulas exponentially larger than the initial ProbLog program [Vla+16]. In particular, it is known that reliability estimation problems possess such positive loops if the graph is cyclic. Hence, encoding such problems in ProbLog results in boolean formulas that might be exponentially larger than the initial graph, contrary to encodings based on projected weighted model counting. Finally, once the positive loops are broken, the weighted model count of the formula can be computed using state-of-the-art weighted model counters.

Schlandals Modelling Language

3

This chapter introduces Schlandals, a new modelling language based on CNF formulas designed explicitly for probabilistic inference problems. First, we motivate the development of a new modelling language for probabilistic inference tasks. Then, we formally define Schlandals and prove that the tasks described in Chapter 2 can be encoded in that language.

This chapter is based on the first part of the following article:

- A. Dubray, P. Schaus, and S. Nijssen. "Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses". In: *LIPIcs, Volume 280, CP 2023* 280 (2023). Ed. by R. H. C. Yap. ISSN: 1868-8969. DOI: 10.4230/LIPICS.CP.2023.15. (Visited on 03/07/2025)

In addition to the content from the original paper, the proofs that our new language can be used to model ProbLog programs and Bayesian networks are given.

3.1 Motivations for a New Modelling Language

One of the goals of this thesis is to explore how model counters can be specialised for probabilistic inference. While modern model counters are efficient, they are not designed with the assumption that the CNF formulas they count encode a probabilistic inference problem. For example, distributions are transformed into variables linked by clauses (i.e., *at least one* and *at most one* constraints). Model counters reason over the problem's structure (e.g., for the branching heuristic) based on the variables, but not how they are linked. Hence, one of Schlandals' goals is to have distributions as first-class citizens. We show in Chapter 5 that such design choice allows us to compute an upper bound on the weighted model count during the search.

We also observe that all the encodings presented in Chapter 2 share a similarity: all clauses except those encoding the distributions are Horn clauses (i.e., implications). A particularity of Horn formulas is checking their satisfiability, which can be done during Boolean Unit Propagation [DG84], a significant simplification. Hence, imposing such a structure in a boolean formula simplifies the solving process. Moreover, we show in Chapter 4 how

the structure of Horn formulas can be used to allow for further simplification during propagations.

Another key observation is that encoding a probabilistic model into a CNF formula requires adding variables unrelated to the model's parameters. Such variables must have a weight when solving a weighted model counting problem and are assigned a weight of 1 so as not to impact the weighted model count. Moreover, reducing some inference tasks to *unweighted* model counting problem results in boolean formulas that are exponentially larger than the initial probabilistic model (e.g., reliability estimation problems in ProbLog [Vla+16]). To simplify the encoding and ensure polynomial-size boolean formulas, we propose solving all the abovementioned problems using *projected* weighted model counting. Although it is already the state-of-the-art encoding for reliability problems, it is new for Bayesian networks and ProbLog.

All these specificities of the Schlandals language do not require developing a new solver; they could be integrated into modern model counters. However, another goal of this work is to explore how to design an efficient, simple model counter. Modern model counters incorporate years of improvements, often extending previous counters (e.g., both `Ganak` and `sharpSAT-TD` extends `sharpSAT`). Extending such solvers with new ideas can be challenging at times. For example, working on distributions rather than boolean variables requires rethinking the branching heuristics and solver data structures. For this reason, we decided to start from scratch, implementing the most basic functionalities required for search-based model counters.

3.2 A Modelling Language Designed for Probabilistic Inference

Schlandals is a language based on propositional logic and relies on solving a projected weighted model counting problem. Let us remember that F is a boolean formula over variables $\boldsymbol{B}$ partitioned between projected variables $\boldsymbol{P}$ and non-projected variables $\boldsymbol{X}$. Since this work focuses on probabilistic problems, the weights on the projected variables are used to model probabilities. Hence, we also refer to those as *probabilistic variables* while the non-projected variables are called *deterministic variables*. Two novelties are introduced in Schlandals compared to classical propositional logic. First, a constraint on the clauses in F is imposed: only *Horn* clauses are allowed.

Definition 5 (Horn clause)**.** *A Horn clause C is a formula of the form*

$$B_1 \wedge \cdots \wedge B_n \Rightarrow B_t$$

where $\mathbf{I} = B_1 \wedge \cdots \wedge B_n$ is called the implicant *of the clause and $H = B_t$ is* the head *of the clause. Here $B_i \in \boldsymbol{B}$ is a boolean variable, and B_t is either a boolean*

variable in $\boldsymbol{B}$ or $\bot$. If $n = 0$ (the implicant is empty), then the left-hand side reduces to $\top$.

It can be observed that when a Horn clause is written as an implication, as above, all the literals in the implication have the same (positive) polarity. Hence, we only discuss variables, not literals, to simplify our discussion and notation. A Horn clause C_i can be identified uniquely by its implicant $\mathbf{I}_i$ and head H_i. Hence, when clear of the context, we use the notation C_i and $(\mathbf{I}_i, H_i)$ interchangeably. For simplicity of notation, we also denote by $B \in \mathbf{I}_i$ that $B \in \boldsymbol{B}$ is a variable of the implicant of C_i. Horn clauses have been well-studied in the literature. A significant result is that the SAT problem over a CNF formula of Horn clauses can be solved in linear time [DG84]; given that the SAT problem in its general form is NP-hard, this is a significant simplification.

Secondly, we add the notion of *distributions over the probabilistic variables*. We assume that each probabilistic variable $P \in \boldsymbol{P}$ belongs to exactly one part of a partition $\boldsymbol{P_i} \subseteq \boldsymbol{P}$ of the probabilistic variables. We define a *distribution* over each such a part in the following simple manner: we require that one weight $\omega(P) > 0$ is specified for each probabilistic variable $P \in \boldsymbol{P_i}$ and that $\sum_{P \in \boldsymbol{P_i}} \omega(P) = 1$. Hence, a Schlandals formula can be defined as follows.

Definition 6 (Schlandals formula)**.** *Let $\boldsymbol{B} = \boldsymbol{P} \cup \boldsymbol{X}$ be a set of boolean variables. Let $\boldsymbol{P}$ be partitioned into n subsets $\boldsymbol{P}_1, \ldots, \boldsymbol{P}_n$ and $\omega : \boldsymbol{P} \mapsto [0, 1]$ be a weight function such that $\sum_{P \in \boldsymbol{P}_i} \omega(P) = 1$ for all $\boldsymbol{P}_i$. A Schlandals formula is a literal-weighted boolean formula F over the variables $\boldsymbol{B}$ with weight function ω such that F has the following form.*

$$F = (\mathbf{I}_1 \Rightarrow H_1) \wedge \ldots \wedge (\mathbf{I}_k \Rightarrow H_k)$$

Let $I : \boldsymbol{P} \mapsto \{\top, \bot\}$ be an interpretation on a Schlandals formula F. The weight of that interpretation is based on the weight of each part $\boldsymbol{P_i}$ defined as follows.

$$\omega_I(\boldsymbol{P_i}) = \begin{cases} \omega(P) & \text{if there is exactly one } P \in \boldsymbol{P_i} \text{ for which } I(P) = \top \\ 0 & \text{otherwise,} \end{cases} \tag{3.1}$$

In other words, if precisely one variable in the partition is set to $\top$, the weight of that variable is given to that partition; otherwise, the assignment has a zero weight. Since interpretations with a weight of 0 do not affect the count of a formula, we only consider interpretation respecting the first condition of Equation (3.1). Hence, an interpretation in Schlandals can be seen as assigning a value to each distribution. We can now define the problem of model counting for a Schlandals formula F.

Problem 5 (weighted #∃SSAT)**.** *Let F be a Schlandals formula, in CNF, over variables $\boldsymbol{B} = \boldsymbol{P} \cup \boldsymbol{X}$ and $\omega : \boldsymbol{P} \mapsto [0, 1]$ a weight function. The weighted #∃SSAT problem is to compute the weighted sums of the projected models of F as follows.*

$$\sum_{I \in \mathcal{S}_{\boldsymbol{P}}(F)} \prod_{i=1}^{n} \omega_I(\boldsymbol{P}_i)$$

Problem 5 and Problem 4 are, in essence, very similar. The only differences in the problem formulation are that, in Problem 5, F is a *Horn formula* and the weight function is applied on the partitions $\boldsymbol{P}_i$, the distributions, and not the literals. When clear from the context, we denote the problem of counting the assignments of Schlandals formula the same way as classical boolean formulas, as $\text{pwmc}(F, \mathbf{P})$.

3.3 Encoding Probabilistic Inference Problems in Schlandals

This section shows how the inference problems defined in Chapter 2 can be modelled in Schlandals. We omit the problem of reliability estimation in probabilistic graphs, as the encoding in Schlandals follows the one presented above, with the only difference being that each edge is encoded using a distribution instead of a single boolean variable.

3.3.1 Bayesian Networks

Let $(\boldsymbol{V}, \boldsymbol{\Phi})$ be a Bayesian networks as defined in Chapter 2. The variables used for encoding the network are the same as for ENC1, defined as follows.

$$\lambda_v \in \{\top, \bot\} \qquad \forall v \in \text{dom}(V),\ \forall V \in \boldsymbol{V} \tag{3.2}$$

$$\theta_{\boldsymbol{z}} \in \{\top, \bot\} \qquad \forall \boldsymbol{z} \in \times_{Z \in \text{scope}(\Phi_V)} \text{dom}(Z),\ \forall V \in \boldsymbol{V} \tag{3.3}$$

We define $\boldsymbol{P}$ as the probabilistic variables determined by Equation (3.3) and partition them the following way. Intuitively, a distribution must be defined for each row of a node's CPT. A distribution is defined over its domain for each possible assignment to its parents. The following equation defines the partitions in this manner.

$$\boldsymbol{P}_V^{\boldsymbol{y}} = \bigcup_{v \in \text{dom}(V)} \{\theta_{v\boldsymbol{y}}\} \qquad \forall \boldsymbol{y} \in \bigtimes_{\substack{Y \in \text{scope}(\Phi_V) \\ Y \neq V}} \text{dom}(Y), \forall V \in \boldsymbol{V} \tag{3.4}$$

Equation (3.4) effectively defines distributions according to the rows of the CPTs. For each node $V \in \boldsymbol{V}$, it considers every assignment on the variables in $\text{scope}(V) \setminus V$ - the parents of V - and creates a distribution of size $|\text{dom}(V)|$.

Finally, the clauses encoding the network's structure and the evidence must be defined. The clauses for the structure are very similar to those in ENC1 and are presented next.

$$\left(\wedge_{y\in \boldsymbol{y}}\lambda_y\right)\wedge\theta_{v\boldsymbol{y}}\Rightarrow\lambda_v \quad \forall v\in \mathrm{dom}(V),\ \forall \boldsymbol{y}\in \bigtimes_{\substack{Y\in\mathrm{scope}(\Phi_V)\\ Y\neq V}} \mathrm{dom}(Y)$$
$$\forall V\in \boldsymbol{V} \tag{3.5}$$

The evidence is encoded using the following clauses.

$$\neg\lambda_y \quad \forall y\in\mathrm{dom}(X) \text{ such that } y\neq x, \forall X\in \boldsymbol{E} \text{ with value } x \tag{3.6}$$

Example: Bayesian Network in Schlandals

For example, let us illustrate how to encode our small Bayesian network example in Schlandals. For conciseness, let us encode only the two following CPTs.

	asia	
	$\top$	$\bot$
Θ_1	0.1	0.99

		bronc	
	smoke	$\top$	$\bot$
Θ_2	$\top$	0.6	0.4
Θ_3	$\bot$	0.3	0.7

As in ENC1, the following indicator and parameter variables are needed.

$$\boldsymbol{\Lambda} = \lambda_{A_\top}, \lambda_{A_\bot}, \lambda_{B_\top}, \lambda_{B_\bot}, \lambda_{S_\top}, \lambda_{S_\bot}$$
$$\boldsymbol{\Theta} = \theta_{A_\top}, \theta_{A_\bot}, \theta_{B_\top S_\top}, \theta_{B_\bot S_\top}, \theta_{B_\top S_\bot}, \theta_{B_\bot S_\bot}$$

However, unlike ENC1, the $\boldsymbol{\Theta}$ variables must be partitioned into distributions. The following distributions are created and shown in the corresponding rows of the CPTs.

$$\boldsymbol{\Theta}_1 = \{\theta_{A_\top}, \theta_{A_\bot}\},\ \boldsymbol{\Theta}_2 = \{\theta_{B_\top S_\top}, \theta_{B_\bot S_\top}\},\ \boldsymbol{\Theta}_3 = \{\theta_{B_\top S_\bot}, \theta_{B_\bot S_\bot}\}$$

The following clauses are used to encode the structure of the Bayesian network.

$$\theta_{A_\top}\Rightarrow\lambda_{A_\top} \qquad \lambda_{B_\top}\wedge\theta_{B_\top S_\top}\Rightarrow\lambda_{S_\top} \qquad \lambda_{B_\bot}\wedge\theta_{B_\bot S_\top}\Rightarrow\lambda_{S_\top}$$
$$\theta_{A_\bot}\Rightarrow\lambda_{A_\bot} \qquad \lambda_{B_\top}\wedge\theta_{B_\top S_\bot}\Rightarrow\lambda_{S_\bot} \qquad \lambda_{B_\bot}\wedge\theta_{B_\bot S_\bot}\Rightarrow\lambda_{S_\bot}$$

Finally, a query such as $P(\text{asia}=\top)$ can be encoded using the following unit clause.

$$\neg\lambda_{A_\bot}$$

There is a significant difference between the encoding presented above and the ones described in Chapter 2: the parameter variables are in the *implicant* of the clauses. Without delving into the technical details developed in Chapter 4, this allows for removing clauses and variables from the encoding during pre-processing, thanks to the non-projected variables. However, this choice implies that the evidence must be encoded differently. Similarly to reliability estimation problems, encoding the query using positive literals (i.e., λ_y) does not constrain the problem. Hence, every interpretation is a model of the formula, and its weighted model count is always 1.

Another key difference is that the distributions are defined over the parameter variables. In ENC1, the *at most one* and *at least one* constraints are defined over the indicator variables. At first glance, this seems unintuitive: a valid assignment in Schlandals requires that each distribution has exactly one variable set to $\top$. Hence, in such an interpretation, multiple variables linked to the same CPT are set to $\top$. While this seems problematic, we demonstrate next that, in practice, only one parameter variable per CPT is constrained.

Example: Bayesian Network Interpretations

Let us assume a small Bayesian network with two binary nodes, A and B, such that A is the parent of B. The two following clauses encoded A's CPT:

$$C_1 = \theta_{A_\top} \Rightarrow \lambda_{A_\top} \qquad C_2 = \theta_{A_\bot} \Rightarrow \lambda_{A_\bot}.$$

The four following clauses are needed for B' CPT:

$$C_3 = \lambda_{A_\top} \wedge \theta_{B_\top A_\top} \Rightarrow \lambda_{B_\top} \qquad C_4 = \lambda_{A_\top} \wedge \theta_{B_\bot A_\top} \Rightarrow \lambda_{B_\bot}$$
$$C_5 = \lambda_{A_\bot} \wedge \theta_{B_\top A_\bot} \Rightarrow \lambda_{B_\top} \qquad C_6 = \lambda_{A_\bot} \wedge \theta_{B_\bot A_\bot} \Rightarrow \lambda_{B_\bot}$$

Let F be the boolean formula obtained by using these clauses; models of F must have one variable from $\{\theta_{B_\top A_\top}, \theta_{B_\bot A_\top}\}$ and one variable from $\{\theta_{B_\top A_\bot}, \theta_{B_\bot A_\bot}\}$ set to $\top$. Intuitively, it means that two entries in B's CPT are activated, which is not possible for Bayesian networks. However, depending on the choice for the distribution $\{\Theta_{A_\top}, \Theta_{A_\bot}\}$, one of the two remaining distributions does not constrain the formula anymore.

Let us assume $\theta_{A_\top} = \top$, then, C_1 forces $\lambda_{A_\top} = \top$ and C_2 is always respected, since $\theta_{A_\bot} = \bot$. Hence, C_3 and C_4 are reduced, respectively, to $\theta_{B_\top A_\top} \Rightarrow \lambda_{B_\top}$ and $\theta_{B_\bot A_\top} \Rightarrow \lambda_{B_\bot}$. On the other hand, C_5 and C_6 are unchanged.

The crucial observation is that $\lambda_{A_\bot}$ is a non-projected variable. Hence, for any assignment to the distributions, it is sufficient to find an assignment on $\lambda_{A_\bot}$ that is part of a model of F. In particular, setting $\lambda_{A_\bot} = \bot$ is valid for any assignments of the remaining probabilistic variables (i.e.,

it does not remove interpretations on them). After such assignments, it can be seen that C_5 and C_6 are always respected; hence, the distribution $\{\theta_{B_\top A_\bot}, \theta_{B_\bot A_\bot}\}$ does not constrain the problem anymore. It follows that any valid interpretation $I : \boldsymbol{P} \mapsto \{\top, \bot\}$ can be mapped to a single assignment of the initial Bayesian network's nodes. Moreover, the weighted sum of each valid interpretation corresponding to the same assignment of the BN nodes corresponds to the assignment probability.

Theorem 3. *Let $(\boldsymbol{V}, \boldsymbol{\Phi})$ be a Bayesian network and $\boldsymbol{E} \subseteq \boldsymbol{V}$ a set of variables whose observed values are $\boldsymbol{e}$. Let $\boldsymbol{X}$ and $\boldsymbol{P}$ be the set of boolean variables respectively defined by Equation* (3.2) *and Equation* (3.3)*. Moreover, let $\boldsymbol{P_1}, \dots, \boldsymbol{P_n}$ be a partitioning of $\boldsymbol{P}$ as defined by Equation* (3.4)*. If F is a Schlandals formula over variables $\boldsymbol{B} = \boldsymbol{P} \cup \boldsymbol{X}$, with projected variables $\boldsymbol{P}$, and clauses as defined by Equation* (3.5)*, then the following equality holds.*

$$P(\boldsymbol{E} = \boldsymbol{e}) = \text{pwmc}(F, \boldsymbol{P})$$

Proof. We prove this theorem by first proving that any assignment $\boldsymbol{v}$ to the BN's variable correspond to a subset $\boldsymbol{I_v}$ of the interpretations of F such that

$$P(\boldsymbol{V} = \boldsymbol{v}) = \sum_{I \in \boldsymbol{I_v}} \omega(I).$$

Let $\pi_V : \bigtimes_{Y \in \boldsymbol{V}} \text{dom}(Y) \mapsto \bigtimes_{Z \in \text{scope}(\Phi_V)} \text{dom}(Z)$ be a projection operator from the whole set of variables to the subset of variables in $\text{scope}(\Phi_V)$. A complete assignment $\boldsymbol{v}$ imposes that some parameter variables must be set to $\top$: for each CPT $\Phi_V \in \boldsymbol{\Phi}$, the variable $\theta_{\boldsymbol{z}}$ with $\boldsymbol{z} \in \bigtimes_{Z \in \text{scope}(\Phi_V)} \text{dom}(Z)$ such that $\pi_V(\boldsymbol{v}) = \boldsymbol{z}$ must be true. Let $\boldsymbol{\Theta_v}$ be the set of parameter variables that must be true given the complete assignment $\boldsymbol{v}$. We say that an interpretation I is consistent with $\boldsymbol{v}$ if and only if $I(\theta) = \top \; \forall \theta \in \boldsymbol{\Theta_v}$. Note that other variables in $\boldsymbol{P}$ might be set to $\top$ by I (i.e., probabilistic variables appearing in other rows of the CPTs). We define $\boldsymbol{I_v}$ as the set of interpretations of F that are consistent with $\boldsymbol{v}$.

Let us now compute $\sum_{I \in \boldsymbol{I_v}} \omega(I)$. By definition, $\omega(I) = \prod_{P \in \boldsymbol{P} | I(P) = \top} \omega(P)$. Moreover, by assumptions each interpretation $I \in \boldsymbol{I_v}$ sets the variables in $\boldsymbol{\Theta_v}$ to $\top$; hence, we have

$$\omega(I) = \left(\prod_{\theta \in \boldsymbol{\Theta_v}} P(\theta) \right) \left(\prod_{P \in \boldsymbol{P} | P \notin \boldsymbol{\Theta_v} \wedge I(P) = \top} \omega(P) \right).$$

It follows that:

$$\sum_{I \in \boldsymbol{I_v}} \omega(I) = \sum_{I \in \boldsymbol{I_v}} \left(\prod_{\theta \in \boldsymbol{\Theta_v}} P(\theta) \right) \left(\prod_{P \in \boldsymbol{P} | P \notin \boldsymbol{\Theta_v} \wedge I(P) = \top} \omega(P) \right)$$
$$= \left(\prod_{\theta \in \boldsymbol{\Theta_v}} P(\theta) \right) \sum_{I \in \boldsymbol{I_v}} \prod_{P \in \boldsymbol{P} | P \notin \boldsymbol{\Theta_v} \wedge I(P) = \top} \omega(P)$$

By construction of F, we have that each $\theta \in \boldsymbol{\Theta_v}$ has its weight derived from the CPT; hence, $P(\boldsymbol{V} = \boldsymbol{v}) = \prod_{\theta \in \boldsymbol{\Theta_v}} P(\theta)$. We now prove that

$$\sum_{I \in \boldsymbol{I_v}} \prod_{P \in \boldsymbol{P} | P \notin \boldsymbol{\Theta_v} \wedge I(P) = \top} \omega(P) = 1.$$

The reasoning is the following: the term on the left-hand side of the equation can be seen as computing the weighted sum of interpretations on all distributions except the ones in $\boldsymbol{\Theta_v}$. Since there are no constraints on these distributions (i.e., no specific variables in them must be true), then it is the weighted sum of all possible interpretations on the distributions in $\boldsymbol{P} \setminus \boldsymbol{\Theta_v}$ which, by definition, is 1.

We proved that every assignment $\boldsymbol{v}$ to the BN's nodes corresponds to a specific subset of the interpretations whose weighted sum corresponds to the assignment's probability. Moreover, for two different assignments $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, the sets $\boldsymbol{I}_{\boldsymbol{v}_1}$ and $\boldsymbol{I}_{\boldsymbol{v}_2}$ are disjoint and the union of all possible sets $\boldsymbol{I_v}$ forms the set of all interpretations of F. Finally, by selecting all assignments $\boldsymbol{v}$ respecting F constraints on the evidence (i.e., each node $E \in \boldsymbol{E}$ is forbidden to take any other value than $e \in \boldsymbol{e}$), we have the result that $P(\boldsymbol{E} = \boldsymbol{e}) = \text{pwmc}(F, \boldsymbol{P})$. □

3.3.2 ProbLog

Let $\boldsymbol{R}$ be a grounded ProbLog program over atoms $\boldsymbol{A}$, $\boldsymbol{R}^P = \cup_{i=1}^n \boldsymbol{R}_i^P$ be the n-partitioned set of probabilistic rules of $\boldsymbol{R}$, and Q a query on $\boldsymbol{R}$. Such a program already follows the structure of a Schlandals formula: it contains n distributions, and the set $\boldsymbol{R}$ defines a Horn formula. A Schlandals formula is constructed from $\boldsymbol{R}$ as follows. One boolean variable is created per atom, and one boolean variable is created per value on the left-hand side of the annotated disjunctions.

$$B_A \in \{\top, \bot\} \quad \forall A \in \boldsymbol{A} \tag{3.7}$$
$$\theta_i^j \in \{\top, \bot\} \quad \forall R_i^j \in \boldsymbol{R}_i^P, \quad \forall \boldsymbol{R}_i^P \in \boldsymbol{R}^P \tag{3.8}$$

Then, each rule $R \in \boldsymbol{R}$ is translated as a Horn clause and the query $Q \in \boldsymbol{A}$ is negated.

$$B_{A_1} \wedge \ldots \wedge B_{A_k} \implies B_{A_h} \quad \forall\, A_h \text{ :- } A_1, \ldots, A_k \in \boldsymbol{R} \setminus \boldsymbol{R}^P \tag{3.9}$$

$$B_{A_1} \wedge \ldots \wedge B_{A_k} \wedge \theta_i^j \implies B_{A_h} \quad \forall\, R_i^j = A_h \text{ :- } A_1, \ldots, A_k \in \boldsymbol{R}_i^P \quad \forall \boldsymbol{R}_i^P \in \boldsymbol{R}^P \tag{3.10}$$

$$\neg B_Q \tag{3.11}$$

We define the projected variables $\boldsymbol{P}$ as the variables defined by Equation (3.8), and their weight is given by the initial ProbLog program.

$$\omega(\theta_i^j) = P(R_i^j) \quad \forall R_i^j \in \boldsymbol{R}_i^P, \quad \forall \boldsymbol{R}_i^P \in \boldsymbol{R}^P \tag{3.12}$$

Theorem 4. *Let $\boldsymbol{R}$ be a grounded ProbLog program over the atoms $\boldsymbol{A}$, $\boldsymbol{R}^P = \cup_{i=1}^n \boldsymbol{R}_i^P$ the set of probabilistic rules, and $Q \in \boldsymbol{A}$ a query. Let F be a Schlandals formula over variables $\boldsymbol{B}$ defined by Equations* (3.7)-(3.8)*, projected variables $\boldsymbol{P} \subseteq \boldsymbol{B}$ defined by Equation* (3.8)*, and clauses defined by Equations* (3.9)-(3.11)*. Let the projected variables be weighted following Equation* (3.12)*. Then, the following equality holds.*

$$P(Q \mid \boldsymbol{R}) = 1 - \text{pwmc}(F, \boldsymbol{P})$$

Proof. First, let's examine the formula encoded in the grounded program. As explained in Chapter 2, it is obtained using Prolog's proof system, which finds all proofs in which $Q \Rightarrow \bot$ creates a contradiction; hence, Q is $\top$. Let $\boldsymbol{R}^D = \boldsymbol{R} \setminus \boldsymbol{R}^P$, the probability of the query Q is defined, for ProbLog, as follows:

$$\sum_{\boldsymbol{X} \subseteq \boldsymbol{R}^P \mid \boldsymbol{X} \cup \boldsymbol{R}^D \models Q} P(\boldsymbol{X}).$$

In other words, it is the weighted sum of each logic program induced by the ProbLog program distribution such that they are satisfied given the query Q. Let us now link the ProbLog program distribution and the Schlandals formula F.

By construction, the clauses in F are in one-to-one correspondence with the grounded rules $\boldsymbol{R}$, and a distribution is created for each annotated disjunction. A non-zero weighted interpretation of F correspond to a choice for the distributions and, hence, a logic program in the ProbLog program distribution. The weights of the boolean variables are such that the following equality holds for an interpretation I and the corresponding logic program $\boldsymbol{I}^{\boldsymbol{R}}$.

$$\prod_{i=1}^{n} \omega_{\boldsymbol{P}_i}(I^F) = \prod_{R \in \boldsymbol{I}^{\boldsymbol{R}}} P(R)$$

We now show that the weighted model count of F corresponds exactly to counting the interpretations corresponding to inconsistent logic programs (i.e., programs in which $Q = \bot$). The model count of F is computed as $\text{pwmc}(F, \boldsymbol{P}) = \sum_{I \in \mathcal{S}_{\boldsymbol{P}}(F)} \omega(I)$, with $\mathcal{S}_{\boldsymbol{P}}(F)$ defined as follows:

$$\mathcal{S}_{\boldsymbol{P}}(F) = \{I : \boldsymbol{P} \mapsto \{\top, \bot\} \mid \exists\, I' : \boldsymbol{B} \setminus \boldsymbol{P} \mapsto \{\top, \bot\} \text{ such that } F[I \cup I'] = \top\}.$$

In other words, the interpretations considered in the weighted sums are the ones in which the choice for the distributions results in a satisfiable residual formula. Given the link between the interpretations and the logic programs, it means that the weighted sums are over logic programs *consistent with the formula* F. In our encoding, contrary to the initial logic programs, the query is negated. It follows that $\text{pwmc}(F, \boldsymbol{P})$ is a weighted sum over interpretations corresponding to logic programs in which $Q = \bot$. Hence, we have that $P(Q \mid \boldsymbol{R}) = 1 - \text{pwmc}(F, \boldsymbol{P})$. □

One crucial observation is that encoding a ProbLog program into Schlandals only requires a *grounded program*. That is, there is no need to apply Clark's completion algorithm and, hence, no need for cycle breaking. This is a significant improvement, as reliability problems can now be solved using formulas that are not exponentially larger than the initial ProbLog program.

3.4 Conclusion

This chapter introduced Schlandals, a modelling language for probabilistic inference problems based on propositional formulas. Contrary to propositional formulas, the distributions in Schlandals are first-class citizens; they do not need to be encoded into clauses. Moreover, each Schlandals formula is a Horn formula, as the clauses necessary to encode the considered probabilistic inference problems are Horn. Finally, this language is designed so that its weighted model count is computed using *projected* weighted model counters; hence, only the boolean variables in the distributions are weighted.

Although state-of-the-art reliability estimation encoding already follows the same structure as a Schlandals formula, we showed how to encode Bayesian networks and ProbLog programs into Schlandals. We provide proof that our encodings are correct for these two inference tasks. In particular, our proof for ProbLog programs only required a *grounded* program, removing the need to apply cycle breaking and ensuring polynomial-size encoding for reliability estimation problems.

Exact Inference in Schlandals

4

This chapter details the internal mechanics of the Schlandals solver, a model counter specialised for calculating the weighted model count of a Schlandals formula. More precisely, we show how to develop DPLL-based projected weighted model counters to account for distributions and Horn clauses. Then, we introduce a new propagation algorithm that uses the non-projected variables and Horn structure. Finally, we briefly demonstrate how Schlandals can be adapted for Knowledge Compilation before concluding with an experimental evaluation of the solver.

This chapter is based on the following article:

- A. Dubray, P. Schaus, and S. Nijssen. "Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses". In: *LIPIcs, Volume 280, CP 2023* 280 (2023). Ed. by R. H. C. Yap. ISSN: 1868-8969. DOI: 10.4230/LIPICS.CP.2023.15. (Visited on 03/07/2025)

The exploration of knowledge compilation in Schlandals was not present in the original paper and is an addition to this manuscript. Additional experiments have also been conducted.

4.1 Exhaustive DPLL-style Search

The main conceptual difference between classical model counting and Schlandals is the computation of an interpretation's weight. In the former, an interpretation's weight consists of the products of its literals; in the latter, it relies on the partition of the projected variables. Moreover, using Horn-formulas simplifies the calculation when all projected variables have been assigned. Algorithm 3 gives an overview of the main procedure of Schlandals and highlights these differences. It follows the same structure as a classical DPLL-based model counter: a cache is used to store intermediate results (line 19) and reuse them (line 2), it branches on some variables (lines 4-5), applies a propagation algorithm (line 6), and explore the independent components of the residual formula (line 15).

The first main difference concerns the branching; Algorithm 3 branches on *distributions* and not variables. Since an interpretation that does not set

Algorithm 3: DPLL-based algorithm for solving Problem 5

```
1  Function Schlandals-PWMC(F, P, ω, C)
       input  : F a boolean Horn-formula, over variables B
       input  : P = ∪_{i=1}^{n} P_i ⊆ B a set of projected variables, partitioned
                into n distributions
       input  : ω a literal-weight function
       input  : C a cache of sub-results
       output: weighted #∃SSAT (F, P)
2      if F ∈ C then return C[F]
3      if P = ∅ then return 1
4      i ← a distribution index such that ∃P ∈ P_i | P is not fixed
5      foreach P ∈ P_i do
6          F' ← Propagate(F, P, ⊤)
7          if F' = ⊥ then count_P ← 0
8          else
9              Components ← all connected components of F'
10             fixed ← {P' | P' ∈ P ∧ P' = ⊤}
11             count_P ← ∏_{P'∈fixed} ω(P')
12             count_P ← count_B × ∏_{j|P_j is unconstrained in F'} ∑_{P'∈P_j} ω(P')
13             foreach Comp ∈ Components do
14                 P' ← P reduced to the variables in Comp
15                 count_P ← count_B* DPLL-PWMC(Comp, P', ω, C)
16             end
17         end
18     end
19     C[F] ← ∑_{P∈P_i} count_P
20     return C[F]
```

exactly one variable per partition $\boldsymbol{P}_i$ to $\top$ weights 0, the solver ensures (during the propagation) that each partition's variables are mutually exclusive. Hence, a partition $\boldsymbol{P}_i$ can be seen as a variable whose domain consists of boolean variables, and the solver heuristically selects a distribution to be fixed.

This weighting scheme also impacts how a branch's count is computed (lines 10-12). It is composed of three elements: the weights of the probabilistic variables set to $\top$ during the propagation (lines 10-11), the count of the independent components (line 15), and the count of *unconstrained distributions* (line 12). This last element is not present for classical model counters; we will detail its computation next.

Definition 7 (Unconstrained Distribution)**.** *Let $F = C_1 \wedge \ldots \wedge C_k$ be a Schlandals*

formula over variables $\boldsymbol{B}$ *with* $\boldsymbol{P} = \cup_{i=1}^{n} \boldsymbol{P}_i$ *probabilistic variables. When the following conditions are met, a distribution* $\boldsymbol{P}_i$ *is called* unconstrained.

1. *No clause contains a variable of* $\boldsymbol{P}_i$*:* $\forall 1 \leq j \leq k,\ C_j \cap \boldsymbol{P}_i = \emptyset$
2. *No variable in* $\boldsymbol{P}_i$ *is set to* $\top$.

Detecting such distributions is crucial because branching on them would not yield any propagation. In particular, any partial model on F can be extended by any assignment on unconstrained distributions and be a model of F. The following theorem demonstrates that the contribution of such distributions to the weighted model count can be formulated in a closed form.

Theorem 5. *Let* F *be a Schlandals formula over distributions* $\boldsymbol{P} = \cup_{i=1}^{n} \boldsymbol{P}_i$ *and weight function* $\omega : \boldsymbol{P} \mapsto [0, 1]$*. Without loss of generality, let us assume that the* m *first distributions of* F *are unconstrained, and let us denote* $\boldsymbol{P'} = \cup_{i=m+1}^{n} \boldsymbol{P}_i$ *the remaining distributions. Then, the following equality holds.*

$$\text{pwmc}(F, \boldsymbol{P}) = \left(\prod_{i=1}^{m} \sum_{V \in \boldsymbol{P}_i} \omega(V) \right) \times \text{pwmc}(F, \boldsymbol{P'})$$

Proof. By definition of the weighted model count, we have the following equality:

$$\text{pwmc}(F, \boldsymbol{P}) = \sum_{P \in \boldsymbol{P}_1} \omega(P) \times \text{pwmc}(F[P = \top], \boldsymbol{P} \setminus \boldsymbol{P}_1).$$

By assumption, $\boldsymbol{P}_1$ does not appear in F; hence, for each $P \in \boldsymbol{P}_1$, we have $F[P = \top] = F$. It follows that:

$$\begin{aligned} \text{pwmc}(F, \boldsymbol{P}) &= \text{pwmc}(F, \boldsymbol{P} \setminus \boldsymbol{P}_1) \times \sum_{P \in \boldsymbol{P}_1} \omega(P) \\ &= \left(\sum_{P \in \boldsymbol{P}_2} \omega(P) \times \text{pwmc}(F, \boldsymbol{P} \setminus \boldsymbol{P}_1 \cup \boldsymbol{P}_2) \right) \times \sum_{P \in \boldsymbol{P}_1} \omega(P). \end{aligned}$$

We obtain the following result using the same reasoning as for the m unconstrained distribution.

$$\text{pwmc}(F, \boldsymbol{P}) = \text{pwmc}(F, \boldsymbol{P} \setminus \cup_{i=1}^{m} \boldsymbol{P}_i) \times \prod_{i=1}^{m} \sum_{P \in \boldsymbol{P}_i} \omega(P)$$

□

This result is used to compute the contribution of each unconstrained distribution after propagation (line 12). This term is unnecessary when performing classical weighted model counting, as we assume a setting in which the sum of the weights of a variable's literals is 1.

In addition to these differences, the case when $\boldsymbol{P} = \emptyset$ (line 3) is simplified in Schlandals: F is a Horn formula, and it is known that if F is unsatisfiable, it is detected by BUP. Since Schlandals' propagation algorithm performs BUP, it is known that F has a model and 1 is directly returned without calling an SAT solver. Finally, the computation of independent components (line 9) has also been modified to account for the distributions: two sub-formulas are considered independent if they do not share any variables nor variables that belong to the same distribution.

4.1.1 Propagation

In this section, we describe the propagation algorithm used by Schlandals (line 6 in Algorithm 3). Briefly, it operates in two steps. First, boolean unit propagation and the distribution constraints are applied. Then, an additional propagation removes clauses that do not affect the count by leveraging the non-projected variables.

Algorithm 4 shows how Schlandals performs *Boolean Unit and Distribution Propagation* (BUDP). This algorithm is similar to classical BUP and follows the same structure except for the distribution constraints (lines 18-24). When a probabilistic variable is set to true, all variables in the same distribution must be false (lines 19-20). If it is set to false, and only one other variable remains in the distribution, that variable must be true (lines 21-23).

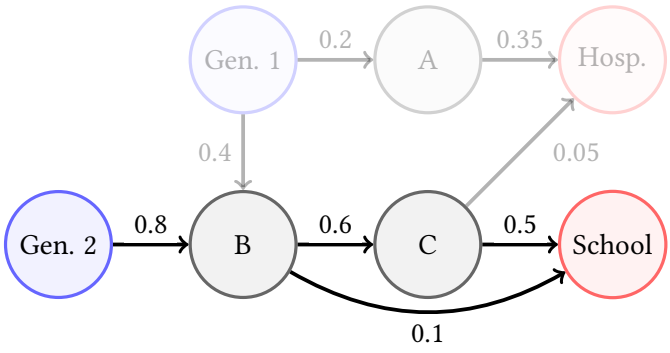
Algorithm 4 is necessary as the distributions are not transformed into clauses. However, this algorithm has the same propagation strength as classical BUP with distributions encoded directly in F. On the other hand, the additional propagation presented below is an addition to BUP and relies on the projected variables. The intuition is that the deterministic variables can be used to remove additional clauses from F. Let us show how this can be done using the following example.

Example: Removing Clauses with Non-projected Variables

Let us consider the graph for the electrical network of our small city, shown below, and assume that the query is to compute the probability that the second generator is connected to the school. To compute the probability of this query, only the sub-graph containing all the paths between the source and the target is necessary. Hence, the faded part of the graph could be removed without impacting the solution. The goal of Schlandals' additional propagation is to remove unnecessary clauses.

Algorithm 4: Boolean Unit and Distribution Propagation

Function BUDP(F, B, v)
  input : F a Schlandals formula over variables $\mathbf{B}$ and distributions $\mathbf{P}$
  input : $B \in \mathbf{B}$ a boolean variable to set to value $b \in \{\top, \bot\}$
  output: F' the residual formula
  $F' \leftarrow F$
  $Q \leftarrow$ `Queue()`; Q.push((B, b))
  while $|Q| > 0$ **do**
    $(V, v) \leftarrow$ Q.pop()
    if V *has been previously assigned to* v **then return** $\bot$
    assign v to V
    if $v = \top$ **then** $F' \leftarrow F' \setminus \{C \mid C \in F' \wedge V \in C\}$
    if $v = \bot$ **then** $F' \leftarrow F' \setminus \{C \mid C \in F' \wedge \neg V \in C\}$
    foreach $C \in F'$ **do**
      if $v = \top$ **and** $\neg V \in C$ **then** $C = C \setminus \{\neg V\}$
      if $v = \bot$ **and** $V \in C$ **then** $C = C \setminus \{V\}$
      `/* C is a clause with only one literal */`
      if $|C| = 1$ **then**
        if $C = V'$ **then** Q.push($(V', \top)$)
        if $C = \neg V'$ **then** Q.push($(V', \bot)$)
      end
    end
    if $V \in \mathbf{P}_i$ *with* $\mathbf{P}_i \subseteq \mathbf{P}$ **then**
      if $v = \top$ **then**
        foreach $V' \in \mathbf{P}_i \mid V' \neq V$ **do** Q.push($(V', \bot)$)
      else if $\mathbf{P}_i = \{V, V'\}$ **then**
        Q.push($(V', \top)$)
      end
    end
  end
  return F'



To illustrate that propagation, let us show the clauses for the graph above. The faded clauses correspond to the unnecessary sub-graph; hence, these are the clauses that can be safely removed.

$$\begin{array}{ccc} \lambda_{G_1} \land \theta_{G_1A} \Rightarrow \lambda_A & \lambda_B \land \theta_{BS} \Rightarrow \lambda_S & \lambda_C \land \theta_{CH} \Rightarrow \lambda_H \\ \lambda_{G_1} \land \theta_{G_1B} \Rightarrow \lambda_B & \lambda_B \land \theta_{BC} \Rightarrow \lambda_C & \lambda_C \land \theta_{CS} \Rightarrow \lambda_S \\ \lambda_{G_2} \land \theta_{G_2B} \Rightarrow \lambda_B & \lambda_A \land \theta_{AH} \Rightarrow \lambda_H & \end{array}$$

The two unit clauses needed to compute $P(\text{path}(G_2, S))$ are λ_{G_2} and $\neg\lambda_S$. Applying the BUDP algorithm gives the following clauses.

$$\begin{array}{ccc} \lambda_{G_1} \land \theta_{G_1A} \Rightarrow \lambda_A & \lambda_B \land \theta_{BS} \Rightarrow \lambda_S & \lambda_C \land \theta_{CH} \Rightarrow \lambda_H \\ \lambda_{G_1} \land \theta_{G_1B} \Rightarrow \lambda_B & \lambda_B \land \theta_{BC} \Rightarrow \lambda_C & \lambda_C \land \theta_{CS} \Rightarrow \bot \\ \top \land \theta_{G_2B} \Rightarrow \lambda_B & \lambda_A \land \theta_{AH} \Rightarrow \lambda_H & \end{array}$$

Only the two unit clauses are removed from the formula, and no more propagation is done. In particular, all unnecessary clauses are still active after the initial propagation.

The question is: "Does an assignment on *a subset of the deterministic* variables exist such that it can extend any projected model of the formula?". In this case, it can be seen that the partial assignment $\lambda_{G_1} = \bot, \lambda_H = \top$ makes all unnecessary clauses evaluate to $\top$ without forcing any assignment on the probabilistic variables.

$$\begin{array}{ccc} \bot \land \theta_{G_1A} \Rightarrow \lambda_A & \lambda_B \land \theta_{BS} \Rightarrow \lambda_S & \lambda_C \land \theta_{CH} \Rightarrow \top \\ \bot \land \theta_{G_1B} \Rightarrow \lambda_B & \lambda_B \land \theta_{BC} \Rightarrow \lambda_C & \lambda_C \land \theta_{CS} \Rightarrow \bot \\ \top \land \theta_{G_2B} \Rightarrow \lambda_B & \lambda_A \land \theta_{AH} \Rightarrow \top & \end{array}$$

A significant consequence of this assignment on λ_{G_1} and λ_H is that all distributions appearing only in the removed clauses become *unconstrained*. Remember that unconstrained distributions do not need to be branched on; their contribution to the weighted model count can be computed using a closed-form formulation. Hence, the assignment above reduces the number of clauses in the formula and significantly reduces the depth of the search tree.

This example demonstrates that the non-projected variables can be leveraged to reduce the number of clauses in our formula F. Schlandals solves a *projected* WMC; hence, it looks for interpretations I on the projected variables

that can be extended into a model of F by another interpretation I' on the non-projected variables. Since the assignment in the example does not force any assignment on the projected variables, it can be used in any extension I'. We now explain how to find such assignments efficiently.

In Schlandals, for an assignment to not be a model of the input formula, it must generate a clause $\top \Rightarrow \bot$. Some clauses cannot contribute to such a contradiction by setting a deterministic variable to $\bot$ in its implicant (e.g., $\lambda_{G_1} = \bot$) or its head to $\top$ (e.g., $\lambda_H = \top$). We formalise this intuition next. First, let us define the following notion, which will help us define clauses that might generate a contradiction.

Definition 8 ($\{\top, \bot\}$-reachability)**.** *Let F be a Schlandals formula over probabilistic variables $\boldsymbol{P}$. A clause $C_i = (\mathbf{I}_i, H_i) \in F$ is $\bot$-reachable if one of the two following conditions is met:*

1. *C_i is of the form $\mathbf{I}_i \Rightarrow \bot$ or $\mathbf{I}_i \Rightarrow P$ with $P \in \boldsymbol{P}$*
2. *There exists a clause $C_j = (\mathbf{I}_j, H_j) \in F$ such that C_j is $\bot$-reachable and $H_i \in \mathbf{I}_j$.*

Similarly, C_i is $\top$-reachable if one of the two following conditions is met:

1. *There exist no deterministic variables in $\mathbf{I}_i$*
2. *There exists a clause $C_j = (\mathbf{I}_j, H_j) \in F$ such that C_j is $\top$-reachable and $H_j \in \mathbf{I}_i$.*

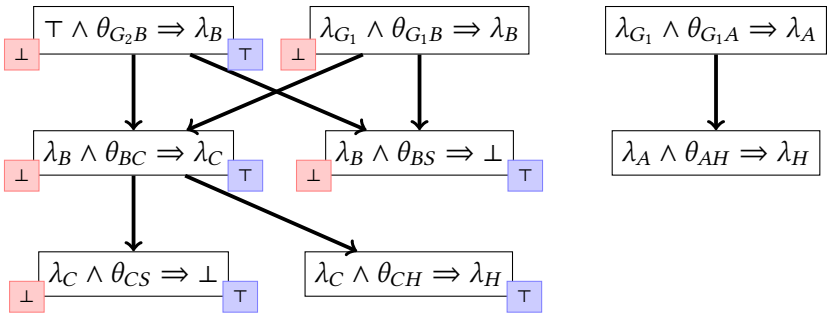
Intuitively, a clause $(\mathbf{I}_i, H_i)$ is $\bot$-reachable if its head might be $\bot$ either by choice (i.e., branching on a probabilistic variable) or constrained (i.e., a unit clause of the form $H_i \Rightarrow \bot$). The same reasoning applies for $\top$-reachable clauses; their implicants might be reduced to $\top$ during the search.

Example: $\{\top, \bot\}$-reachability

Let us show how this notion of $\{\top, \bot\}$-reachability applies to the clauses used in our example. Below are the clauses after propagating the two unit clauses λ_{G_2} and $\neg\lambda_S$ using the BUDP algorithm.

$$\begin{array}{lll} \lambda_{G_1} \wedge \theta_{G_1 A} \Rightarrow \lambda_A & \lambda_B \wedge \theta_{BS} \Rightarrow \lambda_S & \lambda_C \wedge \theta_{CH} \Rightarrow \lambda_H \\ \lambda_{G_1} \wedge \theta_{G_1 B} \Rightarrow \lambda_B & \lambda_B \wedge \theta_{BC} \Rightarrow \lambda_C & \lambda_C \wedge \theta_{CS} \Rightarrow \bot \\ \top \wedge \theta_{G_2 B} \Rightarrow \lambda_B & \lambda_A \wedge \theta_{AH} \Rightarrow \lambda_H & \end{array}$$

Since the notions of $\{\top, \bot\}$-reachability are based on links between the clauses, it is helpful to see the set of clauses as a graph. Below is a graphical representation of the clauses in which there is an edge from a clause $C_i = (\mathbf{I}_i, H_i)$ to a clause $C_j = (\mathbf{I}_j, H_j)$ if and only if $H_i \in \mathbf{I}_j$.



A clause is $\top$-reachable if it contains no deterministic variable (i.e., a λ variable, in our example) in its implicant or is the descendant of a $\top$-reachable variable. The clause $\top \wedge \theta_{G_2B} \Rightarrow \lambda_b$ respects the first condition; hence, all its descendants are $\top$-reachable. The same reasoning can be done for $\bot$-reachability. The clauses $\lambda_B \wedge \theta_{BS} \Rightarrow \bot$ and $\lambda_C \wedge \theta_{CS} \Rightarrow \bot$ have $\bot$ as their head; hence, they, and all their parents, are $\bot$-reachable.

Following our earlier discussion, the clauses that are necessary and sufficient for the computation of the weighted model count are the ones that might produce a contradiction of the form $\top \Rightarrow \bot$. In other words, these clauses are the ones that are both $\top$-reachable and $\bot$-reachable. The following theorem states that removing the unconstrained clauses from a Schlandals formula F does not modify F's projected models.

Theorem 6. *Let $F = C_1 \wedge \ldots \wedge C_n$ be a Schlandals formula, on projected variables $\boldsymbol{P}$, reduced by the BUDP algorithm. Assume, without loss of generality, that the $m \leq n$ first clauses are either not $\top$-reachable or not $\bot$-reachable and let $F' = C_{m+1} \wedge \ldots \wedge C_n$. The following equality holds.*

$$\text{pwmc}(F, \boldsymbol{P}) = \text{pwmc}(F', \boldsymbol{P})$$

Proof. Let $\boldsymbol{B}$ be the set of variables in F, $\boldsymbol{P}$ be the projected variables, and $\boldsymbol{X} = \boldsymbol{B} \setminus \boldsymbol{P}$ be the non-projected variables. We define $\boldsymbol{B'}$, $\boldsymbol{P'}$, and $\boldsymbol{X'}$ similarly for F'.

We will prove this theorem by construction. First, we define a specific set of deterministic variables and construct an interpretation I on that set. Then, we prove that for any interpretation $I' : \boldsymbol{B'} \mapsto \{\top, \bot\}$ such that $F'[I'] = \top$ (resp. $F'[I'] = \bot$), we have $F[I \cup I'] = \top$ (resp. $F[I \cup I'] = \bot$).

Let us define the non $\top$- and $\bot$-reachable clauses.

$$\boldsymbol{C}^{\neg\top} = \{C \in F \mid C \text{ is not } \top\text{-reachable } \}$$
$$\boldsymbol{C}^{\neg\bot} = \{C \in F \mid C \text{ is not } \bot\text{-reachable } \} \setminus \boldsymbol{C}^{\neg\top}$$

By definition, for every clause $C_i = (\mathbf{I}_i, H_i) \in \boldsymbol{C}^{\neg\top}$ there exists a variable $X_i \in \boldsymbol{X}$ such that $X_i \in \mathbf{I}_i$. Similarly, every clause $C_j = (\mathbf{I}_j, H_j) \in \boldsymbol{C}^{\neg\bot}$ is

such that $H_j \in \boldsymbol{X}$. Let $\boldsymbol{X}^{\neg\top} = \bigcup_{(\mathbf{I},H)\in\boldsymbol{C}^{\neg\top}}\{X \in \mathbf{I} \mid X \in \boldsymbol{X}\}$ as the set of deterministic variables appearing in the implicant of a non $\top$-reachable clause. Similarly, let $\boldsymbol{X}^{\neg\bot} = \{H \mid (\mathbf{I},H) \in \boldsymbol{C}^{\neg\bot}\}$ as the set of heads of non $\bot$-reachable clauses. Lastly, let $\boldsymbol{X}' = \boldsymbol{X}^{\neg\top} \cup \boldsymbol{X}^{\neg\bot}$.

We now prove that the interpretation $I : \boldsymbol{X}' \mapsto \{\top, \bot\}$, defined as follows, is a valid extension for every model of F'.

$$I(X) = \begin{cases} \bot & \text{if } X \in \boldsymbol{X}^{\neg\top} \\ \top & \text{if } X \in \boldsymbol{X}^{\neg\bot} \end{cases}$$

Let $I' : \boldsymbol{B} \mapsto \{\top, \bot\}$ any model of F' (i.e., $F'[I'] = \top$). The interpretation $I' \cup I$ must satisfy the clauses in $\boldsymbol{C}^{\neg\top}$ and $\boldsymbol{C}^{\neg\bot}$ to be a model on F. By the construction of F, every clause in $\boldsymbol{C}^{\neg\top}$ has its implicant evaluating to $\bot$ and every clause in $\boldsymbol{C}^{\neg\bot}$ has its head to $\top$. Moreover, by construction, if I' is not a model of F', then $I' \cup I$ can not be a model of F since F' is a sub-formula of F. □

Algorithm 5 shows the complete propagation operated by Schlandals when a decision is taken for a distribution. First, *Boolean Unit and Distribution Propagation* is applied, setting the selected variable to true (line 2). Then, all unnecessary clauses are removed from the residual formula (lines 3-7). This process is done in two steps. First, the clauses are marked as $\{\top, \bot\}$-reachable (lines 4-5). This marking is performed by recursively exploring the set of clauses, starting from those that satisfy the first conditions of Definition 8. Then, the adequate sub-routine is called. These sub-routines mark a clause C and, recursively, all the other clauses whose reachability is induced (i.e., they respect the second condition of Definition 8) by C. Marking them allows stopping the exploration when encountering clauses already explored previously (lines 10,15). Overall, this marking process can be run in time $O(|F| + |E|)$ where $|F|$ denotes the number of clauses in F and $|E|$ is the number of links between the clauses. Then, the clauses are filtered using Theorem 6, only keeping the ones that are both $\top$- and $\bot$-reachable. In practice, no assignment is done on the variables; knowing that an assignment on the deterministic variables exists is sufficient to remove the clauses from the residual formula.

Schlandals' propagation algorithm can be interpreted in terms of the initial problem being solved. We saw that it removes clauses related to subgraphs that are not on a path from the source to the target for reliability estimation problems. Moreover, when a partial assignment on the distributions (i.e., asserting that a subset of the edges is active or not) results in the source and target being disconnected, Algorithm 5 removes all clauses from the formula.

For Bayesian networks, it is known that not all the nodes in the network are necessary for computing the probability of evidence. Indeed, only the

Algorithm 5: Propagation algorithm of Schlandals.

```
1  Function Propagate(F, B, v)
       input  : A boolean formula F over distributions P
       input  : A variable B ∈ B to set to ⊤
       output: The residual formula F'
2      F' ← BUDP(F, B, ⊤)
3      foreach C = (I, H) ∈ F' do
4          if H = ⊥ or H ∈ P then F-Reach(F', C)
5          if V ∈ P ∀V ∈ I then T-Reach(F', C)
6      end
7      F' ← F' \ {C ∈ F' | C not ⊤-reachable ∨ C not ⊥-reachable}
8      return F'
9  Function F-Reach(F, C)
       input  : A boolean formula F, a clause C = (I, H) of F
10     if C is not marked as ⊥-reachable then
11         Mark C as ⊥-reachable
12         foreach C' = (I', H') ∈ F | H' ∈ I do F-Reach(F, C')
13     end
14 Function T-Reach(F, C)
       input  : A boolean formula F, a clause C = (I, H) of F
15     if C is not marked as ⊤-reachable then
16         Mark C as ⊤-reachable
17         foreach C' = (I', H') ∈ F | H ∈ I' do T-Reach(F, C')
18     end
```

ancestors of the observed nodes are necessary to compute the probability of that set. Schlandals' propagation effectively removes clauses not related to such nodes.

Algorithm 5 suggests a specific modelling choice: each head of a clause should be a deterministic variable. Doing so allows the propagation to reduce the problem when possible. Such a design choice is made for both Bayesian networks and reliability estimation problems. However, ProbLog programs do not impose such a restriction, which can hinder performances. Fortunately, as shown in the following example, it is sometimes possible to rewrite a ProbLog program to favour Schlandals propagation.

Example 22: Reliability estimation in ProbLog revisited

Previously, we showed that reliability estimation problems can be encoded in ProbLog using the two following clauses for the graph structure:

```
path(X, Y) :- edge(X, Y).
```

```
path(X, Y) :- edge(X, Z), path(Z, Y).
```

The first rule encodes the graph's direct edges, and the second encodes the path's transitivity property. Although this encoding works and can be used with Schlandals, it is only defined in terms of *probabilistic* variables: the edges. The snippet below demonstrates how to rewrite the rules to leverage Schlandals' propagation. The idea is to add new, *non-probabilistic*, terms corresponding to the edges.

```
0.2::edge(gen1,a).
0.4::edge(gen1,b).
0.8::edge(gen2, b).
0.6::edge(b,c).
0.1::edge(b,school).
0.35::edge(a,hosp).
0.05::edge(c,hosp).
0.5::edge(c,school).
arc(X, Y) :- edge(X, Y).
arc(Y, X) :- edge(X, Y).
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(X, Z), path(Z, Y).
query(path(gen1, hosp)).
```

In this example, the `arc` term is non-probabilistic and will be used during the grounding process. Although it may seem unnecessary to add such a term, our experiments will demonstrate that Schlandals benefits from such an addition.

4.1.2 Branching Heuristic

Many methods have been developed for *variable* selection algorithms in classical model counters. Some are based on the structure of the problem (e.g., literal counting, tree decomposition), others on the activity of the solvers (e.g., conflict-based), or a combination of both (e.g., VSADS). Schlandals' branching heuristics are based solely on the structure of the problem but differ from those previously proposed. Several rationales underlie our choice of a new structure-based branching heuristic.

First, clause learning is not implemented in Schlandals, making the use of heuristics such as VSADS impossible. Early experiments of clause learning showed no significant improvements on our benchmarks while requiring special implementation care to avoid cache corruption [San+04]. Moreover, heuristics based on VSADS require tuning multiple hyperparameters; VSADS has two parameters that weight each component. `Ganak`'s heuristics, CSVSADS, has an additional parameter for the cache factor. One of the aims of Schlandals is to remain simple; hence, we propose a new heuristic solely based on the problem's structure, without hyper-parameters to tune.

Moreover, Schlandals branches on *distributions*, not variables. Hence, it must select a group of variables, not a single one. It is possible to adapt existing branching heuristics for such a case. However, aggregating multiple variables' scores is not a trivial choice (e.g., taking the best score among all variables, their mean, and their median).

The final reason for a new structure-based heuristic is that our heuristic leverages Schlandals' Horn structure. More precisely, we explained that a Schlandals formula F can be seen as a directed graph $G = (\boldsymbol{C}, \boldsymbol{E})$ where $\boldsymbol{C}$ are F' clauses and there is an edge from $(\mathbf{I}, H) \in \boldsymbol{C}$ to $(\mathbf{I}', H') \in \boldsymbol{C}$ if and only if $H \in \mathbf{I}'$. We denote *leaves* the nodes with no outward edges and *sources* the nodes with no inward edges. The reasoning behind our heuristic is the following. An implication $\mathbf{I} \Rightarrow H$ can be seen as a chain reaction: if the implicant $\mathbf{I}$ is true, then the consequence H must be true, which, in turn, can set other implicants to true. The start of these chains of reasoning are clauses not implied by other clauses: G's sources.

Hence, our first heuristic selects a distribution which contains a variable appearing in a source node of G. Schlandals' propagation ensures that a projected variable always appears in G's source nodes. Let us interpret our heuristic when applied to the problems considered in this work.

We also define a symmetrical heuristic that selects distributions near the leaves of G. The intuition is that a leaf in G corresponds to a clause that implies $\bot$; hence, its implicant *must* be $\bot$. Hence, assigning distributions near G's leaves might lead to conflict (i.e., $\top \Rightarrow \bot$) early in the search tree.

Finally, we also implemented a version of the literal counting heuristic (DLCS) [SBK05a]. This heuristic selects the distribution appearing in the most clauses.

4.2 Knowledge Compilation in Schlandals

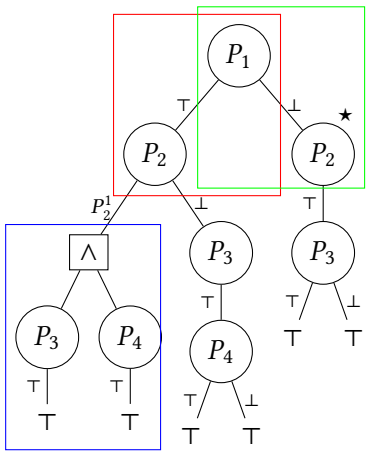
Schlandals is a DPLL-style model counter; hence, the trace of its execution corresponds to a d-DNNF [HD07]. Similarly to `Dsharp`, Schlandals can store its execution trace when it is used as a knowledge compiler. However, unlike `Dsharp`, Schlandals does not output d-DNNF diagrams: it directly returns *arithmetic circuits* (AC) that automatic differentiation tools and NeSy systems can use.

4.2.1 From Depth-First Search to Arithmetic Circuit

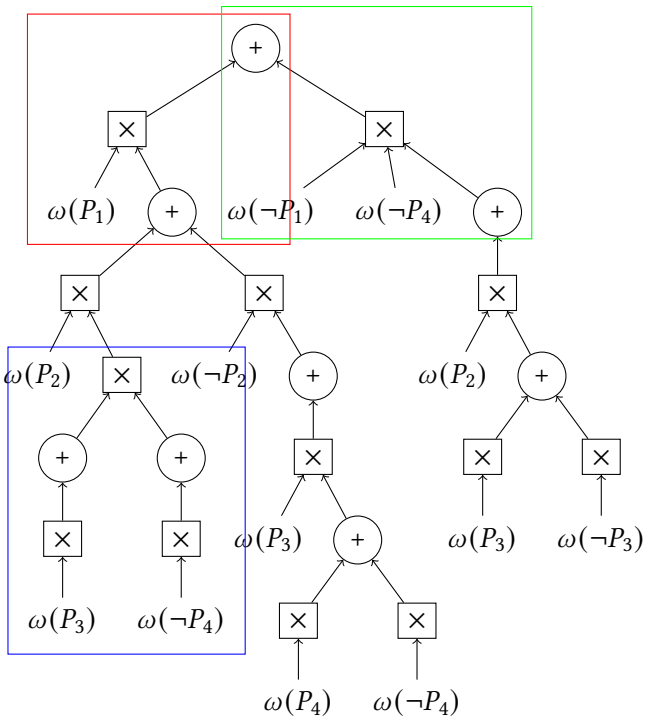
Let us first illustrate how a search tree can be converted to an arithmetic circuit.

Example: Conversion of a Search Tree into an AC

As a small example, let us consider a small Schlandals formula over four binary distributions $\boldsymbol{P}_1, \boldsymbol{P}_2, \boldsymbol{P}_3$, and $\boldsymbol{P}_4$. An example of a possible search tree produced by Schlandals is shown below. For conciseness, only the part of the sub-tree leading to a model is shown. In the node annotated with a star, it is assumed that the $\boldsymbol{P}_4$ distribution is fixed during propagation.



The coloured box indicates the part matching the arithmetic circuit shown below. The intuition behind the conversion algorithm is that the nodes can be transformed into operators (e.g., addition and multiplication), and the propagated variables can be used as multiplicative factors.



For example, the red and green boxes show how the decision at the root is transformed into an arithmetic circuit. The root is transformed into an addition (the sum of the counts of the branches) and has multiplicative nodes as its children. Each multiplicative node has the propagated variables and the sub-circuit corresponding to the remaining sub-problem as input. The blue square shows how independent components are transformed.

This example highlights that Algorithm 3 already contains all the necessary information to compute arithmetic circuits. Hence, the only modification needed is to store that information in the cache and post-process it once the search is finished. These modifications are highlighted in Algorithm 6, which outlines the search procedure for compilation.

In more detail, the cache stores structures instead of the count of each subformula F (line 5). These structures contain all the necessary information to compute F's arithmetic circuit representation. For each residual formula F' resulting of branching (lines 6-23), the variables fixed during propagation and the unconstrained distribution are stored (lines 10-12). Then, each independent component is stored as a child of the current cache entry (line 20).

Once the search is complete, the cache can be parsed using Algorithm 7. This algorithm parses the information stored in the cache to reproduce the computation made in Algorithm 6. For a given formula, all its branches are

Algorithm 6: DPLL Search with additional cached information for compilation

```
1  Function Schlandals-PWMC(F, P, ω, C)
       input  : F a boolean Horn-formula, over variables B, in CNF
       input  : P = ∪_{i=1}^{n} P_i ⊆ B a set of projected variables, partitioned
                into n distributions
       input  : ω a literal-weight function
       input  : C a cache of sub-results
       output: pwmc(F, P)
2      if F ∈ C then return C[F].count
3      if P = ∅ then return 1
4      i ← a distribution index such that ∃P ∈ P_i | B is not fixed
5      C[F] ← {count : _, subs : []}
6      foreach P ∈ P_i do
7          F' ← Propagate(F, P, ⊤)
8          if F' = ⊥ then return 0
9          Components ← all connected components of F'
10         cacheChild ← {fixed : _, unconstrained : _, children : []}
11         cacheChild.fixed ← {P | P ∈ P ∧ P = ⊤}
12         cacheChild.unconstrained ← ⋃_{P_j ∈ F ∧ P_j ∉ F'} P_j
13         fixed ← {P | P ∈ P ∧ P = ⊤}
14         count_P ← ∏_{j | P_j ∈ F ∧ P_j ∉ F'} Σ_{v ∈ P_j} ω(v)
15         count_P ← count_B × ∏_{P ∈ fixed} ω(P)
16         foreach Comp ∈ Components do
17             P' ← P reduced to the variables in Comp
18             C[Comp] ← DPLL-PWMC(Comp, P', ω, C)
19             count_P ← count_B * C[Comp].count
20             cacheChild.children.add(Comp)
21         end
22         C[F].subs.add(cacheChild)
23     end
24     C[F].count ← Σ_{P ∈ P_i} count_P
25     return C[F]
```

explored (lines 3-15) and linked to a sum node. For each branch, the propagated variables (line 5), the unconstrained distributions (lines 6-10), and the independent components (lines 11-13) are linked together using a product node. Once all sub-problems have been explored, the root of the sub-circuit is returned (line 14).

Using the search as a basis to perform knowledge compilation presented

Algorithm 7: Algorithm to convert the trace of Algorithm 7 into an arithmetic circuit

```
1  Function compile(C, F)
       input  : C the cache of sub-results filled using Algorithm 7
       input  : F the CNF (sub-)formula to transform into an arithmetic
                circuit
       output: The root of a (sub-)arithmetic circuit computing
                C[F].count
2      n ← Node (+)
3      foreach sub ∈ C[F].subs do
4          n' ← Node (×)
5          foreach P ∈ sub.fixed do add ω(P) as child to n'
6          foreach U ∈ sub.unconstrained do
7              u ← Node (+)
8              foreach U ∈ U do add ω(U) as child to u
9              add u as child to n'
10         end
11         foreach F' ∈ sub.children do
12             add compile(C, F') as child to n'
13         end
14         add n' as child to n
15     end
16     return n
```

several advantages. First, any improvement made to the search procedure directly impacts the compilation. In Chapter 5, we show how to modify the search to perform incremental and approximate counting; hence, parsing the cache of such a search procedure still produces a valid arithmetic circuit computing an approximate count.

Moreover, the overhead of Algorithm 7 when performing a classical search (i.e., without the need for a compiled AC) is almost null as the compilation-specific parts are easily enclosed in a condition. However, it should be noted that storing the propagated variables, unconstrained distributions, and independent components induces a memory overhead, which we evaluate in our experiments.

4.3 Experimental Evaluation

In this section, we evaluate the performance of the Schlandals solver on the inference tasks defined in Chapter 2. Before analysing the results, we describe the data sets and solvers used in our experiments.

4.3.1 Data sets and Solvers

We compare Schlandals against the following state-of-the-art model counters and reasoning systems: D4 [LM17a; LM19], GPMC [SHS17], sharpSAT-TD [KJ21], Ganak[1] [Sha+19; SM19; SGM], ExactMC [LMY21], Toulbar2 [SdS06], and ProbLog [DKT07]. Note that Ganak computes, in theory, $(0, \delta)$-approximations due to its probabilistic cache. However, in practice, and as noted in the original paper, Ganak always returns the true count in our experiments. All solvers except Schlandals have been compiled and run on their latest available version as of October 2024. Hence, the results presented in this text may differ from those presented in the original papers. Indeed, many solvers are actively developed and periodically enhanced; for example, both D4 and Ganak received significant updates recently, thereby increasing their performance. However, the main takeaways from our experiments are similar to those in the original publications.

These experiments consider the case of exact weighted model counting to compute the desired probability within a 600-second time limit. There is no limit on the size of the cache in our experiments as we compare search-based model counters and knowledge compilers. While all solvers can be run on the Bayesian network instances, only the ones supporting projected weighted model counting (i.e., Schlandals, D4, GPMC, Ganak) and ProbLog can be used on the reliability estimation instances.

Bayesian Network We used Bayesian networks from the bn-learn repository [Scu09] and the Grid network from the Cachet benchmarks [SBK05b]. The networks from the bn-learn repository are used in the literature. They have various sizes (from a few parameters to thousands) and topologies. The Grid network represents a grid of binary random variables. Each network node has two outgoing edges to its right and down neighbours if they exist. Each grid network is a $N \times N$ grid, where N ranges from 10 to 50, with a fixed ratio of deterministic nodes (i.e., nodes whose values are deterministically determined by their parents' values) that ranges from 50% to 90%. For each combination of size and deterministic ratio, ten grid networks are created, with the deterministic nodes selected at random.

The queries are created as follows. For the networks from the bn-learn repository, one query is created per network's leaf, and the goal is to compute the probability of one of its values, selected randomly. The grid networks have

[1] Ganak received a significant update for the 2024 model counting competition. Its code has been merged with approxMC, an approximate model counter, and has been heavily rewritten. Unfortunately, there is no publication describing the changes made to both methods at the time of this writing. However, the authors kindly provided an executable for this updated and more performant version of both Ganak and approxMC.

only one leaf, and the task is to compute the probability that the leaf is true or false, decided randomly. For both types of networks, no additional evidence is encoded. Each query is encoded using the encoding presented in Chapter 3 for Schlandals and the ENC4 encoding [CD06] for the other weighted model counters. We did not use the more efficient ENC4LINP encoding because `ExactMC` requires weights to be between 0 and 1; however, the results presented below are similar when using ENC4 and ENC4LINP. `Toulbar2` works directly with the Bayesian network and evidence as input, and the networks are encoded in ProbLog as shown in Example 2.2.4.

Reliability Estimation For the reliability estimation problem, we use graphs from the GridKit tool [Med+17; Wie16] that represent the power grid networks of Europe and the USA. Each network is further divided by country for Europe and state for the USA, and the resulting sub-networks are used as benchmarks. These graphs are undirected, so each edge is encoded with two clauses. Following previous work, we set each edge's probability of being inactive to 0.125 [Due+17]. Five random queries are created for each sub-network by randomly selecting a pair of connected nodes.

4.3.2 Results

Let us now look at the experimental results. We will answer the following questions: I) How efficient is Schlandals compared to state-of-the-art solvers? II) What is the impact of Schlandals' additional propagation? III) What is the overhead of Schlandals' compilation algorithm? IV) Is it beneficial to use Schlandals as a counting algorithm in ProbLog?

Comparison between Schlandals and state-of-the-art solvers First, let us compare Schlandals against the state-of-the-art solvers. Figure 4.1 shows each solver's proportion of Bayesian networks solved over time. We consider the networks from the bn-learn (upper part of the figure) and the grid networks (lower part) separately as they exhibit different structural properties. For bn-learn networks, the difficulty of the query is primarily determined by the queried node and the network topography. On the other hand, grid networks have a fixed structure, and the grid size and the amount of determinism mainly determine their complexity. Moreover, we have two settings for the bn-learn networks: either the entire network and query are encoded using either ENC4 or Schlandals' encoding (left), or the network is pre-processed to keep only the ancestors of the queried node, and then it is encoded (right). Such pre-processing is useless for grid networks, as all nodes are required for query computation.

Let us first analyse the case of bn-learn networks. It can be seen that the pre-processing of the network impacts all solvers except Schlandals and ProbLog. Schlandals' additional propagation, based on $\{\top, \bot\}$-reachability, is equivalent to removing all clauses unnecessary to the query. Hence, the only gain from pre-processing the network is a faster initial propagation for Schlandals. ProbLog grounding procedure proceeds similarly; every unnecessary rule is removed. On the other hand, all other solvers benefit significantly from the pre-processing, allowing them to solve almost all instances. Overall, Schlandals is outperformed by other model counters when the network is pre-processed, but it can still solve roughly 90% of the instances.

Interestingly, `Toulbar2` performance increases slightly with longer run times; it solves most instances instantaneously and then only a few more. However, regardless of the pre-processing, it still solves roughly the same number of instances as the weighted model counters. The best-performing model counters are `D4`, `GPMC`, and `ExactMC`: they perform relatively similarly and solve all instances in a few seconds when the networks are pre-processed. `Ganak` and `sharpSAT-TD` take more time to solve the instances, but this is due to their pre-processing, which requires a lot of time. For example, `sharpSAT-TD` computes a tree-decomposition of the formula's primal graph using the `FlowCutter` [Str17] anytime algorithm. It runs until a user-defined timeout, 120 seconds in our experiments, and then the solver starts the model count algorithm. `Ganak`, among other things, computes a tree decomposition and independent support for difficult instances, both of which require time.

Finally, ProbLog does not perform well, which is expected. Indeed, while it is possible to encode BN in ProbLog, it can not benefit from the optimisations developed for the CNF encodings (e.g., reusing the same variable for multiple entries in a CPT). Hence, it can solve only the easiest instances.

The situation is different for the grid networks. In this case, Schlandals perform the worst, solving only 40% of the instances, while the best-performing weighted model counters solve almost 90% of the instances. After analysing Schlandals' behaviour on these instances, we observed that it struggles to solve most instances due to a very deep search tree (e.g., up to a depth of 200). One possible explanation for such a deep search tree is our simple distribution selection heuristic. Because it is too simplistic, it does not make appropriate decisions (e.g., it does not favour decomposition or consider conflicts), drastically increasing the search space.

Figure 4.2 shows the proportion of instances solved for the reliability estimation problems. All solvers behave similarly; most of the instances they solve are solved in a few seconds, and then there is a long plateau with very few instances additionally solved. Overall, it can be seen that Schlandals is the best-performing model counter, and ProbLog performs better than pure

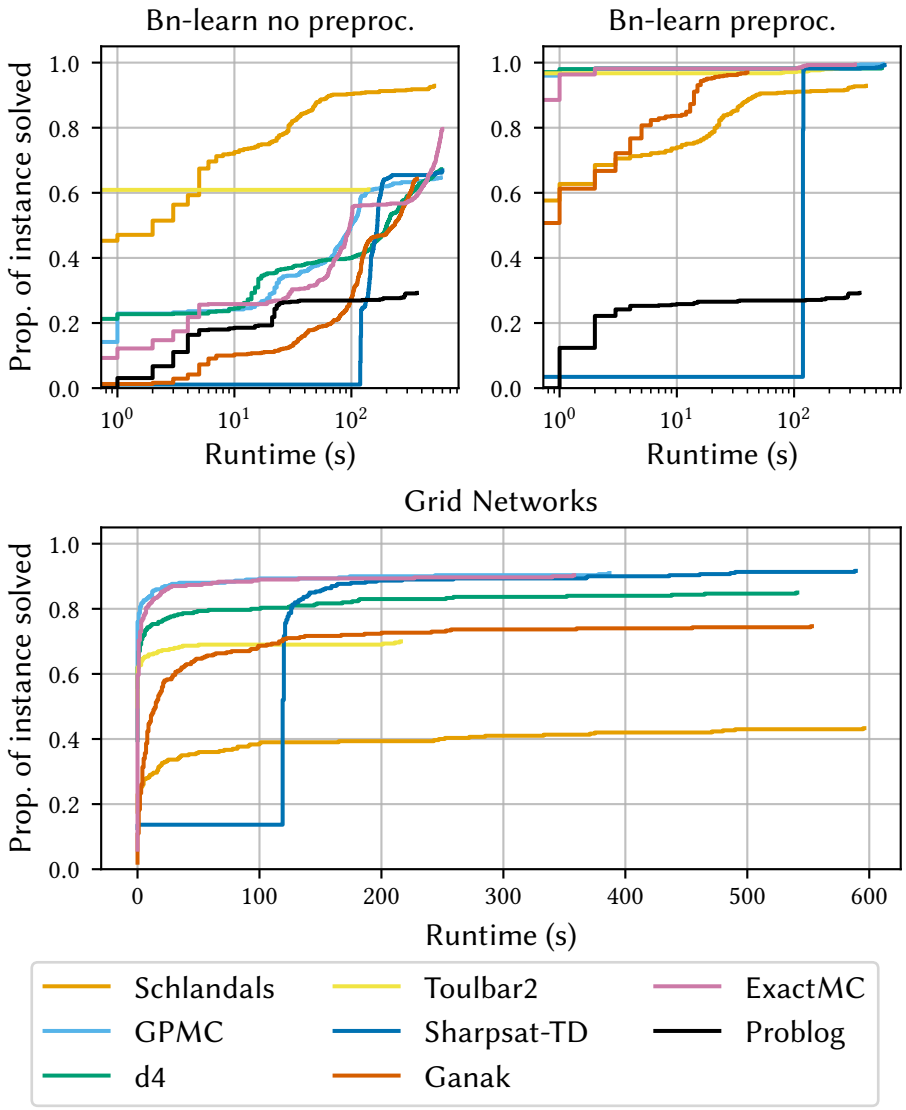


Figure 4.1: Proportion of solved instances over time, with a timeout of 600 seconds, for Bn-learn (up) and grid (down) Bayesian networks. The Bn-learn networks are either fully encoded in CNF (left) or pre-processed to keep only nodes relevant to the query (right). The lines stop when no more queries are solved.

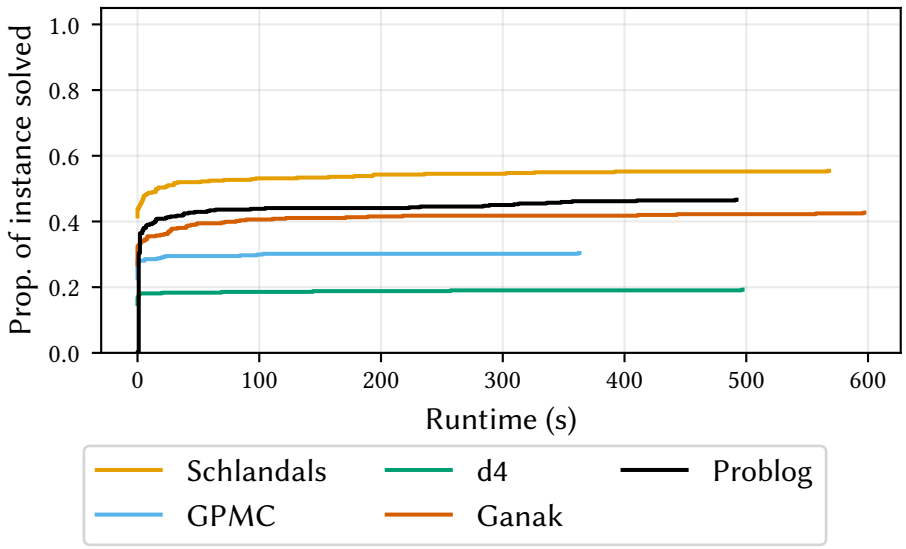


Figure 4.2: Proportion of solved instances over time, with a timeout of 600 seconds, for reliability estimation problems. The lines stop when no more queries are solved.

model counters.

Overall, it can be seen that Schlandals performance depends on the problem type. It performs less well for Bayesian networks than state-of-the-art model counters with an optimised encoding. Schlandals' encoding is less optimised than ENC4; it lies between ENC1 and ENC3. Hence, it is expected that Schlandals performs less well on Bayesian networks. However, for reliability estimation problems, all solvers share the same encoding, and Schlandals performs the best.

Overhead of the compilation Let us now analyse the relative performance of various settings for Schlandals. First, let us compare Schlandals' search and compilation algorithms. Figure 4.3 (left) shows the relative performance of the search against the compilation. Before analysing the result, let us briefly explain how to read such a graph. It is a scatter plot in which each data point is an instance (i.e., a Bayesian network or a reliability estimation problem). The coordinates of each point are given by the run time of each method: an instance at $(x = 10, y = 30)$ takes 10 seconds to be solved by the search and 30 seconds to be compiled. The solid black line represents the $y = x$ line. Hence, computing the count of instances on the diagonal takes as much time for the search as it does for the compilation. Instances above (resp. below) the diagonal are solved faster with the search (resp. with the

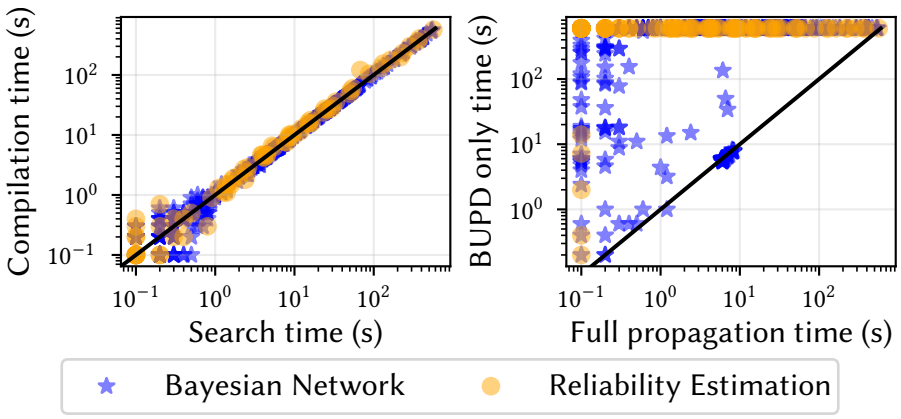


Figure 4.3: Relative performance of Schlandals' search against Schlandals' compilation (left) and Schlandals' search with only the BUPD algorithm as propagation (right)

compilation). Finally, each instance was run five times, and the average run time is reported. Instances not solved by both instances are not plotted; if an instance is solved by one method but not the other, it is placed at 600 on the axis corresponding to the latter method.

Let us now analyse the overhead induced by the compilation. It can be seen that all instances align on the diagonal; hence, there is no overhead for performing knowledge compilation in Schlandals. There are a few exceptions, but they can be solved in under a second; hence, a slight variance is expected for such small instances. It can be surprising given Schlandals' compilation algorithm: it first executes the search with a modified cache structure and then parses it to create an arithmetic circuit. Hence, the compilation takes at least as much time as the search. Let us first observe that most of the overhead comes from the modified cache structure and the memory overhead it induces. Indeed, post-processing the cache is fast: only the satisfiable part of the search space is compiled, and there is no need to perform branching and propagation as all the necessary information is stored in the cache. Our experimental results imply that the memory overhead is limited. In practice, the overhead induced by our modified cache structure is equivalent to the size of the arithmetic circuit being built. Indeed, each element stored in the cache is used to create nodes in the circuit: the elements propagated to $\top$, the unconstrained distributions, and the references to children. Hence, the additional memory used corresponds to the number of edges in the circuit. In particular, the memory footprint of our compilation algorithm is equivalent to directly creating the NNF diagram during the search.

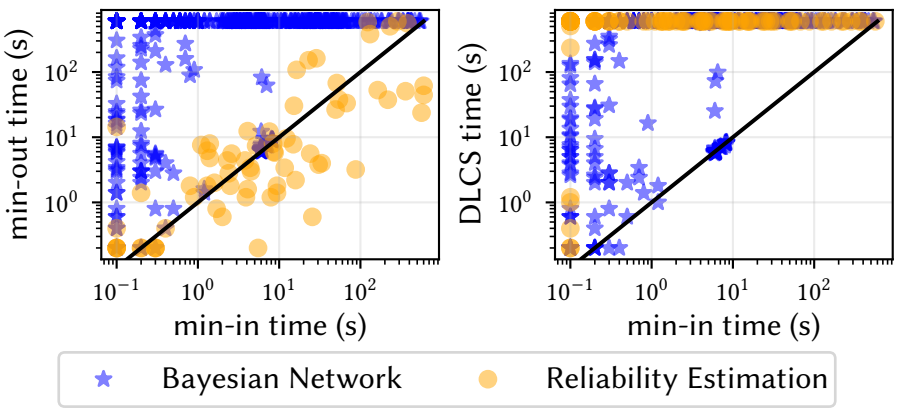


Figure 4.4: Relative performance of Schlandals' min-in degree branching heuristic against Schlandals' min-out branching heuristic (left) and the DLCS heuristic (right).

Impact of Schlandals' additional propagation Next, we analyse the benefits of Schlandals' additional propagation (i.e., removing clauses not $\top$-reachable or $\bot$-reachable). Figure 4.3 (right) shows the relative performances of Schlandals with this propagation against Schlandals search using only the boolean unit propagation and the distribution constraints. In both cases, the pre-processing still applies the additional propagation; hence, any benefit of our additional propagation is gained *during the search*. It can be seen that the additional propagation is crucial for Schlandals efficiency; if only the boolean unit propagation and the distribution constraints are applied, Schlandals solves much fewer instances and less quickly. For example, several Bayesian networks that were solved in a few seconds with complete propagation are solved in more than a hundred seconds without it.

It can be easily understood when considering reliability estimation problems. When the choices for the distributions (i.e., the edges) result in the source and target being disconnected, the problem is solved, and Schlandals' propagation removes all clauses from the residual formula. However, when such propagation is not done, the solver explores the residual formula even if it does not impact the count. More generally, our additional propagation allows Schlandals to detect and remove parts of the problem that do not impact the count. Without it, additional branching must occur, which increases the search tree's depth and degrades its performance.

Comparison of Schlandals' branching heuristics Figure 4.4 shows the comparisons of Schlandals' min-in degree heuristic against its min-out degree heuristic and the DLCS heuristic [SBK05a]. While it is known that the

literal-counting heuristic underperforms compared to mixed heuristics such as VSADS, it can be implemented in Schlandals as it is not a failure-based heuristic. Hence, we use it as a baseline for our heuristic.

Overall, the min-in-degree heuristic outperforms the two others, especially on Bayesian networks. However, for reliability estimation problems, the min-out degree heuristics performs slightly better. For some networks' topologies, starting branching near the target node might be advantageous (e.g., a target node linked by a single edge to the rest of the graph). Such behaviour opens the question of more refined branching heuristics, such as ones based on tree decomposition. Analysing the problem's structure (i.e., the graph of clauses) can be beneficial for reliability estimation problems.

Schlandals as Inference Mechanism in ProbLog We analyse the impact of using Schlandals as the inference mechanism in ProbLog. We have seen previously that Schlandals, when used as a standalone tool, performs better than ProbLog. We now analyse in more detail if using Schlandals as the inference tool in ProbLog has benefits. Figure 4.5 (left) shows the relative performance of using Schlandals in ProbLog against the default ProbLog inference algorithm. Several small instances (i.e., solved in less than ten seconds) can be solved by both methods, and the ProbLog default counting method is sometimes slightly faster than using Schlandals. However, many instances are solved faster using Schlandals or can only be solved using Schlandals. In particular, for the reliability estimation problems used in this work, there are positive cycles in the initial ProbLog program; hence, using Schlandals allows for drastically decreasing the size of the CNF formula.

Finally, we analyse the impact of the initial ProbLog program on the performance of Schlandals. In particular, reliability estimation problems can be encoded in various ways in ProbLog. We have explored the classical way of encoding this problem in Chapter 2: each edge (x, y) with probability p is encoded with a probabilistic fact `p::edge(x, y).`, and the path transitive property is encoded using the rules `path(X, Y) :- edge(X, Y).` and `path(X, Y) :- edge(X, Z), path(Z, Y).`. However, such a structure is not optimal for Schlandals as it only relies on probabilistic facts. In Chapter 3, we proposed an alternative encoding by adding new non-probabilistic facts `arc(X, Y) :- edge(X, Y).` and by rewriting the transitive property rules using the `arc` terms instead of the `edge` terms. Figure 4.5 (right) shows the relative performance of this new arc-encoding against the classical ProbLog encoding. For both encodings, ProbLog uses Schlandals as a weighted model counting algorithm. It can be seen that, in addition to solving the instances more quickly, many instances can not be solved without the arc-encoding.

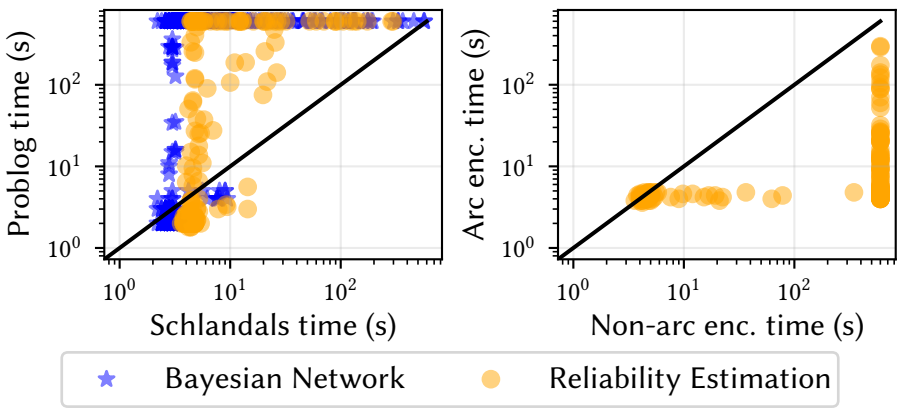


Figure 4.5: Left) the relative performance of ProbLog with Schlandals as counting algorithm against default ProbLog inference mechanism. Right) the relative performance of ProbLog using Schlandals as counting algorithm on reliability problem estimation with the classical encoding against the arc-based encoding.

4.4 Conclusion

This chapter explored how to modify a classical DPLL-style search-based model counter to calculate the weighted model count of a Schlandals formula. We developed a new search-based, DPLL-style model counter (also named Schlandals) for this purpose. We showed that minor modifications to the classical exhaustive DPLL search algorithm allow computing the count of a Schlandals formula. Moreover, we developed a new algorithm that leverages the Horn structure of the formulas to simplify them during the search. Additionally, an interface to Schlandals has been implemented in ProbLog, allowing us to run ProbLog programs with Schlandals as a counting algorithm.

We evaluated the performance of Schlandals against state-of-the-art weighted model counters and reasoning systems on two problems: Bayesian network inference and reliability estimation problems. Our experiments showed that Schlandals is competitive with other solvers: on Bayesian networks, it solves most instances, except for the more difficult Grid networks, and it outperforms the other solvers on reliability estimation problems. Our experiments also highlighted that pre-processing the Bayesian networks is essential for the performance of model counters, but Schlandals propagation performs such operation natively. We also analysed the impact of various Schlandals' components on its performance and showed that our algorithm for simplifying a Horn formula is essential for Schlandals' efficiency. Finally, our experiments demonstrate that ProbLog benefits from using Schlandals on the studied problems; however, it can require rethinking the structure of ProbLog programs

by introducing non-probabilistic terms for reliability estimation problems.

Approximate Inference in Schlandals

5

The algorithm developed in Chapter 4 computes the weighted model count of a formula and returns it when the whole search space has been explored. Given that computing the WMC of a formula is #*P*-Complete, it might be infeasible in a reasonable amount of time. However, most of the time, computing the exact probability is unnecessary. For example, in medical diagnosis [Shw+91; Pra+94], a disease is diagnosed in a patient if the probability that the patient has the disease exceeds a threshold set by the expert. Hence, computing the exact probability is unnecessary; a lower bound is sufficient.

This chapter introduces algorithms that can be used to compute approximations of the WMC. In particular, the methods proposed in this work provide both a lower *and an upper bound* on the true WMC. We first explain how Schlandals' search algorithm can be modified to compute *bounds* on the weighted model count and then provide two algorithms that provide error-bounded approximations of the true weighted model count.

This chapter is based on the following articles:

- A. Dubray, P. Schaus, and S. Nijssen. "Anytime Weighted Model Counting with Approximation Guarantees for Probabilistic Inference". In: *LIPIcs, Volume 307, CP 2024* 307 (2024). Ed. by P. Shaw. ISSN: 1868-8969. DOI: 10.4230/LIPICS.CP.2024.10. (Visited on 03/07/2025)
- L. Dierckx, A. Dubray, and S. Nijssen. "Learning from Logical Constraints with Lower- and Upper-Bound Arithmetic Circuits" International Joint Conference on Artificial Intelligence (IJCAI) 2025.

The content presented in Section 5.5 has not been published before, and supplementary experiments have been conducted.

5.1 Intuition and Motivation

Many algorithms have been developed for approximate (weighted) model counting or probabilistic inference (e.g., [Cha+14; Gom+07; LMY22; SGM; SM19; Vla+15; GD11]) They often rely on some sampling and provide statistical guarantees. For example, `SampleSAT` [Gom+07] and `PartialKC` [LMY22]

are *unweighted* model counters providing estimations on the true model count. `SampleSAT` provides a correct lower bound with a given probability, while `PartialKC` aims to provide an unbiased estimate of the count. There are only a few methods explicitly designed for *weighted* problems [Cha+14; GD11], but they often suffer from the same problem: the bounds returned on the WMC might be invalid. ProbLog allows the computation of true bounds on the probability of a query, but it is limited to the inference of a ProbLog program [Vla+15].

One benefit of sampling-based approaches is that they are *anytime*. That is, they can be stopped at any time during their execution and still return a result, and the longer they run, the more accurate the result is (e.g., the probability of `SampleSAT`'s lower bound being wrong decreases with the runtime). Hence, it is possible to obtain an excellent estimate of the true (weighted) model count by running the methods for a sufficiently long time.

To our knowledge, no weighted model counter provides *deterministic lower and upper bounds* on the true weighted model count. This chapter explores how Schlandals can be adapted to produce such bounds. Subsequently, we demonstrate that adapting Schlandals' search strategy allows providing these bounds in an anytime fashion: the solver iteratively produces bounds that become tighter at each iteration. For the rest of this chapter, we consider Schlandals formulas that cannot be solved exactly within the allowed time limit. In practice, this means that some of the children of some nodes in the search tree are unexplored.

5.2 Counting Unsatisfying Assignments

Calculating the weighted model count of a propositional formula can be seen from two perspectives. The first chapters of this work present the classical approach. It consists of finding all satisfying interpretations of the formula and summing their weights. The other perspective can be seen as solving a *dual problem*; that is, computing the weighted sum of the non-satisfying assignments of the formula. These two problems are complementary. In particular, the weighted sum of satisfying and non-satisfying assignments is 1 for probabilistic inference.

Example 24: Duality of Weighted Model Counting

Let us demonstrate this duality on the boolean formula considered in Chapter 2: $F = (A \vee \neg B) \wedge (\neg A \vee C \vee D) \wedge (\neg A \vee \neg C)$ with $\boldsymbol{P} = \{A, B\}$. The weight function ω is defined as follows: $\omega(A) = 0.6, \omega(\neg A) = 0.4, \omega(B) = 0.2, \omega(\neg B) = 0.8$. We computed in Chapter 2 that

$$\text{pwmc}(F, \{A, B\}) = 0.92.$$

In particular, the only assignment on $\{A, B\}$ that cannot be extended in a model of F is $I(A) = \bot, I(B) = \top$ because the first clause always evaluates to $\bot$. The weight of this assignment is as follows.

$$\omega(\neg A) \times \omega(B) = 0.4 \times 0.2 = 0.08 = 1 - 0.92$$

This example illustrates two methods for computing a formula's weighted model count: summing the number of satisfying or non-satisfying assignments. This observation is not new and has been made by Möhle and Biere in the case of *unweighted* projected model counting [MB18]. Möhle and Biere proposed to create two propositional formulas to represent the input formula F or its negation $\neg F$. Then, they observed that finding unsatisfying assignments on the formula representing $\neg F$ correspond to models of F. On the contrary, our approach does not create new formulas but reasons about F's unsatisfying assignments to derive an upper-bound on the weighted model count. Hence, our method differs conceptually from the one of Möhle and Biere, leading to two different solvers.

Following the naming convention introduced by Möhle and Biere, we call the problem of computing the weighted sum of unsatisfying assignments the *dual (projected) weighted model counting problem* and denote it $\overline{\text{pwmc}}(F, \boldsymbol{P})$ for a Schlandals formula F and projected variables $\boldsymbol{P}$. The following proposition connects the two counting problems.

Proposition 1. *Let F be a literal-weighted boolean formula over variables $\boldsymbol{B}$ with weight function ω and distributions $\boldsymbol{P} = \cup_{i=1}^{n} \boldsymbol{P}_i$. The following equality holds.*

$$\text{pwmc}(F, \boldsymbol{P}) + \overline{\text{pwmc}}(F, \boldsymbol{P}) = \prod_{i=1}^{n} \sum_{P \in \boldsymbol{P}_i} \omega(P) \tag{5.1}$$

Proof. The proof is similar to that of Theorem 5. The weight of each possible assignment must be counted as both satisfying and unsatisfying assignments are summed up; this is similar to the case in which distributions are unconstrained. □

This proposition states that the sum of all interpretations of a boolean formula can be expressed in closed-form formulation. Using this proposition, we can derive an upper bound on the true weighted model count when the search space is partially explored. Let us denote $LB_{\top}(F, \boldsymbol{P})$ (resp. $LB_{\bot}(F, \boldsymbol{P})$) a partial evaluation of the weighted sum of satisfying (resp. unsatisfying)

assignments on F. Then, we have the following relationships.

$$LB_{\top}(F, \boldsymbol{P}) \leq \text{pwmc}(F, \boldsymbol{P}) \tag{5.2}$$

$$\prod_{i=1}^{n} \sum_{P \in \boldsymbol{P}_i} \omega(P) - LB_{\bot}(F, \boldsymbol{P}) \geq \text{pwmc}(F, \boldsymbol{P}) \tag{5.3}$$

Equation (5.2) follows directly from the definition of $LB_{\top}(F, \boldsymbol{P})$ and Equation (5.3) is easily derived from Proposition 1. Equation (5.2) can be used to calculate a lower bound on the true model count. A key observation is that DPLL-style search algorithms, such as Schlandals' search algorithm, compute, by design, a valid value for $LB_{\top}(F, \boldsymbol{P})$ if the search space is partially explored. Intuitively, when Schlandals branches on a distribution $\boldsymbol{P}_i = \{P_i^1, \dots, P_i^k\}$, the weighted model count of F can be written as follows.

$$\text{pwmc}(F, \boldsymbol{P}) = \sum_{j=1}^{k} \omega(P_i^j) \times \text{pwmc}(F[P_i^j = \top], \boldsymbol{P} \setminus \boldsymbol{P}_i)$$

Hence, evaluating partially this sum results in a lower bound of the true count. The remaining question is how to compute the sum of unsatisfying assignments to obtain $LB_{\bot}(F, \boldsymbol{P})$.

5.3 Solving the Dual Weighted Model Counting Problem

With few modifications, Schlandals can solve the dual-weighted model counting problem. We first present the particularity of counting unsatisfying assignments and then give a modified DPLL-style search that solves both counting problems simultaneously. For the rest of this section, let us assume that F is the Schlandals formula being considered and $\boldsymbol{P} = \cup_{i=1}^{n} \boldsymbol{P}_i$ its distributions. Similarly to classical weighted model counting, the count of unsatisfying interpretations is computed from the propagation result and the count of sub-problems.

Detecting Unsatisfying Assignments from Propagation When descending in the search tree, computing the WMC of a Schlandals formula using Algorithm 3 can be seen as constructing models (i.e., satisfying assignments to the distributions). Setting a probabilistic variable to $\top$ during the propagation adds this assignment to the model under construction. However, computing unsatisfying assignments is slightly different: forcing a distribution's variable $P_i \in \boldsymbol{P}_i$ to $\bot$ means that any interpretations with $P_i = \top$ *cannot* be a model of the formula. Hence, when setting a variable to $\bot$ during propagation, the weight of all interpretations containing this variable can be added to the sum of unsatisfying assignments.

Example: Unsatisfying Assignments from Propagation

Let $\boldsymbol{P}_1 = \{P_1^1, P_1^2\}$ and $\boldsymbol{P}_2 = \{P_2^1, P_2^2\}$ be two distributions and

$$F = (P_1^1 \Rightarrow A) \wedge (P_2^1 \wedge A \Rightarrow \bot)$$

be a Schlandals formula. The truth table for F is shown below.

$\boldsymbol{P}_1$	$\boldsymbol{P}_2$	Projected model of F
P_1^1	P_2^1	$\bot$
P_1^1	P_2^2	$\top$
P_1^2	P_2^1	$\top$
P_1^2	P_2^2	$\top$

Let us assume that the solver branches on $\boldsymbol{P_1} = P_1^1$. It forces $A = \top$ which, in turn, forces $\boldsymbol{P}_2 = P_2^1$. Hence, assigning $\boldsymbol{P}_1 = P_1^1$ forces $P_2^2 = \bot$, disallowing the solver to branch on the only unsatisfying assignment of F. After such assignments, that branch's weighted count of unsatisfying assignments can be computed as $\omega(P_1^1) \times \omega(P_2^2)$.

Let us formalise the intuition of the previous example. Let us assume, without loss of generality, that Schlandals is solving a formula F on n distributions and branches on $\boldsymbol{P}_1 = P_1^1$. Let F' be the residual formula without $\boldsymbol{P}_1$ and $\text{dom}(\boldsymbol{P}_i)$ (resp. $\text{dom}'(\boldsymbol{P}_i)$) denotes the domain (i.e., either all unassigned variables in $\boldsymbol{P}_i$ or its only variable set to $\top$) before (resp. after) propagation. Then, the following equality holds:

$$\prod_{i=2}^{n} \sum_{P' \in \text{dom}'(\boldsymbol{P}_i)} \omega(P') \leq \prod_{i=2}^{n} \sum_{P \in \text{dom}(\boldsymbol{P}_i)} \omega(P). \tag{5.4}$$

Indeed, since the propagation only removes elements from the domains, we have that $\forall i \ \text{dom}'(\boldsymbol{P}_i) \subseteq \text{dom}(\boldsymbol{P}_i)$. We argue that the weighted sum of interpretations set as unsatisfying during the propagation is given by

$$\omega(P_1^1) \times \left(\prod_{i=2}^{n} \sum_{P \in \text{dom}(\boldsymbol{P}_i)} \omega(P) - \prod_{i=2}^{n} \sum_{P' \in \text{dom}'(\boldsymbol{P}_i)} \omega(P') \right) \tag{5.5}$$

The reasoning is the following. Both side of Inequality 5.4 represents the maximum weighted model count of F' in two different settings. On the right-hand side, it is its maximum WMC *if only* $\boldsymbol{P}_1$ *is removed from* F. The left-hand side represents the actual maximum WMC, taking into account the effect of Schlandals' propagation. The left-hand side is smaller because the propagation removed some interpretations from the set of possible models. Hence, the difference between these two values corresponds to the weighted sum of the

interpretations detected as unsatisfying during the propagation. Finally, this computation is performed for F''s interpretations, which do not contain $\boldsymbol{P}_1$. Hence, this value must be multiplied by the weight of the variable branched on, P_1^1 in this case.

Example: Unsatisfying Assignments from Propagation (cont.)
Following our previous example, the only distribution remaining after branching on $\boldsymbol{P}_1$ is $\boldsymbol{P}_2$. We have $\text{dom}(\boldsymbol{P}_2) = \{P_2^1, P_2^2\}$ and $\text{dom}'(\boldsymbol{P}_2) = \{P_2^1\}$. Hence, the weighted sum of unsatisfying assignments in the branch is given by

$$\omega(P_1^1) \times \left((\omega(P_2^1) + \omega(P_2^2)) - \omega(P_2^1)\right) = \omega(P_1^1) \times \omega(P_2^2)$$

Unsatisfying Assignments for Independent Components In classical model counting, when a formula F is decomposed into k independent components $F^1, \ldots, F^k$, their count is multiplied by each other. Intuitively, a model of F is an element in the cartesian product of its components model space: it must be a model of each component. However, for unsatisfying assignments, it does not work like that: an interpretation on F results in a contradiction if *at least* one of the components is unsatisfied. Fortunately, it is possible to derive an equation for the dual-weighted model count of F using proposition Proposition 1.

Let us denote $\boldsymbol{P}^i$ the projected variables appearing in the i-th independent component and $\max(F, \boldsymbol{P}) = \prod_{i=1}^{n} \sum_{P \in \boldsymbol{P}_i} \omega(P)$. We define $\max(F^i, \boldsymbol{P}^i)$ similarly for the independent components. The following equation provides the formula for calculating the weighted sum of unsatisfying assignments for independent components.

$$\max(F, \boldsymbol{P}) - \underbrace{\prod_{i=1}^{k} \left(\underbrace{\max(F^i, \boldsymbol{P}^i) - \overline{\text{pwmc}}(F^i, \boldsymbol{P}^i)}_{\text{max value for } pwmc(F^i, \boldsymbol{P}^i)} \right)}_{\text{max value for } pwmc(F, \boldsymbol{P})} \tag{5.6}$$

Equation (5.6) can be interpreted as follows. Each component is possibly partially solved; hence, $\overline{\text{pwmc}}(F^i, \boldsymbol{P}^i)$ is the (partial) sum of unsatisfying assignments of component F^i. It follows that $\max(F^i, \boldsymbol{P}^i) - \overline{\text{pwmc}}(F^i, \boldsymbol{P}^i)$ is the maximum value that the weighted model count of F^i can take (i.e., all unexplored interpretations of F^i are models). Hence, the product in Equation (5.6) is the product of component *maximum possible weighted model count*; that is, the maximum possible weighted model count for F. Subtracting this value from the theoretical maximum value for $\text{pwmc}(F, \boldsymbol{P})$ results in the weighted

sum of all assignments detected as unsatisfying. The benefit of Equation (5.6) is that it provides a closed-form formulation to compute the dual-weighted model count for independent components as a product. Hence, it can be easily integrated into Schlandals' search algorithm.

Algorithm 8: DPLL-based algorithm for solving the dual weighted model count problem

Function `Dual-PWMC`$(F, \mathbf{P}, \omega, C)$
- **input :** F a boolean Horn-formula, over variables $\mathbf{B}$, in CNF
- **input :** $\mathbf{P} = \cup_{i=1}^{n} \mathbf{P}_i \subseteq \mathbf{B}$ a set of projected variables, partitioned into n distributions
- **input :** ω a literal-weight function
- **input :** C a cache of sub-results
- **output:** The weighted count of the unsatisfying assignments of F

if $F \in C$ **then return** $C[F]$
if $\mathbf{P} = \emptyset$ **then return** 0
$i \leftarrow$ a distribution index such that $\exists P \in \mathbf{P}_i \mid P$ is not fixed
$maxResidual \leftarrow \prod_{j=1 \mid j \neq i}^{n} \sum_{P' \in \mathbf{P}_j} \omega(P')$
foreach $P \in \mathbf{P}_i$ **do**
  $F' \leftarrow$ `Propagate`$(F, P, \top)$
  if $F' = \bot$ **then** $count_{\bot}^{P} \leftarrow maxResidual \times \omega(P)$
  else
    $prop_{\bot} \leftarrow \omega(P) \times \left(maxResidual - \prod_{j=1 \mid j \neq i}^{n} \sum_{P' \in \mathbf{P}_j} \omega(P')\right)$
    fixed $\leftarrow \{P' \mid P' \in \mathbf{P} \wedge P' = \top\}$
    $p \leftarrow \prod_{P' \in \text{fixed}} \omega(P') \times \prod_{j \mid \mathbf{P}_j \text{ is unconstrained in } F'} \sum_{P' \in \mathbf{P}_j} \omega(P')$
    $Components \leftarrow$ all connected components of F'
    foreach $Comp \in Components$ **do**
      $\mathbf{P}' \leftarrow \mathbf{P}$ reduced to the variables in $Comp$
      $Comp_{\bot} \leftarrow$ `Dual-PWMC`$(Comp, \mathbf{P}', \omega, C)$
      $Comp_{\bot} \leftarrow \max(Comp, \mathbf{P}') - Comp_{\bot}$
    end
    $count_{\bot}^{P} \leftarrow \max(F, \mathbf{P}) - \prod_{Comp} Comp_{\bot}$
    $count_{\bot}^{P} \leftarrow count_{\bot}^{P} \times p + prop_{\bot}$
  end
end
$C[F] \leftarrow \sum_{P \in \mathbf{P}_i} count_{\bot}^{P}$
return $C[F]$

A DPLL-Style Algorithm for Solving the Dual Counting Problem Algorithm 8 shows how to solve the dual weighted model counting on a Schlandals formula. The algorithm's structure is similar to the classical Schlandals' search algorithm: a cache is used to store count of sub-formulas (lines 2,23), it branches on all the value of a distribution (lines 4-22), and it recursively solves the independent components (lines 13-18) of the residual formula after propagation (line 7).

As explained above, Algorithm 3 differs from Algorithm 8 when computing the weighted model count. When the formula is empty, a satisfiable leaf has been found, and 0 is returned (line 3). On the contrary, when the residual formula, after propagation, is unsatisfiable, then the weighted sum of all its interpretations is computed using Proposition 1 and multiplied by $\omega(P)$ as P is forced in that branch (line 8).

If the residual formula is not $\bot$, then the count is computed as described above. First, the count of the interpretations detected as unsatisfying during propagation is computed using Equation (5.5) (line 10). Then, the dual weighted count of the independent components is computed using Equation (5.6) (lines 17,19). Finally, these two values are summed together (line 20). It is crucial to multiply the results of the recursive calls by the weights of the variables set to $\top$ during the propagation and the weight of the unconstrained distributions. Indeed, in the same way that classical weighted model counting builds interpretations using propagation, the unsatisfying assignments found during recursive calls extend these assignments.

For simplicity, we presented Algorithm 3 and Algorithm 8 as two separate algorithms; however, given their similar structure, it is clear that both problems can be solved during a single search. All computations stay the same; the only modification needed is to store a tuple in the cache instead of a single count. Hence, we can assume that Schlandals computes both counts simultaneously and returns a tuple $(p_\top, p_\bot)$ such that $p_\top + p_\bot = 1$ at the root. If it times out, then a lower bound is given by $p_\top$ and an upper bound by $1 - p_\bot$.

Schlandals is, to the best of our knowledge, the only model counter computing an upper bound on the weighted model count in this manner. However, computing upper bounds has been explored in other solvers, and in particular in `Toulbar2` [Vir+16]. It allows computing a lower and an upper bound on the true probability in the following manner. During its search, it computes an upper bound on the probability mass of the sub-problem. Then, based on pre-defined approximation criteria, it chooses to explore or not these sub-problems. If a sub-problem is not explored, its probability mass is added to a global counter. At the end of the search, the computed probability serves as a lower bound, as some parts of the search space remain unexplored. The unexplored probability mass can be used to provide an upper bound on the probability.

However, several key differences exist between our algorithm and the one proposed in `Toulbar2`. Our method can be stopped anytime, and it provides an upper bound on the count. On the other hand, `Toulbar2`'s approximation algorithm suffers from a key limitation: if the method times out, it can only compute an upper-bound on *the explored search space*[1]. Moreover, our upper-bound calculation is optimised for the type of model we work with: it relies on the fact that only projected variables have weights, integrates the bound calculation with domain propagation, and can be integrated into any DPLL-style algorithm with little overhead.

5.3.1 From Bounds to ε-approximations

Most existing works do not return bounds on the true WMC; they provide ε-approximation from which bounds can be derived. As a reminder, a value $\hat{p}$ is an ε-approximation of a value p if the following equation holds:

$$\frac{p}{1+\varepsilon} \leq \hat{p} \leq p(1+\varepsilon). \tag{5.7}$$

An ε-approximation give bounds on the true probability; from Equation (5.7) it follows that $p \leq \hat{p}(1+\varepsilon)$ and $p \geq \frac{\hat{p}}{1+\varepsilon}$. We prove that the relation can also be in the other direction: it is possible to derive an ε-approximation from bounds on p. To do so, let us first show in which condition it is possible to derive an ε-approximation for a pre-defined ε.

Theorem 7. *Let F be a boolean formula, p its weighted model count, and $\varepsilon \geq 0$ an error factor. Moreover, let $p_l \leq p$ be a lower bound on p and $p_u \geq p$ an upper bound on p. If $p_u \leq p_l(1+\varepsilon)^2$, then the following inequalities hold.*

$$\frac{p}{1+\varepsilon} \leq \sqrt{p_l \times p_u} \leq p(1+\varepsilon)$$

Proof. Let us first prove the left part of the inequality. By definition, we have that $p \leq p_u$ and, by assumption, $\frac{p_u}{(1+\varepsilon)^2} \leq p_l$. Hence, the following relations hold.

$$p^2 \leq p_u^2 \Leftrightarrow \frac{p^2}{(1+\varepsilon)^2} \leq \frac{p_u^2}{(1+\varepsilon)^2} = \frac{p_u p_u}{(1+\varepsilon)^2} \leq p_l \times p_u$$
$$\Leftrightarrow \sqrt{\frac{p^2}{(1+\varepsilon)^2}} = \frac{p}{1+\varepsilon} \leq \sqrt{p_l \times p_u}$$

[1] After discussing this issue with the developers of `Toulbar2`, they changed this behaviour. Hence, recent implementations of `Toulbar2` compute valid bounds even when the solver times out.

We prove the second inequality similarly.

$$p \geq p_l \Leftrightarrow p^2(1+\varepsilon)^2 \geq p_l^2(1+\varepsilon)^2 = p_l p_l(1+\varepsilon)^2 \geq p_l p_u$$
$$\Leftrightarrow \sqrt{p^2(1+\varepsilon)^2} = p(1+\varepsilon) \geq \sqrt{p_l \times p_u}$$

□

Theorem 7 can be seen in two ways, leading to different algorithmic perspectives. First, if the lower- and upper-bound are close enough, with respect to a pre-defined ε error factor, the search can be stopped before a timeout. On the other hand, if the search times out, it is still possible to compute a *minimal ε required to provide an ε-approximation*. Indeed, from the condition on the bounds in Theorem 7, given a lower bound p_l and an upper bound p_u, the minimum required ε is given by the following equation.

$$\varepsilon_{min} = \sqrt{\frac{p_u}{p_l}} - 1 \tag{5.8}$$

From an algorithmic perspective, if an ε is provided, it is possible to stop the search when the bounds are close enough. The idea is to approximate parts of the search tree so that an ε-approximation can be computed at the root. Such an algorithm is not anytime; the goal is to compute an ε-approximation, and it returns a solution only when the approximation is computed or it times out. On the other hand, it is also possible to return an approximation error and let the user decide if it is good enough. Such an algorithm is, by nature, anytime; it must frequently return solutions to the user. We provide algorithms for both cases in the rest of this chapter.

5.4 Anytime Bounded Weighted Model Counting

Let us first explore the case in which there is no need to provide an ε. In such a case, the goal is to design a method that solves the problem iteratively, tightening the bounds at each iteration. In this work, we propose modifying the search strategy to achieve such results.

5.4.1 Limited Discrepancy Search

We leverage *Limited Discrepancy Search* (LDS) [HG95] as a way of computing successive bounds on the weighted model count. LDS was initially developed for constraint satisfaction problems (CSP) and naturally extended to constraint optimisation problems (COP). It is an incremental search; it explores a progressively larger part of the search space, trying to find good solutions first. Using such an approach is natural for CSPs and COPs; if the

algorithm finds a good solution early in the search, it can either return (for CSPs) or prune large parts of the search space (for CSPs). On the other hand, it is not straightforward to see that counting problems can benefit from LDS; indeed, the whole search space must be explored to count the number of solutions. However, Schlandals is well-suited for LDS. To explain that, let us first formalise how LDS works.

Limited Discrepancy Search has been designed for *tree search* algorithms in which the successors of a node are heuristically ordered. That is, given a node (n) of the search tree with k children, there is an order $(c_1), \ldots, (c_k)$ such that the node (c_i) is assumed to be better (i.e., containing a solution or the optimal solution) than node (c_{i+1}). Hence, the heuristic can be viewed as a function h such that $h(c_i)$ returns a score for the child (c_i). LDS makes the following two assumptions on the heuristic h.

Assumption 1: The heuristic is correct. It effectively sorts the children according to the solutions they contain. It means that the node (c_1) either contains a solution (for CSPs) or the optimal solution (for COPs).

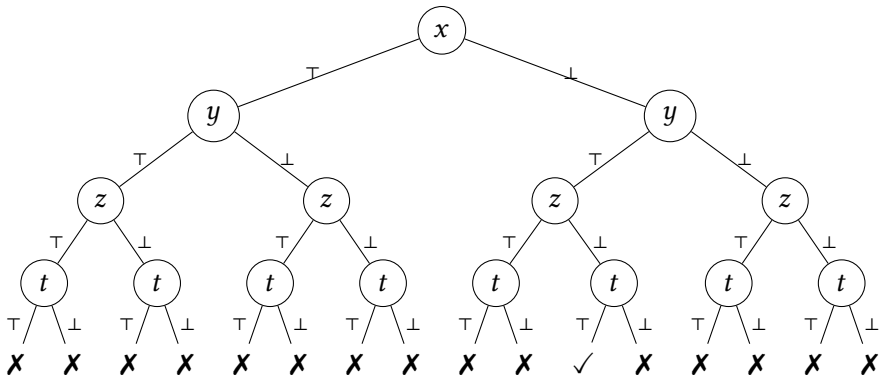
Assumption 2: If the heuristic fails, it is only for a decision point on a branch. For CSPs, it means that if a contradiction is reached (i.e., a constraint is unsatisfied), changing a few decisions on that branch would lead to a solution.

Hence, LDS strongly relies on the quality of the heuristic h: the idea is to follow the heuristic as much as possible while allowing deviations when it fails to provide an optimal solution. The decision points at which the heuristic is not followed (i.e., we branch towards a different child than (c_1)) are called "discrepancies".

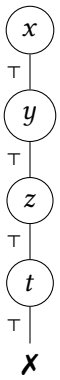
Limited Discrepancy Search is a classical depth-first search in which the number of discrepancies is limited. It iteratively starts a DFS at the root node, increasing the number of discrepancies allowed each time until a solution is found (for CSPs) or the entire search space is explored (for COPs).

Example 27: Limited Discrepancy Search

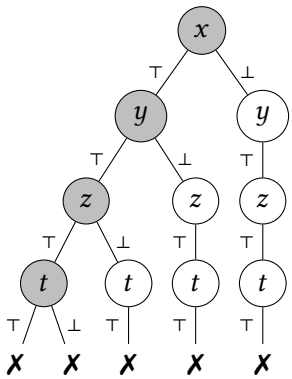
Let us illustrate how limited discrepancy search works on a small CSP with four binary variables, x, y, z, and t containing only one solution, which is $x = \bot, y = \top, z = \bot$, and $t = \top$. The value selection heuristic h always prefers to assign a variable to $\top$ first and then $\bot$. The entire search tree for such a problem is shown below; the non-solution leaves are marked with ✗ while the solution is marked with ✓.



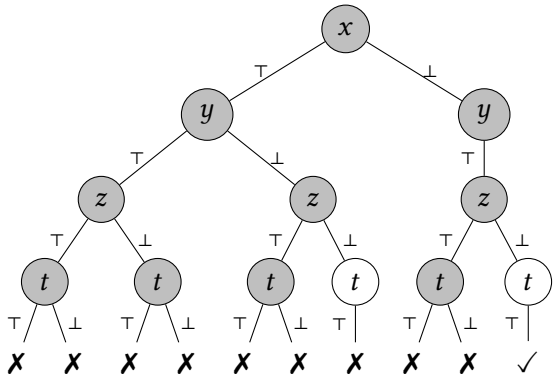
The search space explored after the first iteration of LDS is shown below. During this first iteration, the maximum allowed discrepancy is 0; hence, only the left-most branch of the search space is explored.



The maximum discrepancy is increased to 1 in the second iteration. The nodes previously explored are coloured in grey. Since it can deviate once from the heuristic, this iteration already explores the right part of the search tree.



Finally, a solution is found at the third iteration, and the search can be stopped.



5.4.2 LDS for Weighted Counting Problems

Limited Discrepancy Search assumes that the value selection heuristic leads to an optimal solution. There are no solutions to counting problems; the whole search space must be counted, and it is impossible to reduce the search space. We argue that, in the specific case of *approximate weighted* model counting, and in particular for Schlandals, LDS presents advantages.

Let us assume that LDS is the search strategy for Schlandals. Remember that Schlandals solves, at the same time, the weighted model counting (i.e., computing $p_\top$) problem and its dual (i.e., computing $p_\bot$), allowing to compute bounds at the root node. Let h be a heuristic that orders the values of its distribution. In the context of LDS, h must favour a good solution; hence, for the *weighted model counting*, we assume that h is such that *highly-weighted* interpretations are found first. Launching LDS with such a heuristic enables the quick identification of highly weighted interpretations; hence, the bounds at the root are expected to converge rapidly towards the true count. A key aspect is that h does not need to favour models; finding highly-weighted unsatisfying assignments results in a tighter upper bound.

We designed the most straightforward value heuristic h: given a distribution $\boldsymbol{P}_i = \{P_i^1, \ldots, P_i^m\}$, we set $h(P_i^j) = \omega(P_i^j)\ \forall j$. We order the variables in *decreasing* order of weight. The intuition is that, when branching on a variable P, both counts (i.e., $p_\top^P$ and $p_\bot^P$) are multiplied by $\omega(P)$. Hence, selecting the one with the highest value is the most natural choice.

Algorithm 9 shows the pseudo-code for solving the weighted model counting with LDS. For conciseness, we omit the computations of the $p_\top$ and $p_\bot$ counts.

Given a formula F and a maximum number of discrepancies d, Algorithm 9 follows the same structure as classical DPLL-based model counters.

Algorithm 9: LDS-based weighted model counting

```
1  Function LDS-DPLL(F, P, ω, C, d)
       input  : F a Schlandals formula over distributions P = ∪_{i=1}^{n} P_i
       input  : ω a weight function
       input  : C a cache of sub-results
       input  : d a maximum discrepancy
       output: Possibly partial p_⊤ and p_⊥ count
2      if F ∈ C as (p_⊤, p_⊥, d') and d' ≤ d then return (p_⊤, p_⊥)
3      if P = ∅ then return 1
4      p_⊤ ← 0; p_⊥ ← 0
5      i ← a distribution index such that ∃P ∈ P_i | P is not fixed
6      discrepancy ← 0
7      foreach P ∈ P_i in decreasing order of weight ω(P) do
8          F' ← Propagate(F, P, ⊤)
9          if F' = ⊥ then update p_⊥
10         else
11             update p_⊥ from propagation
12             Components ← all connected components of F'
13             d' ← d − discrepancy
14             foreach Comp ∈ Components do
15                 P' ← P reduced to the variables in Comp
16                 (p_⊤^Comp, p_⊥^Comp) ← LDS-DPLL(Comp, P', ω, C, d')
17             end
18             update p_⊤ and p_⊥ from the p_⊤^Comp and p_⊥^Comp values
19             if discrepancy == d then break;
20             discrepancy ← discrepancy + 1
21         end
22     end
23     C[F] ← (p_⊤, p_⊥, d)
24     return (p_⊤, p_⊥)
```

One of the differences compared to Schlandals' classical search is that when branching on a distribution (line 7), the distribution's variables are explored in *decreasing order* of weight.

When the maximum number of discrepancies is reached (line 19), the algorithm stops the search and returns the (possibly partial) counts. These counts are stored in the cache with the maximum number of discrepancies allowed (line 23). Storing this additional information is necessary; when encountering a formula already explored, the algorithm can not directly return its counts. Indeed, if more discrepancies are allowed (i.e., $d > d'$), then new

parts of the search space can be explored.

Finally, we include an outer loop that uses the LDS-based counting algorithm, shown in Algorithm 10. While the whole search space has not been explored (line 4), the algorithm launch an iteration of LDS (line 5), compute the associate $\varepsilon_{\min}$ (line 6), and outputs it with the bounds (line 7). This simple algorithm continues to run until the entire search space has been explored. However, it is possible to stop it after each iteration of LDS based on user-defined preferences.

Algorithm 10: LDS-based weighted model counting

```
1 Function LDS-Schlandals(F, P, ω)
      input : F, P, ω are the same as Algorithm 9
2     C ← newCache()
3     maxDiscrepancy ← 0, p⊤ ← 0; p⊥ ← 0
4     while p⊤ + p⊥ ! = 1 do
5         (p⊤, p⊥) ← LDS-DPLL(F, P, ω, C, maxDiscrepancy)
6         ε_min ← sqrt((1 − p⊥)/p⊤) − 1
7         output (p⊤, 1 − p⊥, ε_min)
8         maxDiscrepancy ← maxDiscrepancy + 1
9     end
```

5.5 Computing ε-approximation in Schlandals

Let us now consider the case in which the goal is to provide an ε-approximation for a known ε. The intuition behind our method is that it is possible to approximate each sub-problem encountered during the DPLL search in such a way that it provides an ε-approximation at the root. For convenience, let us denote by OR nodes in the search tree those corresponding to branching on a distribution, and by AND nodes those corresponding to a decomposition into independent components.

Theorem 8. *Let F be a (possibly residual) Schlandals formula solved at a node (n) of a Schlandals search tree and $(c_1), \ldots, (c_k)$ be its children. Let $p_\top^{(c)}$ and $p_\bot^{(c)}$ respectively denote the, possibly partial, sum of weighted satisfying and unsatisfying interpretations of the child (c). Moreover, let $\max^{(c)}$ be the weighted sum of all interpretations at node (c). If it holds, for every child (c), that*

$$\begin{cases} \max^{(c)} - p_\bot^{(c)} \leq p_\top^{(c)}(1+\varepsilon)^2 & \text{if } (n) \text{ is a OR node} \\ \max^{(c)} - p_\bot^{(c)} \leq p_\top^{(c)} \sqrt[k]{(1+\varepsilon)^2} & \text{if } (n) \text{ is a AND node} \end{cases}$$

then, the following inequality holds.

$$\max^{(n)} - p_{\bot}^{(n)} \leq p_{\top}^{(n)}(1+\varepsilon)^2$$

Theorem 8 uses Theorem 7 to ensure that an ε-approximation can be computed for the formula F. Indeed, the left-hand side of the inequalities in Theorem 8 correspond to an upper-bound on the (sub-)problems and partial sums for the weighted model count (i.e., $p_{\top}^{(n)}$) gives a lower bound on the true count. Hence, the theorem states that if every child of a node (n) is such that they can provide an ε-approximation, then so do the bounds computed for (n).

Proof. Let us first define $p_{\top}^{(n)}$, $p_{\bot}^{(n)}$, and $\max^{(n)}$, as computed by Schlandals' search procedure.

if (n) is a OR-node:

$$p_{\top}^{(n)} = \sum_{i=1}^{k} p_{\top}^{(c_i)}, \quad p_{\bot}^{(n)} = \sum_{i=1}^{k} p_{\bot}^{(c_i)}, \qquad \max^{(n)} = \sum_{i=1}^{k} \max^{(c_i)}$$

if (n) is a AND-node:

$$p_{\top}^{(n)} = \prod_{i=1}^{k} p_{\top}^{(c_i)}, \quad p_{\bot}^{(n)} = \max^{(n)} - \prod_{i=1}^{k} (\max^{(c_i)} - p_{\bot}^{(c_i)}), \quad \max^{(n)} = \prod_{i=1}^{k} \max^{(c_i)}$$

Note that for ease of notation, we do not include the weights of the variables that are branched on for a OR-node. The computations presented are still valid if we assume that the counts returned by the children incorporate these weights. That is, if the child (c_i) is created by setting variable P_i to true, then we assume that the weights of all interpretations in (c_i) are multiplied by $\omega(P_i)$. Let us now prove the inequalities.

Case 1: (n) is a OR node By hypothesis, we have, $\max^{(c_i)} - p_{\bot}^{(c_i)} \leq p_{\top}^{(c_i)}(1+\varepsilon)^2$ for each child $c_i (1 \leq i \leq k)$. Since both sides of the inequalities are

positive, the following inequalities can be derived.

$$\begin{aligned}
&\sum_{i=1}^{k}\left(\max^{(c_i)} - p_{\bot}^{(c_i)}\right) \leq \sum_{i=1}^{k}\left(p_{\top}^{(c_i)}(1+\varepsilon)^2\right)\\
\Leftrightarrow &\sum_{i=1}^{k}\left(\max^{(c_i)} - p_{\bot}^{(c_i)}\right) \leq (1+\varepsilon)^2\sum_{i=1}^{k} p_{\top}^{(c_i)} = (1+\varepsilon)^2 p_{\top}^{(n)}\\
\Leftrightarrow &\sum_{i=1}^{k}\max^{(c_i)} - \sum_{i=1}^{k} p_{\bot}^{(c_i)} \leq (1+\varepsilon)^2 p_{\top}^{(n)}\\
\Leftrightarrow &\max^{(n)} - p_{\bot}^{(n)} \leq (1+\varepsilon)^2 p_{\top}^{(n)}
\end{aligned}$$

Case 2: (n) is a AND node In this case, we have $\max^{(c_i)} - p_{\bot}^{(c_i)} \leq p_{\top}\sqrt[k]{(1+\varepsilon)^2}$ for all $c_i (1 \leq i \leq k)$. We prove this case similarly to the precedent:

$$\begin{aligned}
&\prod_{i=1}^{k}\left(\max^{(c_i)} - p_{\bot}^{(c_i)}\right) \leq \prod_{i=1}^{k}\left(p_{\top}\sqrt[k]{(1+\varepsilon)^2}\right)\\
\Leftrightarrow &\prod_{i=1}^{k}\left(\max^{(c_i)} - p_{\bot}^{(c_i)}\right) \leq \prod_{i=1}^{k} p_{\top}\prod_{i=1}^{k}\sqrt[k]{(1+\varepsilon)^2} = (1+\varepsilon)^2 p_{\top}^{(n)}\\
\Leftrightarrow &\max^{(n)} - \max^{(n)} + \prod_{i=1}^{k}\left(\max^{(c_i)} - p_{\bot}^{(c_i)}\right) \leq (1+\varepsilon)^2 p_{\top}^{(n)}\\
\Leftrightarrow &\max^{(n)} + (-1)\times\left(\max^{(n)} - \prod_{i=1}^{k}\left(\max^{(c_i)} - p_{\bot}^{(c_i)}\right)\right) \leq (1+\varepsilon)^2 p_{\top}^{(n)}\\
\Leftrightarrow &\max^{(n)} - p_{\bot}^{(n)} \leq (1+\varepsilon)^2 p_{\top}^{(n)}
\end{aligned}$$

□

The particularity of this theorem is that the approximation factor must be tighter when the formula is decomposed into independent components. Indeed, since the counts are multiplied by each other, in that case, they must be tighter to produce an ε-approximation at (n). On the other hand, for a OR node, the weighted sum of the counts is computed; hence, producing an ε-approximation is easier at (n).

Theorem 8 leads to a straightforward approximation algorithm based on Schlandals' search, shown in Algorithm 11. We omit again the computations of the $p_{\top}$ and $p_{\bot}$ counts (lines 9, 11, and 20) for ease of notation, but they are the same as in Algorithm 3 and Algorithm 8.

Algorithm 11: DPLL-style search providing ε-approximation

1 **Function** ε-DPLL$(F, \mathbf{P}, \omega, C)$
  input : F a Schlandals formula over distributions $\mathbf{P} = \cup_{i=1}^{n} \mathbf{P}_i$
  input : ω a weight function
  input : C a cache of sub-results
  input : ε an approximation factor
  output: Partial $p_\top$ and $p_\bot$ such that $\max^F - p_\bot \leq p_\top (1+\varepsilon)^2$
2 **if** $F \in C$ *as* $(p_\top, p_\bot, \varepsilon')$ *with* $\varepsilon' \leq \varepsilon$ **then return** $(p_\top, p_\bot)$
3 **if** $\mathbf{P} = \emptyset$ **then return** 1
4 $p_\top \leftarrow 0; p_\bot \leftarrow 0$
5 $\max^F \leftarrow \prod_{i=1}^{n} \sum_{P \in \mathbf{P}_i} \omega(P)$
6 $i \leftarrow$ a distribution index such that $\exists P \in \mathbf{P}_i \mid P$ is not fixed
7 **foreach** $P \in \mathbf{P}_i$ **do**
8 $F' \leftarrow$ `Propagate`$(F, P, \top)$
9 **if** $F' = \bot$ **then** update $p_\bot$
10 **else**
11 update $p_\bot$ from propagation
12 **if** $\max^F - p_\bot \leq p_\top \times (1+\varepsilon)^2$ **then** break;
13 $Components \leftarrow$ all connected components of F'
14 $k \leftarrow |Components|$
15 $\varepsilon' \leftarrow \sqrt[k]{1+\varepsilon} - 1$
16 **foreach** $Comp \in Components$ **do**
17 $\mathbf{P}' \leftarrow \mathbf{P}$ reduced to the variables in $Comp$
18 $(p_\top^{Comp}, p_\bot^{Comp}) \leftarrow \varepsilon$-DPLL$(Comp, \mathbf{P}', \omega, C, \varepsilon')$
19 **end**
20 update $p_\top$ and $p_\bot$ from the $p_\top^{Comp}$ and $p_\bot^{Comp}$ values
21 **end**
22 **if** $\max^F - p_\bot \leq p_\top \times (1+\varepsilon)^2$ **then** break;
23 **end**
24 $C[F] \leftarrow (p_\top, p_\bot, \varepsilon)$
25 **return** $(p_\top, p_\bot)$

The main difference between Algorithm 11 and the classical search algorithm is that the procedure can return before considering all values for the distribution being branched on. Each time the counts are updated, the algorithm checks if the bounds induced by the counts are close enough (with respect to a given approximation factor) (lines 12,22); if so, the search stops and the partial count are returned.

When encountering independent components, the new ε factor is computed as $\sqrt[k]{1+\varepsilon} - 1$ (line 15), which gives the expected approximation factor

for each component. Indeed, we have that $(1 + \sqrt[k]{1+\varepsilon} - 1)^2 = \sqrt[k]{(1+\varepsilon)^2}$.

Finally, the approximation factor used for a formula must also be stored in the cache, similar to how the discrepancy was stored with the counts. Indeed, a formula can be encountered multiple times with different approximation factors, as it decreases when there are independent components. Hence, the formula must be re-explored if the stored factor (ε') is greater than the required factor (ε).

5.6 Partial Knowledge Compilation

Before analysing the performance of the two algorithms presented above, let us briefly show how our compilation algorithm can, with minimal changes, compute partial circuits. As a reminder, Schlandals' compilation algorithm works by post-processing the cache of intermediate results. When used as a compiler, the necessary information to produce an arithmetic circuit is stored in the cache; that is, pointers to children in the search tree and the propagation result (i.e., variables assigned to $\top$ and unconstrained distributions). Then, the cache can be parsed, following the links to children, to create an arithmetic circuit that computes the weighted model count of the boolean formula.

The first key observation is that a similar procedure can be applied to compute arithmetic circuits that calculate the sum of unsatisfying assignments; the only difference is that variables removed from the distributions must be stored in the cache. Hence, Schlandals can produce, with a single search, two ACs that compute a lower and an upper bound on the weighted model count.

The second key observation is that Schlandals' compilation algorithm still works when the cache is *partially filled* (i.e., the search space has not been fully explored). Partially exploring the search space results in nodes having links to only a subset of their children. It is still possible to build a partial AC from this subset, resulting in ACs that compute a partial sum over the model or unsatisfying assignments. Hence, both approximate algorithms presented above can be used to produce partial ACs.

Combining limited discrepancy search with partial compilation is particularly interesting. Let us assume that the goal is to compute two circuits (one for the lower bound and one for the upper bound) that provide an ε-approximation of the weighted model count. Moreover, the two circuits should be as small as possible in size. In a NeSy context, smaller circuits result in faster training loops, which can significantly reduce the learning time. Hence, this is a bi-objective optimisation problem: finding the pair of circuits whose size is as small as possible while providing an ε-approximation.

We describe how limited discrepancy search can be used to solve such an optimisation problem. The first observation is that adding the discrepancy of each search-tree node in the cache allows compiling circuits representing

the search space up to a given discrepancy. Moreover, it is straightforward to compile the circuit of each bound with a different discrepancy. After each LDS iteration, it is possible to store the following elements: 1. The discrepancy, 2. The bounds, 3. and the sizes of the circuits if they were compiled after this iteration. Note that this last element is easily computed using two counters that are updated when entries are added to the cache.

Then, for two discrepancies d_1 and d_2 (with d_1 not necessarily different than d_2), it is possible to construct an AC computing the lower bound for discrepancy d_1 and an AC computing an upper bound for discrepancy d_2. Such a combination has a certain size and ε-error factor. Let S be the set of such pairs; then, the Pareto frontier of S represents possible solutions to our bi-objective optimisation problem. Deciding to favour either the circuit size or the approximation error is a user choice.

This use case highlights the benefits of our simple compilation algorithm. With minor modifications to the cache structure, which adds negligible overhead to the compilation runtime, it is possible to produce partial circuits by running another search scheme.

5.7 Experimental Evaluation

This section evaluates the two approximation algorithms developed in this chapter. We limit our comparison to `Ganak` and `Toulbar2`, as they are the only solvers that provide methods for computing approximations with relative errors. `Ganak` computes (ε, δ)-approximations using the `ApproxMC` algorithms [CMV16; SM19; SGM] and `Toulbar2` has a dedicated algorithm for computing ε-approximations [Vir+16]. For `Ganak`, we set $\delta = 0.8$ as its performance heavily depends on the choice of δ; when computing (ε, δ)-approximations with a very small δ (e.g., 0.01), it can only solve the easiest instances. Our analysis excludes approximate unweighted model counters. It is known that weighted model counting can be reduced to unweighted model counting by adding variables and clauses for the weights [Cha+15]; however, the overhead of such an approach makes it unusable in practice.

One challenge in evaluating these three solvers is that they efficiently solve different instances. Hence, an instance considered difficult for Schlandals might not be for `Ganak` and vice versa. For this reason, we compare the three solvers in various settings, including different sub-sets of instances. We denote by `Schlandals-eps` the version of Schlandals computing an ε-approximation (Algorithm 11) and by `Schlandals-lds` the LDS-based version of Schlandals (Algorithm 9). When running `Schlandals-lds` to compute an ε-approximation, the algorithm runs with increasingly higher discrepancy until the minimum required approximation factor is small enough (i.e., $\varepsilon_{\min} \leq \varepsilon$) All versions of Schlandals (i.e., the classical search, `Schlandals-eps`,

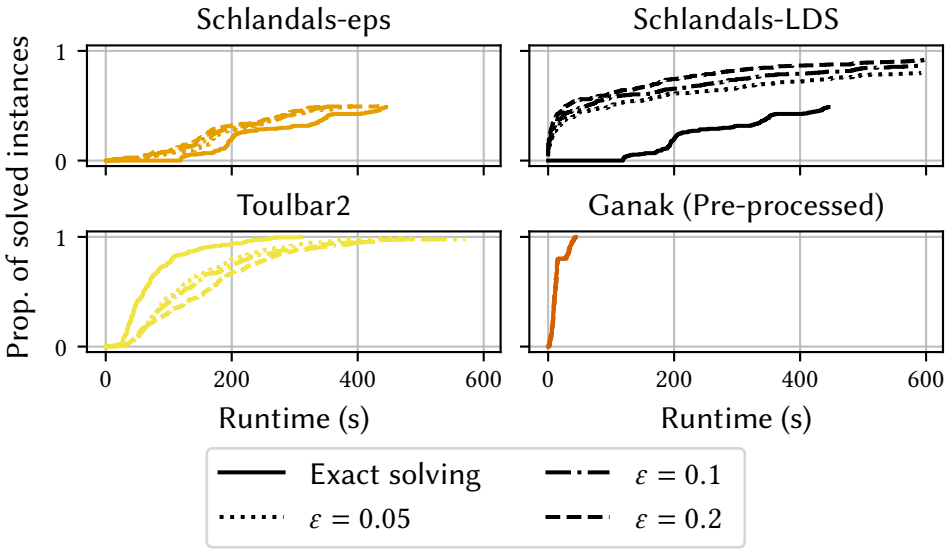


Figure 5.1: Proportion of solved instances over time for various approximation factors for the `munin1` Bayesian network. The solid lines represent each solver's exact solving (i.e., without approximation), while the dotted and dashed lines represent the approximations.

and `Schlandals-lds`) use the same value selection heuristic: when branching on a distribution, the values are ordered in decreasing order of their weight.

We evaluate three aspects of the approximation algorithms:

1. Does computing approximations allow solving more instances?
2. How precise are the computed approximations?
3. How do Schlandals' bounds converge?

Performance improvements from approximation First, let's examine how the approximation affects performance. In order to take benchmarks that are difficult for each solver, we use the `munin1` network, a sub-network of the `munin` network, instead of the bn-learn networks used in the previous chapter. It contains 186 nodes, 273 edges, and more than 15, 000 parameters, making it a challenging instance to solve. The queries are not pre-processed to retain only nodes relevant to them, except for `Ganak`; if no such pre-processing is done, it cannot solve the instances. However, this does not significantly affect the analysis.

Figure 5.1 shows the proportion of instances created from `munin1` solved over time for various ε values. The solid lines represent the exact solving

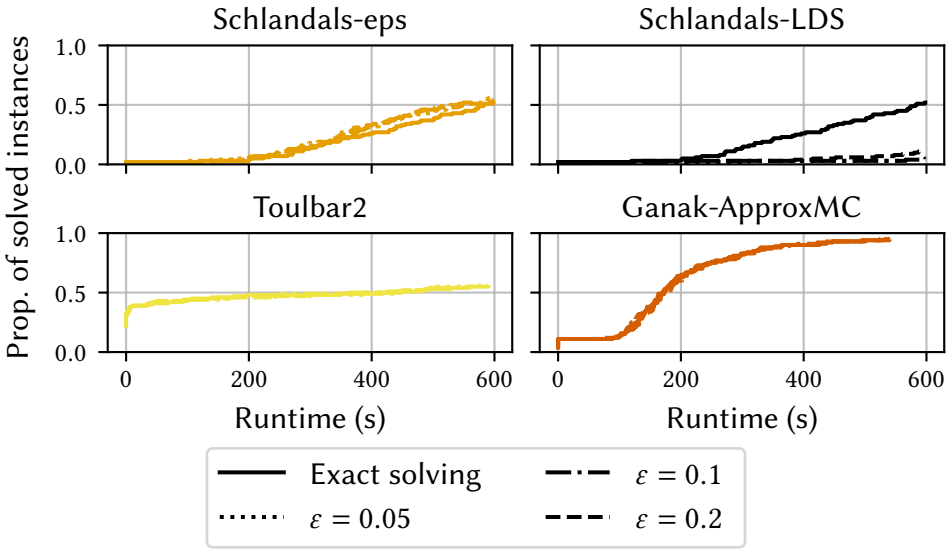


Figure 5.2: Proportion of solved instances over time for various approximation factors for grid Bayesian networks. The solid lines represent each solver's exact solving, without approximation), while the dotted and dashed lines represent the approximations. The grids are of size 10 by 10, with 50% of deterministic nodes for Schlandals; of size 25 by 25, with 70% of determinism; and of size 45 by 45, for Ganak with 70% of determinism.

of the solvers, as evaluated in Chapter 4. It can be seen that Schlandals can solve half of the instances within the given timeout, while `Toulbar2` solves them all within 300 seconds. `Ganak` also solves them all, which is expected since they are pre-processed. However, computing ε-approximations does not affect the time it needs to solve the instances; it solves the instances the same with or without approximations.

Let us now analyse how `Schlandals-eps` and `Schlandals-lds` benefit from approximations. On the one hand, `Schlandals-eps` does not result in more instances being solved, but they are solved faster. On the other hand, `Schlandals-lds` solves many more instances than the classical search. Hence, for this particular data set, `Schlandals-eps` does not have much benefit.

It is possible to explain these two behaviours from the structure of the algorithms. `Schlandals-eps` is still a classical depth-first search over the possible assignments; hence, it can still lose time in parts of the search space with a small probability mass. Additionally, when a formula is decomposed into independent components, each component must be solved to approximate the overall formula. Such problems do not appear when using LDS;

using a limited discrepancy at each iteration allows for the search to quickly diversify, exploring parts of the search space likely to contain interpretations with a high weight. Moreover, when independent components are encountered for the first time, they are explored with a discrepancy of 0, leading to a fast evaluation.

For `Toulbar2`, the behaviour is unexpected as it solves the instances *slower* when approximations are computed. We observed that when computing ε-approximations, the search tree explored by `Toulbar2` is generally larger. It has been observed by Viricel et al. that the upper-bound computation and leaf-ordering can negatively impact performance. One possible explanation for this degradation in performances is that when computing approximations, `Toulbar2`'s search strategy changes, resulting in larger search trees.

A similar analysis can be done on the grid Bayesian networks, as shown in Figure 5.2. In this experiment, we do not use the instances of Chapter 4. Indeed, the three solvers can solve grids of very different sizes. For example, `Ganak` can solve grids of size 40 with 50% of determinism while Schlandals struggles to solve instances of size 12 with 50% determinism. Hence, we created a dataset of 100 queries for each solver so that it cannot solve all of them exactly. For `Ganak`, the grids are of size 45 with 70% determinism; for `Toulbar2`, the grid is of size 25 with 70% determinism; and for Schlandals, the grid is of size 10 with 50% determinism. Although it results in graphs that can not be directly compared, it is still possible to analyse the behaviour of the solvers in these instances. Obviously, such an analysis is biased due to the selection of parameters; it is possible to select parameters that result in approximate methods being more efficient. We have selected these parameters because they yield results that differ from those for `munin1`, thereby highlighting that no algorithm is superior in all cases.

`Ganak` behaves in the same way as for the `munin1` network; it does not benefit from computing approximations. Interestingly, Schlandals and `Toulbar2` behaviour change on the grid networks. While computing ε-approximations was detrimental to `Toulbar2` performance, it is no longer the case. In practice, the search space is often *reduced* in this case, but not significantly. On the other hand, `Schlandals-lds` solves fewer instances than the classical search. In this case, LDS is ineffective due to the structure of the search space. As explained previously, the search space is typically deep when solving grid instances, resulting in interpretations with low weight. Hence, the bounds converge slowly, and the overhead of restarting the search from the root is not compensated. `Schlandals-eps` does not suffer from such a drawback since it applies the same search as Schlandals, there is no overhead, and both methods solve the same number of instances.

Finally, Figure 5.3 shows the proportion of instances solved over time for

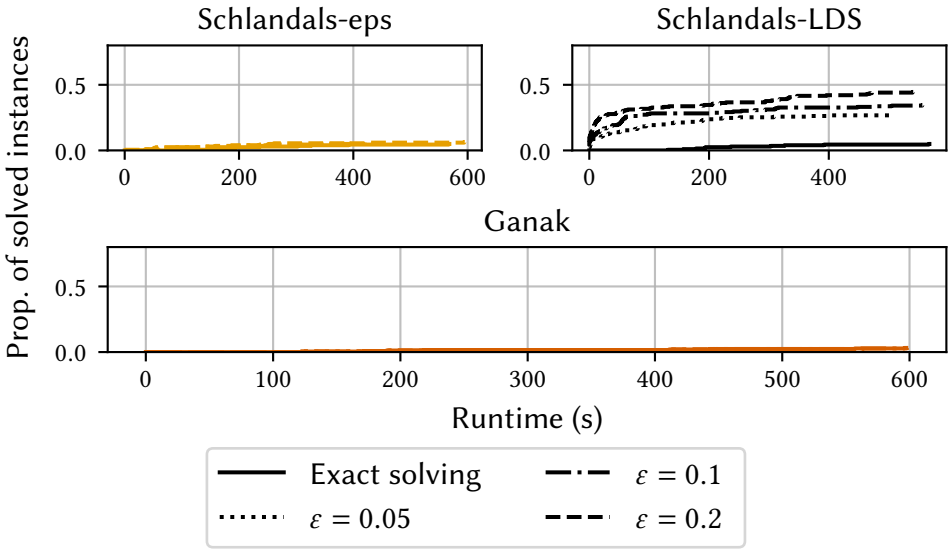


Figure 5.3: Proportion of solved instances over time for various approximation factors for reliability estimation queries. Only queries that take at least one solver more than 120 seconds are considered. The solid lines represent each solver's exact solving (i.e., without approximation), while the dotted and dashed lines represent the approximations.

reliability estimation problems. We used the same datasets; only `Ganak` is compared to our methods. For these problems, it can be seen that `Schlandals-lds` is the only method that truly benefits computing approximations.

How accurate are the approximations? An ε-approximation is an error-bounded approximation but can be relatively close to the true weighted model count in practice. Figure 5.4 shows how well each solver approximates the true weighted model count for Bayesian networks (i.e., *munin1* and grid networks). For each instance for which the true weighted model count p can be computed, we compute the metric $\frac{\hat{p}}{p}$ where $\hat{p}$ is an ε-approximation. A value of 1 means that $p = \hat{p}$, a value lower than 1 means that the count is under-approximated, and a value greater than 1 means that the true count is over-approximated.

The three solvers have very different behaviours. `Ganak` mostly returns the true weighted model count. As it approximates very few instances, it computes the exact weighted model count most of the time. On the other hand, `Toulbar2`, while not solving many more instances, still provides approximations. Its behaviour can explain this: during its search, when it evaluates that a sub-problem does not contain enough probability mass (with re-

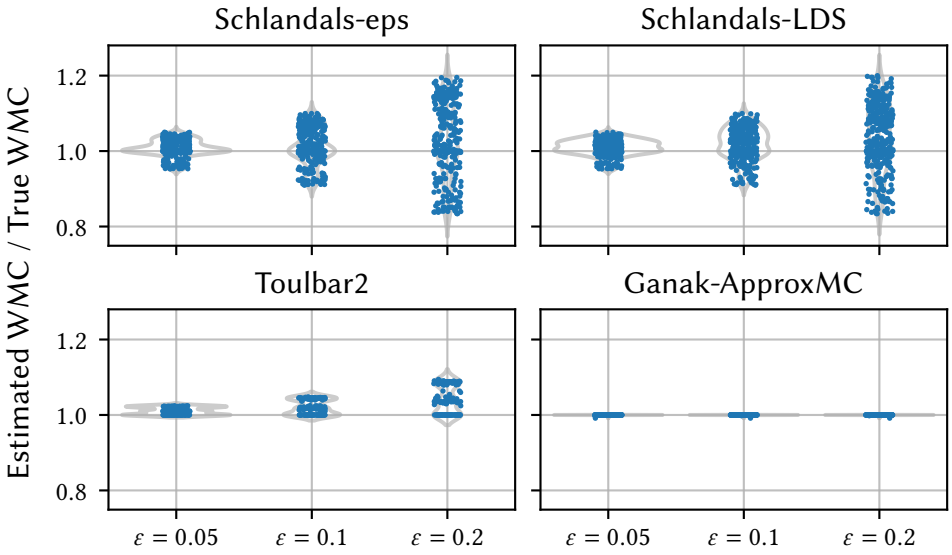


Figure 5.4: Accuracy of the approximations returned by each solver, each data point represents an instance. For Schlandals and `Toulbar2`, the geometric mean of the computed bounds is returned. `Ganak` directly provides its approximations.

spect to the given ε), it does not explore it and adds its total probability mass to an upper bound. Hence, even if an instance can be solved exactly, it is still approximated. `Toulbar2` returns a lower and an upper bound on the true probability; hence, to provide an approximation, we decided to use their geometric mean as taking one of the two bounds would always under- or over-approximate the true probability. We also experimented with the arithmetic mean of the bounds, but the results did not change significantly. Such an approximation mostly over-approximates the probabilities. Interestingly, `Toulbar2` provides probabilities with stronger guarantees than required. For example, when computing approximation with $\varepsilon = 0.2$, all data points are below the 1.1 bar, meaning they are also approximations for $\varepsilon = 0.1$.

Finally, Schlandals has approximations ranging in the full spectrum of possibilities for both approximation algorithms. Both `Schlandals-eps` and `Schlandals-lds` have a similar behaviour: as soon as an approximation can be made, the algorithm stops exploring the (sub-)problem. Hence, the bounds computed at the root are generally less tight than they could be in the given time limit.

Bound convergence in Schlandals Finally, let us analyse how Schlandals' bounds converge towards the true model count. We define two metrics eval-

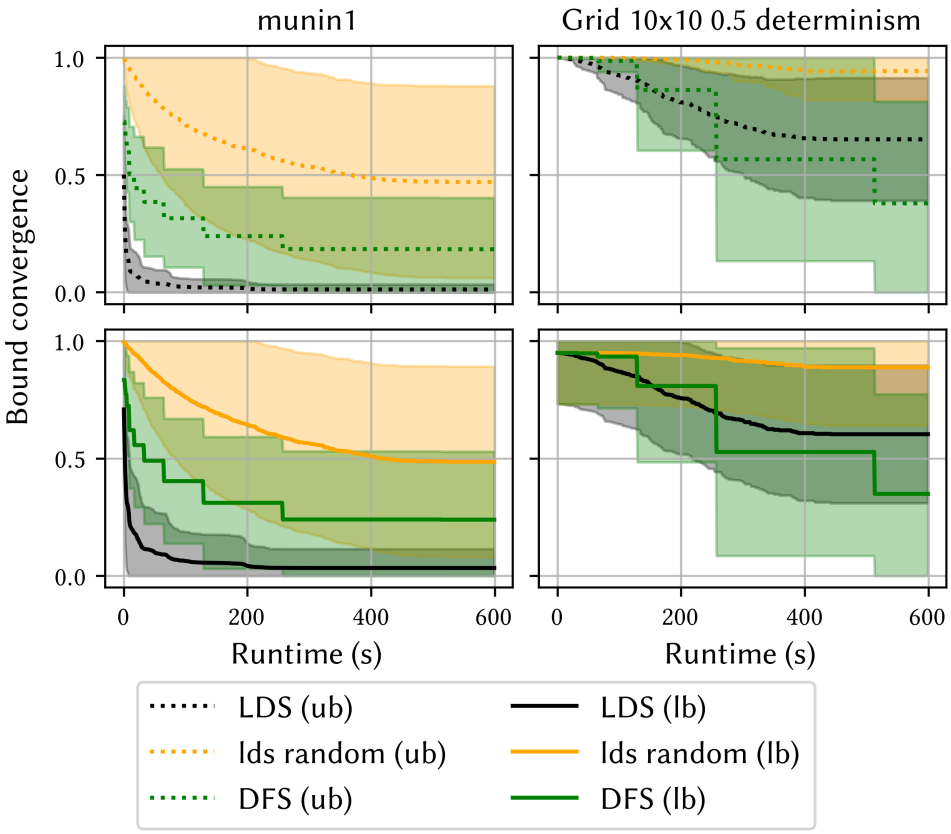


Figure 5.5: Bound convergence of the DFS and LDS search of Schlandals on the *munin1* Bayesian network (left) and the grid networks (right). For each method, the distance of the bounds towards the true count is computed for each time epoch. Unsatisfiable instances (i.e., a weighted model count of 0) are excluded from this analysis. The average distance for each instance is reported.

uating the distance between each bound and the model count. Let p be the true model count, p_l be a lower bound on p, and p_u be an upper bound on p. Then we define the distance between p_l and p as $1 - \frac{p_l}{p}$, and the distance between p_u and p as $1 - \frac{1-p_u}{1-p}$. The intuition behind these metrics is that when no information is available (i.e., $p_l = 0$ and $p_u = 1.0$), they are equal to 1. On the other hand, when the bounds reach the true count, the distance is 0.

Figure 5.5 shows the average distance of the bounds over time on the *munin1* and grid networks used previously for `Schlandals-lds` (black) and the classical depth-first search (green). Since the depth-first search is not anytime, it was launched with increasingly higher timeouts, and the bounds at the timeout were recorded. The depth-first search used the same variable ordering as `Schlandals-lds`. Moreover, to evaluate the impact of the variable ordering heuristic, we also ran LDS with the variables ordered randomly (orange). The coloured areas represent the standard deviation from the mean. The two upper graphs show the convergence of the upper bound towards the true probability, and the lower graphs show the convergence of the lower bound.

All methods have, on average, better bounds on the `munin1` dataset than for the grid data set; more instances are solved for this set, resulting in tighter bounds. For this data set, `Schlandals-lds` has the best convergence: on average, the bounds are tighter, faster and with less variability. It can also be observed that the variable ordering is crucial for `Schlandals-lds` efficiency; with a random order on the variables, `Schlandals-lds` performs worse than the classical depth-first search. Moreover, `Schlandals-lds` reaches a plateau very quickly: the lower and upper bounds converge to a small distance of the true probability before stagnating until a time out. In particular, the upper bound reaches an average distance 0.02 ± 0.05 after 100 seconds.

On the other hand, the classical depth-first search has, on average, the best bounds for the grid networks. As explained before, `Schlandals-lds` does not perform well in these instances; hence, the overhead of the incremental search is not compensated by a quick tightening of the bounds. Overall, the grid network exhibits higher variability due to many instances not being well-solved, resulting in poor approximations (i.e., lower bounds close to 0 and upper bounds close to 1). Once again, `Schlandals-lds` with a random order is the worst-performing method, with bounds very far away from the true probability, even after 600 seconds.

5.8 Conclusion

This chapter considered the case of approximate weighted model counting and, in particular, the computation of ε-approximations. First, we showed how the distributions can be leveraged to compute an upper bound on the

count of a (sub-)formula. In addition, Schlandals can, like other weighted model counters, extract a lower bound from a partial evaluation of its search space. It is known that ε-approximation also provides such bounds; this work extends this relationship by showing that an ε-approximation can also be computed from a lower and an upper bound.

Then, we presented two algorithms to perform approximate weighted model counting in Schlandals. The first one, `Schlandals-lds`, is based on a limited discrepancy search: an incremental exploration of the search space to find highly weighted interpretations first. It is an anytime method that can be stopped when an approximation criterion is met, if it times out, or when the search space is fully explored. An interesting property of `Schlandals-lds` is that it does not require a pre-defined ε value to be used. On the other hand, the second approximate algorithm, `Schlandals-eps`, is designed to compute an ε-approximation for a given ε. It approximates each sub-problem so that the probability computed at the root is a valid ε-approximation.

We evaluated `Schlandals-lds` and `Schlandals-eps` against the approximate algorithms developed in `Toulbar2` and `Ganak`. We showed that `Schlandals-lds` is the method allowing approximating the most instances. Our experiments demonstrated that our variable ordering heuristic effectively favours finding highly-weighted interpretations first, resulting in LDS outperforming classical depth-first search.

However, for the grid, Bayesian networks, `Schlandals-lds` does not work. On such networks, Schlandals search space is deep (e.g., a search tree with a depth greater than 100), and the overhead of the incremental search hinders the performance. On the other hand, `Schlandals-eps` does not suffer from such a problem; it always performs slightly better than Schlandals classical search. A similar problem has been observed on `Toulbar2`, for which the gain of using approximation on our benchmarks is limited. However, when `Toulbar2` does produce an approximation, it is often of better quality than the ones produced by `Schlandals-lds` and `Schlandals-eps`. `Ganak` does not seem to benefit from computing approximations: it solves the instances in the same way regardless of the allowed error factor. Moreover, even when computing approximations, the `munin1` bayesian network must be pre-processed for `Ganak` to solve it. Overall, `Schlandals-lds` provides the most gain compared to its corresponding exact method, but it still needs to be improved to handle larger search spaces.

Conclusion 6

This work considered probabilistic inference; more precisely, given a model of a joint probability distribution over random variables, the task studied was to compute the probability of observation of some of them. In the most general case, this problem is #*P*-Complete; that is, it requires counting the number of solutions of a *NP*-Complete problem. This work focuses on model counting algorithms for probabilistic inference. The inference task is first encoded as a boolean formula with a weighting scheme such that the weighted count of its satisfying assignments corresponds to the desired probability. This thesis considers three probabilistic inference tasks: computing the probability of evidence in Bayesian networks, reliability estimation in probabilistic graphs, and probabilistic logic programming in ProbLog.

This work investigates how the model counting task can be specialised for probabilistic inference. We introduced the Schlandals language, a new language based on propositional logic that incorporates elements specific to the probabilistic inference task studied. First, probability distributions are not transformed into clauses; they are first-class citizens. Moreover, a restriction is imposed on the structure of the boolean formula: only Horn clauses are allowed. Finally, Schlandals formulas are specifically designed to be solved using *projected* weighted model counting. We showed in Chapter 3 that all three studied problems can be encoded as a Schlandals formulas.

Then, in Chapter 4, we detailed the algorithms at the core of the Schlandals solver, a solver designed to compute the count of a Schlandals formula. We demonstrated that a straightforward modification of an exhaustive DPLL-style search can be employed. Moreover, we developed a new method that leverages the non-projected variables to simplify the formula. We demonstrate that this algorithm can also be applied within the DPLL-style search to reduce the search space.

Our experiment showed that Schlandals is competitive with state-of-the-art model counters and reasoning systems. On Bayesian networks, Schlandals is even better than other model counters when the networks are not reduced in pre-processing since our new algorithm for simplifying Schlandals formulas performs such reduction naturally. When the networks are reduced, Schlandals still solve most instances, but slower than the other competitors. On the other hand, Schlandals is the best-performing solver on reliability esti-

mation problems. Additionally, we implemented an interface in ProbLog that utilises Schlandals as an inference system; for both Bayesian networks and reliability estimation problems, ProbLog solves more instances when coupled with Schlandals.

Then, we developed algorithms for approximate model counting in Chapter 5. We first demonstrated that thanks to the probability distributions, it is possible to compute the weighted sum of satisfying and unsatisfying assignments of a Schlandals formula simultaneously. An upper bound can then be computed from the sum of unsatisfying assignments if the search times out. Moreover, we prove that a relationship exists between bounds on the count and the well-known ε-approximations.

We leverage this relationship in two new algorithms for approximate weighted model counting in Schlandals. The first one, `Schlandals-eps`, recursively approximates the sub-problems encountered during the depth-first search so that the bounds computed at the root node provide an ε-approximation. The second method, `Schlandals-lds`, is based on another search strategy: limited discrepancy search. It incrementally explores larger parts of the search space and tries to find highly weighted interpretations first. By doing so, `Schlandals-lds` aims to make the bounds at the root converge faster, thereby providing a good approximation more quickly. Our experiments confirmed this behaviour, `Schlandals-lds` is able to make bounds converge faster than `Schlandals-eps`, resulting in faster ε-approximations.

`Schlandals-lds` performed overall the best on the studied benchmark. However, there are certain cases when it decreases the performance of the classical depth-first search when the overhead of the incremental search becomes too high. On the other hand, although `Schlandals-eps` performs less well, it always performs at least as well as Schlandals depth-first search.

6.1 Future Research Directions

Finally, we briefly present some possible future work based on Schlandals. There are two axes developed in this section. First, there is still work that can be done on Schlandals itself to enhance its capability of solving model counting problems. Next, we outline possible research directions for NeSy AI in a more general context.

6.1.1 From Schlandals to Other Model Counters and Vice-versa

One goal for developing Schlandals was to quickly test new ideas for solving the weighted model counting problems. We demonstrated that utilising distributions as first-class citizens enables the computation of an upper bound on the true weighted model count and that limited discrepancy search is an

effective strategy for quickly obtaining good approximations. These features can be integrated into other model counters; for example, LDS can be implemented in search-based model counters such as GPMC or Ganak. Integrating distributions as first-class citizens into other model counters is more challenging as it changes the basic unit of the boolean formulas (i.e., distributions instead of variables). One challenge is that most model counters are generic and solve multiple variants of the counting problem, such as unweighted and weighted model counting. Probability distributions are not helpful for unweighted model counting, making their integration into standard model counters difficult.

On the other hand, many techniques in state-of-the-art model counters could be implemented in Schlandals. For example, further research is needed to explore how branching heuristics can be generalised to select distributions rather than variables. Similarly, various preprocessing techniques have been proposed in the literature (e.g., vivification [PHS08], gate detection [BW04; EB05; Ost+02], backbone detection [Mon+99]) and are beneficial for model counting [LM17b] but are not yet implemented in Schlandals.

6.1.2 Solving More Inference Tasks

The inference tasks considered in this work fall into the sum-product problems category, i.e., summing the weights of possible worlds. However, many more inference tasks can be defined on the probabilistic models considered. For example, a typical query on a Bayesian network $(\boldsymbol{V}, \Phi)$ is to find the most likely assignments to a set of node $\boldsymbol{M} \subseteq \boldsymbol{V}$, given a set of observed variables $\boldsymbol{E} = \boldsymbol{V} \setminus \boldsymbol{M}$. This is the *Maximum A Posteriori* (MAP) problem, which is known to be *NP*-Hard. Another popular example is the *Marginal MAP* (MMAP) problem, which combines the MAP problem with the counting problem presented in this work. Contrary to the MAP problem, the sets $\boldsymbol{M}$ and $\boldsymbol{E}$ are not a partition of $\boldsymbol{V}$; hence, for each possible assignment of the variables in $\boldsymbol{M}$, computing its probability requires solving a #*P*-Hard problem. These two problems can also be solved by reasoning over a boolean formula encoding the Bayesian network. An interesting line of work would be to analyse how Schlandals can be adapted to solve such problems.

Moreover, additional probabilistic models can also be considered. For example, factor graphs, a generalisation of Bayesian networks, are a natural extension of our work. It is known that classical encoding algorithms such as ENC4 can be used to encode factor graphs. However, Schlandals has a slightly different structure for its clauses (i.e., the parameter variables are on the left-hand side of the implications), and encoding a factor graph into a Schlandals formula is not as direct as with state-of-the-art encodings.

Finally, removing the Horn constraint on the clauses can increase the

number of problems that can be modelled in Schlandals. Although Schlandals performance relies heavily on its ability to simplify formulas during propagation, many inference tasks are modelled with non-Horn clauses. For example, the ROAD-R challenge is a popular NeSy benchmark for autonomous driving, where the goal is to detect road events in videos [Sin+22; Giu+23]. This benchmark defines a set of logical constraints that contains distribution constraints, Horn clauses, and a few non-Horn clauses. Schlandals can not be applied to this challenge because of these few non-Horn clauses, but the benefits of Schlandals (e.g., its approximate LDS-based algorithm) extend beyond the Horn-structure of its formulas. The impact of allowing non-Horn clauses in a Schlandals formula can be evaluated in future work. In particular, we demonstrated that simplifying the formula using non-projected variables is crucial for Schlandals' performance; however, this algorithm relies on the Horn structure of the formula. Hence, new algorithms must be developed to apply this simplification to general boolean formulas.

6.1.3 Counting Constraint Satisfaction Problems for NeSy Systems

One of the most common ways of encoding logical constraints in the NeSy literature is propositional logic. The literature contains a variety of knowledge compilers that can transform a boolean formula into a differentiable form (e.g., NNF diagrams, arithmetic circuits), allowing NeSy systems to be learned end-to-end with automatic differentiation tools. Schlandals' compilation algorithm is based on post-processing the cache, a depth-first search in which propagation is applied after each decision.

From a compilation point of view, only the propagation result (i.e., which variables have been assigned) is necessary. Hence, constraints can be encoded in other language than CNF formulas. In particular, the Constraint Programming (CP) paradigm offers a variety of global constraints with dedicated, powerful filtering algorithms (i.e., algorithms that remove values from variables based on assignments to other variables).

For example, let us consider a NeSy Sudoku verifier. When encoded in propositional logic, the Sudoku constraints are transformed into pairwise difference constraints (i.e., $\neg X \vee \neg Y$). In the CP literature, the `allDifferent` constraint encodes Sudoku constraints compactly, stating that a set of n variables must take distinct values. In addition to encoding the set of constraints compactly, there are filtering algorithms for the `allDifferent` constraint that are stronger than boolean unit propagation. Let us consider four variables, on which the `allDifferent` constraint is applied, with the following domains: $X_1 \in \{1, 2, 3, 4\}, X_2 \in \{2, 3\}, X_3 \in \{1, 2, 4\}, X_4 \in \{2, 3\}$. Any assignment on these variables must set X_2 and X_4 to either 2 or 3; hence, X_1 and X_3 can not take these values and can be removed from their domain.

While an algorithm exists to find such filtering on the `allDifferent` constraint [Rég94], such propagation is impossible when decomposed into binary constraints.

This small example highlights the reasoning power of global constraints in CP. Moreover, numerous research studies in the constraint community are currently focused on compiling constraints as decision diagrams [Ber+16]. Further research could investigate the benefits of incorporating global constraints within NeSy systems and how current state-of-the-art algorithms for decision diagrams can be adapted. Moreover, this would extend the applicability of NeSy systems to more real-world applications that require the modelling of complex and high-level constraints.

6.1.4 Learning NeSy Systems with Approximate Counting

Finally, as highlighted in the introduction to this manuscript, exact probabilistic inference is the primary computational bottleneck of NeSy systems. Approximate methods can reduce the runtime of the inference task, but they do so at the cost of not representing all solutions to the logic constraints. These methods come with various guarantees, and an open question currently being investigated is whether the learning phase of a NeSy system benefits from such guarantees [DDN24; Kri+23; MMD21]. In other words, when representing the set of models of a boolean formula in a partial arithmetic circuit, is it necessary to have strong guarantees (e.g., ε-approximations) on the probability computed by the circuit?

Moreover, the weights are continually updated during the learning, but the circuits are constructed from the original weights. Hence, a partial circuit with, for example, ε-approximation at the beginning of the training might not maintain this guarantee during the training. This also suggests that there are algorithmic questions about how to integrate partial compilation into the learning pipeline, allowing for the best representation of the sets of models of the boolean formula while minimising excessive overhead by avoiding repeated recompilation.

Using our LDS-based approximation, we began exploring such questions, particularly from the symbolic aspect of NeSy systems. For example, we showed that there is a trade-off between the accuracy of the approximations and the size of the ACs. Further research is needed to explore the compromise between these two aspects for NeSy systems.

Bibliography

[Abr+96] B. Abramson, J. Brown, W. Edwards, A. Murphy, and R. L. Winkler. "Hailfinder: A Bayesian System for Forecasting Severe Weather". In: *International Journal of Forecasting* 12.1 (1996). (Visited on 03/28/2025).

[AMD08] S. C. Amstrup, B. G. Marcot, and D. C. Douglas. "A Bayesian Network Modeling Approach to Forecasting the 21st Century Worldwide Status of Polar Bears". In: *Arctic sea ice decline: observations, projections, mechanisms, and implications* 180 (2008). (Visited on 03/28/2025).

[And+91] S. Andreassen, R. Hovorka, J. Benn, K. G. Olesen, and E. R. Carson. "A Model-Based Approach to Insulin Adjustment". In: *AIME 91*. Ed. by O. Rienhoff, D. A. B. Lindberg, M. Stefanelli, A. Hasman, M. Fieschi, and J. Talmon. Vol. 44. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991. ISBN: 978-3-540-54144-8 978-3-642-48650-0. DOI: 10.1007/978-3-642-48650-0_19. (Visited on 03/28/2025).

[Aug+22] E. Augustine, C. Pryor, C. Dickens, J. Pujara, W. Wang, and L. Getoor. "Visual Sudoku Puzzle Classification: A Suite of Collective Neuro-Symbolic Tasks". In: *International Workshop on Neural-Symbolic Learning and Reasoning (NeSy)*. 2022. (Visited on 10/16/2024).

[Azi+15] R. A. Aziz, G. Chu, C. Muise, and P. Stuckey. "#∃ SAT: Projected Model Counting". In: *International Conference on Theory and Applications of Satisfiability Testing 2015*. Springer, 2015.

[Bar+14] A. Bart, F. Koriche, J.-M. Lagniez, and P. Marquis. "Symmetry-driven decision diagrams for knowledge compilation". In: *ECAI 2014*. IOS Press, 2014, pp. 51–56.

[Bar+16] A. Bart, F. Koriche, J.-M. Lagniez, and P. Marquis. "An Improved CNF Encoding Scheme for Probabilistic Inference". In: *Proceedings of the Twenty-second European Conference on Artificial Intelligence*. 2016.

[Bar82] J. Barwise. *Handbook of Mathematical Logic*. Vol. 90. Elsevier, 1982. (Visited on 03/07/2025).

[BDP03] F. Bacchus, S. Dalmao, and T. Pitassi. "Algorithms and Complexity Results For# SAT and Bayesian Inference". In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 2003. (Visited on 03/07/2025).

[Bea+03] P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. "Memoization and DPLL: Formula Caching Proof Systems". In: *18th IEEE Annual Conference on Computational Complexity, 2003. Proceedings.* IEEE, 2003. (Visited on 03/07/2025).

[Ber+16] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. *Decision diagrams for optimization.* Vol. 1. Springer, 2016.

[BP00] R. J. Bayardo Jr and J. D. Pehoushek. "Counting Models Using Connected Components". In: *AAAI/IAAI.* 2000. (Visited on 03/07/2025).

[Bry86] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986). (Visited on 03/07/2025).

[BW04] F. Bacchus and J. Winter. "Effective Preprocessing with Hyper-Resolution and Equality Reduction". In: *Theory and Applications of Satisfiability Testing.* Ed. by G. Goos, J. Hartmanis, J. Van Leeuwen, E. Giunchiglia, and A. Tacchella. Vol. 2919. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN: 978-3-540-20851-8 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_26. (Visited on 03/27/2025).

[CD05] M. Chavira and A. Darwiche. "Compiling Bayesian Networks with Local Structure". In: *IJCAI.* Vol. 5. 2005.

[CD06] M. Chavira and A. Darwiche. "Encoding CNFs to Empower Component Analysis". In: *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9.* Springer, 2006.

[CD08] M. Chavira and A. Darwiche. "On Probabilistic Inference by Weighted Model Counting". In: *Artificial Intelligence* 172.6-7 (2008).

[CD13] A. Choi and A. Darwiche. "Dynamic Minimization of Sentential Decision Diagrams". In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 27. 2013. (Visited on 03/09/2025).

[Cha+14] S. Chakraborty, D. Fremont, K. Meel, S. Seshia, and M. Vardi. "Distribution-Aware Sampling and Weighted Model Counting for SAT". In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 28. 2014. (Visited on 04/16/2024).

[Cha+15] S. Chakraborty, D. Fried, K. S. Meel, and M. Y. Vardi. "From Weighted to Unweighted Model Counting." In: *IJCAI.* 2015. (Visited on 02/23/2024).

[CMV16] S. Chakraborty, K. S. Meel, and M. Y. Vardi. *Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls*. Tech. rep. 2016.

[Coo23] S. A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*. 2023. (Visited on 10/14/2024).

[Dar01a] A. Darwiche. "Decomposable Negation Normal Form". In: *Journal of the ACM* 48.4 (July 2001). ISSN: 0004-5411, 1557-735X. DOI: 10.1145/502090.502091. (Visited on 03/07/2025).

[Dar01b] A. Darwiche. "On the Tractable Counting of Theory Models and Its Application to Truth Maintenance and Belief Revision". In: *Journal of Applied Non-Classical Logics* 11.1-2 (Jan. 2001). ISSN: 1166-3081, 1958-5780. DOI: 10.3166/jancl.11.11-34. (Visited on 03/07/2025).

[Dar02] A. Darwiche. "A Logical Approach to Factoring Belief Networks". In: *KR* 2 (2002).

[Dar11] A. Darwiche. "SDD: A New Canonical Representation of Propositional Knowledge Bases". In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011.

[DDN24] L. Dierckx, A. Dubray, and S. Nijssen. "Parameter Learning Using Approximate Model Counting". In: *International Conference on Neural-Symbolic Learning and Reasoning*. Springer. 2024, pp. 80–88.

[DG84] W. F. Dowling and J. H. Gallier. "Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae". In: *The Journal of Logic Programming* 1.3 (1984).

[DKT07] L. De Raedt, A. Kimmig, and H. Toivonen. "Problog: A Probabilistic Prolog and Its Application in Link Discovery." In: *IJCAI*. Vol. 7. Hyderabad, 2007.

[DM02] A. Darwiche and P. Marquis. "A Knowledge Compilation Map". In: *Journal of Artificial Intelligence Research* 17 (2002).

[DP60] M. Davis and H. Putnam. "A Computing Procedure for Quantification Theory". In: *Journal of the ACM* 7.3 (July 1960). ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321033.321034. (Visited on 03/28/2025).

[DPV20] J. Dudek, V. Phan, and M. Vardi. "ADDMC: weighted model counting with algebraic decision diagrams". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 02. 2020, pp. 1468–1476.

[DSN23] A. Dubray, P. Schaus, and S. Nijssen. "Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses". In: *LIPIcs, Volume 280, CP 2023* 280 (2023). Ed. by R. H. C. Yap. ISSN: 1868-8969. DOI: 10.4230/LIPICS.CP.2023.15. (Visited on 03/07/2025).

[DSN24] A. Dubray, P. Schaus, and S. Nijssen. "Anytime Weighted Model Counting with Approximation Guarantees for Probabilistic Inference". In: *LIPIcs, Volume 307, CP 2024* 307 (2024). Ed. by P. Shaw. ISSN: 1868-8969. DOI: 10.4230/LIPICS.CP.2024.10. (Visited on 03/07/2025).

[Dub+20] A. Dubray, G. Derval, S. Nijssen, and P. Schaus. "Mining Constrained Regions of Interest: An Optimization Approach". In: *Discovery Science*. Ed. by A. Appice, G. Tsoumakas, Y. Manolopoulos, and S. Matwin. Vol. 12323. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-61526-0 978-3-030-61527-7. DOI: 10.1007/978-3-030-61527-7_41. (Visited on 03/07/2025).

[Dub+21] A. Dubray, S. Nijssen, I. Thomas, and P. Schaus. "A Seriation Based Framework to Visualize Multiple Aspects of Road Transport from GPS Trajectories". In: *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021. (Visited on 03/07/2025).

[Dub+22] A. Dubray, G. Derval, S. Nijssen, and P. Schaus. "Optimal Decoding of Hidden Markov Models with Consistency Constraints". In: *Discovery Science*. Ed. by P. Pascal and D. Ienco. Vol. 13601. Cham: Springer Nature Switzerland, 2022. ISBN: 978-3-031-18839-8 978-3-031-18840-4. DOI: 10.1007/978-3-031-18840-4_29. (Visited on 03/07/2025).

[Due+17] L. Duenas-Osorio, K. Meel, R. Paredes, and M. Vardi. "Counting-Based Reliability Estimation for Power-Transmission Grids". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 2017.

[EB05] N. Eén and A. Biere. "Effective Preprocessing in SAT Through Variable and Clause Elimination". In: *Theory and Applications of Satisfiability Testing*. Ed. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, F. Bacchus, and T. Walsh. Vol. 3569. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-26276-3 978-3-540-31679-4. DOI: 10.1007/11499107_5. (Visited on 03/27/2025).

[Fie+15] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. "Inference and Learning in Probabilistic Logic Programs Using Weighted Boolean Formulas". In: *Theory and Practice of Logic Programming* 15.3 (2015).

[For+16] J. K. Fortin, K. D. Rode, G. V. Hilderbrand, J. Wilder, S. Farley, C. Jorgensen, and B. G. Marcot. "Impacts of Human Recreation on Brown Bears (Ursus Arctos): A Review and New Management Tool". In: *PLoS One* 11.1 (2016). (Visited on 03/28/2025).

[GD11] V. Gogate and R. Dechter. "SampleSearch: Importance Sampling in Presence of Determinism". In: *Artificial Intelligence* 175.2 (2011). (Visited on 04/17/2024).

[Giu+23] E. Giunchiglia, M. C. Stoian, S. Khan, F. Cuzzolin, and T. Lukasiewicz. "ROAD-R: The Autonomous Driving Dataset with Logical Requirements". In: *Machine Learning* 112.9 (Sept. 2023). ISSN: 0885-6125, 1573-0565. DOI: 10.1007/s10994-023-06322-z. (Visited on 03/28/2025).

[Gom+07] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. "From Sampling to Model Counting." In: *IJCAI*. Vol. 2007. 2007. (Visited on 01/29/2024).

[HD07] J. Huang and A. Darwiche. "The Language of Search". In: *Journal of Artificial Intelligence Research* 29 (2007).

[HG95] W. D. Harvey and M. L. Ginsberg. "Limited Discrepancy Search". In: *IJCAI (1)*. 1995. (Visited on 03/22/2024).

[Jan04] T. Janhunen. "Representing Normal Programs with Clauses". In: *ECAI*. Vol. 16. 2004. (Visited on 03/17/2025).

[JK99] C. S. Jensen and A. Kong. "Blocking Gibbs Sampling for Linkage Analysis in Large Pedigrees with Many Loops". In: *The American Journal of Human Genetics* 65.3 (1999). (Visited on 03/28/2025).

[Kis+14] D. Kisa, G. Van den Broeck, A. Choi, and A. Darwiche. "Probabilistic Sentential Decision Diagrams". In: *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*. 2014.

[KJ21] T. Korhonen and M. Järvisalo. "Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters". In: *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[KM04] S. Khurshid and D. Marinov. "TestEra: Specification-Based Testing of Java Programs Using SAT". In: *Automated Software Engineering* 11.4 (Oct. 2004). ISSN: 0928-8910. DOI: 10.1023/B:AUSE.0000038938.10589.b9. (Visited on 03/28/2025).

[Kor+13] F. Koriche, J.-M. Lagniez, P. Marquis, and S. Thomas. "Knowledge Compilation for Model Counting: Affine Decision Trees." In: *IJCAI*. 2013, pp. 947–953.

[Kri+23] E. van Krieken, T. Thanapalasingam, J. Tomczak, F. Van Harmelen, and A. Ten Teije. "A-nesi: A scalable approximate method for probabilistic neurosymbolic inference". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 24586–24609.

[KS92] H. A. Kautz and B. Selman. "Planning as Satisfiability." In: *ECAI*. Vol. 92. Citeseer, 1992. (Visited on 03/28/2025).

[Llo12] J. W. Lloyd. *Foundations of Logic Programming*. Springer Science & Business Media, 2012. (Visited on 03/17/2025).

[LM06] I. Lynce and J. Marques-Silva. "Efficient Haplotype Inference with Boolean Satisfiability". In: *National Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2006. (Visited on 03/28/2025).

[LM17a] J.-M. Lagniez and P. Marquis. "An Improved Decision-DNNF Compiler." In: *IJCAI*. Vol. 17. 2017.

[LM17b] J.-M. Lagniez and P. Marquis. "On Preprocessing Techniques and Their Impact on Propositional Model Counting". In: *Journal of Automated Reasoning* 58.4 (2017). (Visited on 03/27/2025).

[LM19] J.-M. Lagniez and P. Marquis. "A Recursive Algorithm for Projected Model Counting". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019.

[LM20] J.-M. Lagniez and P. Marquis. "Enhanced Caching For# SAT Solving". In: (2020). (Visited on 03/27/2025).

[LMY21] Y. Lai, K. S. Meel, and R. H. Yap. "The Power of Literal Equivalence in Model Counting". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 2021.

[LMY22] Y. Lai, K. S. Meel, and R. H. Yap. "Fast Converging Anytime Model Counting". In: *arXiv preprint arXiv:2212.09390* (2022). arXiv: 2212.09390.

[LS88] S. L. Lauritzen and D. J. Spiegelhalter. "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems". In: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 50.2 (Jan. 1988). ISSN: 1369-7412, 1467-9868. DOI: 10.1111/j.2517-6161.1988.tb01721.x. (Visited on 10/04/2024).

[Mad+01] C. F. Madigan, S. Malik, M. W. Moskewicz, L. Zhang, and Y. Zhao. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of DAC*. 2001.

[Mar08] J. Marques-Silva. "Practical Applications of Boolean Satisfiability". In: *2008 9th International Workshop on Discrete Event Systems*. IEEE, 2008. (Visited on 03/28/2025).

[Mar99] J. Marques-Silva. "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms". In: *Progress in Artificial Intelligence*. Ed. by G. Goos, J. Hartmanis, J. Van Leeuwen, P. Barahona, and J. J. Alferes. Vol. 1695. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. ISBN: 978-3-540-66548-9 978-3-540-48159-1. DOI: 10.1007/3-540-48159-1_5. (Visited on 02/25/2025).

[MB18] S. Möhle and A. Biere. "Dualizing projected model counting". In: *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2018, pp. 702–709.

[Med+17] W. Medjroubi, U. P. Müller, M. Scharf, C. Matke, and D. Kleinhans. "Open Data in Power Grid Modelling: New Approaches towards Transparent Grid Models". In: *Energy Reports* 3 (2017).

[ML98] S. M. Majercik and M. L. Littman. "Using Caching to Solve Larger Probabilistic Planning Problems". In: *AAAI/IAAI*. 1998. (Visited on 03/07/2025).

[MMD21] R. Manhaeve, G. Marra, and L. De Raedt. "Approximate inference for neural probabilistic logic programming". In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*. IJCAI Organization. 2021, pp. 475–486.

[Mon+99] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. "Determining Computational Complexity from Characteristic 'Phase Transitions'". In: *Nature* 400.6740 (1999). (Visited on 03/27/2025).

[Mui+10] C. Muise, S. McIlraith, J. C. Beck, and E. Hsu. "Fast D-DNNF Compilation with sharpSAT". In: *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010. (Visited on 01/20/2025).

[OD15] U. Oztok and A. Darwiche. "A Top-down Compiler for Sentential Decision Diagrams". In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[Oni03] A. Onisko. "Probabilistic Causal Models in Medicine: Application to Diagnosis of Liver Disorders". In: *Ph. D. Dissertation, Inst. Biocybern. Biomed. Eng., Polish Academy Sci., Warsaw, Poland*. 2003.

[Ost+02] R. Ostrowski, É. Grégoire, B. Mazure, and L. Saïs. "Recovering and Exploiting Structural Knowledge from CNF Formulas". In: *Principles and Practice of Constraint Programming - CP 2002*. Ed. by G. Goos, J. Hartmanis, J. Van Leeuwen, and P. Van Hentenryck. Vol. 2470. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-44120-5 978-3-540-46135-7. DOI: 10.1007/3-540-46135-3_13. (Visited on 03/27/2025).

[PD08] K. Pipatsrisawat and A. Darwiche. "New Compilation Languages Based on Structured Decomposability." In: *AAAI*. Vol. 8. 2008. (Visited on 03/07/2025).

[PD10] T. Pipatsrisawat and A. Darwiche. "A Lower Bound on the Size of Decomposable Negation Normal Form". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 24. 2010. (Visited on 03/07/2025).

[PHS08] C. Piette, Y. Hamadi, and L. Sais. "Vivifying Propositional Clausal Formulae". In: *ECAI 2008*. IOS Press, 2008.

[Pra+94] M. Pradhan, G. Provan, B. Middleton, and M. Henrion. "Knowledge Engineering for Large Belief Networks". In: *Uncertainty in Artificial Intelligence*. Elsevier, 1994. (Visited on 03/24/2025).

[Rég94] J.-C. Régin. "A Filtering Algorithm for Constraints of Difference in CSPs". In: *AAAI*. Vol. 94. 1994. (Visited on 03/27/2025).

[San+04] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. "Combining Component Caching and Clause Learning for Effective Model Counting." In: *SAT* 4 (2004).

[SBK05a] T. Sang, P. Beame, and H. Kautz. "Heuristics for Fast Exact Model Counting". In: *Theory and Applications of Satisfiability Testing*. Ed. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, F. Bacchus, and T. Walsh. Vol. 3569. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-26276-3 978-3-540-31679-4. DOI: 10.1007/11499107_17. (Visited on 02/18/2025).

[SBK05b] T. Sang, P. Beame, and H. Kautz. "Solving Bayesian Networks by Weighted Model Counting". In: *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*. Vol. 1. AAAI Press, 2005.

[Scu09] M. Scutari. "Learning Bayesian Networks with the Bnlearn R Package". In: *arXiv preprint arXiv:0908.3817* (2009). arXiv: 0908.3817.

[SdS06] T. Schiex, S. de Givry, and M. Sanchez. "Toulbar2—an Open Source Weighted Constraint Satisfaction Solver". In: *URL http://mulcyber.toulouse. inra. fr/projects/toulbar2* (2006).

[SGM] M. Soos, S. Gocht, and K. S. Meel. "Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling". In: ().

[Sha+19] S. Sharma, S. Roy, M. Soos, and K. S. Meel. "GANAK: A Scalable Probabilistic Exact Model Counter." In: *IJCAI*. Vol. 19. 2019.

[SHS17] R. Suzuki, K. Hashimoto, and M. Sakai. *Improvement of Projected Model-Counting Solver with Component Decomposition Using SAT Solving in Components*. Tech. rep. JSAI Technical Report, SIG-FPAI-506-07, 2017.

[Shw+91] M. A. Shwe, B. Middleton, D. E. Heckerman, M. Henrion, E. J. Horvitz, H. P. Lehmann, and G. F. Cooper. "Probabilistic Diagnosis Using a Reformulation of the INTERNIST-1/QMR Knowledge Base: I. The Probabilistic Model and Inference Algorithms". In: *Methods of Information in Medicine* 30.04 (1991). ISSN: 0026-1270, 2511-705X. DOI: 10.1055/s-0038-1634846. (Visited on 03/24/2025).

[Sin+22] G. Singh, S. Akrigg, M. Di Maio, V. Fontana, R. J. Alitappeh, S. Khan, S. Saha, K. Jeddisaravi, F. Yousefi, and J. Culley. "Road: The Road Event Awareness Dataset for Autonomous Driving". In: *IEEE transactions on pattern analysis and machine intelligence* 45.1 (2022). (Visited on 03/27/2025).

[SM19] M. Soos and K. S. Meel. "BIRD: Engineering an Efficient CNF-XOR SAT Solver and Its Applications to Approximate Model Counting". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019.

[SS96] J. M. Silva and K. A. Sakallah. "GRASP-a New Search Algorithm for Satisfiability". In: *Proceedings of International Conference on Computer Aided Design*. IEEE, 1996. (Visited on 03/07/2025).

[Str17] B. Strasser. *Computing Tree Decompositions with FlowCutter: PACE 2017 Submission*. Sept. 2017. DOI: 10.48550/arXiv.1709.08949. arXiv: 1709.08949 [cs]. (Visited on 03/04/2025).

[Thu06] M. Thurley. "sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP". In: *Theory and Applications of Satisfiability Testing - SAT 2006*. Ed. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, A. Biere, and C. P. Gomes. Vol. 4121. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-37206-6 978-3-540-37207-3. DOI: 10.1007/11814948_38. (Visited on 03/07/2025).

[Val79a] L. G. Valiant. "The complexity of computing the permanent". In: *Theoretical computer science* 8.2 (1979), pp. 189–201.

[Val79b] L. G. Valiant. "The Complexity of Enumeration and Reliability Problems". In: *SIAM Journal on Computing* 8.3 (1979).

[Val84] L. G. Valiant. "A theory of the learnable". In: *Communications of the ACM* 27.11 (1984), pp. 1134–1142.

[Vir+16] C. Viricel, D. Simoncini, S. Barbe, and T. Schiex. "Guaranteed Weighted Counting for Affinity Computation: Beyond Determinism and Structure". In: *Principles and Practice of Constraint Programming*. Ed. by M. Rueher. Vol. 9892. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-44952-4 978-3-319-44953-1. DOI: 10.1007/978-3-319-44953-1_46. (Visited on 04/24/2024).

[Vla+15] J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt. "Anytime Inference in Probabilistic Logic Programs with Tp-compilation". In: *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*. Vol. 2015. IJCAI-INT JOINT CONF ARTIF INTELL, 2015.

[Vla+16] J. Vlasselaer, A. Kimmig, A. Dries, W. Meert, and L. De Raedt. "Knowledge Compilation and Weighted Model Counting for Inference in Probabilistic Logic Programs". In: *Proceedings of the First Workshop on Beyond NP*. AAAI Press, 2016.

[VVB04] J. Vennekens, S. Verbaeten, and M. Bruynooghe. "Logic Programs with Annotated Disjunctions". In: *Logic Programming*. Ed. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, B.

Demoen, and V. Lifschitz. Vol. 3132. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN: 978-3-540-22671-0 978-3-540-27775-0. DOI: 10.1007/978-3-540-27775-0_30. (Visited on 01/27/2025).

[Wie16] B. Wiegmans. *Gridkit: European And North-American Extracts*. Mar. 2016. DOI: 10.5281/ZENODO.47317. (Visited on 04/17/2023).

[WS05] W. Wei and B. Selman. "A New Approach to Model Counting". In: *Theory and Applications of Satisfiability Testing*. Ed. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, F. Bacchus, and T. Walsh. Vol. 3569. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-26276-3 978-3-540-31679-4. DOI: 10.1007/11499107_24. (Visited on 03/09/2025).

[Zha+01] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver". In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*. IEEE, 2001. (Visited on 02/25/2025).

[Zha97] H. Zhang. "SATO: An Efficient Propositional Prover". In: *Automated Deduction—CADE-14*. Ed. by W. McCune. Vol. 1249. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. ISBN: 978-3-540-63104-0 978-3-540-69140-2. DOI: 10.1007/3-540-63104-6_28. (Visited on 03/07/2025).