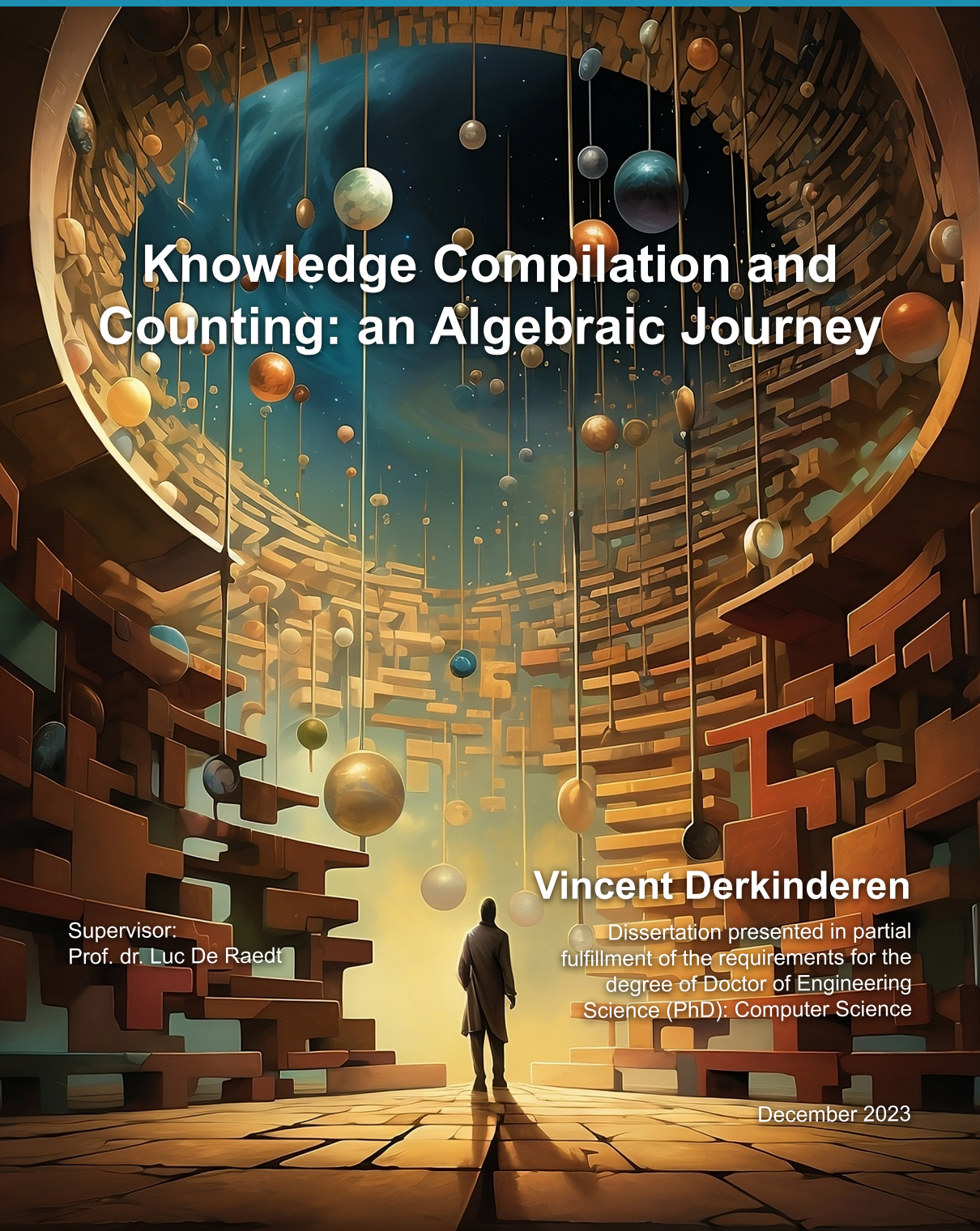
KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science



Knowledge Compilation and Counting: an Algebraic Journey

Vincent Derkinderen

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

Supervisor:
Prof. dr. Luc De Raedt

December 2023

Knowledge Compilation and Counting: an Algebraic Journey

Vincent DERKINDEREN

Examination committee:
Prof. dr. ir. Jean-Pierre Celis, chair
Prof. dr. Luc De Raedt, supervisor
Prof. dr. ir. Tias Guns
dr. ir. Wannes Meert
Prof. dr. ir. Greet Vanden Berghe
Prof. dr. Pierre Marquis
(Université d'Artois)
Prof. dr. Stefano Teso
(Università di Trento)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2023

Cover picture by Thomas Winters and Vincent Derkinderen, resulting from multiple interactions with the generative models of Midjourney (paid license) and of Adobe Firefly, on 28th of August 2023.

Acknowledgements

Like life in general, a PhD journey is the most enjoyable when you have people to share it with. As a token of my gratitude, I would therefore like to acknowledge all those that have helped me complete this journey.

My supervisor Luc De Raedt has not only played a fundamental role throughout my PhD, but also laid the foundations prior. It all began with my master thesis topic that he supervised, *Subgraph search in deep arithmetic graphs*, which allowed me to work with Jonas, Laura, Wannes, Nimish, and Prof. Marian Verhelst. This master thesis initiated my interest in knowledge compilation and model counting, two central topics within this dissertation, and it led me to realise that I liked research! For that, I am very grateful to all those that were involved.

I also appreciate that Luc pushed me to write an FWO project proposal even if he had doubts about its chances. I still remember (and chuckle about) the day he came into my office after we learned that the proposal was accepted! He told me he had not actually anticipated the result and that he now had to find a replacement for the project that I had so far been assigned to. Thanks to the fellowship funding of the Fonds Wetenschapelijk Onderzoek - Vlaanderen (FWO), I had five years to learn a lot from Luc, from writing, presenting, to following a proper research methodology, just to name a few. I am also deeply honored that he asked me to act as a workflow chair for the IJCAI-ECAI2022 conference. The position itself had its ups and downs, but I am glad that I accepted and got to work with the rest of the conference team, contributing to the international research community. Also for the reassuring words leading up to the preliminary defence, Luc, thank you!

I am also grateful for my supervisory and examination committee. For the guidance throughout my PhD journey, for the insightful questions during the (preliminary) defence, and for the constructive feedback that improved this manuscript.

I also wish to thank all my co-authors and (ex-)colleagues who made work more enjoyable and made me be a better researcher. From (random) discussions in the hallway, at Alma, during coffee breaks, to the fun office parties (e.g. the welcome back parties, the Christmas parties, ...), it was really a pleasure working with you all! To those with whom I share(d) an office: Tim, Nitesh, Pietro, Victor, Paolo, Jaron, Robin, Thomas, Yang, and Ying, and Giuseppe back when he was still a visitor in our group, thank you! As many will tell you, a PhD journey is not solely filled with joy; it also encompasses its share of challenges and low points. I therefore thank all of them not only for the interesting research discussions, but also the frustrations we shared and vented (away).

I want to specifically express my appreciation to those who have been following the same track as me these last few months, writing FWO/BOF proposals and the PhD manuscript. Thomas, Pieter, Laurens, Jonas, being able to exchange advice and share frustrations about these processes has been really invaluable to me. Thank you and good luck with your future endeavors!

Most importantly, I wish to thank my friends and family who supported me throughout this process. Notably, my parents, brothers and sister: *voor jullie continue, onvoorwaardelijke liefde en ondersteuning tijdens heel mijn leven; ik kan jullie niet genoeg bedanken!*

Vincent M. Derkinderen
Leuven, Belgium
December 2023

Abstract

The journey captured by this dissertation centers around knowledge compilation and model counting, and their role within state-of-the-art inference algorithms for probabilistic logic programming languages. *Model counting* is the task of computing the number of solutions that satisfy a given set of constraints. For example, the constraint 'A is true or B is false' has three solutions. A particularly important application of this problem is in the domain of probabilistic inference, where each solution may represent a particular scenario that is weighted with a probability. The problem of *weighted model counting* then naturally translates to answering probabilistic queries, making clear the connection with inference in probabilistic logic programming languages.

The weighted model counting problem consists of two operations: addition (counting solutions) and multiplication (computing the probability of a solution). A broader problem that encompasses even more applications is *algebraic model counting* (AMC): it generalizes the two aforementioned operations to those of a semiring, a mathematical construct that guarantees operational properties such as commutativity, associativity and distributivity that can then be exploited for more efficient counting. As we demonstrate throughout this dissertation, a significant insight is that AMC unifies several probabilistic logic programming language variants under a common framework, providing efficient inference algorithms.

Model counting is a hard problem, #P-complete to be exact. Consequently, the chances of discovering a polynomial-time algorithm for model counting are slim. It would prove P = NP, which is widely regarded as unlikely. This issue has also been observed in practice when trying to scale up to larger problems. Nevertheless, many application instances have become solvable in practice due to the development of algorithms that exploit structure present in the problem instances, mirroring the history of SAT solving. From a mathematical point of view, exploiting structure can be regarded as utilising the previously mentioned semiring properties. In this dissertation we contribute an extension

of such counters that expands the exploitable structure by taking advantage of symmetries present within the given constraints. For example, the number of solutions for 'A is true or B is false', remains equivalent when variables A and B are swapped.

Important to efficient model counting is the research domain of *knowledge compilation* which studies, among other things, how to reformulate the set of counting constraints to make counting more tractable. One of our contributions further showcases the generality of algebraic model counting, solved through knowledge compilation, by adapting it to solve a task of decision making under uncertainty. This is non-trivial as it requires three operations instead of two: max, sum, and product.

Finally, we consider counting with respect to a *background theory*, enabling the use of constraints that go beyond Boolean variables that are only true or false. For example, enabling constraints such as $(x + y < 1) \vee (x > 10)$. Knowledge compilation tools for counting in this setting are currently sparse and more often limited to a specific background theory. This motivates our contribution in laying the foundations for knowledge compilation with respect to such background theories.

Beknopte samenvatting

De reis die in deze dissertatie wordt vastgelegd handelt rond kenniscompilatie en het tellen van modellen, en hun rol binnen de state-of-the-art inferentie algoritmen voor probabilistische logische programeertalen. Het *tellen van modellen* is een computationeel probleem bestaande uit het berekenen van het exacte aantal oplossingen die voldoen aan een gegeven verzameling van beperkingen. Bijvoorbeeld, de beperking 'A is waar of B is niet waar' heeft drie oplossingen. Een bijzonder belangrijke toepassing van dit probleem is in het domein van probabilistische inferentie, waar elke oplossing een scenario kan vertegenwoordigen dat wordt gewogen met een bepaalde waarschijnlijkheid. Het *tellen van gewogen modellen* vertaalt zich dan op natuurlijke wijze tot het beantwoorden van probabilistische vragen, wat de connectie met inferentie in probabilistische logische programmeertalen duidelijk maakt.

De gewogen variant van het telprobleem bestaat uit twee operaties: optellen (het tellen van oplossingen) en vermenigvuldigen (de waarschijnlijkheid van een oplossing berekenen). Een breder probleem dat nog meer toepassingen omvat is het *tellen van algebraïsch gewogen modellen* (AMC): het veralgemeent de twee bovengenoemde operaties naar die van een semiring, een wiskundige constructie die operationele eigenschappen garandeert zoals commutativiteit, associativiteit, en distributiviteit, dewelke gebruikt kunnen worden om efficienter te tellen. Zoals we doorheen deze dissertatie demonstreren, is een belangrijk inzicht dat AMC verschillende probabilistische logische programmeertalen verenigt onder een gemeenschappelijk kader, en er efficiënte inferentie algoritmen aan beschikbaar stelt.

Het tellen van modellen is een complex probleem, #P-compleet om precies te zijn. Door dit resultaat is het onwaarschijnlijk dat een algoritme gevonden zal worden met een polynomiale uitvoeringstijd. Dat zou namelijk bewijzen dat P = NP, wat door velen als onwaarschijnlijk wordt beschouwd. Dit resultaat manifesteert zich ook in de praktijk, wanneer we proberen op te schalen naar grotere problemen. Desondanks zijn er reeds veel telproblemen praktisch

oplosbaar geworden door de ontwikkeling van verschillende algoritmen die de structuur van de probleeminstanties uitbuiten om sneller te kunnen tellen. Vanuit een wiskundig perspectief is dit analoog aan het gebruiken van de eerder vermelde semiring eigenschappen. In deze dissertatie stellen we een uitbreiding voor die de uitbuitbare structuur voor de algoritmen vergroot, door gebruik te maken van de symmetrieën die aanwezig zijn binnen de gegeven verzameling van beperkingen. Bijvoorbeeld, het aantal oplossingen voor 'A is waar of B is niet waar' is equivalent wanneer A en B worden omgewisseld.

Belangrijk voor het tellen van modellen is het onderzoeksdomein genaamd *kenniscompilatie*, dat onder meer bestudeert hoe de verzameling beperkingen geherformuleerd kan worden om gemakkelijker te kunnen tellen. Één van de bijdragen in deze dissertatie demonstreert de algemeenheid van het probleem, tellen van algebraïsch gewogen modellen opgelost door kenniscompilatie, door het aan te passen voor een beslissingsprobleem met onzekerheid. Dit is niet triviaal aangezien dergelijk beslissingsprobleem drie operaties omvat in plaats van de gewoonlijke twee: max, som, en product.

We beschouwen ook het probleem van tellen met betrekking tot een *achtergrondtheorie*, die het mogelijk maakt om beperkingen te gebruiken die verder gaan dan Booleaanse variabelen die enkel waar of niet waar zijn, zoals de beperking $(x + y < 1) \vee (x > 10)$. Hulpmiddelen voor het compileren in deze context zijn schaars en vaak beperkt tot een specifieke achtergrondtheorie. Dit motiveert onze bijdrage in het leggen van de fundamenten voor kenniscompilatie met betrekking tot dergelijke achtergrondtheorieën.

List of Abbreviations

$\mathcal{LRA}$ Linear Real Arithmetic. 24, 83–86, 91, 93, 96, 97, 99, 105–107, 110, 143, 144

AI artificial intelligence. 26, 44, 66, 115, 116

AMC Algebraic Model Count. 13, 14, 36, 42, 68, 69, 73, 77–80

CDCL Conflict-driven Clause Learning. 16, 17, 105, 108

CNF Conjunctive Normal Form. 12, 14, 109, 123

d-DNNF deterministic Decomposable Negation Normal Form. 18–21, 24, 82, 86, 104–111, 118

DPLL Davis-Putnam-Logemann-Loveland. 14–16, 48, 103–106, 108–111, 115, 118

MEU Maximum Expected Utility. 69

NNF Negation Normal Form. 19, 20, 109

OBDD Ordered Binary Decision Diagram. 24, 109

PCC Probabilistic Component Caching. 55, 58

PLP Probabilistic Logic Programming. 25–29, 35, 45, 113

PSCC Probabilistic Symmetric Component Caching. 55

SAT Satisfiability problem. 12, 48–50, 59, 64, 105, 108, 111, 121

SDD Sentential Decision Diagram. 22–24, 67, 70, 71, 74, 75, 78, 82, 85, 86, 97–99, 103, 109, 118, 143–145

SMT Satisfiability Modulo Theory. 105, 107, 108, 110

WMC Weighted Model Count. 13, 85

WMI Weighted Model Integration. 83–87, 99, 102, 115, 144

XSDD Extended Sentential Decision Diagram. 85, 99, 143–145

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Knowledge Representation and Reasoning

The term *Artificial intelligence* (AI) was first coined in 1955 in a Darmouth workshop proposal (McCarthy et al., 1955; Russell and Norvig, 2010). The purpose of the workshop, which was organized by John McCarthy, was to find "how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves". Shortly after the workshop, McCarthy proposed the design for an automated reasoning system (McCarthy, 1959) that would later form the basis for knowledge-based systems (Darwiche, 2009). The design consists of two separate components, a knowledge base that encapsulates what is known, and a reasoning component that reasons over the knowledge base to address certain queries. The language to encode the knowledge was envisioned by McCarthy to be logic. This leads us to the topic of this dissertation, which is in the field of *knowledge representation and reasoning* and includes *probabilistic inference* to deal with the uncertainty present when reasoning or acting in a real-world environment.

1.2 The Algebraic Journey

The journey of this dissertation begins with ProbLog (De Raedt et al., 2007), a *probabilistic logic programming language* that has its origins in the similarly named prominent logic programming language called Prolog (Flach, 1994; Körner et al., 2022). Unlike Prolog, however, ProbLog programs may

contain probabilistic facts, that are true with a certain probability. As a brief introduction, consider the ProbLog program below. It is comprised of two probabilistic facts, `cloudy` and `humid`, and logical rules to deduce more information from those facts. In this case there is one rule, which allows us to deduce that `rain` is true when both probabilistic facts are true.

```
0.25 :: cloudy.
0.8  :: humid.
rain :- cloudy, humid.
```

The particular focus of this dissertation is on the counting (and knowledge compilation) based algorithms that have been developed to perform probabilistic inference in this language (Fierens et al., 2015). For example, to efficiently determine what the probability of `rain` being true is, given the information encoded in the ProbLog program. The state-of-the-art approach to address this question consists of translating the ProbLog program into a weighted logical formula. Consider the following motivating example which makes the connection to counting clearer.

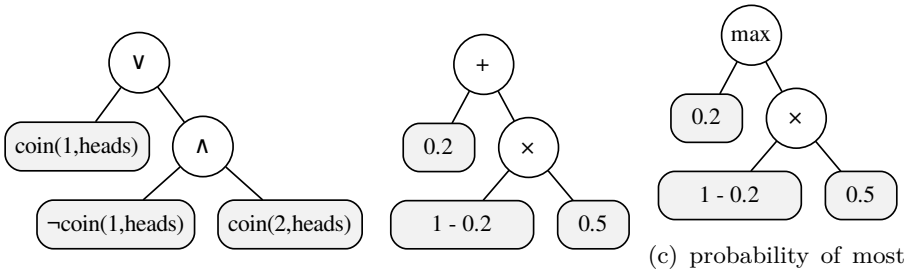
Example 1. *Bob flips two coins. Since each coin has two sides, tossing the coins can lead to one of four possible scenarios (**counting**). Now suppose the first coin is biased, such that the probability of heads for that coin is* 0.2 *instead of* 0.5*. Clearly, the four scenarios are no longer equally likely. Suppose furthermore that Bob considers placing a bet, and is therefore interested in the probability of getting at least one heads. Using the logical symbol* $\vee$ *for disjunction (i.e., 'or'), and* P *to denote the probability, this question is more formally expressed as*

$$P(coin(1, heads) \vee coin(2, heads)) = ? \tag{1.1}$$

To address this question that is a probabilistic query, we can compute the probability of each scenario that satisfies the logical query $coin(1, heads) \vee coin(2, heads)$*, and then add those probabilities (**weighted counting**):*

$$(0.2 \times 0.5) + (0.2 \times (1 - 0.5)) + (0.8 \times 0.5) = 0.6 \tag{1.2}$$

Knowledge compilation. While the previous example had only four possible scenarios, it goes without saying that enumerating all scenarios quickly becomes infeasible for larger problems. In the domain of *knowledge compilation*, tools have been developed that can help to count more efficiently (Darwiche and Marquis, 2002). In particular, these tools reformulate the logical formula, describing the problem in a way that counting becomes easier. For instance in the previous example, we can reformulate the query $coin(1, heads) \vee coin(2, heads)$ to the



(a) logical formula (b) probability heads (c) probability of most likely scenario

Figure 1.1: Formula representations corresponding to the biased coin flip problem in Example 1.

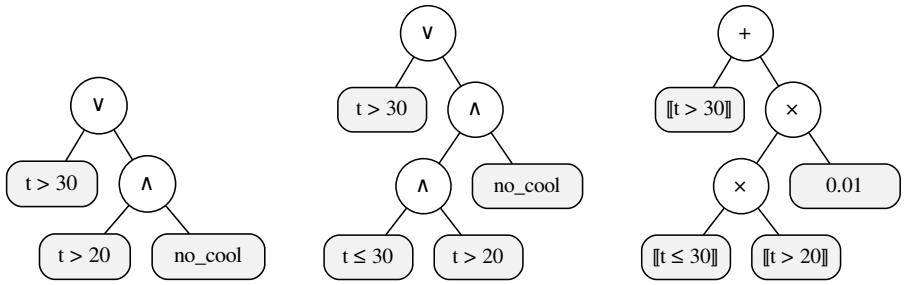
logically equivalent formula displayed below and shown in Figure 1.1a, where ¬ and ∧ are the logical symbols for respectively negation and conjunction (i.e., 'and').

$$coin(1, heads) \vee (\neg coin(1, heads) \wedge coin(2, heads)) \tag{1.3}$$

In this form, weighted counting becomes simple: replace 'or' with addition, 'and' with multiplication, and each coin variable with their associated probability. The resulting equation, shown in Figure 1.1b, is exactly the answer to our probabilistic query, $0.2 + [0.8 \times 0.5] = 0.6$. We explain this process in more detail later in this dissertation.

Algebraic counting. These same knowledge compilation tools also help to solve a more generalized form of counting called *algebraic model counting* (Kimmig et al., 2017). In this form, the addition and product operators have been replaced by more general operations that can be tailored to the task at hand. For instance in the previous example, Bob might wonder what the most likely scenario will be when he does win the bet. The probability of this scenario is easily computed by replacing the addition operator with a maximisation, as illustrated in Figure 1.1c. The description of this most likely scenario can be obtained using a similar technique. As a final example, we consider computing the expected utility. For instance, Bob might wish to compute the yield he can expect to obtain from placing bets. In this case, Bob could compute the probability of each scenario, and multiply it with its yield. Alternatively, the algebraic model counting framework provides Bob with a more efficient approach that only requires selecting the appropriate operations (those of the expectation semiring (Eisner, 2002)).

As the title of this dissertation suggests, we will discuss this technique, and the operation's semiring properties required for it to work, in more detail.



(a) logical formula (b) logical formula (c) equation structure

Figure 1.2: Formula representations corresponding to the broken factory machine problem in Example 2.

Specifically, we will emphasize how the algebraic model counting framework unifies the ProbLog language with its extensions, those that support continuous variables or integration with neural networks, and allows it to more generally solve problems that can be cast to an algebraic model count. Furthermore, we will consider a decision making setting under uncertainty and show how to compute the maximum expected utility, a task that involves three operations instead of two: max, sum, and product. This will help Bob in deciding whether to place a bet.

Modulo theories. Finally, we also consider counting modulo theory problems. In propositional logic, a formula is composed of logical connectives such as $\neg, \vee$, and $\wedge$, and Boolean variables that are only true or false (like in the examples discussed before). In contrast, modulo theory problems move beyond Boolean variables and may require reasoning over integer or real variables. We illustrate this using the following example from Zuidberg Dos Martires (2020), which contains a temperature variable that takes numerical values.

Example 2. *Alice manages the factory machines. As part of her job, she is interested in the probability that a machine breaks down, for which she uses the following formula: a machine breaks down iff the room temperature is above* $30°C$*, or the machine's cooling does not work and the room temperature is above* $20°C$*. After collecting data, she concludes that the cooling does not work properly in* 1% *of the cases, and that the room temperature roughly follows a normal distribution* $\mathcal{N}(20, 9)$*.*

The logical formula corresponding to this example is shown in Figure 1.2a. Again, knowledge compilation tools can help to compute the correct answer

by reformulating the formula such that it decomposes into disjoint regions. Figure 1.2b shows such a reformulation of the previous logical formula. The result can then be turned into the equation shown in Figure 1.2c, where $[\![\cdot]\!]$ represents the Iverson bracket evaluating to 1 when the relation within the brackets is satisfied, and evaluating to 0 otherwise. To compute the correct probability using this equation, we integrate, and push down the integration operator for efficiency reasons when possible:

$$P(broken) = \int f(t)\Big([\![t > 30]\!] + (0.01[\![t \leq 30]\!][\![t > 20]\!])\Big)dt \tag{1.4}$$

$$= \Big(\int f(t)[\![t > 30]\!]dt\Big) + \Big(0.01 \int f(t)[\![t \leq 30]\!][\![t > 20]\!]dt\Big) \tag{1.5}$$

$$= \Big(\int_{t>30} f(t)dt\Big) + \Big(0.01 \int_{20<t\leq 30} f(t)dt\Big) \tag{1.6}$$

$$= 0.00043 + 0.01 \times 0.4996 = 0.005426 \tag{1.7}$$

with $f(t)$ being the probability density function of $\mathcal{N}(20,\, 9)$.

The example is an instance of a **weighted model integration** task (Belle et al., 2015), which benefits from knowledge compilation tools that are capable of reasoning not only over Boolean variables, but also over other types of theories such as (linear) real arithmetic, or integer arithmetic. For example, to realize that $x + y < 5$ and $x > 5$ together imply $y < 0$, techniques beyond Boolean reasoning are required. We discuss this in more detail (for weighted model integration), and will propose a framework for knowledge compilation that works with any quantifier-free theory.

The two overarching research questions of this dissertation are as follows:

> **RQ1** What tasks can be cast into an algebraic model counting problem?
>
> **RQ2** How to then efficiently solve those algebraic model counting problems?

1.3 Contributions

This dissertation presents four main contributions to the research questions stated in the previous section. Each contribution is briefly introduced below.

Algebraic model counting: a unifying framework

RQ1 What tasks can be cast into an algebraic model counting problem?

Algebraic model counting forms a general framework into which several tasks can be cast. It is important to identify these tasks to explore new methods of solving them, and to detect the limitations of algebraic model counting. **Our first main contribution** demonstrates the general applicability of algebraic model counting by considering several inference problems. This contribution comprises the following:

- An overview of the probabilistic, distributional, neural, and algebraic fact, and how they are unifiable under the algebraic model counting framework. We thereby also explain in more detail the connection between counting, knowledge compilation, weighted model counting, and probabilistic inference.
- A demonstration on the ability of algebraic model counting and knowledge compilation to compute the expected utility, and in extension two methods for solving decision making problems in uncertain environments.
- The introduction of a utility learning problem, with a model counting based method to address it.

Dynamically exploiting formula symmetries during model counting

RQ2.1 How to exploit structural symmetry while model counting on propositional logic formulas?

As a second main contribution we investigate how to exploit structural symmetry present in the logical formula to improve the run time of model counters. This contribution is in the context of unweighted counting but is extendable to the weighted and algebraic variants. The idea is relatively simple and revolves around the realization that the model count of a formula is invariant under certain operations. As an example consider that the unweighted model count for each of the following three logical formulas is equivalent.

$$coin(1, heads) \vee \neg coin(2, heads) \tag{1.8}$$

$$\neg coin(2, heads) \vee coin(1, heads) \tag{1.9}$$

$$coin(2, heads) \vee \neg coin(1, heads) \tag{1.10}$$

The second formula is obtained from the first by using the commutativity of $\vee$. The third formula is obtained from the first by renaming the symbols appropriately. Neither of these operations change the unweighted model count. We show how to dynamically exploit this fact by requiring only a minor modification to the caching mechanism of existing model counting algorithms. This optimization effectively leads to a reduction in the model counting search space.

Variable ordering for weighted model integration

> **RQ2.2** Can we extend variable ordering heuristics developed for the discrete domain to also work well for discrete-continuous domains?

Our third contribution focuses on the task of weighted model integration. This is a counting task in the domain of modulo theory formulas, as opposed to the purely propositional formulas. For solving such a task, algebraic model counting and its knowledge techniques have proven to be very useful (Kolb et al., 2019b).

When solving weighted model integration tasks, the variable ordering used by knowledge compilation tools affects not only the representation size but also the performance of the integration performed on top of that representation. Our third contribution is therefore a study on the impact of the variable ordering within weighted model integration. This contribution increased the understanding and led to an adaptation of existing variable ordering heuristics normally used for propositional formulas, as well as a completely novel ordering heuristic.

Top-down modulo theory compilation for counting problems

> **RQ2.3** How to perform knowledge compilation for counting over modulo theory formulas?

Our fourth and final main contribution is a discussion on knowledge compilation for quantifier-free modulo theory formulas, for counting tasks. This contribution includes a new top-down knowledge compilation algorithm, that generalizes compilation for propositional formulas to any quantifier-free modulo theory. This helps, for instance, answer Alice's inference task involving the factory machines where she needed to compile the following modulo theory

formula (cf. Example 2 and Figure 1.2)

$$(t > 30) \vee \big((t > 20) \wedge no\_cool\big) \tag{1.11}$$

We furthermore discuss the complications of modulo theory compilation on the desired representation properties, and the ability to obtain those properties. The main difficulty within this setting comes from the presence of implicit logic connecting multiple atoms. For example while using the linear real arithmetic background theory, when $(x + y \leq 0)$ and $(x \geq 0)$ are both true, then $(y \leq 0)$ must also be true. This statement implicitly holds true regardless of the actual formula.

1.4 Structure of the Thesis

The topic of this dissertation is exact model counting using knowledge compilation. The initial required background is provided in **Chapter** 2. Additional background information is introduced when necessary in the background section of each following chapter.

As a starting point, **Chapter** 3 presents the algebraic journey of the ProbLog (Fierens et al., 2015) programming language, elaborating on the connection between inference in the language on the one hand, and counting and knowledge compilation on the other hand. This also makes clear our primary motivation behind investigating counting over logical formulas. Additionally, it provides the specific insight that ProbLog, its DeepProbLog variant that supports neural network integration (Manhaeve et al., 2018), and its variant DC-ProbLog (Zuidberg Dos Martires et al., 2023) that supports continuous variables, can all be generalised under the algebraic framework using algebraic facts (Kimmig et al., 2011). This chapter thereby contributes to **RQ1** and is based on the following journal article currently under review.

> V. Derkinderen, R. Manhaeve, P. Zuidberg Dos Martires, and L. De Raedt (2023c). "Semirings for Probabilistic and Neural-Symbolic Logic Programming". Accepted with minor revision in International Journal of Approximate Reasoning

After this synthesis we discuss in more detail the additional contributions we made along this journey, continuing with the contribution to **RQ2.1** in the next chapter.

The focus of **Chapter** 4 is on the unweighted variant of the counting task. This chapter explains how to dynamically reduce the counting search space by

exploiting the symmetry present in the counting task's structure. Empirically, equipping a model counter with this enhancement resulted in an improved PAR-2 score and a greater number of solved benchmarks. The publication listed below forms the basis of this chapter. The discussion at the end, on how to extend the proposed idea to the weighted- and projected model counting task variants, is an additional novel contribution not present in the original publication.

> T. van Bremen, V. Derkinderen, S. Sharma, S. Roy, and K. S. Meel (2021). "Symmetric Component Caching for Model Counting on Combinatorial Instances". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 3922–3930

Chapter 5 proceeds, moving back to the weighted and algebraic counting variants. This chapter further demonstrates the generality of the algebraic model counting framework and its algorithms, by adapting it to solve a decision making task in an uncertain environment. This application is non-trivial as it involves three operations, (arg)max, sum, and product, rather than the two operations that algebraic model counting is normally limited to. The chapter contributes to **RQ1** and is based on the following publication:

> V. Derkinderen and L. De Raedt (2020). "Algebraic Circuits for Decision Theoretic Inference and Learning". In: *Proceedings of the 24th European Conference on Artificial Intelligence, ECAI*. vol. 325. IOS Press, pp. 2569–2576. DOI: 10.3233/FAIA200392

In contrast to the previous chapters which primarily focus on propositional logic formulas, Chapters 6 and 7 instead consider knowledge compilation for the modulo theory setting, addressing **RQ2.2** and **RQ2.3** respectively. In this setting, weighted model integration is a relevant counting task (Kolb et al., 2019b). **Chapter** 6 investigates the impact of the variable ordering used during knowledge compilation for such integration tasks. **Chapter** 7 increases the theory awareness of the compilation process, proposing a new algorithm for compiling quantifier-free modulo theory formulas. The two chapters are respectively based on the following publications.

> V. Derkinderen, E. Heylen, P. Zuidberg Dos Martires, S. Kolb, and L. De Raedt (2020). "Ordering Variables for Weighted Model Integration". In: *Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence, UAI*. ed. by R. P. Adams and V. Gogate. Vol. 124. AUAI Press, pp. 879–888

V. Derkinderen, P. Zuidberg Dos Martires, S. Kolb, and P. Morettin (2023d). "Top-Down Knowledge Compilation for Counting Modulo Theories". In: *CoRR* abs/2306.04541. accepted at Workshop on Counting and Sampling at SAT 2023. DOI: 10.48550/arXiv.2306.04541

Finally, **Chapter** 8 concludes this dissertation and provides future research directions.

All my publications, including those not covered in this dissertation, are listed at the end.

Chapter 2

Background

In this chapter we provide the necessary background. We formalize the problem of model counting, weighted model counting, and the more general algebraic model counting. Then we introduce the #DPLL algorithm that performs model counting, and explain how it relates to knowledge compilation.

2.1 Propositional Logic

Propositional logic allows us to formally express propositions precisely, unlike natural language which may be ambiguous. In this logic, propositions such as "a burglary triggers an alarm" are encoded using a combination of so-called atomic propositions, also referred to as *propositional variables*. These variables are Boolean, meaning they are either true or false, and we denote them using uppercase symbols such as X when they are part of a propositional formula. A *propositional formula* ψ is used to represent more complex propositions and is inductively defined as a propositional variable X, a negation of a formula $\neg\psi_1$, a conjunction (meaning 'and') of two formulas $\psi_1 \wedge \psi_2$, or a disjunction (meaning 'or') of two formulas $\psi_1 \vee \psi_2$. We may use parenthesis when necessary for clarity, but otherwise $\neg$ binds stronger than $\wedge$ which in turn binds stronger than $\vee$. This is similar to the usual convention in mathematics of $\times$ binding stronger than $+$. We denote all variables involved in formula ψ as $vars(\psi)$, and use the term *literal* to refer to a propositional variable or its negation.

Example 3. *The proposition "a burglary triggers an alarm", which we interpret as "burglary $\implies$ alarm", can be encoded as the propositional formula $\neg B \vee A$ where B is used to denote a burglary and A denotes the alarm being triggered.*

The *interpretation* m of a propositional formula is a truth assignment to each of its variables. When convenient we may also represent this assignment as a set of literals. When given an interpretation m for a formula ψ, we can also assert the truth of ψ under m, which works as expected: if ψ is a variable X then its truth is equivalent to the truth assigned to X, if ψ is $\neg\psi_1$ then its truth is the negation of ψ_1's, if ψ is $\psi_1 \wedge \psi_2$ then it is true iff both ψ_1 and ψ_2 evaluate to true, and if ψ is $\psi_1 \vee \psi_2$ then it is true iff ψ_1 or ψ_2 evaluate to true. When ψ is true under m, formally denoted as $m \models \psi$, we say m is a model of ψ. When ψ has no models, we say ψ is unsatisfiable. To denote the set of all models of ψ, we use R_ψ.

Example 4. *The interpretation* $m = \{A \mapsto true, B \mapsto true\}$ *is a model of the previous formula* $\neg B \vee A$*, while* $\{A \mapsto false, B \mapsto true\}$ *is not. For brevity, we may instead represent those interpretations as* $\{A, B\}$ *and* $\{\neg A, B\}$*.*

A *propositional theory* is a set of propositional formulas, and it is interpreted as true under m iff each of its formulas is true under m. A *clause* is a disjunction of (multiple) literals, e.g. $A \vee \neg B \vee C$. A propositional formula is said to be in *conjunctive normal form* (CNF) when it is a conjunction of clauses.

2.2 Model Counting

The problem of determining whether a propositional formula ψ has a model, i.e., whether $|R_\psi| > 0$, is called the (Boolean) *satisfiability problem* (SAT). Connected to the satisfiability problem is the model counting problem: "given a propositional formula ψ, the *model counting* problem, also referred to as #SAT, seeks to compute the number of satisfying assignments (or models) of ψ, i.e., $|R_\psi|$"(van Bremen et al., 2021).

Definition 1 (model count)**.** *The model count of a propositional formula* ψ *over variables* $\mathbf{V}$ *is the number of models that* ψ *has over* $\mathbf{V}$*. Evidently it holds that the model count lies within the interval* $[0, 2^{|\mathbf{V}|}]$*.*

We denote the model count as $|R_\psi|$, implying $\mathbf{V} = vars(\psi)$ unless specified otherwise.

Example 5. $A \vee \neg B$ *has model count three:* $\{A, B\}$*,* $\{A, \neg B\}$*, and* $\{\neg A, \neg B\}$*.*

Weighted model counting is a generalisation of model counting where each model is assigned a weight. Since it would be impractical to define a weight for each individual model, the weight of a model m is commonly defined as the product of the weight of each literal l in that model: $\prod_{l \in m} w(l)$.

Definition 2 (weighted model count (WMC))**.** *Given a propositional formula ψ over variables $\mathbf{V}$, and a weight function w mapping each literal to a real, the weighted model count is*

$$WMC(\psi, \mathbf{V}, w) = \sum_{m \models \psi} \prod_{l \in m} w(l) \tag{2.1}$$

Example 6 (weighted model count)**.** *Given formula $\psi = A \vee \neg B$ over $\mathbf{V} = \{A, B\}$, and weight function $w = \{A \mapsto 0.2, \neg A \mapsto 1, B \mapsto 0.5, \neg B \mapsto 2\}$, the weighted model count is* 2.5*.*

$$WMC(\psi, \mathbf{V}, w) = (0.2 \times 0.5) + (0.2 \times 2) + (1 \times 2) = 2.5$$

Applications of weighted model counting include probabilistic inference, which we discuss in more detail in Chapter 3 and 5.

Algebraic model counting generalizes model counting even further, replacing the addition and product operations with commutative semiring operations (Kimmig et al., 2017). From Derkinderen and De Raedt (2020),

Definition 3 (commutative semiring)**.** *A commutative semiring $\mathcal{S}$ is an algebraic structure $(\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$ where*

- *$\mathcal{A}$ defines the domain of the values,*
- *$\oplus$ and $\otimes$ are associative, commutative binary operations over $\mathcal{A}$,*
- *$\otimes$ distributes over $\oplus$,*
- *$e^{\otimes} \in \mathcal{A}$ and $\forall a \in \mathcal{A}$: $e^{\otimes} \otimes a = a$, i.e., $e^{\otimes}$ is a neutral element for $\otimes$.*
- *$e^{\oplus} \in \mathcal{A}$ and $\forall a \in A$: $e^{\oplus} \oplus a = a$ and $e^{\oplus} \otimes a = e^{\oplus}$, i.e., $e^{\oplus}$ is a neutral and absorbing element for $\oplus$ and $\otimes$ respectively.*

The semiring relevant to (weighted) model counting is $(\mathbb{R}, +, \times, 0, 1)$, and the one for satisfiability problems is $(\mathbb{B}, \vee, \wedge, \bot, \top)$, where $\bot$ is the symbol for 'false', and $\top$ is the symbol for 'true'. From Derkinderen and De Raedt (2020),

Definition 4 (algebraic model count (AMC))**.** *Given a propositional formula ψ over variables $\mathbf{V}$, a commutative semiring $\mathcal{S} = (\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$, and a weight function w mapping each literal to an element of $\mathcal{A}$, the algebraic model count (Kimmig et al., 2017) is*

$$AMC(\psi, \mathbf{V}, \mathcal{S}, w) = \bigoplus_{m \models \psi} \bigotimes_{l \in m} w(l) \tag{2.2}$$

Example 7 (algebraic model count)**.** *Suppose semiring* $\mathcal{S} = (\mathbb{R}, max, \times, 0, 1)$*, formula* $\psi = A \vee \neg B$ *and* $w = \{A \mapsto 1, \neg A \mapsto 2, B \mapsto 3, \neg B \mapsto 4\}$ *then* $AMC(\psi, \mathbf{V}, \mathcal{S}, w) = 8$*, the highest weight out of all models of* ψ*.*

$$AMC(\psi, \mathbf{V}, \mathcal{S}, w) = max\big((1 \times 3), (1 \times 4), (2 \times 4)\big) = 8$$

By selecting the correct semiring and labeling function, several tasks can be cast into an AMC task, including sensitivity analysis and computing gradients (Kimmig et al., 2017). In Chapter 5 we adapt this framework to perform decision making in an uncertain environment.

Constraint programming. The Boolean satisfiability (SAT) problem is focused on propositional logic formulas. The larger field of constraint programming also encompasses other types of constraints. For example, integer linear programs (ILP) more generally contain linear constraints over integers. Satisfiability in a $0{-}1$ ILP specifically, where the integers are limited to 0 or 1, is an NP-complete problem (Karp, 1972). Hence, they are poly-time reducible to SAT problems and vice versa. In contrast, model counting is a counting task (as opposed to a decision or optimisation task) and is as such not directly comparable. However, the algorithms developed for counting do relate to the algorithms developed for satisfiability or optimization. That is, the #DPLL algorithm that is explained in the next section relates to the branching algorithms used for solving ILPs, like the branch-and-bound based algorithms (Morrison et al., 2016). Consequently, our contributions to model counting, like those discussed later in Chapter 4, may also contribute to other areas of constraint programming.

2.3 Counting via the #DPLL Algorithm

The problem of model counting, both in its unweighted and weighted form, serves as a well-known example of the #P-complete complexity class (Valiant, 1979a). Chances of finding a polynomial-time algorithm for model counting are therefore believed to be slim, since it would proof P = NP, which is widely regarded to be unlikely. This issue has also been observed in practice when trying to scale to larger counting problems. However, over the years many more instances have become solvable in practice by algorithms exploiting the structure present in the instances. The #DPLL algorithm is a counting algorithm that does exactly that. We discuss a basic version of this algorithm, extended with component caching. In Chapter 4 we contribute an improvement to it. Parts of Section 2.3.2 are based on our publication discussed in Chapter 4 (van Bremen et al., 2021).

2.3.1 The Basic #DPLL Algorithm

The Davis–Putnam–Logemann–Loveland algorithm, DPLL for short, was designed to address satisfiability problems (Davis et al., 1962; Davis and Putnam, 1960). The algorithm assumes that formula ψ is in conjunctive normal form (CNF), i.e., a conjunction of clauses. When this is not the case ψ must first be transformed, using for example the Tseitin transformation algorithm (Kuiter et al., 2023; Tseitin, 1983).

Roughly, the DPLL algorithm works by iteratively branching on literals l until ψ is satisfied or until a conflict occurs. In case of the latter, the algorithm backtracks and tries an opposite literal $\neg l$. Consider what happens when conditioning on literal l, using the following example.

Example 8 (conditioning)**.** *Consider the CNF formula ψ depicted below without the $\wedge$ operator that is implicitly present between the two clauses. When conditioning C to be true, denoted as $\psi|_C$, then the second clause $\neg B \vee C$ becomes satisfied and the first clause is transformed into $A \vee B$. Indeed, then $\neg C$ evaluates to false and the first clause can only be satisfied through $A \vee B$.*

$$\psi \begin{cases} A \vee B \vee \neg C \\ \neg B \vee C \end{cases} \qquad\qquad \psi|_C \begin{cases} A \vee B \end{cases}$$

The basic DPLL algorithm. A basic DPLL algorithm is shown in Algorithm 1. First it checks whether any of the clauses is empty, in line 2. An empty clause indicates that a conflict occurred, i.e., that our current truth assignments have made it impossible for the clause, and by consequence the whole ψ, to be satisfied. Hence, false is returned. In contrast, when no clauses remain at all, i.e., all are satisfied, the current (partial) assignment forms a model of ψ and true is returned (line 4). For instance after conditioning on C and B in the previous example, all clauses of ψ are satisfied. This means $\{B, C, A\}$ and $\{B, C, \neg A\}$ are both models of ψ, positively addressing the satisfiability question of whether there exists a model for ψ. In line 6 a variable is selected and in line 7 the actual conditioning and branching occur. Note for the final line that the lazy or-operator will first perform DPLL($\psi|_l$) and only when that assignment fails, i.e., false was returned, only then is the second argument DPLL($\psi|_{\neg l}$) computed.

The DPLL algorithm has been significantly researched and extended over the years. For example, a variety of variable selection heuristics have been developed (line 6) (Bliem and Järvisalo, 2019; Lagniez and Marquis, 2017; Sang et al., 2005; Sharma et al., 2019; Vaezipoor et al., 2021). Also worth mentioning is *unit propagation*: when a clause becomes unit, meaning it only has one literal

Algorithm 1: Basic DPLL algorithm

```
1 function DPLL(ψ):
2     if ψ contains empty clause then
3         return false
4     else if ψ contains no clauses then
5         return true
6     pick a literal l in ψ
7     return DPLL(ψ|_l) or DPLL(ψ|_¬l)
```

Algorithm 2: Basic #DPLL algorithm

```
1 function #DPLL(ψ, w):
2     if ψ contains empty clause then
3         return 0
4     else if ψ contains no clauses then
5         return ∏_{unassigned variable v}(w(v) + w(¬v))
6     pick a literal l in ψ
7     return #DPLL(ψ|_l, w) × w(l) + #DPLL(ψ|_¬l, w) × w(¬l)
```

X left, then assigning $\neg X$ definitely leads to a conflict and we should instead assign X to be true. The assignment of X may introduce more unit clauses, causing even more literals to be assigned. This optimisation, of assigning and propagating unit clause literals, reduces the exploration of invalid assignments and is used by all modern solver implementations.

The basic #DPLL algorithm. While the DPLL algorithm is designed to address the satisfiability problem, a slight adaptation of the algorithm, which we refer to as *#DPLL*, can be used to address (weighted) model counting problems (Birnbaum and Lozinskii, 1999). In Algorithm 2 we have replaced return false and true to instead return weighted model counts. When an assignment leads to a satisfied ψ, it returns 1. When the assignment is partial the weights of the unassigned variables must be considered, i.e., return $\prod_{\text{unassigned variable } v}(w(v) + w(\neg v))$. In line 7 the weight of each assignment l is incorporated.

CDCL. *Conflict-driven clause learning* (CDCL) is a variant of the DPLL algorithm through which the solving of many practical satisfiability problems became feasible (Sang et al., 2004; Schrag, 1997; Silva and Sakallah, 1996). This variant analyses the cause of conflicts, learns from it and, in contrast to

DPLL which backtracks to the previous assignment, back jumps to an earlier assignment that caused the conflict. This efficiently avoids exploring many invalid assignments, and all modern satisfiability solvers and model counters based on search have adopted this optimisation. The CDCL algorithm can also be used for counting, similar to how DPLL was extended to #DPLL. In this dissertation we do not differentiate between the #DPLL and the #CDCL algorithm, i.e., the statements that concern #DPLL are also applicable to #CDCL unless stated otherwise.

2.3.2 #DPLL with Component Caching

As previously mentioned, the #DPLL algorithm and its CDCL variant have seen several optimisations. Arguably one of the most impactful optimizations for model counting specifically has been *component caching* (Bacchus et al., 2003; Sang et al., 2004). Component caching identifies subformulas (or *components*) that can be solved independently and memoizes them, allowing for a shallower search tree.

Definition 5 (component)**.** *Consider a partitioning of a formula ψ into sets of clauses $\psi = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n$ such that $vars(\mathcal{C}_i) \cap vars(\mathcal{C}_j) = \emptyset$ for $i \neq j$. Then each $\mathcal{C}_i$ is called a component of ψ, and we have $|R_\psi| = \prod_{i=1}^{n} |R_{\mathcal{C}_i}|$.*

Example 9 (component)**.** *Consider the formula ψ and its conditioning $\psi|_B$:*

$$\psi \begin{cases} A \vee \neg C \\ \neg A \vee \neg B \vee C \\ D \vee \neg E \\ \neg D \vee E \vee B \end{cases} \qquad \psi|_B \begin{cases} A \vee \neg C \\ \neg A \vee C \\ D \vee \neg E \end{cases}$$

The first two clauses of $\psi|_B$ do not share any variables with the third clause. This means that $\psi|_B$ can be split up into the two components $\mathcal{C}_1 = \{A \vee \neg C, \neg A \vee C\}$ and $\mathcal{C}_2 = \{D \vee \neg E\}$ such that $|R_{\psi|_B}| = |R_{\mathcal{C}_1}||R_{\mathcal{C}_2}|$.

To more clearly illustrate the computational advantage of decomposing ψ into components $\mathcal{C}_1$ and $\mathcal{C}_2$, consider that ψ implies checking $2^{|vars(\psi)|}$ assignments while by decomposing we must only check $2^{|vars(\mathcal{C}_1)|} + 2^{|vars(\mathcal{C}_2)|}$ assignments. This high-level intuition also explains why decomposing ψ does not provide the same significant benefit to solving satisfiability problems as it does with solving counting problems, because the satisfiability problems are only concerned with finding just one model.

Algorithm 3 illustrates the basic #DPLL algorithm from before but now includes unit propagation and component decompositioning. While the combination of component decompositioning with clause learning and backjumping (CDCL) has proven to be beneficial in practice (Sang et al., 2004), we exclude it from our discussion as it is not important to illustrate our contributions. In our implementation of Chapter 4 we do employ the combination. Also note that Algorithm 3 performs unweighted model counting. A weighted version is possible through minor adaptations, but it is unnecessary for Chapter 4 that focuses on unweighted counting.

As illustrated in the previous example and seen in the algorithm, component decompositioning (line 20) is not restricted to the algorithm's beginning and can instead be performed after every literal assignment, more specifically after unit propagation (line 12). Component decompositioning is usually also paired with caching: when a component is solved, it is stored alongside its model count (line 8) so that when an identical component is encountered later in the search tree the cached value can be reused (line 3).

Algebraic model counting. It turns out that the #DPLL algorithm can also easily be used to perform algebraic model counting. After adapting the algorithm to perform weighted model counting, update the weights of each literal and replace the operations $+$ and $\times$ with $\oplus$ and $\otimes$ respectively.

2.4 Counting via Knowledge Compilation

We now discuss the topic of knowledge compilation and start by reiterating that model counting is #P-complete in general (Valiant, 1979a). There is however a class of formula representations where counting is tractable: the class of (s)d-DNNF formulas (Darwiche and Marquis, 2002). Compiling any formula into such a form is being studied in the field of knowledge compilation. Even though counting via d-DNNF compilation may appear as a very different approach compared to using the #DPLL algorithm, the two are actually strongly related as the search trace of the #DPLL algorithm forms a d-DNNF representation.

We first elaborate on the d-DNNF and sd-DNNF class in Section 2.4.1, before elaborating more on the connection with #DPLL in Section 2.4.2. Finally, in Section 2.4.3, we briefly discuss decision diagrams that belong to the d-DNNF class but are compiled using a different approach than the traces of a #DPLL algorithm. Section 2.4.3 is based on our publication discussed in Chapter 5 (Derkinderen and De Raedt, 2020).

Algorithm 3: #DPLL algorithm with component caching.

```
1  function GetModelCount(ψ):
2      if ψ in cache then
3          return CacheGet(ψ)
4      else
5          pick a literal l in ψ
6          |R_{ψ_l}| ← CountConditioned(ψ, l)
7          |R_{ψ_¬l}| ← CountConditioned(ψ, ¬l)
8          CacheInsert(ψ, |R_{ψ_l}| + |R_{ψ_¬l}|)
9          return |R_{ψ_l}| + |R_{ψ_¬l}|
10     end
11 function CountConditioned(ψ, l):
12     ψ_l ← propagate units on ψ|_l
13     if ψ_l contains empty clause then
14         return 0
15     else if ψ_l contains no clauses then
16         v ← number of unassigned variables in ψ_l
17         return 2^v
18     else
19         |R_{ψ_l}| ← 1
20         C ← DisjointComponents(ψ_l)
21         for C_i ← C do
22             |R_{ψ_l}| ← |R_{ψ_l}| × GetModelCount(C_i)
23         end
24         return |R_{ψ_l}|
25     end
```

2.4.1 sd-DNNF Formulas

The class of sd-DNNF formulas is composed of those that satisfy the following properties: smooth (s), deterministic (d), decomposable (D), and are in negation normal form (NNF) (Darwiche and Marquis, 2002).

Definition 6 (negation normal form (NNF))**.** *A formula ψ is in negation normal form iff negation only occurs on the literals in ψ, and the only other Boolean operators are $\wedge$ and $\vee$.*

Definition 7 (deterministic)**.** *An NNF formula ψ is deterministic iff for all disjunctions $\bigvee_i \alpha_i$ in ψ, the disjuncts are pairwise logically inconsistent, i.e., $\alpha_i \wedge \alpha_j$ is unsatisfiable for each $i \neq j$.*

Definition 8 (decomposable)**.** *An NNF formula ψ is decomposable iff for all conjunctions $\bigwedge_i \alpha_i$ in ψ, no variables are shared between the conjuncts, i.e., $vars(\alpha_i) \cap vars(\alpha_j) = \emptyset$ for each $i \neq j$.*

Definition 9 (smooth)**.** *An NNF formula ψ is smooth iff for all disjunctions $\bigvee_i \alpha_i$ in ψ, each disjunct α_i contains the same variables, i.e., $vars(\alpha_i) = vars(\alpha_j)$.*

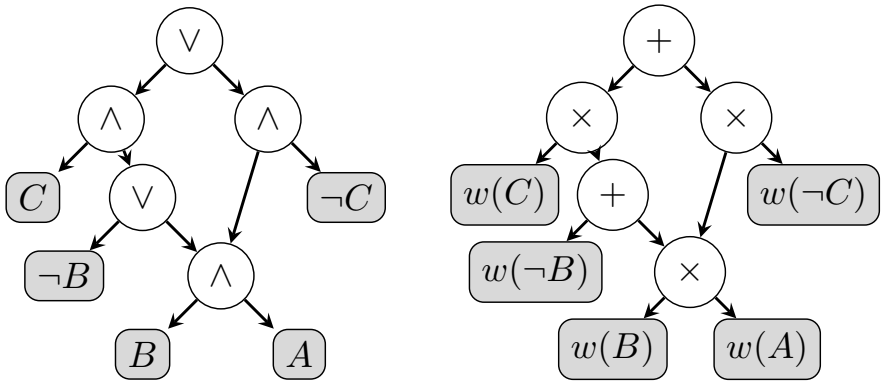
The weighted model count of a formula ψ in sd-DNNF can be computed in time linear in the size of the representation. Since smoothness can be obtained in polynomial time when ψ is a d-DNNF (Darwiche and Marquis, 2002; Shih et al., 2019), we consider the d-DNNF properties as most important and primarily focus on obtaining those. The procedure to obtain the weighted model count of an sd-DNNF formula consists of replacing each literal l with its weight $w(l)$, and each $\vee$ and $\wedge$ with $+$ and $\times$ respectively. This results in a computational graph that, when evaluated bottom-up, exactly yields the weighted model count (Darwiche, 2000). We call the computational graph resulting from this procedure an *arithmetic circuit* (Darwiche, 2002). This procedure is not just limited to weighted model counting but is also applicable to the algebraic variant by replacing each $\vee$ and $\wedge$ with the semiring operations $\oplus$ and $\otimes$ respectively. We refer to Kimmig et al. (2017) for a formal exposition on why this works.

Example 10. *The propositional formula $(B \vee C) \wedge (\neg B \vee A)$ satisfies the NNF property but is neither deterministic, nor decomposable, nor smooth. It is not deterministic because, for example, B and C in $(B \vee C)$ are not logically inconsistent. It is not decomposable because B occurs in both branches of the $\wedge$-node. It is not smooth because, for example, the branches of $(B \vee C)$ do not mention the same set of variables. In contrast, Figure 2.1a illustrates a logically equivalent formula that does satisfy the d-DNNF properties (but is not smooth because of the bottom left $\vee$-node). Figure 2.1b shows the same representation after its transformation into an arithmetic circuit, that can be used to compute the weighted model count (while smoothing during the evaluation of the representation as to account for $w(C) + w(\neg C)$).*

2.4.2 Traces of the #DPLL Algorithm

The traces of the #DPLL algorithm introduced in Section 2.3 form a d-DNNF formula (Darwiche, 2004; Huang and Darwiche, 2005):

decision Each decision in the #DPLL algorithm corresponds to a deterministic $\vee$-node $(X \wedge \psi|_X) \vee (\neg X \wedge \psi|_{\neg X})$ of which the inner $\wedge$-nodes are decomposable.



(a) d-DNNF (b) arithmetic circuit

Figure 2.1: A d-DNNF and arithmetic circuit representation of the propositional formula $(B \vee C) \wedge (\neg B \vee A)$.

propagation By determining literal X through (unit) propagation on ψ, the algorithm produces a decomposable $\wedge$-node of the form $X \wedge \psi|_X$.

component decompositioning The component decompositioning introduced before decomposes ψ into independent components $\mathcal{C}_1, \ldots, \mathcal{C}_n$ that by definition do not share any variables. This results in a decomposable $\wedge$-node $\mathcal{C}_1 \wedge \cdots \wedge \mathcal{C}_n$ (slightly abusing notation since $\mathcal{C}_i$ is a set of clauses).

The traces are also in negation normal form since the only operands are $\{\wedge, \vee, \neg\}$ and negation only occurs on the literal leaf nodes.

The formulas produced by the traces of a #DPLL algorithm belong to a subclass of d-DNNF, namely Decision-DNNF. The Decision in this class refers to the fact that all $\vee$-nodes are of a specific form, namely $(X \wedge \psi|_X) \vee (\neg X \wedge \psi_{\neg X})$ with X a propositional variable (Darwiche and Marquis, 2002; Oztok and Darwiche, 2014).

Knowledge compilers that are based on amortizing the trace of a #DPLL algorithm are also called top-down knowledge compilers. Examples include c2d (Darwiche, 2004), dSharp (Muise et al., 2012), miniC2D (Oztok and Darwiche, 2015), D4 (Lagniez and Marquis, 2017), and sharpSAT-TD (Kiesel and Eiter, 2023).

This connection between the #DPLL algorithm and knowledge compilation also makes clearer how these procedures operate: they essentially solve the weighted model counting equation by applying distributivity, associativity, and

commutativity to drastically reduce the number of required computations. This is the mathematical view. However, they recognize that formula ψ dictates the computations and therefore take a more logical perspective instead (Derkinderen et al., 2023c).

2.4.3 Decision Diagrams

Opposite to top-down compilers are bottom-up compilers, which, as the name implies, process a formula ψ from the bottom of its expression to the top. Suppose for example that ψ is $(A \vee \neg B) \wedge (\neg A \vee C)$. In this case, a bottom-up compiler would first represent the two conjuncts (here each a disjunction) before performing the conjunction operation itself. This process implies the existence of an *apply*-operation defined for the target language.

Ordered binary decision diagrams (Bryant, 1986) and *sentential decision diagrams* (Darwiche, 2011) are two examples of target languages that are subclasses of d-DNNF and for which bottom-up compilers have been developed (Choi and Darwiche, 2013; Somenzi, 1997).

A *sentential decision diagram* (SDD) represents a propositional logic formula and is either a constant (true $\top$ or false $\bot$), a literal or a decomposition node $\{(p_1, s_1), \ldots, (p_n, s_n)\}$. The latter represents $\bigvee_{i=1}^{n} p_i \wedge s_i$ with p_i and s_i both SDDs. A decomposition node is an $\vee$-node that partitions the theory into disjoint children (p_i, s_i) called the elements of the decomposition node. Each element is graphically represented as a paired box where the left and right boxes are respectively called the prime p_i and sub s_i (cf. Figure 2.2). The pair (p_i, s_i) represents a conjunction of both ($\wedge$-node). s_i represents the parent theory conditioned on p_i (cf. Example 11). The disjointness of the elements is specifically caused by the prime of each element, i.e., $\forall i \neq j : p_i \wedge p_j = \bot$. For a more thorough explanation of the decomposition we refer to Darwiche (2011).

Example 11. *The root r in Figure 2.2 represents the propositional formula $\psi = (A \wedge B) \vee (C \wedge D) \vee (B \wedge C)$. Call the children of r, from left to right, r_1, r_2 and r_3 and the formula they represent $\langle r_1 \rangle, \langle r_2 \rangle$ and $\langle r_3 \rangle$. The prime of r_1 represents $\neg B$, and the sub of r_1 represents ψ conditioned on $\neg B$ which equals $C \wedge D$. The conjunction of both the prime and sub of r_1 forms the formula $\langle r_1 \rangle = \neg B \wedge C \wedge D$. The formula of r is the disjunction of each of its children: $\langle r \rangle = \psi = \langle r_1 \rangle \vee \langle r_2 \rangle \vee \langle r_3 \rangle$.*

A *vtree* is a full binary tree where each SDD variable appears in a leaf node (Figure 2.3). Denote with v^l and v^r the left and right subtree of vtree v. A vtree guides the construction of an SDD by determining the variables present in the primes and subs of each SDD node. When node $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ respects

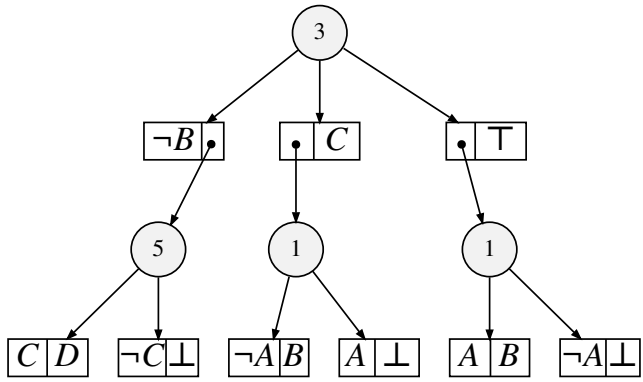


Figure 2.2: A sentential decision diagram representing propositional formula $(A \wedge B) \vee (C \wedge D) \vee (B \wedge C)$. A circle represents an $\vee$-node, while a paired-box represents an $\wedge$-node consisting of two children: the prime and sub.

vtree node v, the variables in each p_i and s_i are determined by respectively the variables in v^l and v^r, and each p_i (s_i) respects v^l (v^r). Graphically, the number in the decomposition node refers to the vtree node that it respects (Figure 2.2 and 2.3). The following is the formal SDD definition introduced by Darwiche (2011). The definition uses $\langle \alpha \rangle$ to denote the boolean function represented by the SDD α.

Definition 10 (sentential decision diagram (SDD))**.** *α is an SDD that respects vtree v iff:*

- $\alpha = \bot$ *or* $\alpha = \top$.
  Semantics: $\langle \bot \rangle = false$ *and* $\langle \top \rangle = true$.
- $\alpha = X$ *or* $\alpha = \neg X$ *and v contains variable X.*
  Semantics: $\langle X \rangle = X$ *and* $\langle \neg X \rangle = \neg X$.
- $\alpha = \{(p_1, s_1), \ldots (p_n, s_n)\}$, *$v$ is internal, $p_1, \ldots, p_n$ are SDDs that respect the subtrees of v^l, $s_1, \ldots, s_n$ are SDDs that respect the subtrees of v^r, and $\langle p_1 \rangle, \ldots, \langle p_n \rangle$ is a partition.*
  Semantics: $\langle \alpha \rangle = \bigvee_{i=1}^{n} \langle p_i \rangle \wedge \langle s_i \rangle$.

Example 12. *We denote the root node in Figure 2.2 as r and the root of the vtree in Figure 2.3 as v. The primes of r only involve A and B since v^l only contains A and B. The subs of r only involve C and D since v^r only contains C and D. We say r respects v (node label 3).*

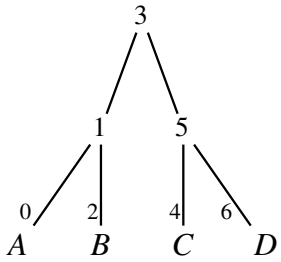


Figure 2.3: The vtree used for the sentential decision diagram in Figure 2.2.

The *strong determinism* (Pipatsrisawat and Darwiche, 2010) and *structured decomposability* (Pipatsrisawat and Darwiche, 2008) of SDDs allow for an efficient apply operation that is key to a bottom-up compiler. For the purpose of this dissertation we neither discuss in detail these two properties, nor the apply-algorithm, and instead refer to Darwiche (2011). However, strong determinism relates to the specific form of the $\vee$-node where each branch is disjoint because of the disjointness of the primes, and structured decomposability refers to the vtree guidance.

Counting via SDD. Because SDD is a subclass of d-DNNF, we can again replace the $\vee$-nodes (circle) and $\wedge$-nodes (paired box) in an SDD with respectively $+$ and $\times$, and the literals with their weights, to obtain an arithmetic circuit that represents the weighted model count (after accounting for the smoothness requirement). Furthermore, when replacing the $+$ and $\times$ with the more general $\oplus$ and $\otimes$ operations, and the leaf values with semiring elements, we obtain what we refer to as an *algebraic circuit*. The latter circuit can be used to compute the algebraic model count (Kimmig et al., 2017).

Relation to OBDD. Ordered binary decision diagrams (OBDD), which were developed prior to SDD, form a subclass of SDD that arise when only conditioning on single variables (i.e., a right-linear vtree) rather than sentences. More specifically, they arise when each vtree node v is a Shannon vtree node, i.e., when each left vtree branch v^l only contains a single variable (Oztok and Darwiche, 2015).

2.5 Background Theories

Chapters 6 and 7 move beyond the limitation of propositional logic to discrete

variables, extending the focus to background theories. In this setting, the semantics of a formula are extended with additional types of atoms that are interpreted by specific background theories. The theory most relevant to this dissertation is *linear real arithmetic* ($\mathcal{LRA}$). Other notable theories include *(linear) integer arithmetic* ($\mathcal{LIA}$) and *fixed-size bit vectors* ($\mathcal{BV}$). Compared to a propositional formula, an $\mathcal{LRA}$ formula may additionally contain $\mathcal{LRA}$ atoms. More formally,

Definition 11 ($\mathcal{LRA}$ formula)**.** *(Morettin et al., 2019) An $\mathcal{LRA}$ formula ψ is inductively defined as a propositional formula ψ, a conjunction or disjunction of two $\mathcal{LRA}$ formulas, the negation of an $\mathcal{LRA}$ formula $\neg\psi$, or an $\mathcal{LRA}$ atom of the form $\sum_i c_i x_i \bowtie c$ with $c_i, c \in \mathbb{R}$, $x_i \in \mathbb{R}$, and $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$.*

Importantly, this dissertation will only focus on quantifier-free theories, which is reflected in the previous definition.

Example 13 ($\mathcal{LRA}$ formula)**.** *Consider the following $\mathcal{LRA}$ formula:*

$$[(x \leq 0) \vee (x \geq 1)] \wedge [A \vee (x \leq 0)]$$

As a consequence of the background theory, $x \leq 0$ implies $\neg(x \geq 1)$, even when this is not explicitly part of the formula representation.

Chapter 3

Probabilistic and Neural-Symbolic Logic Programming

This chapter is based on the following submission:

> V. Derkinderen, R. Manhaeve, P. Zuidberg Dos Martires, and L. De Raedt (2023c). "Semirings for Probabilistic and Neural-Symbolic Logic Programming". Accepted with minor revision in International Journal of Approximate Reasoning

The preliminaries in the previous chapter have made clear the connection between counting and knowledge compilation. In this chapter we discuss in more detail their application within *probabilistic logic programming* (PLP), a field that focuses on integrating probabilistic models into programming languages based on logic. While originally PLP focused on discrete probabilities, more recent approaches have incorporated continuous distributions as well as neural networks, effectively yielding neural-symbolic methods. This chapter provides an overview and synthesis of this domain, through the 15-year journey of the ProbLog PLP language (De Raedt et al., 2007) and its variants, thereby contributing a unified algebraic perspective on the different flavors of PLP.

This chapter contributes to answering the research question:

RQ1: What tasks can be cast into an algebraic model counting problem?

The remainder of this chapter is organised as follows: in Section 3.1 we provide a historical overview of probabilistic logic programming and list the contributions of this chapter. In Section 3.2 we give a brief introduction to logic programming and how logic programs can be extended to a wide variety of domains such as statistical relational AI and neural-symbolic AI. We also show how these extensions are generalized by the concept of the algebraic fact and the use of semirings. In Section 3.3 we give a brief explanation of how inference is performed for algebraic logic programming, and in Section 3.4 we study learning for such programs. Finally, we give an overview of related work and applications in Section 3.5, followed by concluding remarks in Section 3.6.

Throughout the chapter we focus on intuitions and on the simplest setting for PLP based on labeled facts, rather than exhaustively covering all (syntactic) variations for which we refer to the literature for technical details.

3.1 Introduction

3.1.1 History of Probabilistic Logic Programming

Probabilistic logic programming (De Raedt and Kimmig, 2015; Riguzzi, 2018) integrates probabilistic programming (Goodman et al., 2016) with logic programming (Flach, 1994; Lloyd, 2012). It has a rich tradition dating back to the early 1990s. In particular, Dantsin (1990), Ng and Subrahmanian (1992), Poole (1993) adapted ideas by Nilsson (1986) and Pearl (1988) on probabilistic graphical models and logics towards a logic programming framework. Sato (1995) and Poole (1997) then introduced the ideas of distribution semantics and independent choice logic, respectively, which allows for the extension of (deterministic) logic programs (Flach, 1994) with probabilistic facts. Probabilistic facts play a role similar to the parentless nodes in Bayesian networks: they are marginally independent of one another, and dependencies are induced by the rules of a logic program. Sato (1995) also introduced the first learning algorithm for a programming language constituting – to the best of our knowledge – the first probabilistic programming language with built-in support for machine learning.

Following the works of Sato and Poole, an explosion happened in probabilistic logic programming leading to a plethora of inference and learning techniques, along with extensions of the original distribution semantics (Kersting and

De Raedt, 2000; Sato and Kameya, 1997; Vennekens et al., 2004). Some of these works are concerned with faster inference using knowledge compilation technology (in ProbLog (De Raedt et al., 2007)) and approximate inference (Vlasselaer et al., 2015) as well as extensions towards continuous distributions (Gutmann et al., 2011a), the use of semirings (Eisner and Filardo, 2010; Kimmig et al., 2011; Orsini et al., 2017), neural networks (Manhaeve et al., 2018; Yang et al., 2020), and dynamics (Vlasselaer et al., 2016). For a broad overview of probabilistic logic programming and related techniques we refer the reader to (Riguzzi, 2018) and (De Raedt and Kimmig, 2015).

It is noteworthy that in parallel to the developments in probabilistic logic programming, similar advances were made in the field of probabilistic relational databases (Van den Broeck, Suciu, et al., 2017). Just like for probabilistic logics, the idea of probabilistic databases dates back to the mid-eighties (Barbará et al., 1992; Cavallo and Pittarelli, 1987; Gelenbe and Hebrail, 1986) and has consequently been developed since (Antova et al., 2006; Benjelloun et al., 2006; Dalvi and Suciu, 2007; Fuhr, 1995; Fuhr and Rölleke, 1997; Grohe et al., 2022; Lakshmanan et al., 1997; Olteanu et al., 2009).

3.1.2 Synthesizing Probabilistic Logic Programming Variations

The key contribution of this chapter is the insight that many extensions of the basic PLP framework can be cast within a unified algebraic logic programming framework (Eisner and Filardo, 2010; Kimmig et al., 2011), in which the standard probability semiring is replaced by another, sometimes special purpose semiring.

Replacing the probability semiring with an arbitrary semiring allows us to generalize probabilistic logic programming towards many other inference and learning tasks. This is akin to the use of semirings in graphical models, in which the sum-product algorithm can be replaced by a max-product to obtain the most probable state instead of the probability.

To provide evidence for our claim we provide a synthesis of the many variants of the ProbLog language (De Raedt et al., 2007) that were developed in the 15-year journey in probabilistic logic programming. More specifically, we will introduce a unified semiring framework that generalizes probabilistic logic programming to an algebraic logic programming framework with corresponding inference and learning algorithms based on semirings. We will also show how to obtain 1) pure Prolog, that is definite clause logic, 2) ProbLog, the extension of Prolog with the probabilistic facts, 3) DeepProbLog, an extension of ProbLog with neural

predicates, and 4) DC-ProbLog, an extension of ProbLog towards continuous distributions, as special cases of the unified framework. The key advantage of casting these PLP frameworks with a unified algebraic PLP framework is that it leads to a surprisingly simple synthesis of different complex language constructs. Furthermore, it turns out that – just like for the sum-product and max-product algorithms of graphical models – a single semiring-based algorithm can be used for inference and learning with all these frameworks. While the use of semirings in PLP is not new (Eisner and Filardo, 2010; Kimmig et al., 2017; Orsini et al., 2017), it is the first time that it is used to describe the four frameworks mentioned above in a unified way.

3.2 From Logic Programs to Algebraic Logic Programs

We will first introduce definite clause logic, which forms the basis of logic programming and the programming language Prolog. Afterwards we will consider variations in which facts are labeled, that are used in probabilistic and neural-symbolic logic programs.

3.2.1 Logic Programming

Logic programming is based on definite clauses. These are expressions of the form $h :\!- b_1, ..., b_N$ where h and b_i are logical atoms. A logical atom $a(t_1, ..., t_K)$ consists of a predicatesymbol a of arity K (often denoted a/K), followed by K terms t_i. Terms then are either constants, logical variables, or structured terms of the form $f(t_1, ..., t_L)$ with f a functor and the t_i terms. A clause of the form $h :\!- b_1, ..., b_N$ states that h is true whenever all b_i's are true. When $N = 0$, the clause is a fact and it is assumed to be true. A substitution θ is an expression of the form $\{V_1 = t_1, \ldots, V_N = t_N\}$ where the V_i are different variables and the t_i's are terms. Applying a substitution θ to an expression e (term or clause) yields the instantiated expression $e\theta$ where all variables V_i in e have been replaced by their corresponding terms t_i in e. For example, applying the substitution $\theta = \{X = ann, Y = bob\}$ to `parent(X,Y)` results in `parent(ann,bob)`. When an expression does not contain any variables, it is called ground. Logic programs consist of two main components: 1) a set of facts $\mathcal{F}$ that define the atoms that are considered true, and 2) a set of rules (or clauses) $\mathcal{C}$ that allow the program to derive new atoms from the given set of facts through resolution. A logic program combined with its semantics defines the entailment relationship ($\models$), which defines all the atoms that can be derived

using the given facts and rules. For further details on logic programming, we refer to Flach (1994).

Facts are a basic constituent of logic programs, they represent atoms that are true. We will now show how they can be extended to cope with discrete, continuous, neural, and algebraic labels that form the basis of modern PLP.

3.2.2 Probabilistic Facts

The probabilistic fact is a generalisation of the logic fact, in which the fact is annotated with a probability of being true instead of being deterministically true. This lifts logic programming to probabilistic logic programming (PLP).

Definition 12 (probabilistic fact)**.** *A probabilistic fact is an expression of the form $p :: f$ where f is a ground fact that is true with a probability $p \in [0, 1]$.*

Introducing the probabilistic fact to the logic programming language Prolog resulted in the PLP language ProbLog (De Raedt et al., 2007).

Definition 13 (ProbLog program)**.** *A ProbLog program is a triple $(\mathcal{F}, w, \mathcal{C})$ where $\mathcal{F}$ is a set of probabilistic facts, w is a function mapping each ground probabilistic fact $f \in \mathcal{F}$ to its probability p, its negation $\neg f$ to $1 - p$, and $\mathcal{C}$ is a set of definite clauses. Syntactically, for each probabilistic fact $p :: f$ in $\mathcal{F}$, the symbol f must be unique (with respect to $\mathcal{F}$), and must not appear as the head h of any definite clause in $\mathcal{C}$.*

Example 14 (Bayesian network)**.** *The ProbLog program below models a variant of the well-known sprinkler Bayesian network. It contains three probabilistic facts and three rules.*

```
% Probabilistic facts
0.25 :: cloudy.
0.8  :: humid.
0.5  :: sprinkler.

% Rules
rain :- cloudy, humid.
wet  :- rain.
wet  :- sprinkler.
```

When considering all probabilistic facts in a program, the probability that a set of probabilistic facts $F' \subseteq \mathcal{F}$ is true, and all other facts in the program $(\mathcal{F} \setminus F')$

false, is given by (Kimmig et al., 2017):

$$P_{\mathcal{F}}(F') = \Big(\prod_{f_i \in F'} w(f_i) \Big) \Big(\prod_{f_i \in \mathcal{F} \setminus F'} (1 - w(f_i)) \Big) \tag{3.1}$$

A set of facts $F' \subseteq \mathcal{F}$ of a program, combined with the rules $\mathcal{C}$ again form a deterministic program. In this way, the probabilistic facts induce a probability distribution over all the deterministic programs. The set of all atoms that are entailed to be true from F' and $\mathcal{C}$ together is often referred to as a *possible world*.

Example 15 (possible worlds)**.** *Consider the ProbLog program in Example 14. The set of all ground probabilistic facts $\mathcal{F}$ is* $\{\texttt{cloudy}, \texttt{humid}, \texttt{sprinkler}\}$*. An example F' is* $\{\texttt{cloudy}, \texttt{humid}\} \subset \mathcal{F}$*, which has probability*

$$P_{\mathcal{F}}(F') = (0.25 \times 0.8) \times (1 - 0.5) = 0.1 \tag{3.2}$$

The possible world entailed by F' in the ProbLog program is the set

$$\{\texttt{cloudy}, \texttt{humid}, \texttt{rain}, \texttt{wet}\} \tag{3.3}$$

A different F' would lead to a different possible world; the table below shows all the possible worlds of this ProbLog program along with their probability.

Possible world	$P_{\mathcal{F}}(F')$
{}	*0.075*
{`cloudy`*}*	*0.025*
{`humid`*}*	*0.3*
{`cloudy`*,* `humid`*,* `rain`*,* `wet`*}*	***0.1***
{`sprinkler`*,* `wet`*}*	***0.075***
{`sprinkler`*,* `cloudy`*,* `wet`*}*	***0.025***
{`sprinkler`*,* `humid`*,* `wet`*}*	***0.3***
{`sprinkler`*,* `cloudy`*,* `humid`*,* `rain`*,* `wet`*}*	***0.1***

Definition 14 (success probability)**.** *The success probability $P(G)$ of a query G considers all possible worlds in which G is entailed, and sums their probabilities.*

$$P_{(\mathcal{F},w,\mathcal{C})}(G) = \sum_{\substack{F' \subseteq \mathcal{F} \\ F' \cup \mathcal{C} \models G}} P_{\mathcal{F}}(F') \tag{3.4}$$

$$= \sum_{\substack{F' \subseteq \mathcal{F} \\ F' \cup \mathcal{C} \models G}} \Big(\prod_{f_i \in F'} w(f_i) \Big) \Big(\prod_{f_i \in \mathcal{F} \setminus F'} (1 - w(f_i)) \Big) \tag{3.5}$$

Example 16. *The success probability for query $G =$* `wet` *considers all possible worlds where G is true (those with a bold probability in Example 15's table):* $P(\texttt{wet}) = 0.1 + 0.075 + 0.025 + 0.3 + 0.1 = 0.6$

3.2.3 Neural Facts

Just like the probabilistic fact is a generalisation of the fact, the neural fact is a generalisation of the probabilistic fact. Instead of the probability being fixed, or treated as a single learnable parameter, the probability of the neural fact is parameterized by a neural network. This allows for the introduction of a neural probabilistic logic programming language, which is a type of neural-symbolic integration. One of the simplest forms of the neural fact is where its probability is defined by a neural network binary classifier. Introducing this neural fact to the ProbLog language leads to the introduction of DeepProbLog (Manhaeve et al., 2021a).

Definition 15 (neural fact)**.** *A neural fact is an expression of the form*

$$nn(n_r, [x_1, \ldots, x_k]) :: r(x_1, \ldots, x_k).$$

where r is a predicate symbol, nn is a reserved functor, n_r uniquely identifies a neural network model that defines a probability distribution over the Boolean domain $\{true, false\}$, conditioned on the ground inputs to the neural network $x_1, \ldots, x_k$.

The semantics of the neural fact are defined in terms of the semantics of regular probabilistic facts. A neural fact of the form $nn(n_r, [x_1, \ldots, x_k]) :: r(x_1, \ldots, x_k)$ represents a probabilistic fact $f_{n_r}(x_1, \ldots, x_k) :: r(x_1, \ldots, x_k)$ where f_{n_r} is the function defined by network n_r.

Example 17 (neural fact)**.** *Extending on Example 14, we can use a neural network to predict whether the day will be cloudy, based on additional information that is provided, such as pressure and temperature.*

```
nn(cloudnet, [18°C, 998hPa]) :: cloudy(18°C, 998hPa).
0.8 :: humid.
0.5 :: sprinkler.

% Rules
rain(T,P) :- cloudy(T,P), humid.
wet(T,P) :- rain(T,P).
wet(_,_) :- sprinkler.
```

We can now query this model for the probability of `wet`*, given a certain temperature and pressure:* $P(\texttt{wet}(18°\texttt{C}, 998\ \texttt{hPa}))$.

The concept of the neural fact, which can be used to encode binary classifiers, can be extended to the neural annotated disjunction in order to encode multiclass classifiers. For more details, we refer to Manhaeve et al. (2021a).

3.2.4 Distributional Facts and Indicator Facts

As ProbLog and DeepProbLog only allow for (neural) probabilistic facts, they are inherently restricted to discrete random variables. We alleviate this by introducing so-called distributional facts and indicator facts[1] (Zuidberg Dos Martires et al., 2023).

Definition 16 (distributional fact)**.** *A distributional fact is of the form* $x \sim d$*, with* x *being a ground term, and* d *a ground term whose functor denotes a probability distribution. The distributional fact states that the ground term* x *is a random variable distributed according to* d*.*

Definition 17 (indicator fact)**.** *An indicator fact is an expression of the form* $\sigma :: f$ *where* f *is a ground fact labeled with a measurable set* σ*.*

Similar to probabilistic facts in regular ProbLog programs (cf. Definition 13), we require from each indicator fact $\sigma :: f$ that f is unique across all facts, and does not occur in the head of any clause. Also for each distributional fact $x \sim d$, x must be unique.

Example 18. *We rewrite the program in Example 14 using distributional and indicator facts. Note how the program separates into two layers. On the one hand, we have the distributional facts that define how random variables are distributed. On the other hand, we have logical rules. The link between the two layers is made by the indicator facts that are each labeled with a measurable set using the random variables introduced by the distributional facts.*

```
% Distributional facts
x_c ~ flip(0.25).
x_h ~ flip(0.8).
x_s ~ flip(0.5).

% Indicator Facts
```

[1] Zuidberg Dos Martires et al. (2023) implicitly introduced indicator facts by means of Boolean comparison predicates.

```
7  [x_c = 1]::cloudy.
8  [x_h = 1]::humid.
9  [x_s = 1]::sprinkler.
10
11 % Rules
12 rain :- cloudy, humid.
13 wet  :- rain.
14 wet  :- sprinkler.
```

Example 19. *In Example 14 and Example 18 we modeled the humidity as a Boolean random variable. Extending ProbLog with continuous random variables allows us now to model the humidity as a continuous variable. Using, for instance, a beta distribution, we model the relative humidity as a random variable that takes values in the* $[0, 1]$ *interval.*

```
1  % Distributional facts
2  x_c ~ flip(0.25).
3  x_h ~ beta(4,2).
4  x_s ~ flip(0.5).
5
6  % Indicator Facts
7  [x_c = 1]::cloudy.
8  [x_h > 0.6]::humid.
9  [x_s = 1]::sprinkler.
10
11 % Rules
12 rain :- cloudy, humid.
13 wet  :- rain.
14 wet  :- sprinkler.
```

As we model the humidity as a continuous random variable we use an inequality instead of an equality in Line 8.

We now define the probability of a query G being true in the distributional program, similar to Definition 14. This requires a new definition due to the indicator facts, previously probabilistic facts, that are each associated with an indicator function $\sigma(\cdot)$ rather than a probability w. An additional expectation operator is needed to obtain a probability (cf. Definition 18) because, due to the indicator functions, the result otherwise would merely represent a set of random events that are described through those indicators. We illustrate this in Example 21.

Definition 18 (expected success value (Zuidberg Dos Martires et al., 2023))**.** *The expected success probability $P(G)$ of a query G is given by*

$$P_{(\mathcal{F},\sigma,\mathcal{C})}(G) = \mathsf{E}\left[\sum_{\substack{F' \subseteq \mathcal{F} \\ F' \cup \mathcal{C} \models G}} \left(\prod_{f_i \in F'} \sigma(f_i)\right)\left(\prod_{f_i \in \mathcal{F} \setminus F'} \sigma(\neg f_i)\right)\right] \tag{3.6}$$

where $\sigma(\cdot)$ is a function that maps indicator facts to measurable indicator functions.

Example 20 (indicator functions)**.** *The function $\sigma(\cdot)$ in Definition 18 maps the indicator facts from Example 18 to indicator functions as follows:*

$$\sigma(\texttt{cloudy}) = [\![x_c = 1]\!] \qquad \sigma(\neg\texttt{cloudy}) = [\![x_c = 0]\!]$$

$$\sigma(\texttt{humid}) = [\![x_h > 0.6]\!] \qquad \sigma(\neg\texttt{humid}) = [\![x_h \leq 0.6]\!]$$

$$\sigma(\texttt{sprinkler}) = [\![x_s = 1]\!] \qquad \sigma(\neg\texttt{sprinkler}) = [\![x_s = 0]\!]$$

Here we use Iverson brackets $[\![\cdot]\!]$ to denote the indicator function: whenever the relation inside an Iverson bracket holds it evaluates to 1. Otherwise, it evaluates to 0.

Example 21. *Using Definition 18 and the fact that we can interchange the expectation and the summation (Miosic and Zuidberg Dos Martires, 2021; Zuidberg Dos Martires et al., 2023; Zuidberg Dos Martires et al., 2019b), we can compute the success probability for the query $G = \texttt{wet}$ in a similar fashion to Example 16. For the world $\{\texttt{cloudy}, \texttt{humid}, \texttt{rain}, \texttt{wet}\}$ we get:*

$$\begin{aligned}\mathsf{E}\Big[[\![x_c = 1]\!][\![x_h > 0.6]\!][\![x_h = 0]\!]\Big] &= \mathsf{E}\Big[[\![x_c = 1]\!]\Big]\mathsf{E}\Big[[\![x_h > 0.6]\!]\Big]\mathsf{E}\Big[[\![x_h = 0]\!]\Big] \\ &= 0.25 \times 0.66304 \times 0.5 = 0.08288\end{aligned}$$

Note that the expectation operator is necessary to map the random event (encoded with Iverson brackets) to a probability. We can compute in a similar fashion the probabilities of the remaining worlds.

The program in Example 19 is rather simple in terms of functional dependencies between random variables. For instance, none of the parameters of any of the distributions depends on the parameters of another random variable. Furthermore, the labels of the indicator facts are all univariate. This means, effectively, that all the random variables are independent of each other, which allowed us to break down the expectation in Example 21. Note however that

Definition 18 is more general and does allow for functional dependencies between random variables (e.g., when the mean of a normal distribution is a random variable itself) and also for multivariate labels of indicator facts (e.g., when an indicator depends on more than one random variable). We refer the interested reader to Zuidberg Dos Martires et al. (2023) for an in-depth and formal exposition of such cases. We also note that De Smet et al. (2023) generalize distributional facts to so-called *neural distributional facts*, which allows them to unify neural and discrete-continuous PLP. The main idea is that parameters of distributional facts are allowed to be the output of neural networks.

3.2.5 Algebraic Facts

The previously introduced concepts of probabilistic fact, neural fact, and indicator fact can each be generalised into an *algebraic fact*, a concept first discussed in aProbLog (Kimmig et al., 2011).

Definition 19 (algebraic fact)**.** *An algebraic fact is an expression of the form $a :: f$ where f is a fact and a is an element of a commutative semiring's domain.*

Recall from Chapter 2, Definition 3, that the commutative semiring is an algebraic construct that defines a multiplication $\otimes$ and addition $\oplus$ operation over a domain of values. As such, it defines the actual computation that is executed for a given formula. The properties of the commutative semiring ensure that the calculated result is correct. As shown in Kimmig et al. (2011), using a different semiring amounts to solving a different task that used to be considered separately. We take this idea further and show how using different semirings allows a single framework to generalize a variety of frameworks that were originally considered to be distinct.

In this generalisation a program has four components $(\mathcal{F}, w, \mathcal{C}, \mathcal{S})$ where $\mathcal{S}$ is a commutative semiring and w is a function mapping both facts f and their negation $\neg f$ to elements of $\mathcal{S}$'s domain $\mathcal{A}$. In contrast to the weight function w in a regular ProbLog program, the weight of $\neg f_i$ may be different from $1 - w(f_i)$. The inference task within such a program is generalised to Equation 3.7. Note the use of the semiring operations $\otimes$ and $\oplus$, which replaced $\times$ and $+$. We also replaced $P_{(\mathcal{F},w,\mathcal{C})}(G)$ with $AMC_{(\mathcal{F},w,\mathcal{C},\mathcal{S})}(G)$ since it is no longer necessarily a probability.

$$AMC_{(\mathcal{F},w,\mathcal{C},\mathcal{S})}(G) = \bigoplus_{\substack{F' \subseteq \mathcal{F} \\ F' \cup \mathcal{C} \models G}} \left(\bigotimes_{f_i \in F'} w(f_i) \right) \left(\bigotimes_{f_i \in \mathcal{F} \setminus F'} w(\neg f_i) \right) \tag{3.7}$$

Example 22. *Consider the program in Example 14. When using the semiring $(\mathbb{R}, max, \times, 0, 1)$, the output of the query instead becomes the most probable*

explanation. Indeed, $\otimes = \times$ means that the weight of a model is still its probability, and by choosing $\oplus = max$ the most probable model is selected.

While the example above is also solving a probabilistic task, the algebraic framework is certainly not restricted to probabilities alone. In fact, the semiring elements $\mathcal{A}$ can be anything as long as we define the proper operations over them: preference values, distances, weights, tuples, sets, ... Several other extensions built around this framework include reasoning over second-order queries (Verreet et al., 2022a), decision making via the expected utility semiring (Derkinderen and De Raedt, 2020) as shown in Chapter 5, and parameter learning via the gradient semiring (Kimmig et al., 2011). More information on the latter is provided in Section 3.4.

Comparing Equation 3.7 to Equation 3.6 we see that the probability of a query to a discrete-continuous probabilistic program can be formulated in terms of an algebraic model count as well. Indeed, the structure within the expectation operator in $P_{(\mathcal{F},\sigma,\mathcal{C})}(G)$ (Equation 3.6) is equivalent to the structure of an AMC call (Equation 3.7), resulting in:

$$P_{(\mathcal{F},\sigma,\mathcal{C})}(G) = \mathbb{E}\left[AMC_{(\mathcal{F},\sigma,\mathcal{C},\mathcal{S})}(G)\right] \tag{3.8}$$

A similar formulation is also used by De Smet et al. (2023), Miosic and Zuidberg Dos Martires (2021), Zuidberg Dos Martires et al. (2023), and Zuidberg Dos Martires et al. (2019b).

The necessity of the semiring properties originates from the AMC equation, which inherently does not impose any restrictions on the ordering of the $\oplus$ and $\otimes$ operations: commutativity and associativity must therefore hold for both. Furthermore, the distributivity property, together with the aforementioned properties, enables the equation to be factorized which allows it to be computed more efficiently. In contrast, systems wherein the operation ordering does matter operate with more procedural semantics. As such, the aforementioned procedure using AMC does not work as is. As an example we refer to Orsini et al. (2017), where, when the semiring properties are violated, the semantics rely on the ordering of the (Prolog) rules.

3.3 Inference

We now discuss inference for algebraic logic programs. Inference happens in three steps: 1) logical inference, 2) translation to an algebraic model counting problem, 3) calculating the algebraic model count. This pipeline is illustrated in Figure 3.1. We now discuss each step in detail.

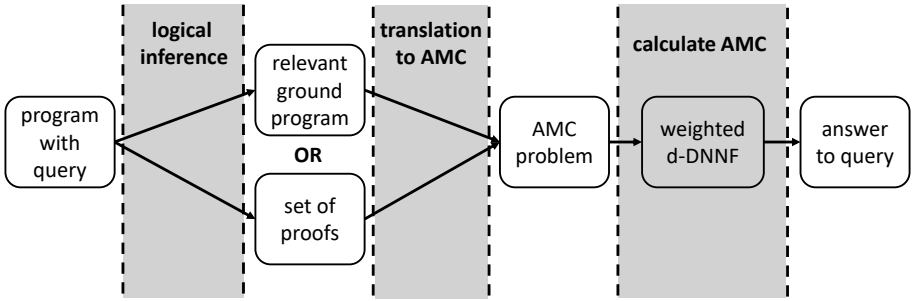


Figure 3.1: The three step pipeline to perform inference in algebraic logic programs, consisting of logical inference (Section 3.3.1), translation to an AMC problem (Section 3.3.2), and finally, solving that problem to obtain the query answer (Section 3.3.3).

3.3.1 Logical Inference

The first step concerns the logical inference, for which we discern two different but related approaches.

The first approach is proving, which uses SLD resolution to calculate the set of all proofs for a query G. SLD resolution uses a backward chaining, goal-oriented approach. A goal is a sequence of atoms $?\text{–}\, l_1, ..., l_n$. The initial goal is the query. At each step, the algorithm chooses a clause $h \text{:–}\ b_1, \ldots, b_n$ whose head h unifies with the first atom l_1 in the goal, with the substitution θ, i.e. $h\theta = l_1\theta$. The application of resolution yields a new goal, $?\text{–}(b_1, \ldots b_n, l_2, \ldots, l_j)\theta$. This is repeated until the goal is empty, resulting in a successful proof, or until no more clauses can be applied, in which case the proof fails. It is possible that multiple clauses can be applied to a goal, which leads to different branches in the SLD tree and the possibility of multiple proofs for a single query.

The second approach is to construct the relevant ground program $\mathcal{P}_G$. Grounding replaces each rule c containing variables $\{V_1, ..., V_k\}$ by all instances $c\theta$ where θ is a substitution $\{V_1 = c_1, ...V_k = c_k\}$ and the c_i are constants or other ground terms appearing in the domain. If G is not ground, the grounding will compute all possible answer substitutions $G\theta$. To keep inference tractable, it is key to only consider the part of the ground program that is relevant to the query (i.e. the grounded facts and rules are used in the derivations of the query). Again, SLD resolution is used to find the relevant grounding.

Example 23 (logical inference)**.** *We demonstrate the different steps of inference by extending Example 14 to reason about separate days of the week. For ease of modeling (and grounding), we also use rules annotated with probabilities, a*

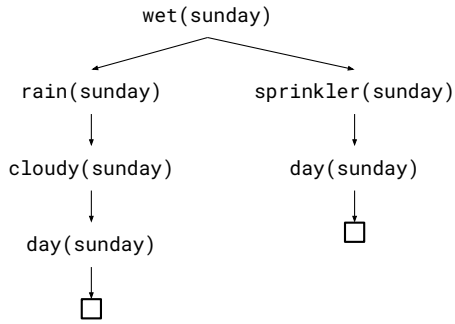


Figure 3.2: The SLD tree for Example 23. The two branches represent the two separate proofs for the query `wet(sunday)`.

purely syntactical construct[2].

```
day(monday).
...
day(sunday).

0.25 :: cloudy(Day) :- day(Day).
0.5 :: sprinkler(Day) :- day(Day).

0.8 :: rain(Day) :- cloudy(Day).

wet(Day) :- rain(Day).
wet(Day) :- sprinkler(Day).
```

We query the probability of the grass being wet on Sunday, i.e. $P($`wet(sunday)`$)$. *We first consider the proving approach. The proving procedure is easily visualized as an SLD tree, shown in Figure 3.2. There are two proofs for our query, one where the grass is made wet by the sprinkler, and one where it is rainy and cloudy.*

For the grounding approach, we only consider the part that is relevant to the query `wet(sunday)`*, so the variable* `Day` *only needs to be substituted with* `sunday`*. The resulting relevant ground program is:*

[2]For example, the rule `0.25 :: cloudy(sunday) :- day(sunday)` is syntactic sugar for a rule `cloudy(sunday) :- cloudy_on_day(sunday), day(sunday)` and a fact `0.25 :: cloudy_on_day(sunday)`.

```
day(sunday).

0.25 :: cloudy(sunday) :- day(sunday).
0.5 :: sprinkler(sunday) :- day(sunday).

0.8 :: rain(sunday) :- cloudy(sunday).

wet(sunday) :- rain(sunday).
wet(sunday) :- sprinkler(sunday).
```

3.3.2 Translation to Algebraic Model Counting

In the next step of the inference we map our results onto propositional logical formulas.

It is important to realise that the possible worlds of a ProbLog program discussed in Section 3.2.2 are simply models of a theory formed by that program. Considering this equivalence, the probability query $P(G)$ (Equation 3.4) can be identified as an instance of the weighted model counting problem. Consequently, the problem of computing $P(G)$ can be addressed by transforming it into a weighted model counting problem for which several solvers exist.

Example 24 (weighted model count)**.** *Consider the propositional theory ψ and weight function w below. The models of ψ, denoted as R_ψ, correspond exactly to the possible worlds of Example 14. Additionally, because we have chosen the weights w appropriately, the weighted model count is the probability of* `wet` *being true, $WMC(\psi, w) = 0.6$.*

$$\psi = \big(\texttt{rain} \iff \texttt{cloudy} \wedge \texttt{humid}\big) \wedge \big(\texttt{wet} \iff \texttt{rain} \vee \texttt{sprinkler}\big)$$

$$\begin{aligned} w = \{&\texttt{cloudy} \mapsto 0.25, \neg\texttt{cloudy} \mapsto 0.75, \texttt{humid} \mapsto 0.8, \neg\texttt{humid} \mapsto 0.2, \\ &\texttt{sprinkler} \mapsto 0.5, \neg\texttt{sprinkler} \mapsto 0.5, \texttt{rain} \mapsto 1, \neg\texttt{rain} \mapsto 1, \\ &\texttt{wet} \mapsto 1, \neg\texttt{wet} \mapsto 0\} \end{aligned}$$

Similarly, the algebraic query (Equation 3.7) can be identified as an instance of algebraic model counting.

Example 25 (algebraic model count)**.** *(Derkinderen and De Raedt, 2020) Consider Example 24 but using semiring $\mathcal{S} = (\mathbb{R}, max, \times, 0, 1)$, then $AMC(\psi, \mathcal{S}, \alpha) = 0.3$, the highest model weight out of all models in R_ψ.*

The construction of the logical formula itself depends on how the previous inference step was performed. In the proving approach, the proofs for a query G are combined into a logical formula

$$G \leftrightarrow \bigvee_{E \in \text{Proofs}(G)} \bigwedge_{f_i \in E} f_i$$

where $f_i \in E$ are the probabilistic facts used in proof E.

In the grounding approach, Clark's completion can be used for cycle-free programs. Clark's completion constructs a formula

$$h \leftrightarrow \bigvee_{(h :- b_1, \ldots, b_n) \in \mathcal{P}_G} b_1 \wedge \cdots \wedge b_n$$

for each set of rules with the same ground head h, whose bodies are $b_1, \ldots, b_n$. Cyclical programs first need to be turned into equivalent acyclic programs through cycle breaking. For this, we refer to Fierens et al. (2015). The propositional theory ψ in Example 24 would for instance follow from this grounding approach.

3.3.3 Solving Model Counting

The process of efficiently computing the weighted model count has already been extensively discussed in Chapter 2, and will be touched upon more in Chapter 4 and 5. A major functional advantage of the compilation approach that ProbLog uses, and that is discussed in Section 2.4, is that the computational graph can be re-used for several queries, or with a different w. In this way the largest computational cost is amortized, and re-evaluation is linear in the graph size. This is especially useful for parameter learning as then only w varies, e.g., learning in DeepProbLog (Manhaeve et al., 2021a).

Example 26 (sd-DNNF)**.** *Figure 3.3 shows the ProbLog program of Example 14 in sd-DNNF, the theory ψ of which was already illustrated in Example 24. The corresponding representation of $WMC(\psi)$ is illustrated in Figure 3.4.*

Importantly, the ability to easily extend the procedure to algebraic model counting allows ProbLog to generalise to other inference tasks, and to other language variants.

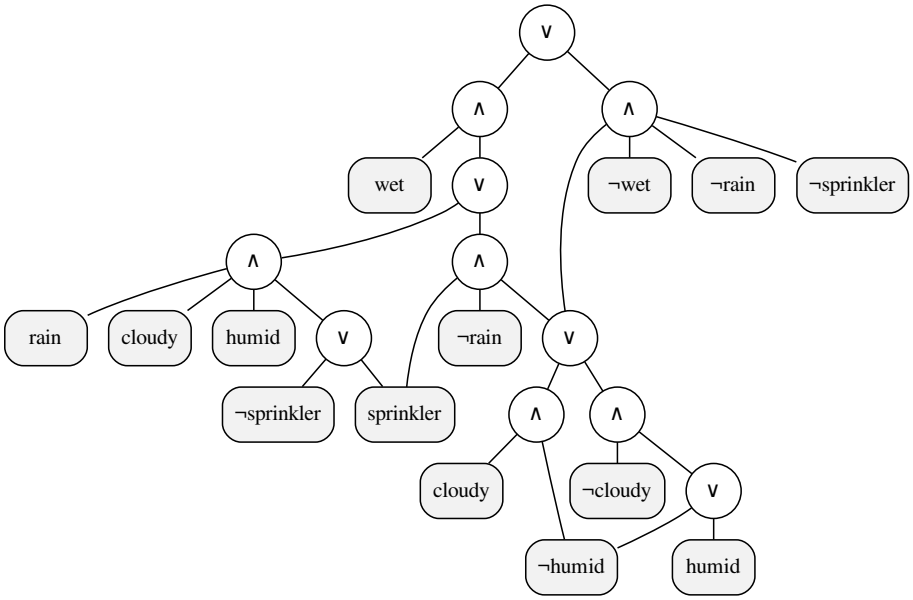


Figure 3.3: An sd-DNNF corresponding to the ProbLog program in Example 14.

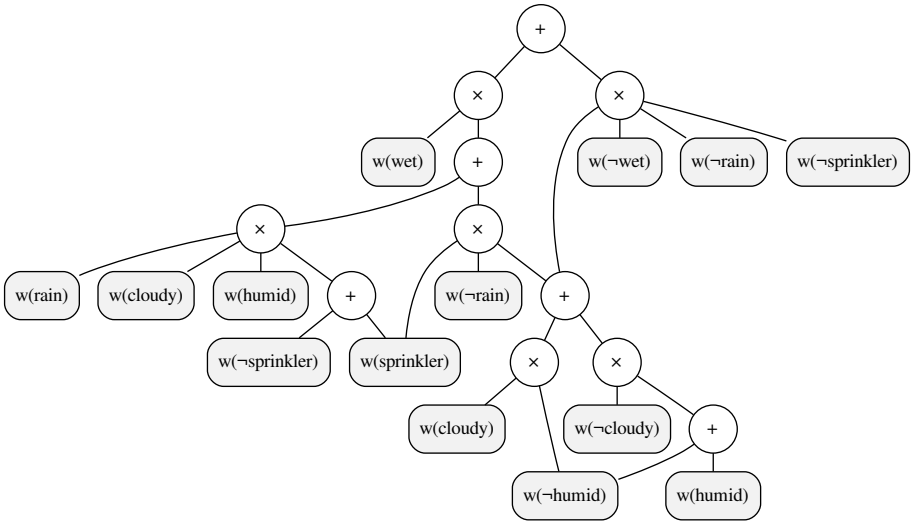


Figure 3.4: A $WMC(\psi)$ representation of Figure 3.3.

3.4 Learning

The generalisations of the fact as discussed in Section 3.2 can also introduce parameters to the logic program. For probabilistic facts, the probability itself can be learned, and would thus be a parameter of the model. Similarly, the distributions in the distributional facts can have learnable parameters, e.g. the mean and standard deviation of a normal distribution. For the neural predicate, the weights of the neural networks are usually considered to be learnable parameters.

We now consider the parameter learning setting. Given a program with parameters $\mathcal{W}$, a set $\mathcal{Q}$ of tuples (G, p) with G a query and p the target probability, and a loss function $\mathcal{L}$, compute:

$$\arg\min_{\mathcal{W}} \frac{1}{|\mathcal{Q}|} \sum_{(G,p)\in\mathcal{Q}} \mathcal{L}(P_{\mathcal{W}}(G), p) \tag{3.9}$$

Earlier approaches to parameter learning used an expectation-maximization approach. Recently, gradient-based optimization has become the dominant strategy for learning. AMC enables us to automatically derive gradients for the parameters in the program through the use of the gradient semiring, which we explain in Section 3.4.1. After the gradients have been calculated, standard gradient-based optimizers can be used. When the parameters are contained in differentiable structures (e.g. in a neural network), they are easy to optimize in conjunction with other parameters, as the same gradient-based techniques can be used.

3.4.1 Gradient Semiring

To derive gradients, we use the gradient semiring (Kimmig et al., 2011). The elements of this semiring are tuples

$$\left(p, \frac{\partial p}{\partial x}\right)$$

where p is a probability, and $\frac{\partial p}{\partial x}$ is the partial derivative of that probability with respect to a parameter x. This is easily extended to a vector of parameters $\vec{x} = [x_1, \ldots, x_N]^T$, the concatenation of all N parameters in the ground program. The elements of the semiring then become tuples

$$(p, \nabla p)$$

where p is a probability and ∇p the gradient of p with respect to all parameters in $\vec{x}$. Addition $\oplus$, multiplication $\otimes$ and the neutral elements with respect to these operations are defined as follows:

$$(p_1, \nabla p_1) \oplus (p_2, \nabla p_2) = (p_1 + p_1, \nabla p_1 + \nabla p_2) \tag{3.10}$$

$$(p_1, \nabla p_1) \otimes (p_2, \nabla p_2) = (p_1 p_2, p_2 \nabla p_1 + p_1 \nabla p_2) \tag{3.11}$$

$$e^{\oplus} = (0, \vec{0}) \tag{3.12}$$

$$e^{\otimes} = (1, \vec{0}) \tag{3.13}$$

Note that the first element of the tuple performs ProbLog's probability computation, whereas the second element computes the gradient of the first element.

To perform parameter learning in ProbLog, we use the following mapping:

$$w(f) = (p, \vec{0}) \qquad \text{for } p :: f \text{ with fixed } p \tag{3.14}$$

$$w(f_i) = (p_i, \mathbf{e}_i) \qquad \text{for } t(p_i) :: f_i \text{ with learnable } p_i \tag{3.15}$$

$$w(\neg f) = (1 - p, -\nabla p) \qquad \text{with } w(f) = (p, \nabla p) \tag{3.16}$$

where the vector $\mathbf{e}_i$ has a 1 in the i-th position and 0 in all others.

Example 27. *We demonstrate the joint learning of probabilistic parameters and the neural network's parameters for the program in Example 17. For this example, we learn the parameter of* `humid` *and jointly train the* `cloudnet` *network. We use cross-entropy as our loss function* $\mathcal{L} = -(p \log(P(G)) + (1 - p) \log(1 - P(G)))$*, with* p *the target probability and* $P(G)$ *the probability predicted using the current weight parameters. Since the target probability is* 1*,* $\mathcal{L}$ *is* $-\log(P(G))$*. To update the parameters, we need to compute*

$$\frac{\partial \mathcal{L}}{\partial \mathcal{X}} = \frac{\partial \mathcal{L}}{\partial P(G)} \frac{\partial P(G)}{\partial \mathcal{X}} = \frac{-1}{P(G)} \frac{\partial P(G)}{\partial \mathcal{X}}$$

To compute this gradient, we need both $P(G)$ *and* $\frac{\partial P(G)}{\partial \mathcal{X}}$*, which we can calculate using the gradient semiring. The resulting arithmetic circuit after all stages of inference is given in Figure 3.5. The final gradients are thus* $\frac{\partial \mathcal{L}}{\partial \texttt{humid}} = -0.41$ *and* $\frac{\partial \mathcal{L}}{\partial \texttt{cloudy(18°C,998 hPa)}} = -0.54$*.*

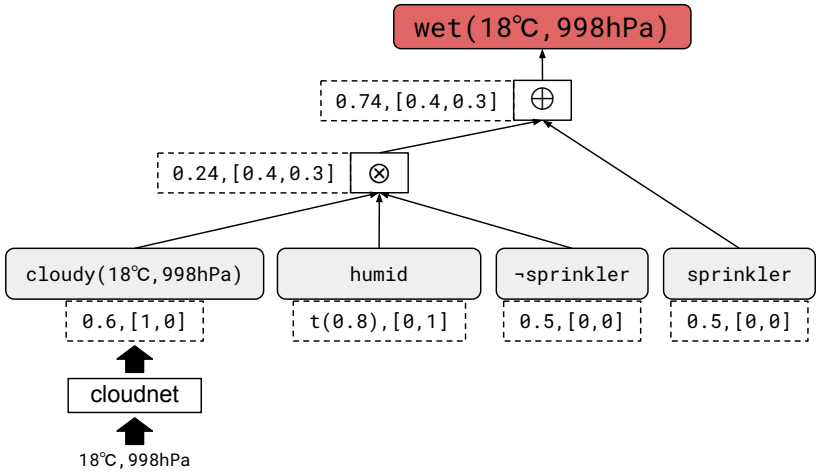


Figure 3.5: The arithmetic circuit for Example 27. Each node is annotated with an element from the gradient semiring, where the two elements of the gradient represent the partial derivative of the probability for the output with respect to the neural network, and the probabilistic parameter respectively.

3.5 Related Work and Applications

There a many more variants of the logical fact introduced in related work. We discuss a few more in this section. DTProbLog (Van den Broeck et al., 2010) is a decision-theoretic variant of ProbLog that adds the decision fact, along with the possibility of assigning utilities to atoms. The probabilistic facts, rules, and utilities define an expected utility given a set of values for the decision facts. These values thus define a decision problem that needs to be solved to maximize the expected utility. BetaProbLog (Verreet et al., 2022a) further generalizes the concept of the probabilistic fact by replacing the single probability (i.e. a point estimate) with a beta distribution which additionally models the epistemic uncertainty over the probability of the fact. NeurASP (Yang et al., 2020) takes the idea of the neural fact and applies it to answer set programming, a different logic programming language where it is called the neural atom. In Belle and De Raedt (2020), the authors introduce *semiring programming*, a declarative framework where semirings are used to model and solve a wide variety of tasks in AI. In Scallop (Huang et al., 2021), semirings are used to define approximate inference to alleviate the intractability that methods such as DeepProbLog encounter. In (Zuidberg Dos Martires, 2021), the authors investigate the setting in which the semiring operations are functions that have to be learned. Other work involving semirings and their application onto functional programming

includes Dolan (2013) and Berg et al. (2022).

An application domain well suited for algebraic and probabilistic logic programming is the field of robotics. Here, the probabilistic modelling and reasoning capabilities have been used for object tracking, affordance, and object manipulation (Antanas et al., 2019; Moldovan et al., 2012a, 2018, 2012b, 2011; Nitti et al., 2014; Persson et al., 2020; Zuidberg Dos Martires et al., 2020), for representing and tracking cognitive knowledge about the environment (Mekuria et al., 2019; Veiga et al., 2019; Yang et al., 2023), and for performing the uncertain decision making itself (Bueno et al., 2016; Derkinderen and De Raedt, 2020; Latour et al., 2017; Nitti et al., 2015, 2017; Van den Broeck et al., 2010; Venturato et al., 2022).

Other application tasks include activity recognition (McAreavey et al., 2017; Skarlatidis et al., 2015; Smith et al., 2021; Sztyler et al., 2018), consistency-based diagnosis (Hommersom and Bueno, 2016), modeling incomplete and imprecise information (Doherty and Szalas, 2022), system prognostics (Vlasselaer and Meert, 2012), ontology matching and querying (van Bremen et al., 2019, 2020; Wang, 2015), probabilistic argumentation (Hung, 2017; Mantadelis and Bistarelli, 2020; Totis et al., 2021), solving word-problems (Dries et al., 2017; Suster et al., 2021), automating video montages (Aerts et al., 2016), epidemiological modelling (Weitkämper et al., 2021), game-playing (Thon et al., 2008, 2011), event processing (Apriceno et al., 2021; Roig Vilamala et al., 2023; Xing et al., 2019), modelling probabilistic routing networks (Berg et al., 2021), and biology (De Maeyer et al., 2013, 2016, 2015; De Raedt, 2007; Groß et al., 2019; Kimmig and Costa, 2012).

3.6 Conclusion

In this chapter we provided an overview and synthesis, contributing a unified algebraic perspective on PLP that describes how logic programming can be extended to a wide variety of settings by generalizing the concept of the fact. We have shown how these extensions are all special cases of the concept of the algebraic fact, where facts are labeled with elements from commutative semirings, and the conjunction and disjunction are replaced with multiplication and addition respectively. We have further shown a recipe for efficient inference and learning for programs that include such algebraic facts. Finally, we have discussed other works that perform similar extensions, and where these systems have been applied. Going forward, it would be valuable to look into what benefits the use of semirings has for other languages and programming paradigms.

Chapter 4

Exploiting Symmetry for Model Counting

A large part of this chapter was previously published as:

> T. van Bremen, V. Derkinderen, S. Sharma, S. Roy, and K. S. Meel (2021). "Symmetric Component Caching for Model Counting on Combinatorial Instances". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 3922–3930

Timothy van Bremen, Shubham Sharma, and I, have each contributed equally to the publication. Timothy and I conceived the idea and created a prototype. During the development process we found the other now co-authors had independently worked on this as well. We collaborated, implementing the idea within GANAK and have equally contributed to the publication writing.

Parts of the conference article have been restructured and rewritten to function as a chapter in this dissertation. The discussion on applying our contributions to projected model counting and weighted model counting is a novel addition.

Model counting is a fundamental problem in artificial intelligence with a wide variety of applications such as probabilistic inference (cf. the previous chapter and Chavira and Darwiche (2008)), neural network verification (Baluta et al., 2019), computational biology (Sashittal and El-Kebir, 2020), and the like. Consequently, the problem of model counting has been subject to intense

theoretical and practical investigations over the past four decades. The seminal work of Valiant (1979b) showed that model counting is #P-complete. Despite this complexity result, many problem instances are feasible in practice due to the ability of modern model counting algorithms to exploit the structure present in those instances.

In this chapter we increase the amount of exploitable structure by taking advantage of certain structural symmetries present within the instance formula ψ. We explain how these symmetries can be detected and exploited through a minor change in the #DPLL model counting algorithm that was introduced in Chapter 2. This change reduces the counting search space, thereby improving the algorithm's run time (when many symmetries are present).

Our contribution answers the question:

> **RQ2.1: How to exploit structural symmetry while model counting on propositional logic formulas?**

The remainder of this chapter is organized as follows: we first situate our contribution on a high-level in Section 4.1, and present the related work in Section 4.2. Notations and preliminaries are presented in Section 4.3, and the primary technical contribution, i.e., symmetric component caching, is presented in Section 4.4 and 4.5. Afterwards, we present an empirical analysis of our contribution in Section 4.6 and finally conclude in Section 4.7.

4.1 Introduction

Practical strategies for model counting span a variety of approaches, from approximate techniques (Soos and Meel, 2019; Stockmeyer, 1983) with probabilistic error bounds, to exact counting (Aziz et al., 2015; Bayardo Jr and Pehoushek, 2000; Birnbaum and Lozinskii, 1999; Lagniez and Marquis, 2017; Oztok and Darwiche, 2015; Sang et al., 2004; Thurley, 2006). Many solvers use variants of the classic DPLL algorithm for SAT solving (Davis et al., 1962), with optimizations geared towards model counting (Birnbaum and Lozinskii, 1999). One prominent optimization used in such algorithms is *component caching*: during the search process subsets of clauses that can be solved independently (referred to as *components*) are identified, solved, and cached. When the same component appears again along a different search path, the model count of the component can simply be returned from the cache, alleviating the need to recompute it (Bacchus et al., 2003).

The exact representation scheme used for storing components in the cache differs between solvers: Cachet (Sang et al., 2004, 2005) uses a simple encoding where the literals in each clause are represented as integers with clauses separated by a sentinel. sharpSAT (Thurley, 2006) uses a *hybrid encoding* that achieves a more compact representation. D4 expanded on the idea of compact representations, obtaining even better results (Lagniez and Marquis, 2021). Ganak (Sharma et al., 2019) introduced the notion of a *probabilistic* cache: the component encodings are hashed into a yet smaller representation to enable better cache utilization but, in the process, paying a price with a (small) probability of incorrect counts due to hash collisions. The algorithm is parametrized by the probability of collision, which can be set as small as the user desires, at the expense of poorer cache utilization due to longer hash lengths. In addition, Ganak adds several other optimizations that allowed it to significantly outperform other state-of-the-art model counters.

However, all existing cache indexing schemes (including that of Ganak) declare a cache hit only on *exact* matches on components. We make an important observation that there are components that are structurally identical but differ only in the variables appearing in the formula. Because components employ variables disjoint from the rest of the formula, the model counts can also be transferred across such structurally identical components. Such *symmetric components* occur naturally in many instances, particularly those arising from combinatorial problems. It is worth remarking that the counting variants of many combinatorial problems also enjoy straightforward reductions to #SAT, such as n-queens, quasigroup (Latin square) completion, and graph k-colouring (Aloul et al., 2002; Gomes and Shmoys, 2002; Lauria et al., 2017; Wang et al., 2020; Yang, 1991).

Our primary contribution is exploiting the inherent symmetry exhibited in combinatorial problems for component caching-based model counters. To this end, we propose and formalise the notion of *symmetric component caching*—allowing for the use of cached model counts even across components that are only structurally identical (*symmetric*) and not exact matches. We first prove that the proposed scheme is sound when combined with clause learning. Then, we augment the state-of-the-art counter Ganak with *symmetric component caching*, along with several low-level but crucial technical improvements. The resulting counter, called SymGanak, outperforms the state-of-the-art model counter Ganak on PAR-2 score and number of instances solved, achieving significant performance gains in terms of run time.

4.2 Related Work

We are not the first to explore symmetry in propositional logic: the use of precomputed symmetry-breaking predicates to speed up SAT solving dates back to 1996 (Crawford et al., 1996). More recent work has extended this idea with more efficient symmetry-breaking formulas (Devriendt et al., 2016). Taking a different approach, others have examined how symmetry information can be used at run time for SAT solvers (Sabharwal, 2009).

Outside of SAT solving, Kitching and Bacchus (2007) explored symmetry in the context of solving constraint optimization problems with decomposable objective functions. Salmon and Poupart (2019) exploited symmetry while solving partially observable Markov decision processes, which were encoded as stochastic satisfiability problems. Due to their setting, they used a different encoding to detect symmetries. Bart et al. (2014) exploited symmetry to achieve space savings in knowledge compilation. We also note that a rich literature on symmetry exists in the adjacent domain of *lifted inference*, in which the aim is to develop algorithms that exploit symmetries in graphical models to speed up probabilistic inference. Although many such algorithms assume a relational representation of the input, several approaches do target non-relational input models; see, for example, Bui et al. (2013), Holtzen et al. (2019), and Niepert (2012).

In the context of propositional model counting specifically, Wang et al. (2020) studied the use of existing model counting algorithms on formulas conjoined with symmetry-breaking predicates, thus effectively counting models up to isomorphism. SymGanak differs in that it counts all models of the formula, and does not rely on symmetries of the input formula itself—rather, it exploits symmetry amongst the components encountered *during run time* of the algorithm. Note that, in principle, this does not require symmetry to be present in the input formula for our approach to be effective: if a variable ordering can be chosen in such a way that propagating variables in this order leads to structurally identical components, this will suffice to see performance gains over existing counters. We compare different variable ordering heuristics later in this paper and also examine the implications of our approach when integrating with many of the features (such as clause learning) present in modern model counters.

4.3 Background

Before proceeding to the primary contribution of this chapter we list the needed preliminaries, the majority of which is already provided in Chapter 2. We specifically, however, remind the reader of the concepts related to components and component decompositioning. Afterwards, we explain a few notions related to graph theory (Section 4.3.2) that are key to detecting structural symmetries in formulas.

As is common in many model counters, we assume the input propositional formula ψ to be in conjunctive normal form (cf. Chapter 2).

4.3.1 #DPLL with Component Caching

The concepts of components and component decompositioning were both previously introduced in Section 2.3.2. As a reminder, component decompositioning partitions the formula ψ into a set of components $\mathcal{C}$ such that the components do not share any variables with each other. These components can be solved (i.e., counted) independently, resulting in a more shallow search process. A #DPLL-based model counter that implements component caching furthermore caches the model counts from each solved component such that the results can be reused when the same component is encountered again later on.

Algorithm 4 illustrates a #DPLL algorithm with component caching. This algorithm is similar to the one shown in Chapter 2 but makes explicit the encoding call that occurs while caching ($\mathsf{Encode}(\psi)$ in line 2). Designing efficient encodings for the components has been an important research direction (Sang et al., 2004, 2005; Sharma et al., 2019; Thurley, 2006). More compact encodings improve cache utilization, allowing the memoization of more components for a given cache size. Our contribution will alter the encoding.

4.3.2 Isomorphism

Definition 20 (coloured graph)**.** *A coloured graph is a three-tuple* $G = (V, E, P)$*, where* (V, E) *specifies an undirected graph and* $P = \{V_i\}_{i=1}^{k}$ *is a partition of the vertices into* k *distinct colours. We further denote* $\mathsf{colour}(v) = i$ *if* $v \in V_i$*.*

Given two coloured graphs, one can ask if they are *isomorphic*.

Algorithm 4: #DPLL algorithm with component caching.

```
1  function GetModelCount(ψ):
2      encoding ← Encode(ψ)
3      if encoding in cache then
4          return CacheGet(encoding)
5      else
6          pick a literal l in ψ
7          |R_{ψ_l}| ← CountConditioned(ψ, l)
8          |R_{ψ_¬l}| ← CountConditioned(ψ, ¬l)
9          CacheInsert(encoding, |R_{ψ_l}| + |R_{ψ_¬l}|)
10         return |R_{ψ_l}| + |R_{ψ_¬l}|
11     end
12 function CountConditioned(ψ, l):
13     ψ_l ← propagate units on ψ|_l
14     if ψ_l contains empty clause then
15         return 0
16     else if ψ_l contains no clauses then
17         v ← number of unassigned variables in ψ_l
18         return 2^v
19     else
20         |R_{ψ_l}| ← 1
21         C ← DisjointComponents(ψ_l)
22         for C_i ← C do
23             |R_{ψ_l}| ← |R_{ψ_l}| × GetModelCount(C_i)
24         end
25         return |R_{ψ_l}|
26     end
```

Definition 21 (coloured graph isomorphism)**.** *Given two coloured graphs* $G = (V_1, E_1, P_1)$ *and* $H = (V_2, E_2, P_2)$*,* G *and* H *are said to be isomorphic if there exists a bijection* $\pi : V_1 \to V_2$ *such that:*

- $\forall v, w \in V_1$*:* $(v, w) \in E_1 \iff (\pi(v), \pi(w)) \in E_2$
- $\forall v \in V_1$*:* $\mathsf{colour}(v) = \mathsf{colour}(\pi(v))$

The (coloured) graph isomorphism problem is to determine whether or not two (coloured) graphs are isomorphic. The coloured graph isomorphism problem is polynomial-time reducible to its uncoloured counterpart (Schweitzer, 2009), so we will omit the word "coloured" when appropriate. Although the complexity-theoretic status of the graph isomorphism problem remains open, relatively

efficient algorithms exist in practice (McKay and Piperno, 2014). A closely related problem is that of *graph canonization*, which is to compute the *canonical labelling* of a given graph.

Definition 22 (canonical labelling)**.** *Given graphs G and H, a canonical labelling of a graph G is a new graph $\mathsf{Canon}(G)$, such that H is isomorphic to G if and only if $\mathsf{Canon}(G) = \mathsf{Canon}(H)$.*

As implied by the name, $\mathsf{Canon}(G)$ is effectively a relabelling of G. Thus, given an oracle for graph canonization, verifying isomorphism between graphs can be done by computing the canonical labelling for each graph and checking whether the resulting graphs are identical. The strength of this approach is that it allows checking isomorphism of a graph with many graphs at once, through for example a hash table of canonized graphs.

4.4 Symmetric Components

As a contribution, we observe that the model count of a component is not affected by the following operations:

- re-ordering the literals in a clause
- re-ordering the clauses in a component
- renaming the variables within a component to new variable symbols.

We propose an approach that captures these symmetries to improve the reuse of cached model counts. Let us first formally define the notion of *symmetric components*,

Definition 23 (symmetric components)**.** *Two formulas ψ_1 and ψ_2 are said to be (semantically) symmetric if there is a bijection $\pi : lits(\psi_1) \to lits(\psi_2)$ such that $R_{\psi_2} = R_{\pi(\psi_1)}$ and $\forall l \in lits(\psi_1) : \neg\pi(l) = \pi(\neg l)$.*

Example 28 (symmetric components)**.** *Suppose we observe the following two components in different places in our search tree. We can show that the two components are semantically symmetric: for the mapping $\pi = \{A \mapsto \neg C, C \mapsto A, D \mapsto B\}$ (fixing all other literals), we have $R_{\mathcal{C}_2} = R_{\pi(\mathcal{C}_1)}$.*

$$\mathcal{C}_1 \begin{cases} \neg A \vee C \\ \\ A \vee C \vee D \end{cases} \qquad\qquad \mathcal{C}_2 \begin{cases} \neg C \vee A \vee B \\ \\ A \vee C \end{cases}$$

Detecting symmetries. We employ the following two-step process to detect if two components are symmetric: (i) first encode the formulas ψ_1 and ψ_2 as graphs $\mathsf{Gr}(\psi_1)$ and $\mathsf{Gr}(\psi_2)$; then (ii) check whether their canonical labellings are equal, i.e., $\mathsf{Canon}(\mathsf{Gr}(\psi_1)) = \mathsf{Canon}(\mathsf{Gr}(\psi_2))$. We first outline the encoding in step (i).

Definition 24 (Aloul et al., 2002)**.** *The graph representation of ψ, denoted $\mathsf{Gr}(\psi) = (V, E, P)$, is a coloured graph constructed in the following manner:*

1. *Add a node n_{c_i} to V with $\mathsf{colour}(n_{c_i}) = \mathsf{red}$ for each clause c_i in ψ.*
2. *Add a node n_{l_i} to V with $\mathsf{colour}(n_{l_i}) = \mathsf{blue}$ for each literal l_i in $lits(\psi)$.*
3. *Add an edge (n_{l_i}, n_{l_j}) joining each literal l_i with its negated counterpart l_j.*
4. *Add an edge (n_{c_i}, n_{l_i}) if l_i occurs in the clause c_i, thus joining every clause node with its constituent literal nodes.*

Example 29. *Using the definition above, both $\mathcal{C}_1$ and $\mathcal{C}_2$ from Example 28 yield a graph with the same structure. This is illustrated in Figure 4.1.*

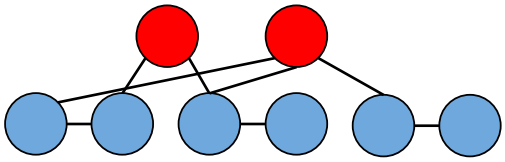


Figure 4.1: The graph representation $Gr(\mathcal{C}_1)$ and $Gr(\mathcal{C}_2)$ of Example 28.

We now state our primary soundness argument for the symmetric component cache:

Theorem 1. *Given two components ψ_1 and ψ_2, if $\mathsf{Canon}(\mathsf{Gr}(\psi_1)) = \mathsf{Canon}(\mathsf{Gr}(\psi_2))$ then $|R_{\psi_1}| = |R_{\psi_2}|$.*

Proof Sketch. It will suffice to show that if $\mathsf{Canon}(\mathsf{Gr}(\psi_1)) = \mathsf{Canon}(\mathsf{Gr}(\psi_2))$, then ψ_1 and ψ_2 are semantically symmetric. Any formula recovered from a graph $\mathsf{Gr}(\psi_1)$ ($\mathsf{Gr}(\psi_2)$) is semantically symmetric to ψ_1 (ψ_2): this is because it is unique up to a reordering of clauses and literals, and relabelling of literals. The same statement holds after canonical labelling: that is, any formula reconstructed from $\mathsf{Canon}(\mathsf{Gr}(\psi_1))$ ($\mathsf{Canon}(\mathsf{Gr}(\psi_2))$) is semantically symmetric to ψ_1 (ψ_2) since the canonical labelling of a graph yields a bijection on the nodes such that

colours and edges are preserved (see Definition 21 and 22). Thus, putting the two statements together we get that if $\mathsf{Canon}(\mathsf{Gr}(\psi_1)) = \mathsf{Canon}(\mathsf{Gr}(\psi_2))$, then ψ_1 and ψ_2 are semantically symmetric.

□

To conclude, we can store each component ψ into the cache using as an index their canonical graph $\mathsf{Canon}(\mathsf{Gr}(\psi))$. By doing so, components that are identical up to a renaming of the variables, and a re-ordering of the literals and clauses, will be identified as such and their model count will be reused.

4.5 Implementation: SymGanak

The previously described approach was incorporated within GANAK (Sharma et al., 2019), a state-of-the-art model counter that won the 2020 model counting competition in the unweighted model counting track (Fichte et al., 2020). We refer to the resulting implementation as SYMGANAK and describe the details of its relevant optimizations below.

Probabilistic symmetric component caching (PSCC). To improve cache utilization, SYMGANAK calculates an m-bit hash of each canonical labelling using the hash family $H_{cl}(n, m)$ mapping $\{0,1\}^n \to \{0,1\}^m$ (Lemire and Kaser, 2016). While probabilistic component caching (PCC) was initially proposed in GANAK, we adapt the scheme to work with cached graphs. The string that is hashed is created from the vertices and edges of the canonical labelling $\mathsf{Canon}(\mathsf{Gr}(\psi))$ in sorted order. This hash (rather than the canonical labelling itself) is stored in the cache. Hashing makes the solver probabilistic due to the risk of a hash collision, but the confidence δ (influencing the hash length m) is configurable by the user and can be set to a small value for high confidence. The probabilistic guarantees proven for PCC in GANAK (Sharma et al., 2019) continue to hold for PSCC in SYMGANAK.

Final encoding. Algorithm 5 shows the final algorithm of the $\mathsf{Encode}(\cdot)$ function (referred to in Algorithm 4) for SYMGANAK. To compute an encoding for a component ψ, SYMGANAK computes the canonical labeling of the graph representation of ψ (line 3). SYMGANAK then randomly samples a hash function from the hash function family $H_{cl}(n, m)$ (line 4), and finally computes a hash of the canonical labelling to add to the component cache (line 5).

Algorithm 5: Encoding symmetric components

```
function Encode(ψ):
    graph ← Gr(ψ)
    canonical_label ← Canon(graph)
    h ← H_cl(n, m)
    return h(canonical_label)
```

Bounded component analysis. Modern model counters typically integrate some form of *conflict-driven clause learning*, recording failed search paths as conflict clauses that guide backtracking. SymGanak employs *bounded component analysis* (Sang et al., 2004), such that the learned clauses are used to prune the search space but are not included in the cached component representation. This is similar to what is done in prior model counters (like sharpSAT and Ganak) and is necessary to take full advantage of the component scheme. Indeed, if learned clauses were included in the cached representation, then they may not only grow much larger in size but may make it harder to find identical components, i.e., get cache hits (Sang et al., 2004). However, note that even for "classical" component caching schemes, bounded component analysis is not trivial: Sang et al. (2004) showed that extra care must be taken when integrating component caching with clause learning. When exploring unsatisfiable parts of the search tree, model counts found for components under this part of the tree must be discarded from the cache, as reusing them may otherwise lead to incorrect results. Fortunately, as long as all components under a given assignment are satisfiable, the approach is sound: below we prove that this result continues to hold when caching symmetric components. As part of the proof, we use $R_{\psi\downarrow\mathbf{P}}$ to denote the projection of R_ψ onto a subset of the variables $\mathbf{P}$ (that is, the models in R_ψ restricted to literals formed only from $\mathbf{P}$).

Lemma 1. *Let π be a partial assignment such that $F|_\pi$ is satisfiable, and let A be a component of $F|_\pi$, and $G|_\pi$ a set of learned clauses of F reduced by π. Then $|R_A| = |R_{A\wedge G|_\pi\downarrow vars(A)}|$.*

Proof. This lemma follows from 1 and 2 below which respectively prove that any projected model of $A \wedge G|_\pi$ is also a model of A, and vice versa.

1. Any model of $R_{A\wedge G|_\pi\downarrow vars(A)}$ is clearly a model of A (because $A \wedge G|_\pi$ implies A).
2. By Theorem 1 of Sang et al. (2004), any model m_A of A can be extended to a model of $F|_\pi \wedge G|_\pi$. Now since $A \wedge G|_\pi$ is implied by $F|_\pi \wedge G|_\pi$

(because A is a component of $F|_\pi$), m_A can also be extended to a model of $A \wedge G|_\pi$. Hence, it follows that for any model m_A of A we have $m_A \in R_{A \wedge G|_\pi \downarrow vars(A)}$.

□

Theorem 2. *Symmetric component caching, in combination with bounded component analysis and clause learning, still yields the correct model count as long as we remove all sibling components and their descendants from the cache when encountering an unsatisfiable component.*

Proof. Using clause learning, when encountering component A as part of a satisfiable formula $F|_\pi$, its model count will be computed as $|R_{A \wedge G|_\pi \downarrow vars(A)}|$. This is because the model count is computed using guidance from the learned clauses (which may contain variables not in $vars(A)$). In bounded component analysis, this value will be cached as the model count of component A: the soundness of this, even for symmetric component caching, is guaranteed by Lemma 1 (subject to pruning unsatisfiable siblings and their descendants). Any component B symmetric to A can safely reuse this value because $|R_A| = |R_B|$ by Theorem 1. □

Handling binary clauses. The component encoding scheme used in CACHET, a precursor to SHARPSAT represents cached components as a combination of the unassigned variables and an identifier for each clause in the component (Sang et al., 2004). Thurley (2006) observed that the presence of binary clauses can be inferred from the presence of the unassigned variables, and therefore proposed a sound caching scheme that did not store the identifier corresponding to binary clauses. Lagniez and Marquis (2021) later expanded on this idea, by realising that the same trick can be more generally applied to clauses that have not been shortened by the current variable assignment. Interestingly, the arguments about the soundness of Thurley's encoding scheme (and the more general scheme of Lagniez and Marquis) do not hold under the symmetric caching scheme. Therefore, in a significant departure from SHARPSAT and its derivatives such as GANAK, we do explicitly encode all clauses. Fortunately, the probabilistic component caching scheme introduced in GANAK alleviates potential space efficiency drawbacks as our cache consists of the hashes of components.

Hybrid thresholding. Searching for a component in the SYMGANAK cache is computationally expensive as compared to previous caching schemes. Thus, there exists a delicate balance between the time spent on cache lookups and the

gains from a cache hit. For this purpose, SymGanak employs the following scheme, which we call *hybrid thresholding*: we fix configurable parameters l and u (empirically determined), for the minimum and maximum number of variables a component must contain to be eligible for symmetric component caching. If the number of variables in a component lies outside of these bounds (either $|vars(\mathcal{C})| > u$ or $|vars(\mathcal{C})| < l$), SymGanak instead uses the traditional caching scheme of Ganak. This scheme is motivated by the following observations:

- the overhead of computing the canonical labeling for small components is often higher than simply solving these components;
- large components have a high cost of computing the canonical labeling and a small likelihood of obtaining a cache hit.

Hence, in both of the above cases, we ignore symmetry detection and resort to PCC (as used in Ganak) which is both fast and has high cache utilization.

Variable selection heuristics. Along with packaging existing heuristics like VSADS (Sang et al., 2005) and CSVSADS (Sharma et al., 2019), we introduce a novel variable selection heuristic, *Isomorphic Cache State and Variable State Aware Decaying Sum* (ICSVSADS), that is motivated by CSVSADS but is also *symmetry aware*. More concretely, whenever a cache hit occurs, we decrease the scores of all variables in that component, as well as the scores of all variables that have previously formed a component symmetric to it.

For example, in Ganak, when $(x \vee y \vee z)$ yields a cache hit under the CSVSADS heuristic, the score of x, y, and z is decreased to discourage branching on those variables in the future. If we were to branch on any of those variables in the future, it would be impossible to obtain a cache hit on the same component below that point in the search tree. We extend this idea to SymGanak with ICSVSADS, such that when $(x \vee y \vee z)$ is hit as a result of the symmetric component $(a \vee \neg b \vee c)$, we not only discourage branching on x, y and z, but also on a, b and c. The heuristic is otherwise identical to CSVSADS.

4.6 Experiments

We integrated the caching scheme proposed above on top of the existing state-of-the-art model counter, Ganak. The code has been released as a branch of the mainline Ganak implementation[1]. We employed Nauty (McKay and

[1] `https://github.com/meelgroup/ganak`

Piperno, 2014) to perform graph canonization. It is worth remarking that SymGanak also provides an option to turn off PSCC, and thereby behaves as a deterministic counter. We performed a detailed empirical evaluation on a large suite of benchmarks arising from combinatorial instances, to answer the following research questions:

RQ1 How do different variable branching heuristics impact the performance of SymGanak?

RQ2 How does the run time performance of SymGanak compare with respect to the state-of-the-art model counter Ganak?

Our empirical study leads to a surprising conclusion: first, we observed that the VSADS heuristic achieves better run time performance than the other branching heuristics. We also observed that SymGanak outperforms Ganak, both in terms of PAR-2 score[2] and the number of instances solved. Our results are in line with often observed behavior in the context of SAT solving: the choice of heuristics depends on the class of benchmarks. As pointed out in Section 4.1, combinatorial benchmarks not only serve as important challenging problems but improvements in automated reasoning have paved a way for the discovery and proofs of challenging mathematical theorems. In this context, we expect our empirical study to motivate further studies on designing efficient counting schemes for combinatorial instances.

4.6.1 Implementation and Experimental Setup

We evaluated SymGanak on 212 instances from a wide range of combinatorial benchmark classes: Battleships, n-queens, grid Bayesian networks, k-colouring of grid graphs, quasigroup (Latin square) completion, FPGA switch-boxes, and logic synthesis, among several others. Details on all of the benchmark classes can be found in the technical Appendix A.

We performed our experiments on a high-performance computer cluster, with each node having an Intel Xeon E5-2690 v3 CPU with 24 cores and 96GB of RAM. We used all 24 cores per node, with a memory limit set to 4GB per core. Every instance, for each tool, was executed on a single core.

For Ganak and SymGanak, we set the default confidence value of $\delta = 0.05$, a maximum component cache size of 2GB, and a timeout of 5000 seconds. For

[2]The PAR-2 score (penalized average run time), as used in the SAT 2018 Competition (Heule et al., 2019), is the average run time assigning a run time of two times the time limit (instead of a "not solved" status) for each unsolved benchmark.

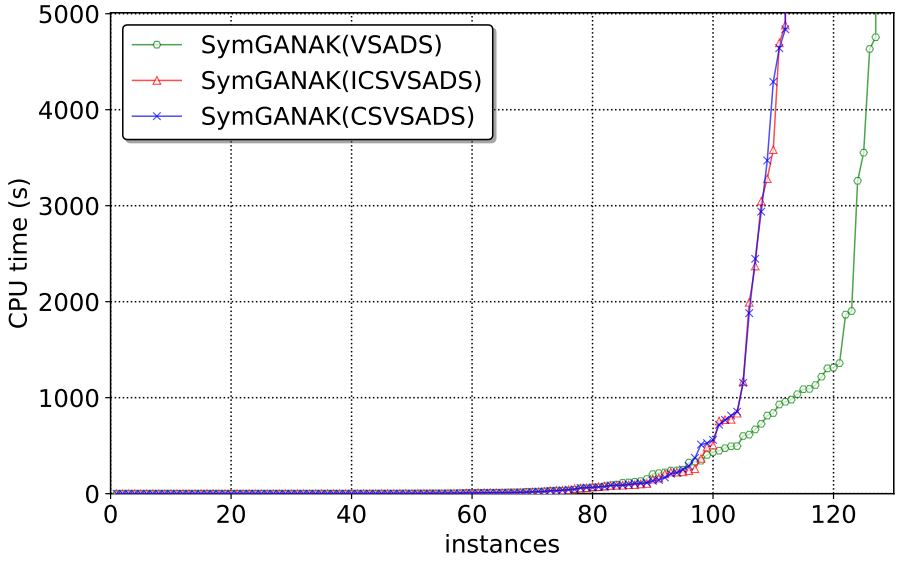


Figure 4.2: Cactus plot comparing different variable branching heuristics in SymGanak

SymGanak, we empirically determined 10 and 250 to be good lower and upper bound values for hybrid thresholding (see Section 4.5). SymGanak (similar to Ganak) uses the *independent support* (IS) (Ivrii et al., 2016) of the formula to accelerate the search; due to cost considerations, IS is used only if fewer than 500 conflicts are detected after 500 000 decisions. We ran both Ganak and SymGanak with this setting. All other parameters were set to their default values as in Ganak.

The cactus plots (Figures 4.2 and 4.3) show the number of instances solved (x-axis) by the respective tool in a given amount of time (y-axis); a point (x, y) on the plots represents that x benchmarks were solved by the counter in y seconds. All run time results are included in Appendix A.

4.6.2 Results

RQ1: Comparing branching heuristics. Figure 4.2 compares the different branching heuristics available in SymGanak. We found it surprising that VSADS outperforms both the cache-aware heuristics (CSVSADS and ICSVSADS). A detailed analysis of these heuristics on our set of benchmarks

shows that these heuristics are incomparable: of all the instances, VSADS, CSVSADS, and ICSVSADS were the most effective heuristic in 34, 16 and 12 instances respectively; they had comparable performance[3] in 60 instances, while they all timed out for 85 instances. So, though VSADS is the dominant heuristic in 34 instances, one of the two cache-aware heuristics emerges as the winner in 28 instances.

Among CSVSADS and ICSVSADS, our symmetry aware heuristic ICSVSADS has the same performance as CSVSADS. A deeper examination revealed that SymGanak (ICSVSADS) made fewer decisions on average than SymGanak (CSVSADS) (149 000 vs 181 000), suggesting that the gains of improved cache utilization may have outweighed the additional bookkeeping required to keep track of variables used in each component. Translating this improved cache performance to run time improvements seems to be an interesting challenge for future work.

RQ2: Impact of the symmetric component cache (SymGanak versus Ganak). As the VSADS branching heuristic performs the best for both SymGanak and Ganak on our benchmarks, we compared both of these tools with VSADS. Figure 4.3 shows that SymGanak outperforms Ganak: while SymGanak solves 16 more instances and achieves a lower PAR-2 score of 0.87× that of Ganak, there was only a single instance solved by Ganak that timed out on SymGanak. Figure 4.4 shows a scatter plot comparing their run time on individual instances.

The performance of SymGanak can be attributed to the fact that by exploiting symmetries, SymGanak can obtain both a higher number of cache hits as well as cache hits on larger components[4]. Over all instances solved by both Ganak and SymGanak, the average component size of each cache hit was 79.7 for SymGanak and only 58.2 for Ganak; the mean number of decisions made by SymGanak was approximately 302 000, compared to 1.4 million for Ganak.

To understand the results above in greater detail, Figure 4.5 shows a detailed distribution of cache hits for a representative n-queens instance: SymGanak has component cache hits with over 100 variables, about an order of magnitude larger than the largest components hit by Ganak. Even on the smaller components, SymGanak manages to obtain substantially more cache hits than Ganak.

[3]Finishing within one second of each other.

[4]We define the size of a component as the number of variables appearing in it.

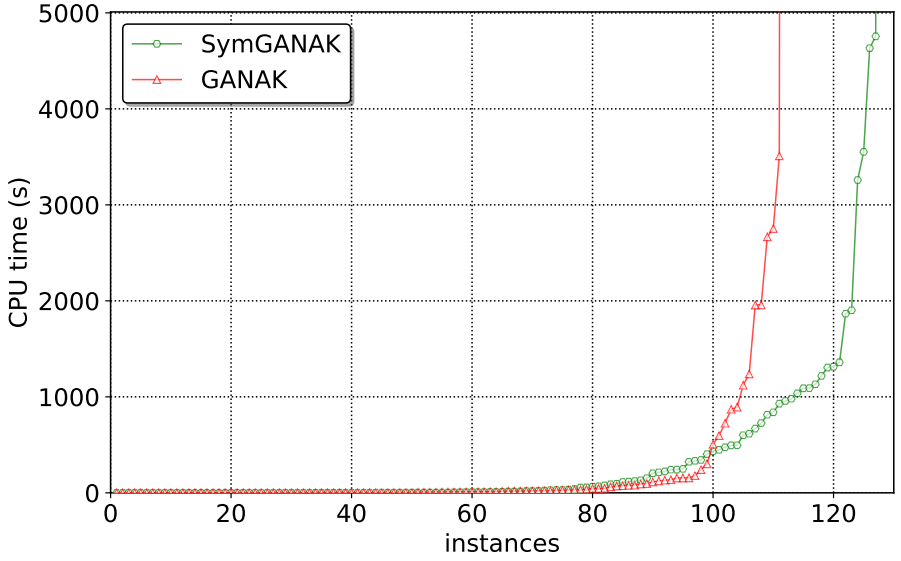


Figure 4.3: Cactus plot comparing SymGanak and Ganak

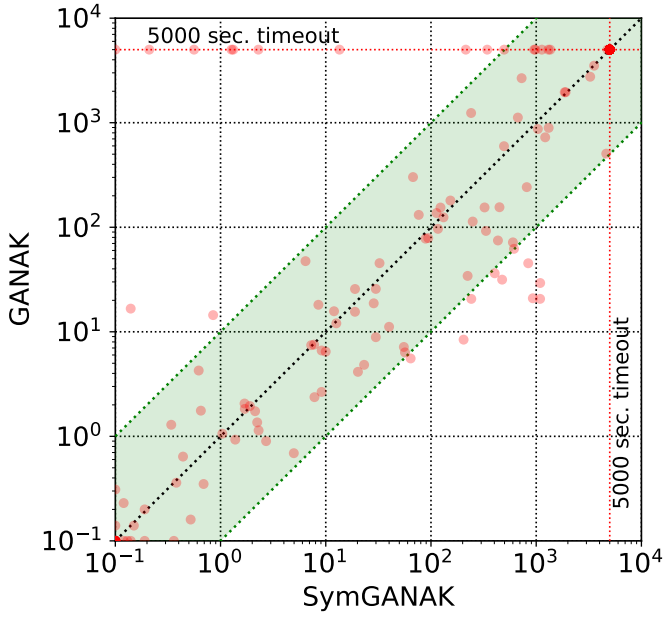


Figure 4.4: Scatter plot comparing SymGanak and Ganak

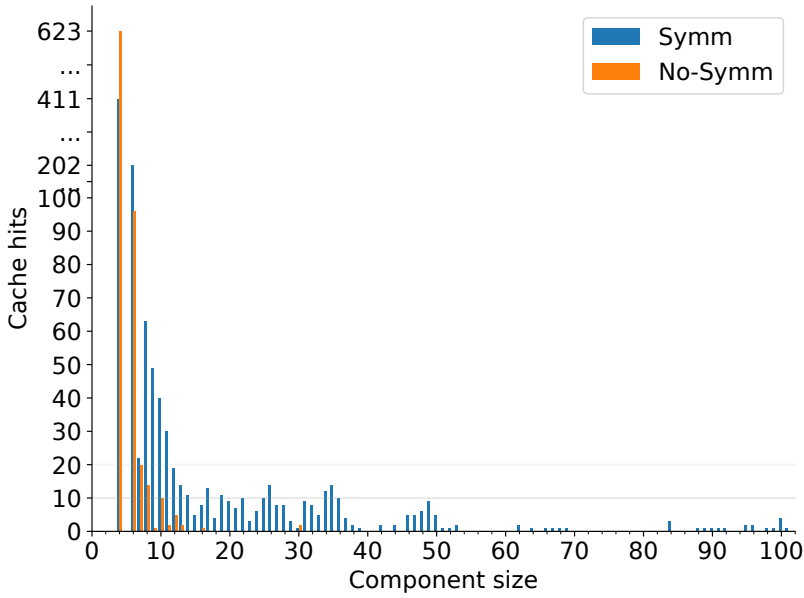


Figure 4.5: Cache hit distribution for an n-queens problem instance ($n = 12$, 144 variables). For each component size (x-axis), the number of cache hits for components of that size (y-axis). The figure was generated using the CSVSADS heuristics, turning off both random restarts and hybrid thresholding for easier analysis. Be mindful of the y-axis values.

4.7 Conclusion

We investigated the effect of caching symmetric components in #DPLL-based model counting algorithms. To evaluate our approach, we implemented the concept into a new counter SymGanak, an extension of Ganak, and compared their performance.

Although the detection of symmetries comes with a computational cost, we showed that detecting larger components more often can reduce the overall time needed to solve them, as illustrated by a reduced PAR-2 score and a greater number of benchmarks solved by SymGanak. This opens the door to further research in faster methods for detecting symmetric components. We also evaluated the performance of SymGanak under several variable selection heuristics. While we made some first steps in identifying a novel variable selection heuristic (ICSVSADS) that could work well with symmetric component caching, improving this remains an open problem for future research.

While our choice of Ganak as the base tool was in line with the typical practice in the SAT community where improvements are shown on top of winning solvers of recent years, it would be interesting to pursue integration of symmetric component caching in other state-of-the-art model counting systems such as D4 (Lagniez and Marquis, 2017).

4.8 Beyond Unweighted Counting

Weighted model counting. Recall from Section 2.2 that the weighted model counting problem generalizes the unweighted variant by associating weights with models, via the weights on their literals. The idea proposed in this work can also be extended to weighted model counting, but it does require a modification to the graph encoding $\mathsf{Gr}(\psi)$. This modification is necessary because variables in the weighted variant are not interchangeable when their literals have different weights. Consider as example, that $A \vee B$ has the same model count as $C \vee D$. However, if $w(A) \neq w(C)$, then the *weighted* model count is different. In order to not consider the variables interchangeable, the graph encoding must be adapted, colouring literals based on their weight such that a different weight implies a different colour and vice versa. The graph isomorphism procedure will consequently differentiate also based on weight.

Projected model counting. In projected model counting problems, the input additionally includes a set of *priority* variables $\mathbf{K} \subseteq \mathbf{V}$ (Aziz et al., 2015). Rather than counting the number of satisfying assignments to $\mathbf{V}$, we are then instead interested in the number of assignments to $\mathbf{K}$ that can be extended to a satisfying assignment for $\mathbf{V}$. Similar to the weighted model counting problem, the proposed symmetry detection procedure must be adapted to distinguish priority variables from non-priority variables, because these are again not interchangeable. Definition 24 can, for example, be adapted to colour the literals of each priority variable light blue, and the literals of each non-priority variable dark blue.

Chapter 5

Decision Making: A Tale of Three Operations

This chapter was previously published as:

> V. Derkinderen and L. De Raedt (2020). "Algebraic Circuits for Decision Theoretic Inference and Learning". In: *Proceedings of the 24th European Conference on Artificial Intelligence, ECAI*. vol. 325. IOS Press, pp. 2569–2576. DOI: 10.3233/FAIA200392

The introduction in Section 5.1 is adapted to better integrate within this dissertation. Section 5.2 is a reduced version of the original because of the preliminaries already discussed in detail in Chapter 2.

Whilst the previous chapter focused primarily on unweighted model counting problems, we now again focus on the weighted and algebraic counting variants. This chapter emphasizes the generality (and limitations) of the algebraic model counting framework by exploring a decision making setting that requires reasoning over uncertainty, decisions, and utility values. This setting involves three operations: addition and multiplication to compute the expected value of each possible scenario, and an (arg)max operation to choose the best set of decisions.

This chapter contributes to answering the research question:

RQ2.1: What tasks can be cast into an algebraic model counting problem?

The remainder of this chapter is organized as follows: in Section 5.1 we situate our contributions and provide a brief introduction, followed by the preliminary background in Section 5.2: a definition of **X**-constrained sentential decision diagrams. Afterwards, we discuss the problem of finding the most optimal decision given an uncertain environment, in Section 5.3, and introduce a learning problem setting that we solve with the proposed algebraic framework, in Section 5.4. We close with related work and a conclusion in Section 5.5 and 5.6 respectively.

5.1 Introduction

Knowledge compilation is important for inference in probabilistic models, which are ubiquitous in artificial intelligence (De Raedt et al., 2016; Koller and Friedman, 2009; Marquis, 2008). Inference and learning for such models is computationally hard. Nonetheless, there has been steady progress in developing tractable representations and algorithms for supporting a wide range of tasks. Especially techniques of knowledge compilation (Darwiche and Marquis, 2002) have been instrumental in speeding up inference in Bayesian networks and Statistical Relational AI (De Raedt et al., 2016).

It is well-known that a wide range of algorithms can be generalised using semirings, for instance, belief propagation with the sum-product and max-product algorithms. The key idea, as explained in Chapter 2 and 3, is to replace the traditional addition and product operations by semiring operations $\oplus$ and $\otimes$ (cf. algebraic model counting Equation 2.2). This has inspired work on algebraic model counting (Kimmig et al., 2011, 2017) where rather than using the standard probabilistic semiring, a range of other semirings are used to solve a wide range of inference tasks, including max-product, sensitivity analysis, gradient computation, and even weighted model integration (Zuidberg Dos Martires et al., 2019b).

In this chapter, rather than considering the standard probabilistic setting, we consider decision theoretic extensions and investigate how we can adapt and apply the compilation approach to cope with some of the resulting inference and learning problems. One way of viewing this is as the transition from standard Bayesian networks to influence diagrams or from a probabilistic programming language (such as ProbLog (Fierens et al., 2015)) to its decision theoretic extension (such as DT-ProbLog (Van den Broeck et al., 2010)).

We specifically focus on the following two inference tasks. First, finding the optimal decision in a given uncertain environment, in a one-shot setting. This means that we seek to find the set of decisions that maximises the expected utility, while all decisions are made at once before any observations. This task requires three operations – (arg)max, addition, and multiplication, which already indicates that it is unclear how to apply the standard counting techniques. We will discuss two approaches to this task, both centered around algebraic model counting. The first approach yields an exact solution but involves adapting the algebraic framework to handle a violation of the semiring properties, by constraining the variable ordering. The approach is based on earlier work on the *Same-Decision Probability* task (Oztok et al., 2016). The second approach is more approximate and views the arithmetic version of the formula representation as a function with unknown values (decisions) that have to be optimised. We then show how to optimise this function by applying the algebraic framework within a gradient ascent algorithm. As a second inference task we briefly discuss a utility learning problem, that we tackle by minimising a loss function using gradient descent, applying again the algebraic framework.

To validate the contributions, we implement them as an extension of DT-ProbLog. The code and data are available at `https://github.com/VincentDerk/Paper-AC-Decisions-Learning`.

5.2 Constrained Sentential Decision Diagram

Necessary to the work in this chapter is the class of sentential decision diagrams (SDD) introduced in Section 2.4.3. For parts of our contribution we require an **X**-constrained SDD: an SDD is **X**-constrained when the vtree satisfies certain conditions. The following definitions are based on Oztok et al. (2016).

Definition 25. *Given a vtree t containing vtree node v, over respectively variables $vars(t)$ and $vars(v)$, the set of variables $\mathbf{X}$ outside of v is defined as $\mathbf{X} = vars(t) \setminus vars(v)$.*

Definition 26 (**X**-constrained vtree node)**.** *A vtree node v is $\boldsymbol{X}$-constrained iff v appears on the right-most path of the vtree and $\boldsymbol{X}$ is the set of variables outside v.*

Definition 27 (**X**-constrained SDD node)**.** *An SDD is $\boldsymbol{X}$-constrained iff it respects an $\boldsymbol{X}$-constrained vtree. An SDD node is $\boldsymbol{X}$-constrained iff it respects an $\boldsymbol{X}$-constrained vtree node.*

Example 30 (**X**-constrained SDD)**.** *The vtree in Figure 5.1 is both $\{A, B\}$- and $\{A, B, C\}$-constrained because of respectively node 5 and 6. The SDD in Figure 5.1 is consequently both $\{A, B\}$- and $\{A, B, C\}$-constrained.*

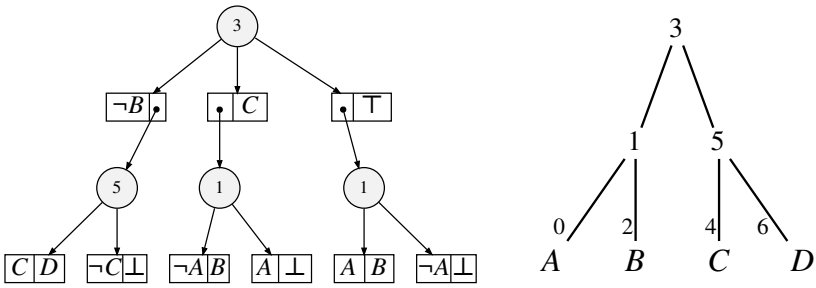


Figure 5.1: A sentential decision diagram representing $(A \wedge B) \vee (C \wedge D) \vee (B \wedge C)$, and its vtree.

We use the SDD package[1] to implement the algorithms proposed in this chapter. The package supports both SDD and **X**-constrained SDDs. The latter will be necessary for Section 5.3.1.

5.3 Maximising Decisions

Proceeding with the decision making setting, we first explain how algebraic model counting can be used to compute an expected utility.

Expected utility. When each literal is associated with both a probability and a utility, then the expected utility can be computed using AMC, similar to the probability approach. We denote with $u(m)$ and $p(m)$ the utility and probability of a model m and with u_v ($u_{\neg v}$) the utility obtained from variable v being true (false). The expected utility of m is defined as $eu(m) = p(m) \times u(m)$. The expected utility of a formula ψ, $eu(\psi)$, is defined in Equation 5.1 [2] and can be computed using the AMC framework.

$$eu(\psi) = \sum_{m \in R_\psi} \underbrace{(\overbrace{\prod_{l \in m} p_l}^{p(m)})(\overbrace{\sum_{l \in m} u_l}^{u(m)})}_{eu(m)} \tag{5.1}$$

Define the labeling function α such that $\alpha(v) = (p_v, p_v \times u_v)$, then the *expected utility semiring* defined below can be used to compute $eu(\psi)$.

[1] The SDD package is available at `http://reasoning.cs.ucla.edu/sdd/`

[2] This is an expectation over the utility of a model, $\mathbb{E}[u(m)]$, which in turn is an additive function of the utilities associated with each literal of the model.

Definition 28 (expected utility semiring)**.** *The expected utility semiring* $(\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$ *with*

- $\mathcal{A} = \{(p, eu) | p \in \mathbb{R}_{\geq 0}, eu \in \mathbb{R}\}$
- $(p_1, eu_1) \oplus (p_2, eu_2) = (p_1 + p_2, eu_1 + eu_2)$
- $(p_1, eu_1) \otimes (p_2, eu_2) = (p_1 p_2, p_1 eu_2 + p_2 eu_1)$
- $e^{\oplus} = (0, 0)$ *and* $e^{\otimes} = (1, 0)$

is an instance of the expectation semiring (Eisner, 2002) and can be used to compute the expected utility of a theory.

Maximising expected utility. A more complicated problem arises when we introduce decision variables and seek the assignment of truth values to those decisions that maximises the expected utility. We refer to this problem as the maximum expected utility (MEU) problem. We focus on its one-shot decision setting where all decisions are made before observing any stochastic variable. That is, each set of instantiated decisions $\mathbf{d}$, leads to a set of possible models (scenarios) making up the expected utility (Equation 5.1) for $\mathbf{d}$.

Definition 29 (maximum expected utility (MEU) problem)**.** *The* maximum expected utility *problem consists of a propositional logic formula* ψ *over a set of decision variables* $\mathbf{D}$ *and a set of stochastic variables* $\mathbf{S}$*. Similar to Equation 5.1, each literal is associated with a probability and a utility such that* $eu(\psi \wedge \mathbf{d})$ *is the expected utility for a particular instantiation* $\mathbf{d}$ *of the decision variables* $\mathbf{D}$*. The MEU task consists of finding the optimal truth assignment* $\mathbf{d}$ *such that the expected utility is maximised:*

$$\underset{\mathbf{d}}{argmax} \sum_{m \in R_{\psi \wedge \mathbf{d}}} (\prod_{l \in m} p_l)(\sum_{l \in m} u_l) \tag{5.2}$$

While this task consists of three operations (max, sum, and product), a semiring is a structure of only two. It is therefore not obvious how to apply AMC.

On a high-level, the MEU problem can be framed as a *one stage stochastic constraint optimisation problem (SCOP)* (Walsh, 2002). That is, there are decisions that afterwards lead to several possible scenarios, whose probability is determined through stochastic variables, and the goal is to optimise the expected utility (Tarim et al., 2006). However, while stochastic constraints play a central role within SCOP, we do not consider these here. In that sense, this work is closer to Latour et al. (2017), who consider in more detail how to compactly

represent probabilities within the stochastic constraints and objective function in a manner that constraint solvers can be used for solving the problem. The MEU problem also relates to the more general Plausibility-Feasibility-Utility framework (Pralet et al., 2007) that was previously introduced to unify several formalisms.

Example 31. *Consider the problem of vaccinating persons in a social network. A logic propositional formula ψ can be used to describe the spreading mechanism of a disease based on the health and social information of each person (e.g., friendship relations) (*$\mathbf{S}$*) and their vaccination status (*$\mathbf{D}$*). There are costs associated with treating an infected person, but also with vaccinations. The MEU problem can be used to optimise and determine which persons to vaccinate.*

Example 32. *Consider a small more concrete problem, deciding whether to use machine A. Using A has a cost of -3 but, when there is no failure, it also yields a reward of 4.*

$$
\begin{gathered}
(profit \wedge useA \wedge \neg failure) \vee \\
(\neg profit \wedge \neg useA) \vee \\
(\neg profit \wedge failure) \\
p_{useA} = 1.0, \quad p_{\neg useA} = 1.0, \quad u_{useA} = -3, \quad u_{\neg useA} = 0 \\
p_{profit} = 1.0, \quad p_{\neg profit} = 1.0, \quad u_{profit} = 4, \quad u_{\neg profit} = 0 \\
p_{failure} = 0.6, \quad p_{\neg failure} = 0.4, \quad u_{failure} = u_{\neg failure} = 0
\end{gathered}
$$

5.3.1 Constrained Algebraic Circuit

To obtain the expected utility of a set of decision assignments $\mathbf{d}$, we need to sum all the models with the same set of decisions. This implies an ordering that the circuit needs to adhere to in order to compare decisions in a valid manner. More specifically, when using SDDs, the algebraic circuit must first condition on the decision variables before considering the rest. $\mathbf{X}$-constrained SDDs with $\mathbf{X} = \mathbf{D}$ have exactly this property (Figure 5.2). This can be seen as follows. By definition, the $\mathbf{D}$-constrained SDD nodes represent the whole formula ψ conditioned on an assignment for each decision, $\psi|_{\mathbf{d}}$. Such a node represents a disjunction of all the models with the same decisions. The algebraic circuit will thus first combine models with the same set of decisions before combining (comparing) with models of other decisions, as was required.

The vtree can be used to determine whether an or-node of the $\mathbf{X}$-constrained SDD represents a summation or a maximisation. Alternatively, the decision information can also be stored in the semiring elements in the form of a decision set $\mathbf{L}$, performing maximisation when the sets of decisions are different. To stay

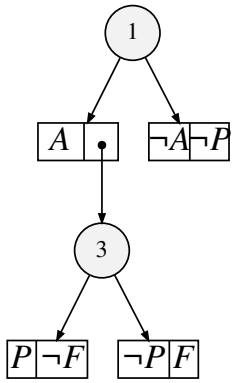


Figure 5.2: An {A}-Constrained SDD modelling Example 32. The variables are abbreviated: $A = useA$, $P = profit$ and $F = failure$.

close to the algebraic framework, we choose to illustrate the latter approach and use the following structure $(\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$, dynamically defining the $\oplus$-operation:

$$\{w(v) = (p_v, p_v \times u_v, \mathbf{L_v}) | v \in \mathbf{D} \cup \mathbf{S} \cup \neg\mathbf{D} \cup \neg\mathbf{S}\} \subset \mathcal{A} \tag{5.3}$$

with $\mathbf{L_v} = \{v\}$ and $\mathbf{L}_{\neg\mathbf{v}} = \{\neg v\}$ or $\mathbf{L_v} = \emptyset = \mathbf{L}_{\neg\mathbf{v}}$ depending on whether v is a decision.

$$a \oplus b = \begin{cases} max(a, b), & \text{if } \mathbf{L_a} \neq \mathbf{L_b} \\ (p_a + p_b, eu_a + eu_b, \mathbf{L_a}), & \text{otherwise} \end{cases} \tag{5.4}$$

$$a \otimes b = (p_a p_b, p_a eu_b + p_b eu_a, \mathbf{L_a} \cup \mathbf{L_b}) \tag{5.5}$$

$$e^{\oplus} = (0, 0, \mathbf{D} \cup \neg\mathbf{D}) \tag{5.6}$$

$$e^{\otimes} = (1, 0, \emptyset) \tag{5.7}$$

$$max(a, b) = \begin{cases} a, & \text{if } b = e^{\oplus} \\ b, & \text{else if } a = e^{\oplus} \\ a, & \text{else if } \frac{eu_a}{p_a} \geq \frac{eu_b}{p_b} \\ b, & \text{otherwise} \end{cases} \tag{5.8}$$

This structure (Equation 5.3 to 5.8) extends the expectation semiring to track the decision sets $\mathbf{L}$ and to perform max when required. For a set of decision assignments, the probability of all models must sum to one. This is not

necessarily the case when constraints are present and we therefore normalize when comparing expected utilities: $\frac{eu_a}{p_a} \geq \frac{eu_b}{p_b}$ (Equation 5.8). Additional cases can be added to prevent division by 0 when $p_a = 0$ or $p_b = 0$. When such a case occurs, the other value must be chosen. Note that this structure is not a semiring as the associativity property is only satisfied within the **X**-constrained context. This is expected and the reason we require an **X**-constrained SDD. If associativity was satisfied in general, then any SDD would have been sufficient.

5.3.2 Unconstrained Algebraic Circuit

Constraining the vtree to be **X**-constrained limits the class of valid SDDs. As a consequence, this could lead to larger circuits. To avoid this, we introduce another approach that maximizes outside of the circuit and does not rely on a constrained ordering. Instead, it treats the algebraic circuit as a function where the decision values are unknown and have to be chosen such that the output of the function, the expected utility, is maximized. This works as follows. When the probability of each decision variable is set to either 0 or 1 (and its negation to 1 or 0), the output of the circuit is the expected utility of that decision set. Hence, optimising these parameters will maximise the expected utility. We solve this maximisation via gradient ascent and compute the gradient using the circuit and the algebraic model counting framework. In contrast to the constrained approach, which is exact and therefore ensures an optimal decision, the unconstrained approach that uses gradient ascent may never find the optimal decision, for example because of local maxima.

Implementation. To ensure that the probability of a decision d remains within $[0, 1]$, we use the sigmoid function: $p_d = \sigma(z) = \frac{1}{1+e^{-z}}$ whose derivative is given by $\sigma'(z) = \sigma(z)(1-\sigma(z))$. We optimise the function represented by the algebraic circuit (Equation 5.9) via gradient ascent and use algebraic circuits to obtain the required gradients (Equation 5.10 to 5.14).

$$EU = \sum_{m \in R_\psi} U(m) \times P(m) \tag{5.9}$$

$$\frac{\partial EU}{\partial d_i} = \sum_{m \in R_\psi} U(m) \times \frac{\partial P(m)}{\partial d_i} \tag{5.10}$$

$P(m)$ is a combination of stochastic variables **S** and decision variables **D** that are in m. In the following equations we will explicitly differentiate between the

positive literals and negative literals (Equation 5.11).

$$P(m) = \Big(\prod_{\substack{s\in m,\\ s\in\mathbf{S}}} p_s\Big)\Big(\prod_{\substack{\neg s\in m,\\ s\in\mathbf{S}}} p_{\neg s}\Big)\Big(\prod_{\substack{d\in m,\\ d\in\mathbf{D}}} p_d\Big)\Big(\prod_{\substack{\neg d\in m,\\ d\in\mathbf{D}}} (1-p_d)\Big) \tag{5.11}$$

thus, if $d_i \in m$

$$\frac{\partial P(m)}{\partial d_i} = (1-\sigma(d_i))P(m) \tag{5.12}$$

and if instead $\neg d_i \in m$

$$\frac{\partial P(m)}{\partial d_i} = -(1-\sigma(d_i))P(m) \tag{5.13}$$

This means that the gradient with respect to d_i can be obtained by computing the sigmoid of d_i, the expected utility where d_i is true and where d_i is false (Equation 5.14). The last two each result in one (parallel) circuit evaluation.

$$\begin{aligned}\frac{\partial EU}{\partial d_i} &= \Big(\sum_{\substack{m\in R_\psi,\\ d_i\in m}} U(m)\frac{\partial P(m)}{\partial d_i}\Big) + \Big(\sum_{\substack{m\in R_\psi,\\ \neg d_i\in m}} U(m)\frac{\partial P(m)}{\partial d_i}\Big)\\ &= (1-\sigma(d_i))\Big(\sum_{m\in R_{\psi\wedge d_i}} EU(m)\Big) + (\sigma(d_i)-1)\Big(\sum_{m\in R_{\psi\wedge\neg d_i}} EU(m)\Big)\end{aligned} \tag{5.14}$$

Alternatively, we can use out-of-the-box automatic differentiation techniques that are present in, for example, PyTorch (Paszke et al., 2019) or TensorFlow (Martín Abadi et al., 2015), with the algebraic circuit as their computational graph (Zuidberg Dos Martires et al., 2019a).

The decision values returned by this approach are in the range of $[0, 1]$ and affect the weight with which models contribute to the expected utility. Insufficient measures to prevent local maxima or insufficient time can cause the decisions to not exactly correspond to either 0 or 1. To obtain the best decisions from the found values, several approaches can be investigated. Examples include rounding to the nearest integer, evaluating different decisions for which the value is far from 0 and 1, treating the results as a stochastic policy, etc. In our implementation, we choose to set each final parameter to the nearest integer in order to have a deterministic policy and use random restarts to mitigate local maxima.

5.3.3 Experiments

In our experiments, rather than using the low-level AMC encodings directly, we use the higher-level probabilistic programming language ProbLog[3] to specify models and queries. These models are then compiled into AMC problems using the mechanics of aProbLog (Kimmig et al., 2011). We implement our approaches for DT-ProbLog by using the mechanics of aProbLog (Kimmig et al., 2011). The data set is constructed as follows. Well-known Bayesian networks (Scutari, n.d.), Survey (Scutari and Denis, 2014), Asia (Lauritzen and Spiegelhalter, 1988) and Earthquake (Korb and Nicholson, 2010) are first compiled into ProbLog programs using ProbLog's existing conversion script. Next, we add decisions to the programs by converting each parent node of the Bayesian network with two possible values into a decision. If this results in less than four decisions, any node of the Bayesian network not yet considered has a chance of 0.5 to introduce a new decision. Each value of the node has an equal probability of being affected by this new decision. Finally, we introduce utilities using two different approaches. The first approach considers each atom t and adds a utility value for t with a probability of 0.8 and for $\neg t$ with a probability of 0.3. The utility values themselves are uniformly sampled from $[-50, 50]$. The second approach instead introduces five new separate atoms with a positive and negative utility, and for each new atom samples five interpretations from the program. The samples serve as rules for the new atom to become satisfied. The second approach happens before adding the decisions. The first approach happens afterwards, to allow decisions to also have utilities. Using this process, we construct 60 DT-ProbLog programs (20 for Asia, Earthquake, and Survey), half of them constructed with the first utility approach and half of them with the second. The number of rules in the resulting programs ranges from 38 up to 108, the number of utilities from 7 to 23, and the number of decisions from 1 to 6. The memory consumption of larger networks (e.g. Sachs (Scutari, n.d.)) was too high to consider here. This is due to the rather naive standard encoding of Bayesian networks as ProbLog programs, which could be optimized using more compact encodings, a better vtree heuristic, or when better configuring the SDD package. Our experiments are designed to answer two questions.

Q1) Does the unconstrained approach provide optimal solutions? That is, do the decisions resulting from the unconstrained approach lead to the highest expected utility? We experimented on the 60 DT-ProbLog programs comparing both approaches. For 85% of the programs, the difference was less than 0.1. The average difference over the experiments is 1.472 and the average relative difference is 0.057. We conclude that overall, the unconstrained approach

[3]ProbLog is available at `https://dtai.cs.kuleuven.be/problog/`

Table 5.1: Statistics on the executions for the constrained (c) and unconstrained (u) approach for each dataset D (Earthquake (e), Asia (a) and Survey (s)) and for all datasets combined (g). The average compile time (CT), run time (RT), and SDD size are provided. The run time includes the compilation time.

D	Appr.	avg. CT (s)	avg. RT (s)	avg. SDD Size (# nodes)
g	c	4.0	4.3	1 480 603
	u	2.5	41.0	1 055 494
e	c	0.0	0.0	2065
	u	0.0	1.1	2244
a	c	0.0	0.1	12539
	u	0.0	3.3	5539
s	c	11.9	12.7	4 427 204
	u	7.5	118.6	3 158 698

provides promising results. Further investigation into problems of a larger size would be interesting.

Q2) How does the constrained ordering impact the circuit size, compile- (CT) and run time (RT) compared to the unconstrained approach? We report on the average compile-, run time, and SDD size in Table 5.1. It is clear that constraining the circuit can lead to larger circuits[4]. However, the unconstrained approach trades compile time for more evaluation time and currently becomes slower than the constrained approach. This part of the implementation can still be optimised, e.g., by using a better random restart configuration or using more optimized tools such as PyTorch (Paszke et al., 2019) or TensorFlow (Martín Abadi et al., 2015).

We conclude that to scale to larger problem sizes, more work is required on the used encoding, vtree heuristics, random restart configuration, etc. Regardless, we have shown that algebraic model counting and algebraic circuits provide an expressive framework that can also solve decision theoretic tasks.

5.4 Learning Utility Parameters

Several techniques have already been introduced to learn the probability parameters in ProbLog (Gutmann et al., 2008, 2011b). The most recent

[4]The reported circuit sizes were obtained from the SDD package and can include dead nodes left over from the construction process.

addition jointly learns the parameters of probabilistic facts and those of neural networks by optimising a loss function using gradient descent (Manhaeve et al., 2018). This approach integrates well with ProbLog's inference as the gradient can also be computed using the algebraic circuit. We will use the same approach and define a loss function that we use as an approximate signal, allowing us to learn a utility parameter for each variable.

Setting. The input of our learning task is based on a set of interpretations $\{m_1, \ldots, m_M\}$ called examples. These examples we only observe partially Q $= \{q_1, \ldots, q_M\}$. For each of the interpretations m_j, we also observe the total utility $\tilde{u}_j$, $\tilde{U} = \{\tilde{u}_1, \ldots, \tilde{u}_M\}$. The output of this learning task consists of the positive and negative utility, respectively $u_{i,p}$ and $u_{i,n}$, associated with each propositional variable f_i. We focus on the utilities here and assume the probability parameter of each variable is already known. This can be relaxed in two ways. On the one hand, we can learn the probabilities first, ignoring the utility values, after which we are in our described setting. On the other hand, the assumption can be relaxed when the loss function is extended with a signal concerning the likelihood of the observations. Though it could be interesting to compare the performance of these different approaches empirically, this is left for further work. We also assume that of each example, all decisions are observed. Our approach works for both one-shot and sequential decision problems.

Approach. To learn the utility parameters, we minimize the following loss function:

$$MSE(Q, \tilde{U}, \psi) = \frac{1}{M} \sum_{j=1}^{M} (ceu(q_j, \psi) - \tilde{u}_j)^2 \tag{5.15}$$

$$ceu(q_j, \psi) = \sum_{m \in R_\psi} P(m|q_j) u(m) \tag{5.16}$$

The intuition behind this equation is that we minimize the difference between the utility we expect $ceu(q_j)$ and the utility that was actually observed $\tilde{u}_j$. The former is defined by $u(m)$ and $P(m|q_j)$. The assumption that all decisions are fully observed, simplifies the calculation of $P(m|q_j)$ and is reasonable as long as the decisions are not made by an unknown third party. Our approach also works for partially observed decisions when each decision has an associated probability. When assuming optimal behavior, the calculation of $P(m|q_j)$ and our minimization approach becomes more complex.

To optimize Equation 5.15, we employ a gradient descent approach and use an algebraic circuit to compute the gradient $\frac{\partial MSE(Q,\tilde{U},\psi)}{\partial u_{i,p}}$ (Equation 5.17).

$$\frac{2}{M}\sum_{j=1}^{M}(\underbrace{ceu(q_j,\psi)}_{Part1}-\tilde{u}_j)\underbrace{\sum_{m\in R_\psi}\delta_{i,m,p}P(m|q_j)}_{Part2} \tag{5.17}$$

$$\delta_{i,m,p}=\begin{cases}1 & \text{if } f_i\in m,\\ 0 & \text{otherwise}\end{cases} \tag{5.18}$$

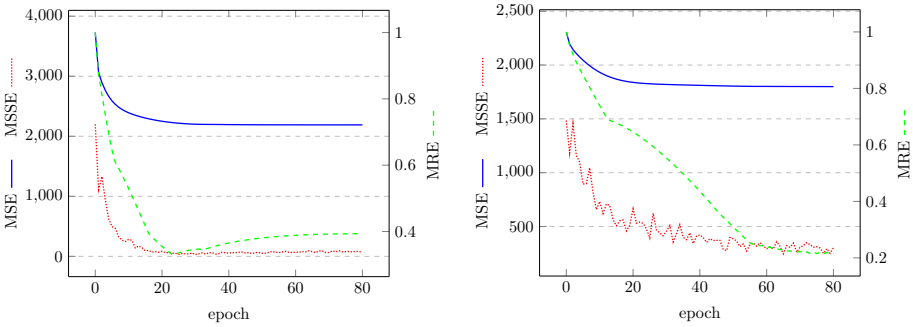
The gradient for the negative utility parameter $u_{i,n}$ is similar to Equation 5.17 except that we use $\delta_{i,m,n}$ instead of $\delta_{i,m,p}$.

$$\delta_{i,m,n}=\begin{cases}0 & \text{if } f_i\in m\\ 1 & \text{otherwise}\end{cases} \tag{5.19}$$

Part 1 of Equation 5.17 can be computed using the expected utility semiring querying for q_j conditioned on q_j. Part 2 of that equation can be computed using the probability semiring querying for $q_j \wedge f_i$ (or $q_j \wedge \neg f_i$ for the $u_{i,n}$ case) conditioned on q_j.

Experiments. The correct DT-ProbLog programs are constructed using the process described for maximisation (Section 5.3). To simplify the experiment setup, we do not add decisions to the programs. To construct the partially observed examples, we sample from the programs and leave out each observed atom with a probability of p_{drop}. The input program of the learning task is the original program with for each utility variable a chance of 0.5 that it is made unknown.

To evaluate our approach we answer the following questions. **Q1)** Is the MSE loss function a good indicator when our aim is to 1) predict the utility of an interpretation, 2) recover the correct values or 3) make good decisions? **Q2)** How does the partial observability affect the results? We consider three metrics to answer both questions, mean squared sampled error (MSSE), mean relative error (MRE), and relative regret. The MSSE (Q1.1, Q2) is closest to what we are optimising and is evaluated by sampling $n_s = 100$ interpretations, comparing the total utility of the interpretation t_c with the total utility of the learned values t_l: $\frac{1}{n_s}\sum_{i=1}^{n_s}(t_c-t_l)^2$. The MRE (Q1.2) is used to compare how close the learned utility values x_i are to the actual values $\tilde{x}_i$, using the relative error to normalise for large (small) $\tilde{x}_i$. When there are n_v learned values, $MRE=\frac{1}{n_v}\sum_{i=1}^{n_v}|\frac{x_i-\tilde{x}_i}{\tilde{x}_i}|$.



(a) with $p_{drop} = 0.7$ (b) with $p_{drop} = 0.8$

Figure 5.3: The learning progress of a Survey network.

If we use the learned program for decision making, then the regret metric is more interesting, but also more complex to compute. The regret (Q1.3) is based on the utility to expect when taking decisions based on the learned model. Denote $\mathbf{d_l}$ as the optimal decisions according to the learned model, $\mathbf{d_t}$ as the true optimal decisions, and $eu(\mathbf{d})$ as the expected utility when taking decisions $\mathbf{d}$. Then the relative regret is defined as $|\frac{eu(\mathbf{d_t}) - eu(\mathbf{d_l})}{eu(\mathbf{d_t})}|$ and computed using the maximisation approaches described in this chapter. This metric requires decisions that we add to the program in a manner equivalent to the procedure described for the maximisation approach (Section 5.3).

Q1) We have tested the learning approach on five different Survey, Earthquake, and Asia networks, each for varying values of p_{drop}[5]. Each experiment was given 80 epochs to converge and 150 partially observed examples to train on. Figure 5.3b shows that while optimising our loss function, the MSSE and MRE successfully decrease as well. This suggests the MSE can be used as an indicator to optimise MSSE and MRE. It is possible that when optimising too long, the MSSE and MRE can increase again due to overfitting to MSE. This is more noticeable for MRE (Figure 5.3a) than for MSSE. The relative regret is low, even for high p_{drop} (Figure 5.4). **Q2)** We investigate the effect of p_{drop} on the MSSE (Figure 5.5). As expected, it generally becomes harder to learn with an increased p_{drop}.

We conclude that AMC techniques can be adapted to perform utility learning. To investigate larger problems with more parameters, we first need improvements to obtain smaller circuits.

[5]Due to a non-deterministic ordering originating in the ProbLog database, an increase in p_{drop} can cause previously unobserved atoms to become observed. It is however still impossible for more atoms to become observed.

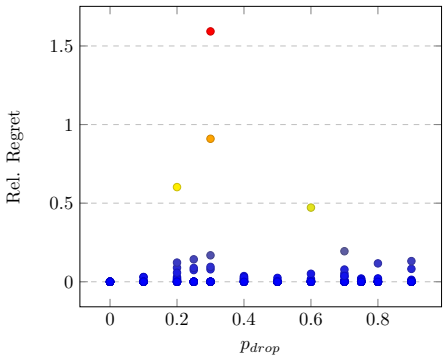


Figure 5.4: Relative regret for 180 Survey networks.

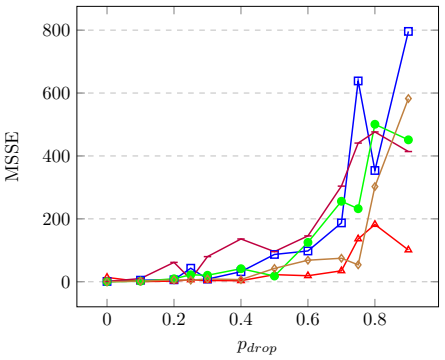


Figure 5.5: The MSSE for five different Survey networks, over different degrees of partial observability (p_{drop}).

5.5 Related Work

Maximisation. Our maximisation approaches address the same problem as DT-ProbLog (Van den Broeck et al., 2010). Their approach consists of manipulating binary decision diagrams while we use a more general AMC approach applied to SDDs. Our approaches also retain the circuit so that it can be used for other tasks (e.g. learning utilities) or an extension of this task (e.g. stochastic constraints). Finally, DT-ProbLog did not yet consider a learning setting. Sum-product-max networks and decision circuits have a structure similar to our constrained circuit approach (Bhattacharjya and Shachter, 2012; Melibari et al., 2016). They determine the operations of each node during the construction process. Our approach emphasises the power of an algebraic circuit and AMC, dynamically defining the operations. Due to the higher-level definition, we can also reuse existing compilation tools. This AMC approach provides more flexibility towards extended or different tasks (Li and Eisner, 2009). Finally, we have also introduced an approach that does not constrain the variable ordering and a learning task. A #DPLL approach can also maximise the expected utility (Apsel and Brafman, 2012; Majercik and Littman, 1998). This is related to our approach as the traces of #DPLL can be used to form an arithmetic circuit. The advantage of materializing the circuit is that it can be reused, significantly reducing the cost of re-evaluating the theory with different input weights. This is especially beneficial for the utility learning approach which requires multiple evaluations to obtain gradients. Another example of a task that requires multiple evaluations is sensitivity analysis (Darwiche, 2000). AND-OR graphs are related to #DPLL and algebraic circuits (Dechter, 1999; Dechter

and Mateescu, 2007). Work in that domain is often used in a probabilistic setting but can also be applied to the maximum expected utility problem (Lee et al., 2019; Marinescu, 2009). Those approaches often start from an influence diagram while we start from an expressive DT-ProbLog program. Furthermore, as a main difference to the work on AND-OR graphs, our contribution includes the application of algebraic circuits to utility learning and an unconstrained circuit approach. This has not been considered by those other approaches.

Utility learning. There is a lot of work already performed in the context of utility learning. However, to the best of our knowledge there is none that is situated in our setting, that is, with partially observed interpretations and the total utility of that interpretation. In terms of data structure, the work on sum-product-max networks (Melibari et al., 2016) is the most similar but it considers a fully observed setting. Markov decision processes and influence diagrams (also known as Bayesian decision networks) are two alternatives for modelling a decision problem. We are not aware of any work for those models that considers our setting. In general, utility information is not provided and instead obtained indirectly for example by preference elicitation (Chajewska et al., 2000; Rothkopf and Dimitrakakis, 2011) or based on interpretations with optimal behavior (Ng and Russell, 2000; Suryadi and Gmytrasiewicz, 1999). The latter is the case in the domain of inverse reinforcement learning (Ng and Russell, 2000) where an unknown utility function, for example of a Markov decision process, is learned from examples containing optimal behavior.

5.6 Conclusion

Algebraic circuits are versatile structures. We have shown at the level of AMC how maximising the expected utility and utility learning can be solved. Because we defined this at the high level of AMC, we were able to reuse the algebraic circuit mechanics of the existing probabilistic languages (aProbLog and DTProbLog) without adding new constructs to them. We have shown two approaches for the maximisation problem and we introduced a novel learning setting where unknown utility values are learned from partially observed interpretations with observed utilities. This learning task can be tackled by a gradient descent approach, using algebraic circuits to compute the gradients. The circuit size and compilation time rapidly increase for larger problems and are currently an obstacle to scaling our approaches. In future work, we plan to investigate ways of improving this by adapting our methods (e.g. encodings) or improving knowledge compilation tools. Finally, we plan to extend the maximisation approaches to sequential problems (Venturato et al., 2022).

Chapter 6

Variable Ordering for Weighted Model Integration

This chapter was previously published as:

> V. Derkinderen, E. Heylen, P. Zuidberg Dos Martires, S. Kolb, and L. De Raedt (2020). "Ordering Variables for Weighted Model Integration". In: *Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence, UAI*. ed. by R. P. Adams and V. Gogate. Vol. 124. AUAI Press, pp. 879–888

The bottom-up min-fill heuristic introduced in the publication was devised during a brainstorming session with P. Zuidberg Dos Martires and S. Kolb. The hypergraph decompositioning algorithm within the publication instead originates from the master thesis of E. Heylen. I formalised the variable ordering analysis based on existing work in the discrete variable domain, providing a more principled basis for all the proposed heuristics. I implemented the additional heuristics, ran all experiments borrowing existing code from the aforementioned master thesis, and contributed to the publication writing process.

The introduction in Section 6.1, and background in Section 6.2 were rewritten to better integrate within this dissertation. Especially the background on weighted model integration and its connection to knowledge compilation was extended with more detail and examples.

We now move beyond classical propositional formulas, and investigate counting with background theories. This alleviates the previous limitation to discrete problems, expanding inference to discrete-continuous domains. A challenging task within this broader setting is *weighted model integration*, which uses a formula ψ to define models of interest similar to propositional counting. In contrast to the propositional setting, ψ may contain continuous variables such that ψ essentially represents the (continuous) regions over which to integrate with respect to a given weight function (Belle et al., 2015). Knowledge compilation techniques have shown to be beneficial also for solving these integration problems (cf. the solver F-XSDD(BR) (Kolb et al., 2019b)).

For d-DNNF compilers, the order in which to branch on variables has shown to greatly influence the size of the final representations. Additionally, when using them for weighted model integration, the variable ordering also heavily impacts the order in which we can optimally integrate out the variables. This chapter discusses that impact and proposes several ordering heuristics for the discrete-continuous domain. Our contribution relates to the research question:

> **RQ2.2 Can we extend variable ordering heuristics developed for the discrete domain to also work well for discrete-continuous domains?**

The remainder of this chapter is organized as follows: Section 6.1 introduces the problem of finding a good variable ordering with continuous variables for weighted model integration, and summarizes our contributions. Section 6.2 formally introduces the problem of weighted model integration and explains its connection to knowledge compilation, partially contributing to the first research question on counting applications. With Section 6.3, we first explain the impact of the variable ordering and discuss ordering heuristics for a chain of variables. Afterwards in Section 6.4, we discuss the case of a tree-structured ordering, important for targeting the SDD class that requires a vtree. The experiments and conclusions are presented in Section 6.5 and 6.6 respectively.

6.1 Introduction

The d-DNNF compilation algorithms are not the only algorithms influenced by a variable ordering. Others that in a similar fashion heavily rely on finding a good ordering include the *sum-product algorithm* (Pearl, 1982), *variable elimination* (VE) (Zhang and Poole, 1994), or *bucket elimination* (Dechter, 1999). Finding the optimal variable ordering in which to marginalize out single random variables is an NP-complete problem (Arnborg, 1985). However, several polynomial time

heuristic schemes have been developed (Darwiche, 2009; Dechter, 2013; Kjærulff, 1990).

Weighted model integration (WMI) (Belle et al., 2015) is a counting task involving continuous variables. It consists of two linked 'subproblems': the combinatorial problem also present in weighted model counting, and the integration of the continuous variables. Algorithms for both subproblems are characterized by a variable order, sometimes a variable tree, that heavily influences their efficiency. In practice, for many WMI problems the integration of continuous variables is the main bottleneck. However, in the WMI literature it is typically assumed that a (good) ordering is provided by the user (Kolb et al., 2018, 2019b), which is unrealistic. Therefore, in this chapter, we study the problem of automatically determining a good variable ordering in the context of WMI.

As a first key contribution of this chapter, we show how variable ordering techniques established for discrete variables can be extended to the discrete-continuous setting (Section 6.3). This is not straightforward as problems with continuous variables exhibit additional dependencies that impact the difficulty of the integration steps. Consequently, this contribution allows us to develop (tree-based) ordering techniques for a state-of-the-art WMI solver F-XSDD(BR) (Kolb et al., 2019b).

The second contribution of this chapter is BU-MiF, a novel heuristic that produces a variable (tree) ordering for the discrete-continuous domain, and which has no direct analog in the discrete domain. We extend F-XSDD(BR) with our new heuristic and experimentally show its benefits on a set of benchmark problems using PyWMI (Kolb et al., 2019a), indicating better performance than currently available orderings. Furthermore, the BU-MiF heuristic allows the F-XSDD(BR) algorithm to perform well on a set of benchmark problems from a tractable subset of WMI, outperforming the specialized SMI solver (Zeng and Van den Broeck, 2019) that was introduced to tackle this class of problems.

6.2 Weighted Model Integration

Weighted model integration as a problem was introduced by Belle et al. (2015), extending weighted model counting from its propositional logic setting to the domain of modulo theories, for instance to support $\mathcal{LRA}$ formulas such as $(x_1 + x_2 > 15) \implies A$ (cf. Chapter 2). In order to explain the relation to weighted model counting, we will rewrite its formulation, step-wise introducing support for continuous variables.

Recall from Chapter 2 the weighted model counting definition, which enumerates

all models m of a formula ψ. Below, we repeat the equation but abstract the weight function of a model $w(m)$ that was previously defined as a product of the weight of its literals.

$$WMC(\psi, \mathbf{V}, w) = \sum_{m \models \psi} w(m) \tag{6.1}$$

Instead of enumerating all models of ψ, we now enumerate all truth assignments over propositional variables $\mathbf{V}$ and only sum those satisfying ψ by using an indicator function denoted with Iverson brackets, $[\![m \models \psi]\!]$, that evaluates to 1 when the indicator is satisfied, and 0 otherwise.

$$WMC(\psi, \mathbf{V}, w) = \sum_{m \in \mathbb{B}^M} [\![m \models \psi]\!] w(m) \tag{6.2}$$

with M equal to the number of propositional variables, i.e., $|\mathbf{V}|$.

Finally, we support continuous variables: we partition $\mathbf{V}$ into a set $\mathbf{b}$ of propositional variables and a set $\mathbf{x}$ of continuous variables. The assignment m is similarly partitioned into $m_{\mathbf{b}}$ and $m_{\mathbf{x}}$.

$$WMI(\psi, \mathbf{V}, w) = \sum_{m_{\mathbf{b}} \in \mathbb{B}^M} \int [\![m \models \psi]\!] \, w(m) \, d\mathbf{x} \tag{6.3}$$

Definition 30 (weighted model integration (WMI))**.** *Given*

- *a set* $\mathbf{b}$ *of* M *Boolean variables,*
- *a set* $\mathbf{x}$ *of* N *real variables,*
- *a weight function* $w : \mathbb{B}^M \times \mathbb{R}^N \to \mathbb{R}_{\geq 0}$*,*
- *and a support* ψ *in the form of a modulo theory formula over* $\mathbf{V} = \mathbf{b} \cup \mathbf{x}$*,*

the weighted model integral is given by Equation 6.3.

When the set of real variables $\mathbf{x}$ is empty, the WMI task reduces to a standard weighted model counting task. Even though the WMI equation is flexible in the type of background theory used, we confine our explanations in this chapter to using $\mathcal{LRA}$ formulas.

Example 33 (WMI example)**.** *Consider a WMI problem, over* $\mathbf{b} = \emptyset$ *and* $\mathbf{x} = \{x_1, x_2\}$*, with weight function* $w(\mathbf{x}) = 2x_1x_2 + x_1^2$ *and the following support* ψ*.*

$$\psi = (0 \leq x_1, x_2 \leq 10) \wedge \Big((x_1 + x_2 < 10) \vee (2x_1 + x_2 < 13) \Big)$$

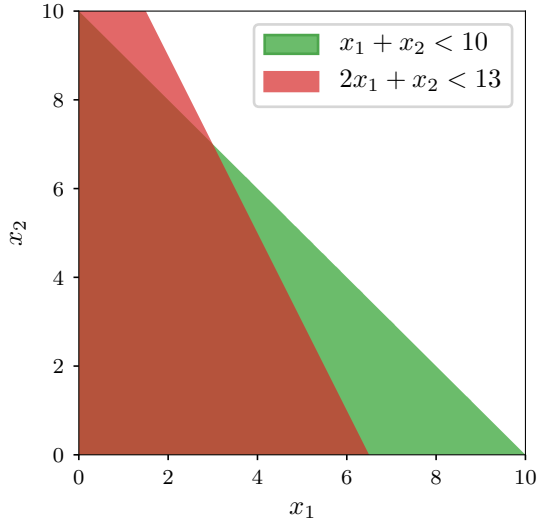


Figure 6.1: The x_1, x_2 space limited to $x_1, x_2 \in [0, 10]$, with the regions satisfying $x_1 + x_2 < 10$ or $2x_1 + x_2 < 13$ coloured. This illustrates the WMI problem of Example 33.

We slightly abused notation to write the $[0, 10]$ *bounds more compactly. The space represented by* ψ *is illustrated in Figure 6.1. The task of weighted model integration is to correctly integrate over this space. The region satisfying* $x_1 + x_2 < 10$ *partially overlaps with the region satisfying* $2x_1 + x_2 < 13$*. This complicates WMI solvers, which must not erroneously consider regions more than once.*

Knowledge compilation in WMI. Similar to knowledge compilation approaches for WMC, WMI has also received attention from this direction in order to build solvers. Such solvers have either been based on *extended algebraic decision diagrams* (Kolb et al., 2018) or *extended SDDs* (XSDDs) (Zuidberg Dos Martires et al., 2019b). These *extended* representations allow, in addition to Boolean variables, also the use of $\mathcal{LRA}$ atoms. In the case of XSDD, this is accomplished by abstracting ψ to a propositional version through the introduction of new fresh Boolean variables for each $\mathcal{LRA}$ atom. The resulting formula is then completely propositional, and is compiled using an existing SDD compiler. The state-of-the-art WMI solver F-XSDD(BR) (Kolb et al., 2019b) uses XSDDs to represent the support ψ. They then analyse the compiled

representation to push the integration of variables closer to the leaf nodes, because that produces smaller intermediate results, and leads to fewer integration calls. This is shown in the example below.

Example 34 (second WMI example)**.** *Suppose* $\mathbf{b} = \emptyset$*,* $\mathbf{x} = \{x_0, x_1, \ldots, x_4\}$*,* $w(\mathbf{x}) = x_1 x_2$*, and*

$$\psi = \begin{cases} (x_0 < 5) \vee (x_0 < x_1) \vee (x_0 < x_2) \vee (x_1 + 2 < x_2) \vee \\ (x_0 < x_3) \vee (x_0 < x_4) \vee (x_3 + 2 < x_4) \end{cases}$$

Figure 6.2 shows the result of 1) creating a d-DNNF representation of the abstracted support ψ*, after which 2) the d-DNNF is turned into an equation and the integration operations along with the weight function are pushed down lower into the equation. More details on how knowledge compilation helps to solve WMI problems are provided in Appendix B, as they are unnecessary for the remainder of this chapter.*

Our ability to push the integration lower into the SDD representation depends on both the formula ψ and the vtree that was used to guide the SDD during construction. Meaning, the ability to push down the integration can be influenced by choosing a good vtree. A major problem, however, is that vtree heuristics aim for succinct representations: they are completely agnostic to the $\mathcal{LRA}$ atoms underlying the abstracted variables, and the integration procedure. We resolve this problem by proposing new heuristics that focus on finding a good order in which to integrate out continuous variables, which can then be used to produce a good variable ordering (or vtree) to guide the SDD construction.

The previous WMI example had a simple weight function consisting of one term. For more complex weight functions — F-XSDD(BR) supports piece-wise polynomial weight functions — we refer to a longer explanation in Appendix B, and the relevant F-XSDD(BR) literature, Kolb (2019) and Kolb et al. (2019b). This does neither affect the analysis in this chapter, nor the contributed variable ordering heuristics.

6.3 Variable Orderings

We first illustrate the impact of the variable ordering (Section 6.3.1) before discussing heuristics (Section 6.3.2), and in both cases we first discuss the discrete domain before moving to the novel continuous part.

The focus of this section is a simple variable ordering, a chain. We therefore use a variable elimination algorithm in our explanation, rather than a knowledge

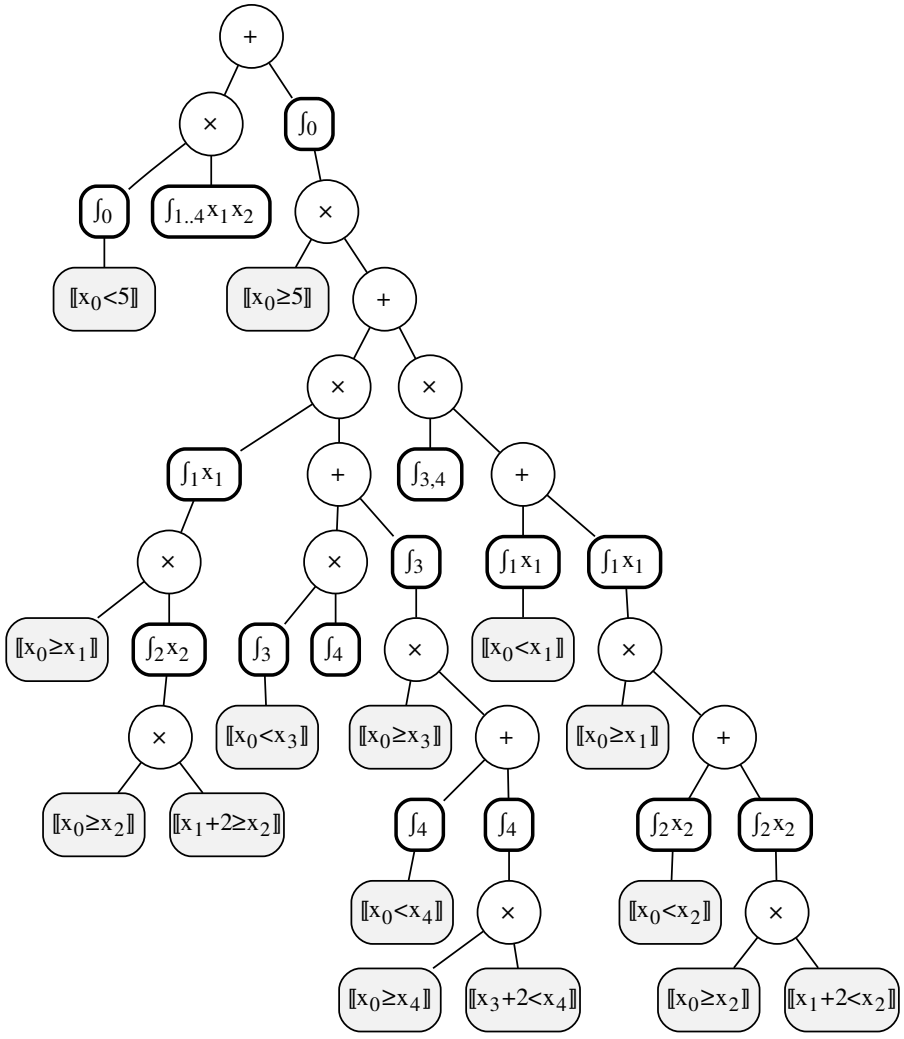


Figure 6.2: The WMI equation of Example 34. Both the weight function x_1x_2 and the integration operations have been pushed lower into the equation. Symbol $\int_i$ denotes the integration of x_i: $\int\ dx_i$. When the node is an intermediate node, the integration is performed over the result of the node below it. When it is instead a leaf node, the result is performed over 1, i.e., $\int 1dx_i$. Usually this means integrating over the variable's domain bounds that are excluded from the representation, for example $\int [\![0 < x_i < 10]\!] dx_i$.

A	P(A)
true	0.4
false	0.6

A	B	P(B|A)
true	true	0.9
true	false	0.1
false	true	0.2
false	false	0.8

B	C	P(C|B)
true	true	0.6
true	false	0.4
false	true	0.7
false	false	0.3

Table 6.1: Conditional probability tables $P(A)$, $P(B|A)$, and $P(C|B)$.

compiled approach. The variable tree (or vtree) structure is discussed afterwards, in Section 6.4.

6.3.1 How to Exploit Structure

Weighted Model Counting. We explain the discrete setting in the context of conditional probabilities, also called factors. Consider the problem of computing the probability $P(C)$ using the factors $P(A)$, $P(B|A)$, and $P(C|B)$ in Table 6.1. This can be done as follows:

$$P(C) = \sum_A \sum_B P(A, B, C) \tag{6.4}$$

$$= \sum_A \sum_B P(C|B)P(B|A)P(A) \tag{6.5}$$

In variable elimination approaches, evaluating this translates to first computing $\sum_B P(C|B)P(B|A)P(A)$, resulting in a new factor $f(A, C)$. The size of an intermediate factor is exponential in the number of its variables causing both the time and space complexity to be exponential. Fortunately, distributivity, commutativity, and associativity can be used to reduce the number of operations that have to be performed. We can for example push inside the summation over A:

$$P(C) = \sum_B P(C|B) \sum_A P(A)P(B|A) \tag{6.6}$$

This leads to an intermediate factor $f(B)=\sum_A P(A)P(B|A)$ depending on only one variable. The complexity is now no longer necessarily exponential in the number of variables but is instead determined by the problem structure and the variable ordering d — here $d=(B, A)$. The importance of the latter becomes apparent if we consider $d=(A, B)$ instead:

$$P(C) = \sum_A P(A) \sum_B P(B|A)P(C|B) \tag{6.7}$$

We obtain the factor $f(A, C)=\sum_B P(B|A)P(C|B)$, depending on two variables instead of one.

Unfortunately, finding a variable ordering that leads to intermediate factors with the lowest maximum number of variables is in general NP-complete (Dechter, 2013).

Weighted Model Integration. Pushing the sum operation inside, as done in the discrete case, has also been studied for the continuous case (Kolb et al., 2019b), the key difference being that integrations are pushed inside instead of summations.

Reconsider the definition of a weighted model integral (cf. Equation 6.3), using $\sum_{\mathbf{b}}$ to denote $\sum_{m_{\mathbb{B}} \in \mathbb{B}^M}$. Let us assume, for the sake of simplicity, that the weight function w does not depend on Boolean variables and fully factorizes, i.e., it is separable into factors depending only on single continuous variables:

$$\mathrm{WMI}(\psi, \mathbf{V}, w) = \sum_{\mathbf{b}} \int [\![m \models \psi]\!] [\overbrace{\prod_{x_i} w_i(x_i)}^{=w(m_{\mathbf{x}})}] d\mathbf{x} \tag{6.8}$$

Such separable weight functions allow us to push inside integrations over specific variables in an integrand. For instance, consider the function $p(z)$:

$$p(z) = \int ([\![0<z<1]\!][\![y \leq z]\!][\![x \leq y]\!] xyz)\, dxdy \tag{6.9}$$

Due to the separable weight function xyz we can push the integrations over x and y inside the integrand, similar to pushing inside summations in the discrete case.

$$p(z) = [\![0<z<1]\!] \left(\int [\![y \leq z]\!] \left(\int [\![x \leq y]\!] x dx\right) y dy\right) z \tag{6.10}$$

Similarly again to the discrete setting, choosing different orders in which to push inside the integrations can have tremendous effects on the space and time requirements of running an inference algorithm.

Example 35. *Given the weight function $w = 1$ and support*

$$\psi = \left(\bigwedge_{i=\{1,\ldots,4\}} (x_0 \leq x_i)\right) \wedge \bigwedge_{i=\{0,\ldots,4\}} (0 \leq x_i \leq 1) \tag{6.11}$$

where x_1, x_2, x_3, and x_4 all interact with x_0. If we first integrate out x_0 we obtain:

$$\begin{aligned}\int [\![\psi]\!] w dx_0 = x_1 [\![x_1<x_2]\!][\![x_1<x_3]\!][\![x_1<x_4]\!] \textstyle\prod_{i=\{1,2,3,4\}} [\![0 \leq x_i \leq 1]\!] + \\ x_2 [\![x_1\geq x_2]\!][\![x_2<x_3]\!][\![x_2<x_4]\!] \textstyle\prod_{i=\{1,2,3,4\}} [\![0 \leq x_i \leq 1]\!] + \\ x_3 [\![x_1\geq x_2]\!][\![x_2\geq x_3]\!][\![x_3<x_4]\!] \textstyle\prod_{i=\{1,2,3,4\}} [\![0 \leq x_i \leq 1]\!] + \\ x_3 [\![x_1<x_2]\!][\![x_1\geq x_3]\!][\![x_3<x_4]\!] \textstyle\prod_{i=\{1,2,3,4\}} [\![0 \leq x_i \leq 1]\!] + \\ x_4 [\![x_1\geq x_2]\!][\![x_2\geq x_3]\!][\![x_3\geq x_4]\!] \textstyle\prod_{i=\{1,2,3,4\}} [\![0 \leq x_i \leq 1]\!] + \\ x_4 [\![x_1\geq x_2]\!][\![x_2<x_3]\!][\![x_2\geq x_4]\!] \textstyle\prod_{i=\{1,2,3,4\}} [\![0 \leq x_i \leq 1]\!] + \\ x_4 [\![x_1<x_2]\!][\![x_1<x_3]\!][\![x_1\geq x_4]\!] \textstyle\prod_{i=\{1,2,3,4\}} [\![0 \leq x_i \leq 1]\!] + \\ x_4 [\![x_1<x_2]\!][\![x_1\geq x_3]\!][\![x_3\geq x_4]\!] \textstyle\prod_{i=\{1,2,3,4\}} [\![0 \leq x_i \leq 1]\!] \end{aligned} \tag{6.12}$$

However, first integrating out x_1 instead yields the more compact intermediate result, resulting in more efficient subsequent computations (the symbolic expression tree representing the integrand is much smaller).

$$\int [\![\psi]\!] w dx_1 = (1-x_0)(\textstyle\prod_{i=\{2,3,4\}} [\![x_0<x_i]\!])(\prod_{i=\{0,2,3,4\}} [\![0 \leq x_i \leq 1]\!]) \tag{6.13}$$

Even though Kolb et al. (2019b) studied pushing inside integrations, they did not develop any heuristics to do so. Their approach relied on hand-crafting specific integration orders for specific problems. In the following we delineate how variable ordering strategies in the discrete setting can be adapted for the continuous domain.

6.3.2 How to Order Variables

Finding the best variable ordering (i.e., smallest intermediate size) is in general NP-complete. In practice, we instead use heuristics. We first introduce additional concepts used to analyse and find good variable orderings. Then, we explain three simple and common variable ordering heuristics that use these concepts. While this explanation is based on existing work for discrete problems (Darwiche, 2009; Dechter, 2013), we show how to apply it to the continuous problem setting, a problem that, to the best of our knowledge, has not yet received much attention prior to the publication of this work.

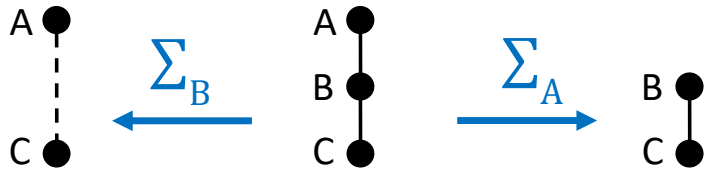


Figure 6.3: The interaction graph for factors $f(A)$, $f(A,B)$ and $f(B,C)$ (middle), the graph when B is summed out (left) and the graph when A is summed out (right).

Interaction Graphs

An important structure used to analyse a discrete problem is the interaction graph of factors (Darwiche, 2009).

Definition 31 (interaction graph)**.** *Let V be a set of vertices and E a set of edges. A* factor interaction graph *$G{=}(V,E)$ of a set of factors $\{f_1,\ldots,f_n\}$ is an undirected graph with a vertex $v_i \in V$ for each variable x_i and an edge between two nodes, v_j and v_k when the corresponding variables x_j and x_k appear together in at least one factor (we say that x_j and x_k co-appear or* interact*).*

Before we can eliminate a variable v when applying variable elimination, we must first multiply all factors in which v appears. After eliminating v, we obtain a factor containing all the variables that were in the multiplied factors. In the factor interaction graph, these two steps correspond to connecting all the neighbors of v and removing v from the graph. The additional edges are called *fill-in edges*. In this process, the number of variables in the resulting factor is equal to the number of neighbors of v while the size of a factor is exponential in the number of variables.

Example 36. *In the discrete example of factors $P(A), P(B|A)$ and $P(C|B)$, A interacts with B and B interacts with C (Figure 6.3, middle). Eliminating B results in new interaction between A and C (Figure 6.3, left) while eliminating A does not result in any new interactions (Figure 6.3, right). This shows that first eliminating A is more beneficial as the intermediate factor $f(A,B)$ is smaller (fewer neighboring variables).*

In order to utilize the concept of interaction graph for continuous domains, we introduce the concept of an **interaction graph of $\mathcal{LRA}$ atoms**. Such an interaction graph is obtained by interpreting the $\mathcal{LRA}$ atoms as factors. The vertices in the interaction graph then correspond to real variables appearing in the atomic literals and they are connected to each other if they jointly appear

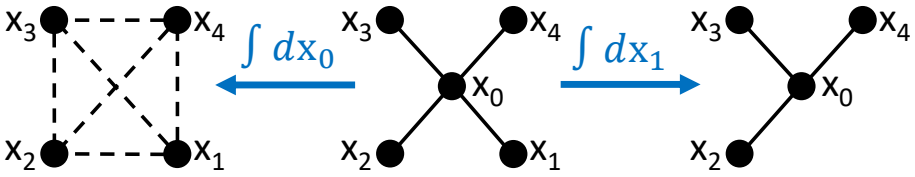


Figure 6.4: The interaction graph of Example 35 (middle), the graph when x_0 is integrated out (left) and the graph when x_1 is integrated out (right).

in at least one atomic literal. This elegant mapping of atomic modulo theory literals to factors allows us to deploy the concepts on variable ordering developed for the discrete setting in the continuous setting.

Example 37. *In Equation 6.11, x_0 interacts with all other continuous variables (Figure 6.4, middle). Integrating out x_0 results in new inequalities in which x_1, x_2, x_3 and x_4 interact with each other (Equation 6.12). In the interaction graph this implies the removal of x_0 and the addition of new edges (dashed, Figure 6.4, left). When integrating out x_1 instead, no new edges are introduced in the interaction graph (Figure 6.4, right), implying a more compact intermediate result (Equation 6.13).*

If a continuous variable v occurs in multiple inequalities with other continuous variables X, integrating out v will yield inequalities between all the variables of X. This process also exhibits an exponential relation for the discrete-continuous setting between the number of neighbors for v and the size of the result after integrating out v. This is related to the exponential complexity of Fourier-Motzkin elimination (Imbert, 1990).

An interaction graph and a variable ordering together form an ordered graph. The following two definitions are from Dechter (2013).

Definition 32 (ordered graph)**.** *Given an undirected graph $G = (V, E)$, the* ordered graph *(G, d) is obtained by ordering the nodes along ordering d. The* parents *of a node v are the nodes connected to v (see E) which occur earlier in the ordering. The* width of node *v in (G, d) is the number of parents v has. The* width of ordered graph *(G, d) is the maximum width of all nodes in (G, d).*

Definition 33 (induced ordered graph)**.** *An* induced ordered graph *(G^*, d) of (G, d) is an ordered graph obtained from (G, d) by processing the nodes in reverse order of d (last to first, top to bottom). A node is processed by adding edges between all its parents. The* induced width of ordered graph *(G, d) is the maximum number of parents any node has in (G^*, d). The* induced width of graph *G is the minimal induced width over all possible orderings d.*

The process to construct (G^*, d) matches the behavior of an interaction graph when eliminating variables in the reverse order of d. Given a variable elimination approach for a discrete setting with starting interaction graph G and elimination ordering d, the number of variables in the largest intermediate factor is equal to the induced width of (G, d) plus one. The time and space complexity of the variable elimination approach is exponential in the induced width (Dechter, 2013).

Heuristic Variable Ordering

Given an interaction graph G, d should be chosen such that the induced width of (G, d) is minimal. This minimizes the size of the intermediate factors for the discrete setting and the size of the resulting equation (symbolic expression tree) for the hybrid setting. Unfortunately, finding the induced width of a graph is NP-complete in general (Dechter, 2013). Nevertheless, there are reasonable heuristics such as min-degree, min-induced-width and min-fill. *Min-degree* constructs the ordering d for interaction graph G in reverse order by iteratively selecting the variable v with the lowest degree in G and removing v and its edges from G. This idea is also used in linear decision diagrams (Chaki et al., 2009) to perform existential quantification of continuous variables. Min-induced-width and min-fill are similar but connect all neighbors of v before removing it. *Min-induced-width* selects the node with the lowest degree but, because of the modification, also accounts for previously added edges. *Min-fill* selects v based on the minimum number of edges required to connect the neighbors (the fill-in edges). None of the heuristics work best on all problems. In general, min-fill has shown to be usually slightly better than min-induced-width, and min-degree has shown to be the worst of the three (Dechter, 2013; Kask et al., 2011; Koller and Friedman, 2009).

The three heuristics originate from the work in the discrete setting. When we construct the interaction graph for the continuous case by treating atomic $\mathcal{LRA}$ literal as factors, the interaction graph provides the same kind of information as for the discrete case (minimise the induced width). We can therefore also apply these heuristics to the continuous setting.

6.4 Variable Trees

Instead of performing computations on factors (cf. Section 6.3), we investigate a search-based approach that consists of recursively conditioning on variables. Consider for example $\sum_A P(B = 1|A)P(A)$, previously solved by taking the

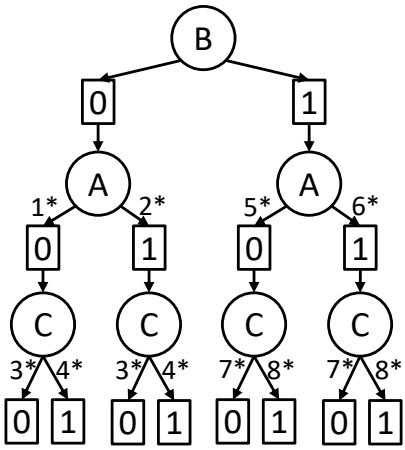
product of both $P(B = 1|A)$ and $P(A)$ before eliminating A. A search-based approach solves this problem by first conditioning on $A = 0$, computing the product, and summing it with the result of conditioning on $A = 1$. The advantage of this approach, when evaluated in a depth-first manner, is that it requires less memory compared to reasoning over complete factors ($A = 0$ and $A = 1$ at the same time). An extension of this approach exploits independencies that result from conditioning on variables. The order in which variables are branched on is in this extension a tree of variables instead of a simple chain variable ordering.

6.4.1 AND/OR Graphs

An *OR-tree* is formed by repeatedly conditioning on variables according to variable ordering d (Figure 6.5). Each conditioning represents an OR-node (circle) branching on the different possible values for the variable. A path or trace in the tree represents an assignment to each variable such that in a leaf node the variables in all factors have been instantiated to a value. Instead of computing the weight for each leaf as the product of instantiated factors, distributivity is used to push instantiated factors as close to the root as possible. This means that as soon as all variables of a factor have been instantiated, the value of the factor can be taken into account (placed on the edge), reducing computations (Dechter, 2013). More formally, each factor $f(\mathbf{X})$ can be taken into account when all values for variables $\mathbf{X}$ have been assigned a value.

Example 38. *The two leaves on the left of Figure 6.5 only differ in the assignment for C. Instead of computing $P(A = 0)P(B = 0|A = 0)P(C = 0|B = 0)$ for the left leaf, $P(A = 0)P(B = 0|A = 0)P(C = 1|B = 0)$ for the right leaf and summing up the results, distributivity can be used to push the shared part, $P(A = 0)P(B = 0|A = 0)$, higher in the tree.*

An AND/OR tree is an extension that exploits more independencies. For example, after conditioning on B, $P(C|B)$ becomes independent from $P(A) \times P(B|A)$ (Figure 6.6). This is used to split the computations into multiple parts (AND-node with a branch for A and one for C), graphically represented by connections going to multiple OR-nodes after a decision. When a subtree occurs multiple times, the parents can refer to the same subtree, reusing the computations. This behavior can be obtained through caching and results in graphs instead of trees. The AND/OR structure is not guided by a simple variable ordering but by a variable tree (Figure 6.6).



Nr.	Factors
1*	$P(A{=}0)P(B{=}0\|A{=}0)$
2*	$P(A{=}1)P(B{=}0\|A{=}1)$
3*	$P(C{=}0\|B{=}0)$
4*	$P(C{=}1\|B{=}0)$
5*	$P(A{=}0)P(B{=}1\|A{=}0)$
6*	$P(A{=}1)P(B{=}1\|A{=}1)$
7*	$P(C{=}0\|B{=}1)$
8*	$P(C{=}1\|B{=}1)$

Figure 6.5: OR-tree with $d = B, A, C$ and table of weights (x*).

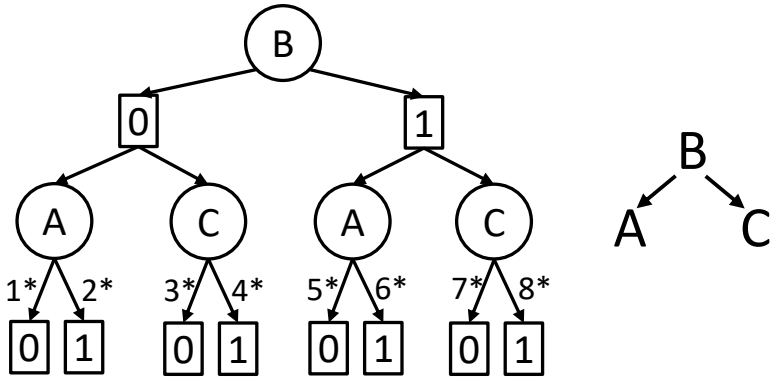
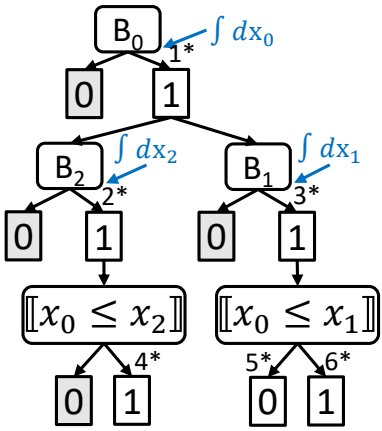


Figure 6.6: AND/OR-tree (left) and its guiding variable tree (right). $B = 0$ (and $B = 1$) splits into two OR-nodes, A and C, indicating an AND-node.

A guiding tree is only valid for a problem when variables that co-occur in a factor are not split over different AND branches. Given a problem with interaction graph G, any tree of G is a valid guiding tree for that problem (Dechter, 2013).

Definition 34 (pseudo tree)**.** *A* pseudo tree *of interaction graph $G = (V, E)$ is a directed rooted tree $\mathcal{T} = (V, E')$ with the back-arc property. This property states that for each edge e, if $e \in E$ and $e \notin E'$ then e is a back-arc edge, i.e., an edge that connects a node with one of its ancestors (Dechter, 2013). The back-arc property ensures that variables that occur in the same factor are not split over different AND branches.*



Nr.	Weights
1*	$[\![0 \leq x_0 \leq 1]\!]$
2*	$[\![0 \leq x_2 \leq 1]\!]$
3*	$[\![0 \leq x_1 \leq 1]\!]$
4*	$[\![x_0 \leq x_2]\!]$
5*	$c_2[\![x_0 > x_1]\!]$
6*	$c_1[\![x_0 \leq x_1]\!]$

Figure 6.7: AND/OR Graph and weight table (x*) for the continuous setting.

For example, when A and C would have co-occurred, an AND node cannot split the factors. In the guiding tree there would be a connection between A and C (Figure 6.6), violating the back-arc property.

So far, we conditioned on discrete variables, branching over the values in their domains. This is more challenging for continuous variables. We propose to branch on the atomic $\mathcal{LRA}$ literals instead, which can be considered as branching over different value intervals. This also implies that the guiding tree will contain $\mathcal{LRA}$ atoms instead of continuous variables.

Example 39. *The AND/OR graph in Figure 6.7 represents the following problem,*

$$\int \Big[c_1 [\![x_0 \leq x_1]\!] [\![x_0 \leq x_2]\!] \left(\textstyle\prod_{i=0,1,2} [\![0 \leq x_i \leq 1]\!]\right) + \tag{6.14}$$

$$c_2 [\![x_0 > x_1]\!] [\![x_0 \leq x_2]\!] \left(\textstyle\prod_{i=0,1,2} [\![0 \leq x_i \leq 1]\!]\right) \Big] dx_0 dx_1 dx_2$$

with $B_i = [\![0 \leq x_i \leq 1]\!]$ *and* c_1 *and* c_2 *as two constants. The weight of all grey decisions is* 0 *and for all others it is equal to the decision itself (see table). The structure allows parallel integration of* x_1 *and* x_2 *(Figure 6.8).*

To evaluate this structure, perform $+$ and $\times$ bottom-up for each OR- and AND-node and integrate out continuous variables as soon as possible. For example, after obtaining $[\![x_0 \leq x_2]\!] B_2$, we can integrate out x_2 as it does not occur at any later point. The order in which continuous variables can be integrated out forms an integration tree (Figure 6.8, left).

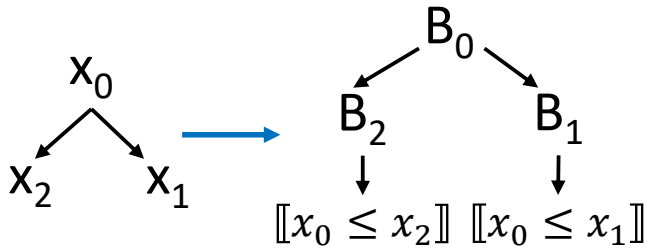


Figure 6.8: Integration tree and a guiding tree respecting it.

Definition 35 (integration tree)**.** *An* integration tree *is a pseudo tree where each node is associated with a continuous variable, except for the root node where it is optional. When the interaction graph contains disconnected subgraphs, the root of the integration tree can be empty.*

Previously the ordering to compute the induced-width was the variable ordering d, now it is specified by the ancestor relation in the integration tree.

Integrating out continuous variables yields new inequalities, much like the intermediate factors created in the discrete setting. However, our continuous approach branches on $\mathcal{LRA}$ atoms (corresponding to factors in the discrete setting). Do note that newly introduced inequalities do not become part of the search structure, only of the intermediate computations. The complexity of evaluating the structure is influenced by the depth of the guiding tree and the complexity of the partial integrations and their intermediate results. The latter is related to the induced-width of the integration tree. While the guiding tree affects the size of the structure, we empirically found minimising the integration time to be more important. Hence, we propose to first use heuristics to find an integration tree of continuous variables and only then convert the tree into a pseudo tree of $\mathcal{LRA}$ atoms that respects this integration order (Figure 6.8).

Sentential Decision Diagrams. We explained the role of variable ordering and how to analyse its influence, in the context of AND/OR graphs. We stress that our AND/OR graph in the context of continuous variables is solely illustrative. F-XSDD(BR), the state-of-the-art approach that we extend and evaluate in the experiments, instead uses an SDD compiler. Even though there are many differences between these two structures (e.g. conditioning on a variable versus a sentence in SDDs), the role of the variable ordering and its influence on the computations remains the same. We chose to illustrate the influence using AND/OR graphs as its existing literature on orderings is closer in focus to our approach. A pseudo tree of $\mathcal{LRA}$ atoms and Boolean variables used to guide

the construction of an AND/OR graph can easily be translated into a vtree to guide an SDD. Boolean variables are not constrained by the integration tree. In our implementation we use them to heuristically balance the vtree. This aspect of the procedure has potential for improvements, by considering the impact on the representation size, which we leave for future work.

6.4.2 Pseudo-Tree Heuristics

The size (and complexity) of AND/OR graphs are controlled by their guiding tree. Finding a minimal height pseudo tree is, similar to minimal induced width, NP-complete (Dechter, 2013). We discuss three heuristics, the first two minimise the induced width, and the third one minimises the tree height. The second heuristic is novel, the first and third were adapted to the continuous setting.

Top-Down Pseudo-Tree. This heuristic constructs a pseudo tree in two steps. First, obtain a variable ordering d through, for example, the previously discussed min-fill approach. Second, given the induced interaction graph G along d, a pseudo tree can be constructed top-down by traversing the induced-ordered G in a depth first manner starting from the first variable in d and prioritising variables earlier in d to break ties (Dechter, 2013).

Bottom-Up Pseudo-Tree. The first step in the previous approach, obtaining d, is unaware of the second step, constructing the pseudo tree. When breaking ties, it hence does not consider the effects on the height of the resulting tree. We propose a new heuristic that interleaves both steps and constructs the pseudo tree bottom-up. By interleaving, the variable selection heuristic can consider the effect on the tree height and make decisions to minimise it. Our heuristic keeps track of several tree roots (branches that are being extended in parallel, bottom-up) which are iteratively extended with new variables. When a variable v is added, it either 1) extends a root, 2) yields a new root or 3) combines multiple roots, depending on whether any of the variables in the current trees were previously neighbors (interacted with) of v. The next variable to add to the trees is selected using a min-fill metric, breaking ties by prioritising the variable that results in the most shallow trees. We refer to our heuristic as *balanced bottom-up min-fill* (**BU-MiF**).

Minimize Height. To minimise the height of a pseudo tree, a hypergraph decompositioning approach can be used to create a (roughly) balanced tree. To

convert the problem into a hypergraph, create a vertex for each factor and a hyperedge for each variable v, connecting all factors that contain v. A pseudo tree can be obtained from the hypergraph by recursively partitioning the vertices into two (roughly) balanced sets while minimising the cut (hyperedges crossing the two sets) (Dechter, 2013). When a variable is instantiated (cut), factors can become independent and can be solved separately (AND-node).

Continuous Setting. The first two heuristics can be applied to continuous variables by changing the variable selection process to use interaction graphs adapted to the continuous setting (Section 6.3). For the continuous setting, these heuristics return an integration tree, providing the order in which to integrate out continuous variables (Figure 6.8, left). When employing an approach that conditions on $\mathcal{LRA}$ literals instead of continuous variables, the integration tree must first be converted into a guiding tree of literals respecting that ordering (Figure 6.8, right). When using SDDs, the integration tree should instead be converted into a vtree. This two-step decomposition is not present in the discrete case and is crucial to apply these heuristics to the hybrid setting.

Using the hypergraph decompositioning approach, a vtree can also be created directly by recursively partitioning the variables in two sets (minimising the cut), forming the left and right subtrees of the vtree. When using $\mathcal{LRA}$ atoms as vertices and hyperedges as shared continuous variables, the min-cut has an additional meaning compared to the discrete setting. The min-cut is the set of variables shared by the $\mathcal{LRA}$ atoms in both sets, indicating the depth at which those variables can be integrated out. By minimising this cut, we minimise the number of variables that can only be integrated out high in the structure, maximising deeper and smaller integrations.

6.5 Experiments

PyWMI is a software package designed to solve WMI problems. It includes the state-of-the-art solver F-XSDD(BR) which compiles WMI problems to XSDDs heuristically minimising the height by balancing the vtree. This heuristic is agnostic to which continuous variables occur in a modulo theory atom and how these continuous variables interact. We extend this solver with the vtree heuristics discussed in Section 6.4.2, yielding a more robust solver that no longer has to rely on a user-provided ordering. This process consists of constructing 1) the interaction graph of the problem, 2) the integration tree using the discussed heuristics, 3) a vtree that respects the ordering of the integration tree, 4) the XSDD using that vtree and 5) evaluating the XSDD to obtain the result.

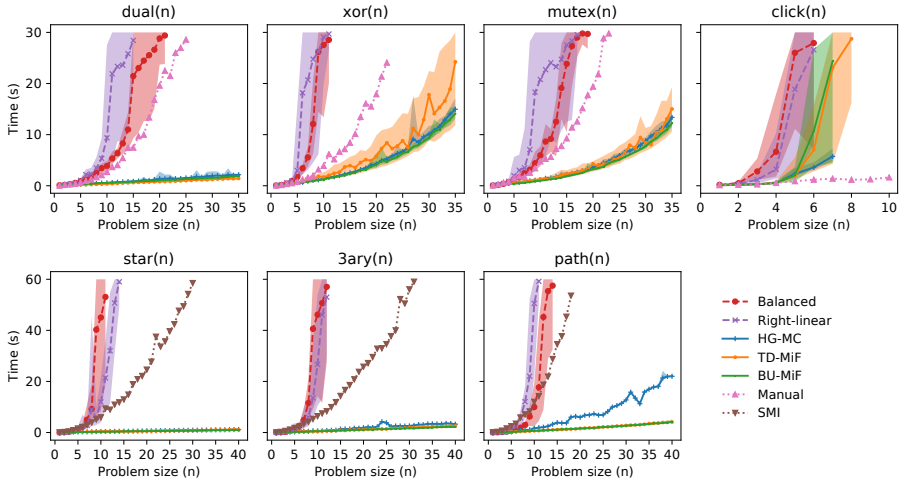


Figure 6.9: Comparison of run times for different variable ordering heuristics. Using the F-XSDD(BR) solver, the run times include time spent on the variable ordering heuristics (negligible), the compilation step, and the evaluation step.

We consider seven problem templates whose size is controlled by parameter n: *dual(n)*, *xor(n)*, *mutex(n)*, *click(n)*, *star(n)*, *3ary(n)* and *path(n)* (Kolb et al., 2019b; Zeng and Van den Broeck, 2019). The last three problem classes belong to a subset of tractable WMI problems. The SMI solver (Zeng and Van den Broeck, 2019) is specialized to exploit this type of problem structure and has outperformed F-XSDD(BR) on these problems. We evaluate three sets of heuristics: 1) hypergraph decompositioning (HG-MC) and top-down min-fill (TD-MiF), both of which we adapted from the discrete setting; 2) our new bottom-up min-fill heuristic (BU-MiF); and 3) F-XSDD(BR)'s current heuristic (*balanced*) and a right-linear heuristic (corresponding to a chain variable order, forming an OBDD). For the first four problems we also compare to the *Manual* approach, which uses the balanced heuristic on a variable order manually adjusted by Kolb et al. (2019b) and is considered to lead to a good ordering. For the last three problems, we compare to SMI.

For every problem, each heuristic ran 10 times with randomized orderings for increasing n. The minimum, maximum, and average run times are recorded. For the first four problems we run up to $n = 35$ with time-out $t = 30s$, and for the last three problems up to $n = 40$ with $t = 60s$. If, in one iteration, a heuristic times out for a given value of n, its run time is set to the time-out, and larger values of n are skipped. All results are shown in Figure 6.9. Code is available at `https://github.com/VincentDerk/BU-MiF`.

Q1: How does top-down min-fill (TD-MiF) compare to the newly introduced balanced bottom-up min-fill (BU-MiF) heuristic? Both BU-MiF and TD-MiF perform similar in terms of total run time on the used benchmarks. The bottom-up approach of BU-MiF, however, always it to be more consistent, breaking ties by focusing on the balance of the integration tree. This is especially apparent on the *xor(n)*) benchmark: BU-MiF's performance is very consistent while TD-MiF's is more variable (larger gap between maximum and minimum run time).

Q2: Do the contributed heuristics improve the problem-agnostic heuristics (balanced and right-linear)? Analysing the run time results of Figure 6.9, we conclude that the proposed heuristics perform significantly better than the previously used heuristics that were problem-agnostic (balanced and right-linear). Only the *click(n)* results are less favorable. Our investigation shows that for the click benchmark the interactions between the continuous variables, on which our heuristics are based, do not have a large impact. In this case it is more advantageous to consider optimising the SDD size itself, which our heuristics currently ignore. Indeed, the Manual heuristic that performed a lot better here had an SDD size of 5 for both $n = 6$ and $n = 7$ while the BU-MiF heuristic averaged an SDD size of 6145 and 1537 respectively. BU-MiF currently converts the integration tree into a vtree that respects the integration order and balances the literals (including Boolean variables) to minimise the depth of the SDD. In future work we can optimize this conversion by analysing the logical formula to obtain more succinct SDDs that still respect the integration order.

There is a relatively large difference between the minimum and maximum run time for the balanced and right-linear heuristics, compared to the newly proposed heuristics. This indicates that the proposed heuristics are less susceptible to unfavorable input orders from the user, yielding a faster and more robust solver. Especially the BU-MiF heuristic is very consistent in terms of resulting run time.

In addition, BU-MiF, HG-MC and TD-MiF also improve over SMI. We found that SMI spends a lot of time finding the integration intervals. The complexity of SMI can be super-exponential in the worst case (Zeng and Van den Broeck, 2019), for instance, with a path primal graph such as in *path(n)*. In contrast, on these benchmark problem instances, the run times of our newly contributed heuristics appear to merely grow linearly.

Q3: Should we minimise the induced-width or the depth of the integration tree? We compared the hypergraph decomposition heuristic (HG-MC) with BU-MiF (min-fill metric to minimise the induced-width). In general, they seem to perform similarly. We computed the induced-width of the solutions returned by both approaches and found that, except for *path(n)*,

they had the same induced-width. The *path(n)* problem, where depth was prioritised over induced-width, suggests that optimising the induced-width is more important. Additional benchmark instances are necessary to reach a conclusive result. However, it is more likely that for some instances it is better to optimise the induced-width while for others it is better to prioritise the depth. A similar conclusion was reached for the discrete-variable setting (Dechter, 2013).

6.6 Conclusion

A crucial element of performing efficient probabilistic inference over discrete random variables is the order in which variables are marginalized out. In this chapter we have shown that the importance of variable ordering also extends to problems in the discrete-continuous domain. We analyzed the influence of the variable ordering in the continuous setting by identifying parallels between probabilistic inference over discrete and continuous random variables and mapping concepts from the discrete setting, such as interaction graphs, to the continuous setting. This allowed us to adapt variable ordering heuristics developed for discrete random variables to perform probabilistic inference over continuous ones.

We introduced a new heuristic (BU-MiF), which significantly outperforms previous heuristics (Balanced, Right-linear) and is more robust than the heuristics adapted from the discrete setting (HG-MC, TD-MiF). BU-MiF also allows F-XSDD(BR) to outrun the specialized SMI solver on a set of benchmark problems from the tractable WMI subclass it addresses.

Future work includes investigating ways to exploit additional information about the logical structure of the WMI support. This could lead to smaller (more succinct) compiled representations for problems such as the *click(n)* problem. An adaptation of our heuristic to an iterative anytime scheme can also be considered (Kask et al., 2011).

Chapter 7

Modulo Theory Compilation

This chapter was previously presented at the Workshop on Counting and Sampling 2023:

> V. Derkinderen, P. Zuidberg Dos Martires, S. Kolb, and P. Morettin (2023d). "Top-Down Knowledge Compilation for Counting Modulo Theories". In: *CoRR* abs/2306.04541. accepted at Workshop on Counting and Sampling at SAT 2023. DOI: 10.48550/arXiv.2306.04541

I helped conceive the problem and initial compilation idea during a brainstorming session with S. Kolb. I then further refined the idea based on existing DPLL(T) literature, developed a prototype version, and led the publication's writing process. Minor changes were made to improve clarity and integration within this dissertation.

We previously discussed variable ordering heuristics for counting in the context of continuous variables, i.e., for weighted model integration. The reader may have noticed, however, that the knowledge compilation techniques themselves remained agnostic to the background theory. Indeed, the F-XSDD(BR) solver used in the experiments of Chapter 6 abstracts the support formula ψ and uses a propositional SDD compiler. This means, for example, that the compiler is unaware that $(x < y) \wedge (y < 0)$ implies $(x < 0)$, which may negatively affect downstream tasks such as weighted model integration.

In this chapter we discuss knowledge compilation for counting in the context of background theories and propose a new framework for compilation in this setting. We thereby contribute to the research question:

> **RQ2.3 How to perform knowledge compilation for counting over modulo theory formulas?**

The remainder of this chapter is organized as follows: Section 7.1 provides a brief introduction. Section 7.2 contains preliminaries, introducing the satisfiability modulo theory problem and how to solve it using the DPLL(T) algorithm. Section 7.3 then discusses the d-DNNF properties in the context of counting with background theories, with a specific focus on (linear) real arithmetic. Afterwards, in Section 7.4, different compilation strategy choices are discussed. Finally, Section 7.5 puts forward the proposed research direction for a future knowledge compiler. The chapter concludes in Section 7.6.

7.1 Introduction

Recall that a key motivation of knowledge compilation is to compile a logical formula into a target language whose properties allow for certain tasks to be performed in time polynomial in the representation size (Darwiche and Marquis, 2002), and that d-DNNF is the target language of interest when counting over a logical formula (Section 2.4.1). Furthermore, Huang and Darwiche realised that d-DNNF compilation can be achieved through exhaustive DPLL search by simply storing the search traces (Darwiche, 2004; Huang and Darwiche, 2005) (Section 2.4.2).

Top-down knowledge compilers, which are based on exhaustive DPLL, have primarily been focused on model counting over propositional logic formulas. In comparison, few works exist that consider formulas with an implicit background theory (Barrett and Tinelli, 2018). Additionally, those few works focus on specific background theories (Chaki et al., 2009; Koriche et al., 2015; Møller et al., 1999; Niveau, 2012; Sanner et al., 2011). A more general algorithmic framework, that we propose here, is missing.

In this chapter we do consider counting in the context of background theories, and discuss compilation strategy choices for the quantifier-free background theory setting. Most importantly, we draw parallels with the propositional counting variant and identify a DPLL(T) (Nieuwenhuis et al., 2006) based approach as a promising direction for a future knowledge compiler.

7.2 Background

The classic satisfiability problem, or SAT, is defined over propositional logic. Its analogue in the background theory setting is called the *satisfiability modulo theory* problem, SMT for short (Barrett et al., 2009). This problem extends SAT with respect to a certain decidable background theory T, notable examples of which include linear real ($\mathcal{LRA}$) or integer ($\mathcal{LIA}$) arithmetic and fixed-size bit vectors ($\mathcal{BV}$), for which we refer to Section 2.5. The SMT task is to determine whether there exists an assignment to the variables such that the input formula ψ is satisfied with respect to T. For example, suppose the $\mathcal{LRA}$ formula ψ is $(x < y) \vee \big(A \wedge (x > 10)\big)$, then $m = \{x \mapsto 11, y \mapsto 0, A \mapsto true\}$ is a model, as part of the *T-satisfying truth assignment* $\{(x < y) \mapsto false, (x > 10) \mapsto true, A \mapsto true\}$. We stress the important distinction between model and T-satisfying truth assignment, the latter of which assigns a truth to each $\mathcal{LRA}$ atom such that it is consistent with both the background theory T and formula ψ.

DPLL(T) is a generalisation of the DPLL algorithm, designed to solve SMT problems (Nieuwenhuis et al., 2006). The key difference with the classic DPLL algorithm is that DPLL(T) involves a *theory-specific solver* that interacts with the DPLL algorithm, using the background theory T to propagate additional atoms when possible. For instance in a prior example, when assigning both $(x < y)$ and $(y < 0)$ to true, the theory-specific solver would automatically propagate $(x < 0)$ to be true. We note that most current implementations are actually extensions of the conflict-driven clause learning (CDCL) algorithm, which augments the DPLL algorithm with conflict clause learning and backjumping (Moura and Bjørner, 2008; Nieuwenhuis et al., 2006; Schrag, 1997). For the purpose of our discussion we will use DPLL(T) to refer to both DPLL(T) and its CDCL(T) augmentation, similar to how we have been using the term DPLL.

Just as SMT generalizes SAT with additional theories, counting modulo theories (#SMT) generalizes model counting (#SAT) with additional theories. In stark contrast with the purely propositional setting, formulas involving numerical theories may have infinitely many models. While in this paper we mainly consider the quantifier-free $\mathcal{LRA}$ setting, we expect the proposed ideas to apply to a broader set of quantifier-free theories.

7.3 d-DNNF for Modulo Theory

We now discuss in more detail the d-DNNF properties (Darwiche and Marquis, 2002) in the context of #SMT.

Similar to the DPLL algorithm, negation only occurs in the leaf nodes when tracing the DPLL(T) algorithm. The fact that atoms can be associated with additional logic (e.g. $\mathcal{LRA}$ atoms such as $x<5$), and that a theory-specific solver aids the propagation process, does not change this. The traces of a DPLL(T) algorithm are also **deterministic**, i.e., the children of each $\vee$-node do not share any models, because every decision associated with $\vee$ still partitions the models. The **decomposability** property is defined as $\wedge$-node branches not sharing any variables. In the modulo theory setting this needs further specification, because different atoms can still be linked together through their inner logic (e.g., shared continuous variables).

Decomposability on the level of atoms is insufficient to ensure the decomposability of counting tasks on the formula. Consider the following example where the $\wedge$-branches do not share any atoms, but do share variables:

$$[(x < 5) \vee (x < y)] \wedge [(x > 10) \vee (y > 10)]$$

Here, $\{(x < 5), (x \geq y)\}$ and $\{(x > 10), (y > 10)\}$ are satisfying truth assignments for respectively the left and right $\wedge$-branch. Together, however, they do not form a theory-satisfying truth assignment because $(x < 5)$ and $(x > 10)$ clearly conflict. The benefit of decomposing on the level of variables is that the previous situation is impossible, making it easier to ensure that each truth assignment is indeed theory-consistent. Unfortunately such decomposability can not be achieved in general. Consider as an example

$$\Big((x < y - 1) \vee (x > y + 1)\Big) \wedge \Big(\neg(x < y - 1) \vee (x > 20)\Big)$$

where every $\mathcal{LRA}$ atom mentions x. Clearly it is impossible to represent this formula while enforcing variable decomposability. For the purpose of our discussion on modulo theory compilation, we will therefore use the d in d-DNNF to refer to decomposability on an atomic level rather than on a variable level. The idea behind this is that it 1) ensures the generality of our compilation approaches to the purely propositional setting, 2) allows us to associate weights to $\mathcal{LRA}$ atoms, because the decomposability prevents situations such as $(x > 10) \wedge (x > 10)$ which would otherwise consider the same weight more than once, and 3) prevents representations from containing $(x \leq 10) \wedge (x > 10)$, thereby improving theory consistency.

In addition to these d-DNNF properties we expect that all truth assignments captured by the compiled formula are theory satisfiable. This is not necessarily

ensured when using a theory-agnostic compilation approach, which we discuss in more detail in the next section. For now, note that the traces of an exhaustive DPLL(T) algorithm will produce exactly such d-DNNF formulas where each truth assignment is theory satisfiable.

7.4 Compilation Strategies

7.4.1 Theory Aware versus Theory Agnostic

Approaches for compiling a formula with respect to a background theory T can be divided into two categories: theory-aware versus theory-agnostic approaches. In case of the latter, a Boolean abstraction of the input formula ψ is first created, i.e., the formula that is obtained by replacing every theory atom in ψ with a fresh Boolean atom. Afterwards, the Boolean abstraction can be compiled into a d-DNNF formula using any off-the-shelf propositional compiler. As a consequential benefit, advancements made to those compilers are automatically inherited. Evidently, theory-agnostic approaches also have a downside compared to approaches that are more theory-aware. Namely, since the compiler is unaware of T, the resulting d-DNNF formula may contain truth assignments that are not consistent with T. For instance, when abstracting $\psi = (x \leq 0) \vee (x \geq 1)$, both literals originating from the abstraction could be assigned to true, forming a T-unsatisfiable truth assignment. It is then up to the downstream inference algorithm that uses the resulting d-DNNF to deal with the inconsistent models, typically impacting negatively the run time of the downstream task. For example, this theory-agnostic approach was adopted by Kolb et al. (2019b) in their F-XSDD(BR) solver discussed in Chapter 6, for solving weighted model integration problems, i.e., weighted #SMT problems over $\mathcal{LRA}$ formulas. As a consequence, a potentially large number of intermediate computations during the online integration procedure would result in regions with no models and would be discarded.

Figure 7.1 shows an example of a formula representation that may result from a theory-agnostic approach (7.1a), compared to a theory-aware approach (7.1b and 7.1c). In this example, the formula representation shrank once the theory was considered. This is not always necessarily the case: the effect that theory awareness has on the representation size heavily depends on the exploitable structure, which may increase or decrease. Depending on the application requirements, we could even consider a more condensed representation that omits theory-implied atoms (Figure 7.1c).

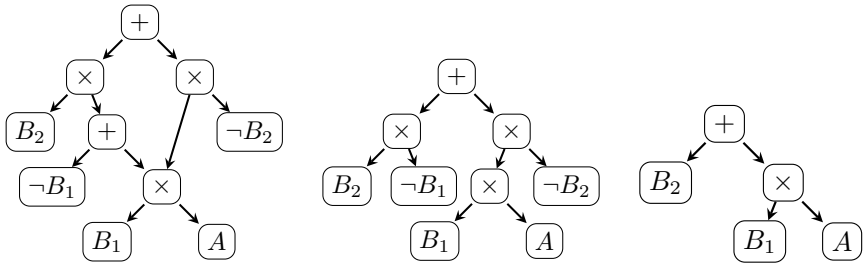


Figure 7.1: Different representations of $(B_1 \vee B_2) \wedge (\neg B_1 \vee A)$, with abbreviations $B_1 = (x < y - 1)$, $B_2 = (x > y + 1)$, and $A = (x > 20)$.

7.4.2 Eager versus Lazy Solving

It is possible to exploit the existing purely propositional tools while still achieving theory awareness, by first adapting the Boolean abstracted input formula ψ in a way that no theory unsatisfiable truth assignments arise. This idea was used by early SMT solvers, where it is called 'eager solving' (Nieuwenhuis et al., 2006). For example, if ψ includes both $(x \leq 0)$ and $(x \geq 1)$, which were respectively abstracted into A_0 and A_1, then also add $A_0 \implies \neg A_1$ or $A_1 \implies \neg A_0$ to ψ. Afterwards, because ψ is purely propositional, any SAT solver (or in our case d-DNNF compiler) can be used. Since the adapted abstraction step is also present in eager SMT solvers, we can simply implement an eager theory compiler by using an existing eager SMT solver as the foundation, swapping their SAT solver for a propositional d-DNNF compiler.

A major disadvantage of the eager approach is that depending on the theory T, it may introduce many additional variables, and greatly increase the formula size of the abstracted input formula ψ. For this reason, numerical theory SMT solvers have instead progressed towards using a so-called 'lazy solving' (Nieuwenhuis et al., 2006) approach that we discuss next. This does not mean, however, that the eager approach has become irrelevant. It still forms the state-of-the-art SMT approach for certain theories such as $\mathcal{BV}$[1]. We hypothesize that similar conclusions can be drawn for the compilation setting, i.e., that a lazy approach is most suitable for numeric background theories.

In the most basic form, the lazy approach involves purely propositional reasoning over ψ, and verifying (partial) assignments with a theory solver to make sure that every model is indeed theory satisfiable. Enhancements of this approach

[1]The 2022 SMT competition winner of the incremental quantifier-free $\mathcal{BV}$ track was Yices 2 (Dutertre, 2014), which uses bit blasting for $\mathcal{BV}$, an eager solving approach.

have progressed to a more proactive design where the theory solver instead helps to propagate atoms and learn conflict clauses. This approach is the DPLL(T) (or CDCL(T)) algorithm discussed in Section 7.2 and is used by most state-of-the-art SMT solvers, including cvc5 (Barbosa et al., 2022), SMTInterpol (Christ et al., 2012), MathSAT (Cimatti et al., 2013), Z3 (Moura and Bjørner, 2008), Yices (Dutertre, 2014), and OpenSMT (Hyvärinen et al., 2016).

7.4.3 Top-down versus Bottom-up Compilation

Recall that DPLL-based compilers are so-called top-down compilers: they start the compilation process from the whole input formula and work their way down (by branching on literals). In contrast, bottom-up compilers process an input formula ψ from the bottom of the expression to the top. For example, if ψ is $(A \vee \neg B) \wedge (\neg A \vee C)$, then the two conjuncts are separately compiled before processing the conjunction itself. This design implies an apply-operation that performs conjunction, disjunction, and negation. The apply-operation enables the incremental construction of (multiple) formulas, while a top-down compiler requires the complete formula to be known upfront. To achieve an efficient apply-operation, however, the target language of bottom-up compilers is usually restricted to strongly deterministic and structured decomposable NNF, such as OBDD and the more general SDD class (Darwiche, 2011; Pipatsrisawat and Darwiche, 2008), strict subsets of d-DNNF. In comparison, #DPLL based top-down compilers instead produce Decision-DNNF, a strict superset of OBDD (Darwiche and Hirth, 2020) (but not of SDD with which it partially overlaps).

An advantage of bottom-up compilation is that it does not restrict the input form. In contrast, top-down compilation tools are commonly restricted to formulas in conjunctive normal form (CNF). While it is possible to translate any propositional formula into CNF, this costs time and may result in an exponential expression (when using De Morgan's law), or requires introducing additional variables (Tseitin, 1983).

An advantage of top-down compilation is that it has available the full information contained within ψ. This assumption allows more informed decisions, in turn leading to more compact compiled formulas. Oppositely, ψ may only become incrementally available to the bottom-up compilation, or it may have a pre-analysis step that considers the whole formula ψ to determine a suitable guiding structure (cf. static variable ordering heuristics for OBDD (Rice and Kulhari, 2008) or vtree heuristics for SDD (Darwiche, 2011)), but that structure can be suboptimal for intermediate formula representations. This naturally leads to the second point, which is that even when the final d-DNNF produced by bottom-up

compilation is small, the intermediate results of the apply-operation may grow to be very large, severely impacting the run time and memory requirements (Huang and Darwiche, 2004).

In terms of compilation in the modulo theory domain, almost all algorithms so far are bottom-up approaches. For example, the work on extended algebraic decision diagrams (XADD) (Sanner et al., 2011) uses a bottom-up approach. Within their implementation, they used a feasibility checker of a linear programming solver to prune theory unsatisfiable paths during- or after bottom-up compilation, observing impressive XADD size reductions. Other bottom-up compilers include work on difference decision diagrams (Møller et al., 1999) and linear decision diagrams (Chaki et al., 2009). Niveau (2012) proposed both a bottom-up and a top-down approach for compiling to interval automata and set-labeled diagrams. Koriche et al. (2015) created a top-down compiler targeting multi-valued decision diagrams.

7.5 Traces of an Exhaustive DPLL(T) Algorithm

In line with top-down compilation for the propositional setting, we propose to compile modulo theory formulas by storing the search traces generated by an exhaustive DPLL(T) algorithm. This is a theory-aware, top-down, lazy approach, which we hypothesize to be very suitable for compiling formulas with numerical background theories such as $\mathcal{LRA}$.

Existing SMT solvers based on the DPLL(T) framework form a natural implementation starting point. As explained in Section 7.2, the traces of such an approach form d-DNNF formulas. To further augment the implementation, it is worth investigating optimisations developed for #DPLL. For example, component decompositioning discussed in Chapter 4 is rarely used for satisfiability problems but has shown to be highly beneficial for model counting and compiling, especially in combination with caching (Bacchus et al., 2003). The augmentations are not straightforward, however, because of the implicit interactions between atoms: components must be decomposed on a variable level (rather than atom level), and previous decisions must be properly considered. Especially the latter is a novel challenge for caching. Consider the following example formula ψ': $\big((x < 3) \vee (x > 5)\big) \wedge \big((y < 0) \vee (y > 4)\big)$. Generally, branching on $(x > 5)$ does not imply $(y < 0)$. However, if $(x + y) < 5$ is a decision that has lead to the intermediate formula ψ', then branching on $(x > 5)$ while compiling ψ' does imply $(y < 0)$. This shows that the compiled form of ψ' is indeed influenced by previous decisions. This is in contrast to the

propositional setting, where after making a decision l and adapting formula ψ, the remaining formula $\psi|_l$ (here ψ') is independent of previous decisions.

Related work. To the best of our knowledge, we are the first to propose a general theory-aware top-down lazy compiler. The closest related works are of De Salvo Braz et al. (2016) and Ma et al. (2009), who use a DPLL(T)-like approach to count over modulo theories, but do not consider storing the traces for compilation. The work of Feldstein and Belle (2021), in contrast, does consider compilation but is positioned further from DPLL(T), excluding optimizations such as conflict clause learning. Moreover, there is limited background theory support, for example, atoms with arithmetic expressions such as $x + y < 2$ are not possible. Koriche et al. (2015) does support linear constraints, compiling a constraint network into a multi-valued decision diagram using a top-down approach, but only considers finite domains.

7.6 Conclusion & Future Work

We have provided a discussion on compilation strategies for (top-down) d-DNNF compilation for quantifier-free modulo theories. We specifically propose compilation through an exhaustive version of the DPLL(T) algorithm, drawing parallels with the propositional setting's evolution from SAT to #SAT. In future work, we aim to finalise the development of a tool based on these ideas, and plan to further investigate component decompositioning and caching for modulo theories. Additionally, refining the d-DNNF properties based on application requirements is of great interest.

Chapter 8

Conclusion

The journey within this dissertation has centered around model counting and knowledge compilation, and both their roles within state-of-the-art inference algorithms. This chapter concludes that journey, providing a summary of the included contributions and a discussion of future research perspectives.

8.1 Summary

This dissertation focussed on two overall research questions. What tasks can be cast into algebraic model counting problems? And how to then efficiently solve those algebraic model counting problems? We now summarize our contributions.

AMC as a Unifying Framework

Algebraic model counting is a very general problem into which several tasks can be cast by selecting the appropriate semiring. In other words, it forms a unifying framework. For instance, reasoning over possible worlds (that is, models) to address probability queries is a task that can naturally be cast as an algebraic model counting problem. Chapter 3 highlights this through a synthesis of the 15-year journey of the probabilistic logic programming language of ProbLog and its variants. It explains how inference in these languages is unifiable under the algebraic framework. Furthermore, the applicability of the algebraic model counting framework is certainly not limited to answering probabilistic questions:

Chapter 5 demonstrates this within a decision making under uncertainty setting. We specifically note the expected utility semiring, the ability to easily compute gradients, and the adaptation that deals with three operations instead of two. These findings contribute to the research question:

> RQ1) What tasks can be cast into algebraic model counting problems?

This question is important: probabilistic inference benefits from algebraic model counting, and others might too. Additionally, it helps to increase our understanding of the limitations of algebraic model counting, and how to adapt for them (e.g., to support three operations). Consequently, future research could continue to address this line of questioning: what other problems can be cast into algebraic model counting, or into the adaptation with three operations? Another related but more theoretical question: can we devise a framework of semirings (see also the work of Belle and De Raedt (2020), in the context of weighted model counting)? For example, nesting the expectation semiring into itself yields a second-order expectation semiring (Li and Eisner, 2009). What transformations on a semiring result again in a semiring? Such questions may (indirectly) help in understanding what sort of tasks can be cast into algebraic model counting.

Exploiting Symmetry

Exploiting structural symmetry present in model counting instances can drastically reduce the search space. Important is how to achieve this; an efficient detection of symmetries is non-trivial. In Chapter 4 we proposed an approach that involves only a minor change to the component caching mechanism of existing #DPLL-based model counters. Specifically, we found that storing the components in a way that cache hits occur when components are symmetrical is a very interesting approach: the idea requires few changes to existing implementations and transforms the challenge of efficient symmetry detection to the problem of efficiently computing canonical labels of a graph. The latter relates to solving graph isomorphism and is solved by existing tools such as Nauty (McKay and Piperno, 2014). This work answers the following research question

> RQ2.1) How to exploit structural symmetry while model counting on propositional logic formulas?

Empirically, we observed that the approach leads to a considerable reduction in the number of decisions required by the #DPLL algorithm. Despite the computational overhead associated with the new component representation, the approach translated into an increased number of solved combinatorial benchmark instances for SymGanak compared to Ganak (Sharma et al., 2019). To reduce the overhead incurred by detecting symmetries, several directions can be explored.

First, it is unclear still how to predict in advance whether many symmetries will be found in a particular instance. The proposed approach is able to exploit symmetries that only emerge after conditioning on variables, which significantly improves its ability to exploit symmetries, but also makes it harder to predict in advance whether many symmetries will be found. In particular because the emergence of symmetry depends on the order in which the algorithm branches on variables. A better understanding of this process would make it easier to selectively decide whether to activate the symmetry detecting caching mechanism for a given instance, or on a lower level, for a specific component encountered within that instance.

Second, we can further investigate variable ordering heuristics to find more symmetries. For example, it is worth exploring the integration of our approach with the D4 model counter (Lagniez and Marquis, 2017), which contains a variable selection and component decompositioning heuristic that might work particularly well together. The intuitive reasoning behind this statement is that the D4 heuristic aims to find many components, while more components also increases the chance of resulting in (symmetric) cache hits.

Third, alternative to the understanding of when the computational overhead is worth incurring, is a broader investigation into the symmetry detection mechanism and how to make it faster. For example, approximate graph isomorphism methods can be considered, as well as intermediate caching schemes.

Finally, while this dissertation has contributed to model counting, the symmetry caching mechanism could also prove useful for knowledge compilation, creating more compact representations. Related to this research direction is the work of Bart et al. (2014).

Our findings, that identifying symmetries in subproblems can be beneficial when solutions are shared among symmetric subproblems, raises an interesting question. Namely, whether these ideas can be applied more generally to search-based approaches beyond those for model counting. For instance, in SAT solving, or more broadly, constraint programming, or planning. In case of SAT solving, we note that existing work in this direction incorporates symmetry

breaking clauses to reduce the SAT search space, which is different from the approach we propose to investigate here. In case of constraint programming, we note that the branch-and-bound based algorithms used in this domain relate to the search-based approach that is DPLL. An interesting direction of research would be to explore this connection deeper and further exploit symmetry in this context (Kitching and Bacchus, 2007). In case of exploiting symmetries within knowledge compilation, we note that compilation tools have also been used while solving optimisation problems, e.g., while solving 0/1-integer linear programs (Becker et al., 2005). We hence speculate that the ideas presented in this dissertation may also benefit such domains.

Orthogonal to the research directions mentioned above, which use symmetry detection to solve subproblems faster, we can also use symmetry detection to unify subproblems (in the context of learning). For instance, the detection of symmetries may also help to identify higher-order abstractions in planning or inductive logic programming (Hocquette et al., 2023).

Knowledge compilation for modulo theories.

While knowledge compilation for counting over propositional theories has been well studied, research for counting with respect to a background theory is a lot less available. Among their use cases is the emerging domain of neuro-symbolic AI that aims to integrate both paradigms of symbolic and neural AI. In Chapter 7 we concluded with an interesting research direction to remedy the gap. Namely, we proposed a framework for compiling formulas with respect to a background theory, inspired by the evolution that occurred within the propositional domain: moving from DPLL to traces of a #DPLL algorithm. We expect this framework to work for any quantifier-free background theory. Our work on variable ordering and knowledge compilation thereby contributes to addressing the following research questions.

> RQ2.2) Can we extend variable ordering heuristics developed for the discrete domain to also work well for discrete-continuous domains?
>
> RQ2.3) How to perform knowledge compilation for counting over modulo theory formulas?

Neuro-symbolic AI is not the only research area that benefits from more general knowledge compilation tools. Another research area that we investigated is weighted model integration. Here, compilation tools are used to more compactly represent and guide the required computations.

The order in which to integrate out continuous variables has a large impact on weighted model integration solvers. This is analogous to the purely propositional case where the order in which to eliminate Boolean variables also has a large impact on the run time of elimination algorithms (e.g. weighted model counting, bucket elimination,...). As it turns out, the foundations on which Boolean variable ordering heuristics have been developed can also be adapted to apply to continuous variables. We showed this in Chapter 6 and proposed a novel heuristic based on these findings, called balanced bottom-up min-fill (BU-MiF). The experiments conducted with the WMI solver indicate a very significant improvement compared to previous ordering heuristics.

Nevertheless, further improvements are still possible. The proposed heuristics, for instance, only consider the integration order and ignore the representation size because the former is generally more important. There are cases, however, where also considering the representation size is needed (see the click graph example in Section 6.5). Ideally, the heuristics are adapted to consider both. As a second source of improvement: we contributed a static variable ordering heuristic, but the proposed #DPLL(T) based framework of Chapter 7 enables us to also investigate the use of a dynamic heuristic that allows the variable order to be different per decision branch.

Based on the observed improvements in our setting, we conclude this contribution with two higher-level takeaways. First, when compiled representations are used in an atypical setting, it is highly recommended to reconsider the existing heuristics. Second, (hyper-)parameters that have a considerable impact on the run time or result of an algorithm must not be left to the user to decide because they do not necessarily have the expertise to make this decision properly. Providing heuristics, in the simplest case a suitable default, furthermore eases integration of the algorithm within other packages.

8.2 Future Perspective

We now look ahead and consider the future of counting and knowledge compilation in the broader context of AI developments. This allows us to identify remaining open challenges, and provide possible broader research directions to help resolve or alleviate them.

Recent successes in artificial intelligence, particularly in the domain of generative models, have caused a large surge of public interest. This, in turn, has led to a large increase in the demand for suitable infrastructure and capable hardware. Also gaining in popularity within the AI community is neuro-symbolic AI (NeSy), a recent research domain that tries to combine the neural approaches,

which have proven to be very powerful, with the more traditional symbolic approaches, that are typically more principled and trustworthy (Hitzler and Sarker, 2021; Hochreiter, 2022; Manhaeve et al., 2021b).

Knowledge compilation is highly relevant to NeSy, as it allows to compactly represent knowledge in ways that allow efficient reasoning and integration with neural approaches. This is proven by the DeepProbLog language (Manhaeve et al., 2021a), a pioneering NeSy framework whose connection to counting and knowledge compilation has been described in Chapter 3. Other NeSy examples using knowledge compilation include semantic loss (Xu et al., 2018) and Scallop (Huang et al., 2021). We can therefore regard the compilation techniques discussed (and improved) in this dissertation as key to enabling further advancements in NeSy. Drawing parallels with the increased demand in hardware capabilities, we can consider knowledge compilation to be part of the infrastructure that enables NeSy and logic based reasoning in general, and anticipate an increase in their use.

The **open challenges** that primarily hinder this progress are scalability and accessibility. Below, we provide more detail on these two challenges.

Theory Aware Knowledge Compilation

The first challenge concerns accessibility. For example, a very common NeSy benchmark problem is MNIST addition (Manhaeve et al., 2018) whose knowledge ψ could be compactly represented as $digit_1 + digit_2 = digit_3$, where each $digit_i$ is an integer variable. Using compilation tools, this could automatically be transformed into a representation that allows efficient counting. Unfortunately, due to the limitations of used libraries and tools, applications often manually resort to a propositional version that introduces several Boolean variables for each digit (one representing $digit_1 = 1$, one representing $digit_1 = 2$, and so on), and introduces many constraints over those Boolean variables.

Within this dissertation we have therefore contributed to laying the foundations for more theory-aware knowledge compilation tools that support ψ. Still, to obtain a formal knowledge compilation map akin to the propositional setting (Darwiche and Marquis, 2002), a lot more work is required from both a theoretical and practical perspective. For instance, open questions include: what properties (like d-DNNF) are required for what applications, how to achieve these properties while compiling, and from a more practical perspective, how to transfer the benefits of caching and component decompositioning in the propositional domain to a more general setting.

Neuro-symbolic AI is not the only domain that would benefit from a

broader formalization of useful logical properties and tools to compile into such representation (Wang et al., 2023). Other examples include formal verification (Spallitta et al., 2022; Tang et al., 2023), and real-time SMT solving (similar to how current propositional compilers facilitate real-time SAT solving), which is for example used in product configuration applications (Popov et al., 2023; Sundermann et al., 2020; Thüm, 2020). We therefore expect this challenge to become even more relevant in the next few years.

Scalability

The primary benefits of neuro-symbolic AI, compared to current neural based methods, are the increase in robustness, trustworthiness, and the ability to learn from less data because knowledge can be manually inserted into the system. Key to the adaptation of NeSy is an efficient integration of the neural and symbolic reasoning. Given the current architectures in this domain, we expect to see improvements across three axis, over the next decade.

First, the improvement of knowledge compilation tools which will enable larger problem settings. Of importance here is the further theoretical development of the knowledge compilation map and the development of better tools. For instance, by exploiting symmetry, which relates to first-order inference. In brief: the ability to obtain smaller knowledge representations faster. Improvements along this axis will most likely also benefit other research areas that use knowledge compilation tools or weighted model counters.

Second, is the encoding of knowledge and how it is used within a neuro-symbolic architecture. This is different from the previous axis, as this focuses on what the knowledge is rather than what representation it is compiled into.

Third, is the adaptation to more efficiently execute on existing hardware, and the creation of new specialized hardware. One of the reasons that neural based methods became so relevant was the increase in data availability and compute, the latter being in the form of GPUs. Computations within the symbolic reasoning part of NeSy architectures currently form irregular dataflows which makes them less suitable to GPU usage (Shah et al., 2023). We expect to see improvements in this area over the next decade.

Appendix A

SymGanak: Results

This appendix chapter contains a description of the problem instances used for the experiments in Chapter 4 (Section A.1), and the empirical results (Section A.2) of those experiments.

A.1 Problem Classes

Our benchmarks are comprised of CNF instances from several different problem classes, that we explain below.

battleship There are 12 Battleship problems that were used in the evaluation of miniSAT-SPFS (Devriendt et al., 2012), and that originate from the 2011 SAT competition. They encode a type of puzzle that is similar to the classical two player board game with the same name. The instances were obtained from `https://github.com/JoD/minisat-SPFS/tree/master/cnf%20test%20files/battleship`.

fpga There are 20 instances that represent FPGA routing problems, introduced by Aloul et al. (2002). Those instances were obtained from `http://www.aloul.net/benchmarks.html`.

counting There are 9 *counting principle* instances where the model count represents the number of different ways that a set of M elements can be

partitioned into sets of size p. The instances were generated using the CNFGen tool (Lauria et al., 2017).

kcolor There are 23 instances of k-colouring graph problems, either with a grid structure (14 instances) or with a r structure (9 instances). The instances were generated using the CNFGen tool (Lauria et al., 2017).

parity There are 21 *parity principle* instances, where the model count of each instance represents the number of different ways that pairs can be formed with the n elements. The instances were generated using the CNFGen tool (Lauria et al., 2017).

tseitin There are 16 instances representing Tseitin transformation problems with either a grid (7 instances) or a random (9 instances) structure. The instances were generated using the CNFGen tool (Lauria et al., 2017).

grid There are 30 instances that represent grid problems: A grid network is an $N \times N$ grid, where each node has at most two directed edges to its neighbors, one right and one down. The model count is related to the number of paths between source (upper-left node) and sink (bottom-right node). The instances were obtained from `https://www.cs.rochester.edu/u/kautz/Cachet/Model_Counting_Benchmarks/Grid.zip`.

latin-squares There are 30 instances that represent quasigroup (Latin square) completion problems. Each model corresponds to the completion of a Latin square. These instances were generated using the code at `https://github.com/HelgeS/lsencode` (Gomes and Shmoys, 2002).

mcnc There are 17 instances from the LGSynth91 benchmark, primarily from the domains of logic synthesis and optimization. The instances were obtained from `https://ddd.fit.cvut.cz/prj/Benchmarks/`.

nqueens-classic There are 17 n-queens problems using the classical encoding where each variable denotes the presence of a queen in a specific cell. The instances were generated using the code from `https://sites.google.com/site/haioushen/search-algorithm/solvean-queensproblemusingsatsolver`.

nqueens-symmetry There are 11 n-queens problems specified using an encoding where each queen is uniquely identified (Wang et al., 2020). The instances were generated as described in that paper, using the Alloys tool (`https://alloytools.org/`).

problog The 6 ProbLog programs were created by compiling Bayesian Networks into the probabilistic logic programming language ProbLog (Fierens et al., 2015), which in turn transformed them into CNFs. The Bayesian Networks can be found at `https://www.bnlearn.com/bnrepository/`.

A.2 Results

The full results of our experiments are given in the tables below. A cactus plot showing that for our benchmarks, GANAK (VSADS) is superior to GANAK (CSVSADS), is shown in Figure A.1.

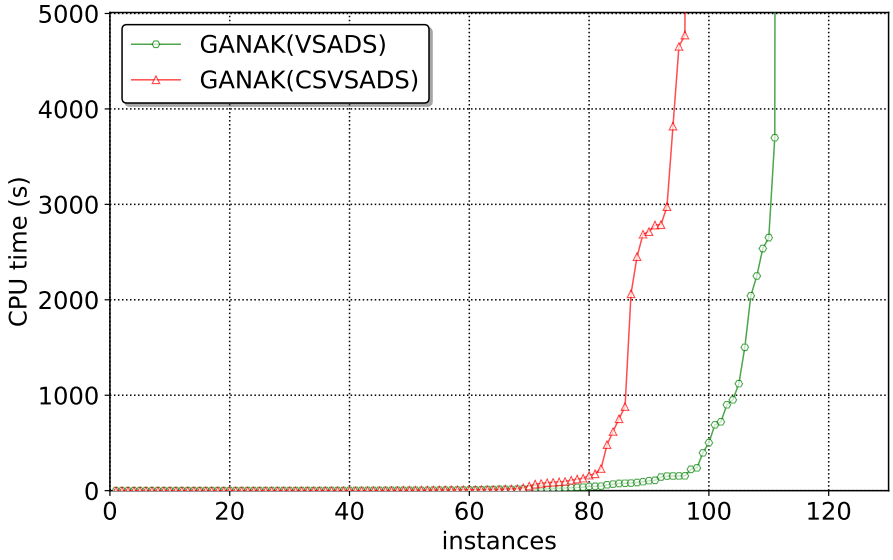


Figure A.1: Cactus plot comparing the VSADS and CSVSADS heuristic for GANAK.

Table A.1: Experimental results comparing different variable selection heuristics for SymGanak (here shortened to Sym): VSADS, CSVSADS (shortened to CSV), and ICSVSADS (shortened to ISCV).

Benchmark	**Vars**	**Clauses**	**Sym (VSADS)**	**Sym (CSV)**	**Sym (ICS)**
battleship					
1	91	322	TO	TO	TO
2	120	484	TO	TO	TO
3	153	693	TO	TO	TO
4	170	865	TO	TO	TO
5	180	910	TO	TO	TO
6	190	955	TO	TO	TO
7	276	1662	TO	TO	TO
8	364	2562	TO	TO	TO
9	378	2653	TO	TO	TO
10	435	3270	TO	TO	TO
11	496	3976	TO	TO	TO
12	1368	16308	TO	TO	TO
count					
1	495	162372	0.44	0.43	0.45
2	455	61440	1.69	1.68	1.69
3	1820	1652576	249.19	251.40	240.35
4	816	165258	222.88	218.51	223.65
5	1330	377076	4632.97	4641.06	4697.19
6	2024	765096	TO	TO	TO
7	2925	1421577	TO	TO	TO
8	4060	2466480	TO	TO	TO
9	70	4768	0.03	0.04	0.03
fpga					
1	120	448	240.65	108.07	227.55
2	120	448	496.11	168.33	224.51
3	135	549	956.80	287.67	92.92
4	135	549	67.49	113.84	78.84
5	180	820	TO	TO	TO
6	198	968	TO	TO	TO
7	198	968	TO	TO	TO
8	216	1128	1359.87	813.65	771.42
9	216	1128	1131.68	1154.13	1168.06
10	144	560	TO	TO	TO
11	144	560	TO	TO	TO

Continued on next page

Table A.1 – *Continued from previous page*

Benchmark	**Vars**	**Clauses**	**Sym (VSADS)**	**Sym (CSV)**	**Sym (ICS)**
12	162	684	TO	TO	TO
13	162	684	TO	TO	TO
14	195	905	TO	TO	TO
15	195	905	TO	TO	TO
16	215	1070	TO	TO	TO
17	234	1242	TO	TO	TO
18	234	1242	TO	TO	TO
19	176	759	TO	TO	TO
20	176	759	TO	TO	TO
grid					
1	460	521	8.51	18.85	19.25
2	460	551	11.98	3.53	3.31
3	672	803	1089.01	TO	TO
4	672	791	928.99	TO	TO
5	672	739	241.60	511.95	504.96
6	924	1115	1091.05	TO	TO
7	924	1121	839.70	TO	TO
8	1216	1479	600.62	TO	TO
9	1216	1417	615.32	TO	TO
10	1548	1783	813.23	TO	TO
11	460	463	0.13	0.16	0.16
12	672	653	2.71	1.09	1.16
13	1216	1197	403.33	TO	TO
14	1548	1515	474.43	TO	TO
15	2121	2091	342.43	TO	TO
16	2121	2115	447.69	TO	TO
17	2332	2299	432.93	TO	TO
18	2553	2579	1217.73	TO	TO
19	2784	2683	TO	TO	TO
20	3276	3167	979.83	TO	TO
21	924	799	0.08	0.06	0.06
22	1065	957	0.36	0.28	0.30
23	1377	1213	0.01	0.02	0.01
24	1377	1211	0.14	0.13	0.13
25	1548	1357	1.70	0.83	0.80
26	1548	1335	0.19	0.16	0.15
27	1729	1515	4.97	3.40	3.60
28	2553	2219	1.38	0.72	0.77

Continued on next page

Table A.1 – *Continued from previous page*

Benchmark	**Vars**	**Clauses**	**Sym (VSADS)**	**Sym (CSV)**	**Sym (ICS)**
29	3276	2841	0.02	0.01	0.01
30	12300	10719	TO	TO	TO
kcolor					
1	360	930	2.23	2.88	3.00
2	360	1080	7.81	9.30	8.83
3	360	1230	0.01	0.01	0.02
4	375	950	23.15	42.04	39.05
5	375	1100	39.98	44.46	52.61
6	375	1250	0.01	0.01	0.01
7	390	970	28.56	11.40	12.09
8	390	1120	333.21	372.50	366.97
9	390	1270	29.91	29.98	34.09
10	300	940	TO	TO	TO
11	400	1420	TO	TO	TO
12	48	136	0.01	0.01	0.01
13	64	208	0.01	0.01	0.03
14	75	220	0.02	0.02	0.02
15	100	335	7.27	6.87	10.52
16	108	324	0.12	0.13	0.14
17	144	492	1305.31	4290.39	TO
18	147	448	9.13	9.48	11.24
19	196	679	TO	TO	TO
20	192	592	323.13	526.61	259.54
21	256	896	TO	TO	TO
22	243	756	TO	TO	TO
23	324	1143	TO	TO	TO
mcnc					
1	1133	2702	TO	TO	TO
2	1793	4138	1036.74	TO	3281.58
3	2543	5655	TO	TO	TO
4	3388	7946	TO	TO	TO
5	549	1430	TO	TO	TO
6	4792	11307	TO	TO	TO
7	4864	12048	TO	TO	TO
8	7230	16680	TO	TO	TO
9	826	1878	TO	TO	TO
10	1429	9648	32.43	35.81	35.24
11	853	2102	56.59	63.87	57.01

Continued on next page

Table A.1 – *Continued from previous page*

Benchmark	Vars	Clauses	Sym (VSADS)	Sym (CSV)	Sym (ICS)
12	2639	6775	TO	TO	TO
13	6502	17723	204.18	207.33	206.89
14	621	2024	55.21	59.87	60.37
15	1196	16806	64.09	73.27	73.19
16	410	2831	20.22	22.21	23.16
17	1322	3858	TO	TO	TO
nqueens-classic					
1	100	1490	0.07	0.07	0.07
2	121	2002	0.65	0.56	0.55
3	144	2620	12.60	12.79	10.95
4	169	3354	153.04	144.17	149.94
5	196	4214	3258.77	2446.95	2371.73
6	225	5210	TO	TO	TO
7	256	6352	TO	TO	TO
8	289	7650	TO	TO	TO
9	324	9114	TO	TO	TO
10	361	10754	TO	TO	TO
11	9	34	0.01	0.01	0.00
12	16	84	0.00	0.01	0.00
13	25	170	0.01	0.01	0.01
14	36	302	0.01	0.01	0.02
15	49	490	0.01	0.01	0.01
16	64	744	0.02	0.02	0.02
17	81	1074	0.05	0.03	0.04
nqueens-symmetry					
1	209655	538129	112.92	96.92	105.57
2	248253	638159	494.38	559.88	485.47
3	290021	746495	TO	TO	TO
4	334959	863131	TO	TO	TO
5	13757	36335	0.08	0.08	0.07
6	20189	53759	0.19	0.19	0.19
7	27771	74409	0.38	0.39	0.38
8	36503	98279	1.04	1.02	1.03
9	112881	287935	1.90	1.96	2.02
10	141969	363011	7.75	6.97	8.76
11	174227	446411	29.90	30.56	29.18
parity					

Continued on next page

Table A.1 – *Continued from previous page*

Benchmark	**Vars**	**Clauses**	**Sym (VSADS)**	**Sym (CSV)**	**Sym (ICS)**
1	4950	485200	TO	TO	TO
2	190	3440	0.34	0.35	0.35
3	231	4642	0.62	0.63	0.62
4	276	6096	0.85	0.86	0.85
5	325	7826	6.42	6.92	7.06
6	378	9856	123.12	105.78	83.11
7	435	12210	726.89	718.18	756.41
8	496	14912	4755.42	4837.29	4881.85
9	561	17986	TO	TO	TO
10	630	21456	TO	TO	TO
11	703	25346	TO	TO	TO
12	780	29680	TO	TO	TO
13	861	34482	TO	TO	TO
14	946	39776	TO	TO	TO
15	1035	45586	TO	TO	TO
16	1128	51936	TO	TO	TO
17	1225	58850	TO	TO	TO
18	1770	102720	TO	TO	TO
19	2415	164290	TO	TO	TO
20	3160	246560	TO	TO	TO
21	4005	352530	TO	TO	TO
problog					
1	1488	4064	2.30	1.58	2.11
2	660	1817	0.52	0.52	0.88
3	8428	24255	TO	TO	TO
4	5494	17313	669.34	767.48	837.41
5	2594	8705	0.15	0.17	0.19
6	2931	7746	88.84	63.58	163.24
qwh					
1	1000	13800	TO	TO	TO
2	1000	9100	TO	TO	TO
3	438	2509	0.69	0.72	0.72
4	438	1591	2.13	2.02	2.17
5	460	2716	9.99	9.96	9.75
6	519	2068	TO	TO	TO
7	506	3118	3551.92	3470.94	3583.48
8	506	1994	TO	TO	TO
9	568	3752	TO	TO	TO

Continued on next page

Table A.1 – *Continued from previous page*

Benchmark	**Vars**	**Clauses**	**Sym (VSADS)**	**Sym (CSV)**	**Sym (ICS)**
10	616	3678	18.95	22.22	17.34
11	616	2279	116.54	87.47	100.76
12	445	2420	0.03	0.03	0.04
13	445	1478	0.12	0.12	0.13
14	648	4281	TO	TO	TO
15	648	2696	TO	TO	TO
16	591	3443	94.11	85.84	94.03
17	517	2997	9.14	9.40	9.68
18	639	3891	76.51	134.17	71.61
19	639	2424	1313.92	854.22	773.68
20	664	4123	1865.32	1881.04	1992.97
21	683	4283	TO	TO	TO
22	1189	7745	131.52	69.63	85.55
23	1189	4850	1902.40	2938.28	3047.77
24	1555	9534	TO	TO	TO
25	125	825	0.14	0.04	0.05
26	125	525	18.97	13.18	6.55
27	3572	26027	TO	TO	TO
28	3572	16789	TO	TO	TO
29	4979	43754	TO	TO	TO
30	4979	28574	TO	TO	TO
tseitin					
1	150	1152	TO	TO	TO
2	200	2418	2.29	2.40	2.29
3	250	4348	13.50	13.60	13.41
4	150	830	0.01	0.00	0.01
5	200	2073	214.13	87.73	86.81
6	250	3442	TO	TO	TO
7	150	772	0.00	0.00	0.00
8	200	1309	1.32	1.37	1.34
9	250	2091	TO	TO	TO
10	180	648	1.27	1.27	1.26
11	24	72	0.01	0.01	0.02
12	40	128	0.02	0.01	0.02
13	60	200	0.04	0.03	0.03
14	84	288	0.08	0.08	0.07
15	112	392	0.21	0.22	0.22
16	144	512	0.56	0.56	0.56

Table A.2: Experimental results comparing Ganak and SymGanak (shortened to Sym) with the variable selection heuristics CSVSADS (shortened to CSV) and VSADS. The first column, B, represents the numbered benchmarks.

B	Vars	Clauses	Ganak (CSV)	Sym (CSV)	Ganak (VSADS)	Sym (VSADS)
battleship						
1	91	322	TO	TO	TO	TO
2	120	484	TO	TO	TO	TO
3	153	693	TO	TO	TO	TO
4	170	865	TO	TO	TO	TO
5	180	910	TO	TO	TO	TO
6	190	955	TO	TO	TO	TO
7	276	1662	TO	TO	TO	TO
8	364	2562	TO	TO	TO	TO
9	378	2653	TO	TO	TO	TO
10	435	3270	TO	TO	TO	TO
11	496	3976	TO	TO	TO	TO
12	1368	16308	TO	TO	TO	TO
count						
1	495	162372	0.74	0.43	0.64	0.44
2	455	61440	647.00	1.68	2.06	1.69
3	1820	1652576	125.86	251.40	113.38	249.19
4	816	165258	TO	218.51	34.26	222.88
5	1330	377076	TO	4641.06	505.81	4632.97
6	2024	765096	TO	TO	TO	TO
7	2925	1421577	TO	TO	TO	TO
8	4060	2466480	TO	TO	TO	TO
9	70	4768	0.01	0.04	0.02	0.03
fpga						
1	120	448	1715.59	108.07	1238.84	240.65
2	120	448	2390.99	168.33	TO	496.11
3	135	549	TO	287.67	TO	956.80
4	135	549	TO	113.84	302.01	67.49
5	180	820	TO	TO	TO	TO
6	198	968	TO	TO	TO	TO
7	198	968	TO	TO	TO	TO
8	216	1128	TO	813.65	TO	1359.87
9	216	1128	TO	1154.13	TO	1131.68
10	144	560	TO	TO	TO	TO
11	144	560	TO	TO	TO	TO

Continued on next page

Table A.2 – *Continued from previous page*

B	Vars	Clauses	ganak (CSV)	Sym (CSV)	ganak (VSADS)	Sym (VSADS)
12	162	684	TO	TO	TO	TO
13	162	684	TO	TO	TO	TO
14	195	905	TO	TO	TO	TO
15	195	905	TO	TO	TO	TO
16	215	1070	TO	TO	TO	TO
17	234	1242	TO	TO	TO	TO
18	234	1242	TO	TO	TO	TO
19	176	759	TO	TO	TO	TO
20	176	759	TO	TO	TO	TO
grid						
1	460	521	TO	18.85	18.13	8.51
2	460	551	TO	3.53	15.73	11.98
3	672	803	TO	TO	20.66	1089.01
4	672	791	TO	TO	20.93	928.99
5	672	739	TO	511.95	20.64	241.60
6	924	1115	TO	TO	29.30	1091.05
7	924	1121	TO	TO	45.25	839.70
8	1216	1479	TO	TO	71.69	600.62
9	1216	1417	TO	TO	62.19	615.32
10	1548	1783	TO	TO	241.99	813.23
11	460	463	0.03	0.16	0.06	0.13
12	672	653	1.27	1.09	0.90	2.71
13	1216	1197	TO	TO	36.28	403.33
14	1548	1515	TO	TO	31.54	474.43
15	2121	2091	TO	TO	TO	342.43
16	2121	2115	TO	TO	155.46	447.69
17	2332	2299	TO	TO	74.73	432.93
18	2553	2579	TO	TO	724.94	1217.73
19	2784	2683	TO	TO	TO	TO
20	3276	3167	TO	TO	TO	979.83
21	924	799	0.03	0.06	0.02	0.08
22	1065	957	0.06	0.28	0.06	0.36
23	1377	1213	0.02	0.02	0.02	0.01
24	1377	1211	0.04	0.13	0.03	0.14
25	1548	1357	0.75	0.83	1.83	1.70
26	1548	1335	0.05	0.16	0.05	0.19
27	1729	1515	0.58	3.40	0.69	4.97
28	2553	2219	0.17	0.72	0.93	1.38

Continued on next page

Table A.2 – *Continued from previous page*

B	Vars	Clauses	ganak (CSV)	Sym (CSV)	ganak (VSADS)	Sym (VSADS)
29	3276	2841	0.01	0.01	0.01	0.02
30	12300	10719	TO	TO	TO	TO
kcolor						
1	360	930	1.05	2.88	1.36	2.23
2	360	1080	2.39	9.30	2.37	7.81
3	360	1230	0.01	0.01	0.02	0.01
4	375	950	7.68	42.04	4.83	23.15
5	375	1100	11.10	44.46	11.16	39.98
6	375	1250	0.01	0.01	0.01	0.01
7	390	970	9.12	11.40	18.73	28.56
8	390	1120	95.29	372.50	92.25	333.21
9	390	1270	9.17	29.98	8.88	29.91
10	300	940	TO	TO	TO	TO
11	400	1420	TO	TO	TO	TO
12	48	136	0.01	0.01	0.01	0.01
13	64	208	0.16	0.01	0.14	0.01
14	75	220	0.03	0.02	0.03	0.02
15	100	335	8.70	6.87	7.45	7.27
16	108	324	0.21	0.13	0.23	0.12
17	144	492	TO	4290.39	TO	1305.31
18	147	448	3.36	9.48	2.66	9.13
19	196	679	TO	TO	TO	TO
20	192	592	234.09	526.61	154.72	323.13
21	256	896	TO	TO	TO	TO
22	243	756	TO	TO	TO	TO
23	324	1143	TO	TO	TO	TO
mcnc						
1	1133	2702	TO	TO	TO	TO
2	1793	4138	4800.98	TO	870.24	1036.74
3	2543	5655	TO	TO	TO	TO
4	3388	7946	TO	TO	TO	TO
5	549	1430	TO	TO	TO	TO
6	4792	11307	TO	TO	TO	TO
7	4864	12048	TO	TO	TO	TO
8	7230	16680	TO	TO	TO	TO
9	826	1878	TO	TO	TO	TO
10	1429	9648	48.04	35.81	45.35	32.43
11	853	2102	5.38	63.87	6.35	56.59

Continued on next page

Table A.2 – *Continued from previous page*

B	Vars	Clauses	ganak (CSV)	Sym (CSV)	ganak (VSADS)	Sym (VSADS)
12	2639	6775	TO	TO	TO	TO
13	6502	17723	9.13	207.33	8.41	204.18
14	621	2024	8.27	59.87	7.16	55.21
15	1196	16806	6.51	73.27	5.56	64.09
16	410	2831	4.63	22.21	4.14	20.22
17	1322	3858	TO	TO	TO	TO
nqueens-classic						
1	100	1490	0.09	0.07	0.08	0.07
2	121	2002	1.93	0.56	1.76	0.65
3	144	2620	11.46	12.79	12.05	12.60
4	169	3354	188.77	144.17	180.15	153.04
5	196	4214	2520.96	2446.95	2753.17	3258.77
6	225	5210	TO	TO	TO	TO
7	256	6352	TO	TO	TO	TO
8	289	7650	TO	TO	TO	TO
9	324	9114	TO	TO	TO	TO
10	361	10754	TO	TO	TO	TO
11	9	34	0.00	0.01	0.01	0.01
12	16	84	0.01	0.01	0.01	0.00
13	25	170	0.00	0.01	0.00	0.01
14	36	302	0.01	0.01	0.01	0.01
15	49	490	0.02	0.01	0.01	0.01
16	64	744	0.01	0.02	0.01	0.02
17	81	1074	0.04	0.03	0.03	0.05
nqueens-symmetry						
1	209655	538129	126.41	96.92	136.80	112.92
2	248253	638159	496.39	559.88	596.34	494.38
3	290021	746495	TO	TO	TO	TO
4	334959	863131	TO	TO	TO	TO
5	13757	36335	0.08	0.08	0.06	0.08
6	20189	53759	0.21	0.19	0.20	0.19
7	27771	74409	0.37	0.39	0.36	0.38
8	36503	98279	1.05	1.02	1.06	1.04
9	112881	287935	1.96	1.96	1.95	1.90
10	141969	363011	7.53	6.97	7.54	7.75
11	174227	446411	28.70	30.56	25.72	29.90
parity						

Continued on next page

Table A.2 – *Continued from previous page*

B	Vars	Clauses	ganak (CSV)	Sym (CSV)	ganak (VSADS)	Sym (VSADS)
1	4950	485200	TO	TO	TO	TO
2	190	3440	1.39	0.35	1.29	0.34
3	231	4642	4.47	0.63	4.26	0.62
4	276	6096	16.31	0.86	14.44	0.85
5	325	7826	108.51	6.92	47.42	6.42
6	378	9856	161.71	105.78	154.36	123.12
7	435	12210	2754.80	718.18	2666.69	726.89
8	496	14912	TO	4837.29	TO	4755.42
9	561	17986	TO	TO	TO	TO
10	630	21456	TO	TO	TO	TO
11	703	25346	TO	TO	TO	TO
12	780	29680	TO	TO	TO	TO
13	861	34482	TO	TO	TO	TO
14	946	39776	TO	TO	TO	TO
15	1035	45586	TO	TO	TO	TO
16	1128	51936	TO	TO	TO	TO
17	1225	58850	TO	TO	TO	TO
18	1770	102720	TO	TO	TO	TO
19	2415	164290	TO	TO	TO	TO
20	3160	246560	TO	TO	TO	TO
21	4005	352530	TO	TO	TO	TO
problog						
1	1488	4064	0.70	1.58	1.14	2.30
2	660	1817	0.15	0.52	0.16	0.52
3	8428	24255	2640.48	TO	TO	TO
4	5494	17313	760.50	767.48	1120.54	669.34
5	2594	8705	0.12	0.17	0.14	0.15
6	2931	7746	66.27	63.58	77.77	88.84
qwh						
1	1000	13800	TO	TO	TO	TO
2	1000	9100	TO	TO	TO	TO
3	438	2509	0.35	0.72	0.35	0.69
4	438	1591	1.45	2.02	1.74	2.13
5	460	2716	5.92	9.96	6.46	9.99
6	519	2068	TO	TO	TO	TO
7	506	3118	3646.05	3470.94	3510.71	3551.92
8	506	1994	TO	TO	TO	TO
9	568	3752	TO	TO	TO	TO

Continued on next page

Table A.2 – *Continued from previous page*

B	Vars	Clauses	ganak (CSV)	Sym (CSV)	ganak (VSADS)	Sym (VSADS)
10	616	3678	16.85	22.22	15.54	18.95
11	616	2279	107.99	87.47	96.55	116.54
12	445	2420	0.03	0.03	0.03	0.03
13	445	1478	0.11	0.12	0.10	0.12
14	648	4281	TO	TO	TO	TO
15	648	2696	TO	TO	TO	TO
16	591	3443	76.12	85.84	78.77	94.11
17	517	2997	5.55	9.40	6.62	9.14
18	639	3891	84.37	134.17	131.35	76.51
19	639	2424	924.06	854.22	893.15	1313.92
20	664	4123	2885.93	1881.04	1958.53	1865.32
21	683	4283	TO	TO	TO	TO
22	1189	7745	86.91	69.63	124.49	131.52
23	1189	4850	3529.40	2938.28	1956.08	1902.40
24	1555	9534	TO	TO	TO	TO
25	125	825	18.01	0.04	16.66	0.14
26	125	525	11.32	13.18	25.67	18.97
27	3572	26027	TO	TO	TO	TO
28	3572	16789	TO	TO	TO	TO
29	4979	43754	TO	TO	TO	TO
30	4979	28574	TO	TO	TO	TO
tseitin						
1	150	1152	TO	TO	TO	TO
2	200	2418	TO	2.40	TO	2.29
3	250	4348	TO	13.60	TO	13.50
4	150	830	0.01	0.00	0.01	0.01
5	200	2073	TO	87.73	TO	214.13
6	250	3442	TO	TO	TO	TO
7	150	772	0.00	0.00	0.00	0.00
8	200	1309	TO	1.37	TO	1.32
9	250	2091	TO	TO	TO	TO
10	180	648	TO	1.27	TO	1.27
11	24	72	0.02	0.01	0.01	0.01
12	40	128	0.06	0.01	0.05	0.02
13	60	200	2.51	0.03	0.31	0.04
14	84	288	TO	0.08	TO	0.08
15	112	392	TO	0.22	TO	0.21
16	144	512	TO	0.56	TO	0.56

Table A.3: Benchmark names with their associated instance number.

instance	Benchmark
	battleship
1	battleship-07-13
2	battleship-08-15
3	battleship-09-17
4	battleship-10-17
5	battleship-10-18
6	battleship-10-19
7	battleship-12-23
8	battleship-14-26
9	battleship-14-27
10	battleship-15-29
11	battleship-16-31
12	battleship-24-57
	count
1	count12-4
2	count15-3
3	count16-4
4	count18-3
5	count21-3
6	count24-3
7	count27-3
8	count30-3
9	count8-4
	fpga
1	fpga10-8-sat
2	fpga10-8-sat-rcr
3	fpga10-9-sat
4	fpga10-9-sat-rcr
5	fpga12-10-sat-rcr
6	fpga12-11-sat
7	fpga12-11-sat-rcr
8	fpga12-12-sat
9	fpga12-12-sat-rcr
10	fpga12-8-sat
11	fpga12-8-sat-rcr
12	fpga12-9-sat
13	fpga12-9-sat-rcr
14	fpga13-10-sat
15	fpga13-10-sat-rcr

Continued on next page

Table A.3 – *Continued from previous page*

instance	Benchmark
16	fpga13-11-sat-rcr
17	fpga13-12-sat
18	fpga13-12-sat-rcr
19	fpga13-9-sat
20	fpga13-9-sat-rcr
	grid
1	grid-50-10-10-q
2	grid-50-10-6-q
3	grid-50-12-3-q
4	grid-50-12-4-q
5	grid-50-12-6-q
6	grid-50-14-5-q
7	grid-50-14-9-q
8	grid-50-16-10-q
9	grid-50-16-7-q
10	grid-50-18-2-q
11	grid-75-10-7-q
12	grid-75-12-10-q
13	grid-75-16-5-q
14	grid-75-18-5-q
15	grid-75-21-5-q
16	grid-75-21-7-q
17	grid-75-22-6-q
18	grid-75-23-3-q
19	grid-75-24-8-q
20	grid-75-26-1-q
21	grid-90-14-6-q
22	grid-90-15-3-q
23	grid-90-17-7-q
24	grid-90-17-8-q
25	grid-90-18-1-q
26	grid-90-18-9-q
27	grid-90-19-10-q
28	grid-90-23-10-q
29	grid-90-26-1-q
30	grid-90-50-6-q
	kcolor
1	kcolor.gnm120-150
2	kcolor.gnm120-200
3	kcolor.gnm120-250

Continued on next page

Table A.3 – *Continued from previous page*

instance	Benchmark
4	kcolor.gnm125-150
5	kcolor.gnm125-200
6	kcolor.gnm125-250
7	kcolor.gnm130-150
8	kcolor.gnm130-200
9	kcolor.gnm130-250
10	kcolor.grid10k3
11	kcolor.grid10k4
12	kcolor.grid4k3
13	kcolor.grid4k4
14	kcolor.grid5k3
15	kcolor.grid5k4
16	kcolor.grid6k3
17	kcolor.grid6k4
18	kcolor.grid7k3
19	kcolor.grid7k4
20	kcolor.grid8k3
21	kcolor.grid8k4
22	kcolor.grid9k3
23	kcolor.grid9k4
	mcnc
1	mcnc-C1355
2	mcnc-C1908
3	mcnc-C2670
4	mcnc-C3540
5	mcnc-C499
6	mcnc-C5315
7	mcnc-C6288
8	mcnc-C7552
9	mcnc-C880
10	mcnc-apex5
11	mcnc-apex6
12	mcnc-pair
13	mcnc-t481
14	mcnc-term1
15	mcnc-too-large
16	mcnc-x1
17	mcnc-x3
	nqueens-classic
1	nqueens-classic-10

Continued on next page

Table A.3 – *Continued from previous page*

instance	Benchmark
2	nqueens-classic-11
3	nqueens-classic-12
4	nqueens-classic-13
5	nqueens-classic-14
6	nqueens-classic-15
7	nqueens-classic-16
8	nqueens-classic-17
9	nqueens-classic-18
10	nqueens-classic-19
11	nqueens-classic-3
12	nqueens-classic-4
13	nqueens-classic-5
14	nqueens-classic-6
15	nqueens-classic-7
16	nqueens-classic-8
17	nqueens-classic-9
	nqueens-symmetry
1	nqueens-symmetry-10-5
2	nqueens-symmetry-11-5
3	nqueens-symmetry-12-5
4	nqueens-symmetry-13-5
5	nqueens-symmetry-4-4
6	nqueens-symmetry-5-4
7	nqueens-symmetry-6-4
8	nqueens-symmetry-7-4
9	nqueens-symmetry-7-5
10	nqueens-symmetry-8-5
11	nqueens-symmetry-9-5
	parity
1	parity100
2	parity20
3	parity22
4	parity24
5	parity26
6	parity28
7	parity30
8	parity32
9	parity34
10	parity36
11	parity38

Continued on next page

Table A.3 – *Continued from previous page*

instance	Benchmark
12	parity40
13	parity42
14	parity44
15	parity46
16	parity48
17	parity50
18	parity60
19	parity70
20	parity80
21	parity90
	problog
1	problog-alarm
2	problog-child
3	problog-hailfinder
4	problog-munin
5	problog-pathfinder
6	problog-pigs
	qwh
1	qwh10.h100
2	qwh10.h100.min
3	qwh15.h120
4	qwh15.h120.min
5	qwh15.h121
6	qwh15.h121.min
7	qwh15.h122
8	qwh15.h122.min
9	qwh15.h125
10	qwh20.h165.r.s10
11	qwh20.h165.r.s10.min
12	qwh20.h165.r.s1337
13	qwh20.h165.r.s1337.min
14	qwh20.h165.r.s20
15	qwh20.h165.r.s20.min
16	qwh20.h165.r.s5
17	qwh20.h166.rb40.s10
18	qwh20.h166.r.s10
19	qwh20.h166.r.s10.min
20	qwh20.h167.r.s10
21	qwh20.h171.rb50.s10
22	qwh30.h320

Continued on next page

Table A.3 – *Continued from previous page*

instance	Benchmark
23	qwh30.h320.min
24	qwh33.h381
25	qwh5.h25
26	qwh5.h25.min
27	qwh50.h750.bal
28	qwh50.h750.bal.min
29	qwh50.h825.bal
30	qwh50.h825.bal.min
	tseitin
1	tseitin.gnm120-150.s1
2	tseitin.gnm120-200.s1
3	tseitin.gnm120-250.s1
4	tseitin.gnm125-150.s1
5	tseitin.gnm125-200.s1
6	tseitin.gnm125-250.s1
7	tseitin.gnm130-150.s1
8	tseitin.gnm130-200.s1
9	tseitin.gnm130-250.s1
10	tseitin.grid10.s1
11	tseitin.grid4.s1
12	tseitin.grid5.s1
13	tseitin.grid6.s1
14	tseitin.grid7.s1
15	tseitin.grid8.s1
16	tseitin.grid9.s1

Appendix B

F-XSDD(BR) with Complex Weight Functions

This appendix chapter is a continuation of Example 34 in Chapter 6 that illustrated a weighted model integration problem where the weight function is a single term. We now describe the procedure that allows F-XSDD(BR) to handle more general weight functions.

Weight function. F-XSDD(BR) natively supports piece-wise polynomial weight functions, which it represents using an *extended algebraic decision diagram* (XADD) introduced by Sanner et al. (2011). This diagram is similar to an ordered binary decision diagram, but allows decisions based on $\mathcal{LRA}$ atoms and maps to polynomial expressions rather than a Boolean value 0 or 1. Furthermore, algorithms have been developed to perform algebraic operations with XADDs. For example, the result of $\text{XADD}_1 \times \text{XADD}_2$ is again an XADD. Consider the example adapted from Kolb (2019), where *ite* denotes an if-then-else:

$$w = ite(A, 2x + y, x^2 y) \times ite(y < 5, 3, 2) \tag{B.1}$$

The XADD corresponding to this weight function is illustrated in Figure B.1.

Support formula. F-XSDD(BR) uses an *extended sentential decision diagram* (XSDD) (Zuidberg Dos Martires et al., 2019b) to represent the support ψ. This representation is based on the more general SDD class, and represents an $\mathcal{LRA}$ formula by abstracting it to a propositional formula, compiling the result using

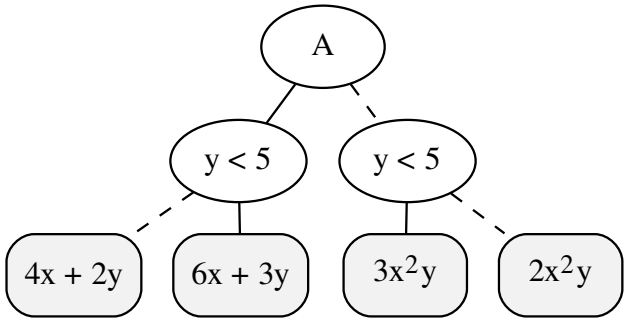


Figure B.1: The XADD representation of the weight function in Equation B.1. A dashed line originating from a node indicates negation. For example, the path following the dashed lines from the top to the bottom right indicates that when A is false, and $y < 5$ is false, the weight is $2x^2y$.

an SDD compiler, while maintaining a mapping of freshly introduced Boolean variables to their corresponding $\mathcal{LRA}$ atom.

Weighted model integration procedure. The XADD weight function can be viewed a set of tuples $\langle \psi_i, w_i \rangle$, one for each polynomial leaf node in the XADD. Essentially, ψ_i represents the collection of truth assignments in w leading to the same weight function polynomial w_i. For example, using Equation B.1, this looks as follows

$$\langle \psi_1, weight_1 \rangle = \langle A \wedge (y < 5)\ ,\ 6x + 3y \rangle \tag{B.2}$$

$$\langle \psi_2, weight_2 \rangle = \langle A \wedge (y \geq 5)\ ,\ 4x + 2y \rangle \tag{B.3}$$

$$\langle \psi_3, weight_3 \rangle = \langle \neg A \wedge (y < 5)\ ,\ 3x^2y \rangle \tag{B.4}$$

$$\langle \psi_4, weight_4 \rangle = \langle \neg A \wedge (y \geq 5)\ ,\ 2x^2y \rangle \tag{B.5}$$

While in this example each path in the XADD leads to a unique weight, it is possible for multiple paths to lead to the same weight, in which case they are combined in one ψ_i. This view of the XADD as tuples is easily established implementation-wise. At this point, each satisfying truth assignment of ψ_i maps to the same weight w_i, so we convert ψ_i into an XSDD conjoined with the support ψ: $\psi \wedge \psi_i$. This transforms the weighted model integration problem over piece-wise polynomial weight functions, into an integration problem over polynomials only, i.e., we perform a WMI computation for each $\psi \wedge \psi_i$ over polynomial w_i, and aggregate the results. The conjunction $\psi \wedge \psi_i$ is possible

through the efficient apply operation available for SDDs (and hence XSDDs). Finally, the sum rule of integration states that integrating over an polynomial is equivalent to integrating independently over each term of the polynomial, and summing the results. This transforms the problem into integrating over a polynomial's term, where we lower the integration into the representation of $\psi \wedge \psi_i$, as discussed in Chapter 6. Figure B.2 illustrates an equation resulting from this procedure, prior to pushing down the integration. The equation that then results from pushing down the integration is included in Chapter 6, Figure 6.2. Afterwards, the procedure performs a bottom-up evaluation, using XADDs to represent the intermediate results because these support algebraic operations and are relatively succinct. For more information on the whole procedure or technical details concerning the integration, we refer to Kolb et al. (2019b) and Kolb (2019).

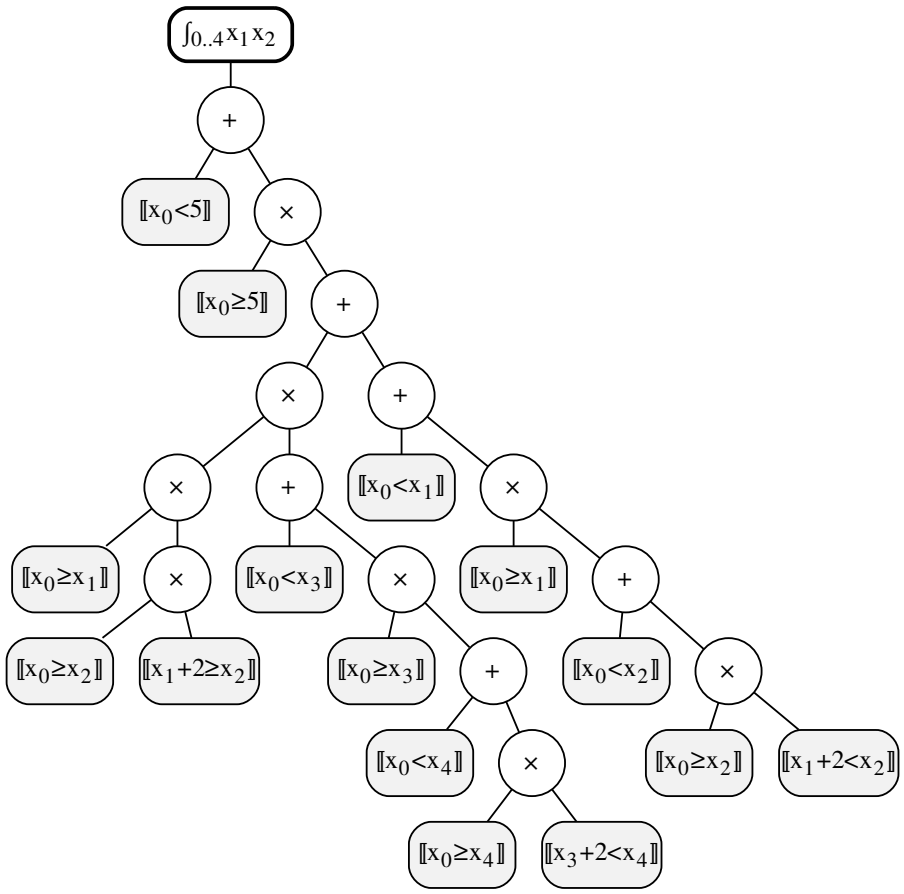


Figure B.2: This is part of the integration problem of Example 34, in Section 6.2. It shows the situation prior to pushing down the integration lower into the equation.

Bibliography

Aerts, B., T. Goedemé, and J. Vennekens (2016). "A Probabilistic Logic Programming Approach to Automatic Video Montage". In: *Proceedings of the 22nd European Conference on Artificial Intelligence, ECAI*. Vol. 285. IOS Press, pp. 234–242 (cit. on p. 47).

Aloul, F. A., A. Ramani, I. L. Markov, and K. A. Sakallah (2002). "Solving Difficult SAT Instances in the Presence of Symmetry". In: *Proceedings of the 39th Design Automation Conference, DAC*. ACM, pp. 731–736 (cit. on p. 51, 56, 123).

Antanas, L., P. Moreno, M. Neumann, R. P. de Figueiredo, K. Kersting, J. Santos-Victor, and L. De Raedt (2019). "Semantic and geometric reasoning for robotic grasping: a probabilistic logic approach". In: *Autonomous Robots* 43.6, pp. 1393–1418 (cit. on p. 47).

Antova, L., C. Koch, and D. Olteanu (2006). "MayBMS: Managing Incomplete Information with Probabilistic World-Set Decompositions". In: *Proceedings of the 23rd International Conference on Data Engineering, ICDE*. IEEE Computer Society, pp. 1479–1480 (cit. on p. 29).

Apriceno, G., A. Passerini, and L. Serafini (2021). "A Neuro-Symbolic Approach to Structured Event Recognition". In: *28th International Symposium on Temporal Representation and Reasoning, TIME*. Vol. 206. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:14 (cit. on p. 47).

Apsel, U. and R. I. Brafman (2012). "Lifted MEU by Weighted Model Counting". In: *Proceedings of the 26th AAAI Conference on Artificial Intelligence*. AAAI Press (cit. on p. 81).

Arnborg, S. (1985). "Efficient Algorithms for Combinatorial Problems on Graphs with Bounded, Decomposability—a Survey". In: *BIT* 25.1, pp. 2–23. ISSN: 0006-3835 (cit. on p. 84).

Aziz, R. A., G. Chu, C. J. Muise, and P. J. Stuckey (2015). "#∃SAT: Projected Model Counting". In: *Theory and Applications of Satisfiability Testing - SAT*. Vol. 9340. Springer, pp. 121–137 (cit. on p. 50, 66).

Bacchus, F., S. Dalmao, and T. Pitassi (2003). "DPLL with Caching: A new algorithm for #SAT and Bayesian Inference". In: *Electronic Colloquium on Computational Complexity, ECCC* 10.003 (cit. on p. 17, 50, 112).

Baluta, T., S. Shen, S. Shinde, K. S. Meel, and P. Saxena (2019). "Quantitative Verification of Neural Networks and Its Security Applications". In: *SIGSAC Conference on Computer and Communications Security, CCS*. ACM, pp. 1249–1264 (cit. on p. 49).

Barbará, D., H. Garcia-Molina, and D. Porter (1992). "The Management of Probabilistic Data". In: *IEEE Transactions on Knowledge and Data Engineering* 4.5, pp. 487–502 (cit. on p. 29).

Barbosa, H., C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar (2022). "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*. Vol. 13243. Lecture Notes in Computer Science. Springer, pp. 415–442 (cit. on p. 111).

Barrett, C. W., R. Sebastiani, S. A. Seshia, and C. Tinelli (2009). "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Vol. 185. IOS Press, pp. 825–885 (cit. on p. 107).

Barrett, C. W. and C. Tinelli (2018). "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Springer, pp. 305–343 (cit. on p. 106).

Bart, A., F. Koriche, J. Lagniez, and P. Marquis (2014). "Symmetry-Driven Decision Diagrams for Knowledge Compilation". In: *Proceedings of the 21st European Conference on Artificial Intelligence, ECAI*. Vol. 263. IOS Press, pp. 51–56 (cit. on p. 52, 117).

Bayardo Jr, R. J. and J. D. Pehoushek (2000). "Counting Models Using Connected Components". In: *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence, AAAI/IAAI*, pp. 157–162 (cit. on p. 50).

Becker, B., M. Behle, F. Eisenbrand, and R. Wimmer (2005). "BDDs in a Branch and Cut Framework". In: *WEA*. Vol. 3503. Springer, pp. 452–463 (cit. on p. 118).

Belle, V. and L. De Raedt (2020). "Semiring programming: A semantic framework for generalized sum product problems". In: *International Journal of Approximate Reasoning* 126, pp. 181–201. ISSN: 0888-613X. DOI: 10.1016/j.ijar.2020.08.001 (cit. on p. 46, 116).

Belle, V., A. Passerini, and G. Van den Broeck (2015). "Probabilistic Inference in Hybrid Domains by Weighted Model Integration". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI*. AAAI Press, pp. 2770–2776 (cit. on p. 5, 84, 85).

Benjelloun, O., A. D. Sarma, A. Halevy, and J. Widom (2006). "ULDBs: Databases with Uncertainty and Lineage". In: *Proceedings of the 32nd*

International Conference on Very Large Data Bases, VLDB. Vol. 6. ACM, pp. 953–964 (cit. on p. 29).

Berg, B. van den, T. van Bremen, V. Derkinderen, A. Kimmig, T. Schrijvers, and L. De Raedt (2021). "From Probabilistic NetKAT to ProbLog: New Algorithms for Inference and Learning in Probabilistic Networks". In: *International Conference on Probabilistic Programming, Extended Abstracts* (cit. on p. 47, 176).

Berg, B. van den, T. Schrijvers, J. McKinna, and A. Vandenbroucke (2022). "Forward- or Reverse-Mode Automatic Differentiation: What's the Difference?" In: *CoRR* abs/2212.11088 (cit. on p. 47).

Bhattacharjya, D. and R. D. Shachter (2012). "Evaluating influence diagrams with decision circuits". In: *CoRR* abs/1206.5257 (cit. on p. 81).

Birnbaum, E. and E. L. Lozinskii (1999). "The Good Old Davis-Putnam Procedure Helps Counting Models". In: *Journal of Artificial Intelligence Research* 10, pp. 457–477 (cit. on p. 16, 50).

Bliem, B. and M. Järvisalo (2019). "Centrality Heuristics for Exact Model Counting". In: *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI*. IEEE, pp. 59–63. DOI: 10.1109/ICTAI.2019.00017 (cit. on p. 15).

Bryant, R. E. (1986). "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* 35.8, pp. 677–691 (cit. on p. 22).

Bueno, T. P., D. Mauá, L. N. de Barros, and F. G. Cozman (2016). "Markov Decision Processes Specified by Probabilistic Logic Programming: Representation and Solution". In: *5th Brazilian Conference on Intelligent Systems, BRACIS*, pp. 337–342 (cit. on p. 47).

Bui, H. H., T. N. Huynh, and S. Riedel (2013). "Automorphism Groups of Graphical Models and Lifted Variational Inference". In: *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence, UAI*. AUAI Press (cit. on p. 52).

Cavallo, R. and M. Pittarelli (1987). "The Theory of Probabilistic Databases". In: *Proceedings of 13th International Conference on Very Large Data Bases, VLDB*, pp. 71–81 (cit. on p. 29).

Chajewska, U., D. Koller, and R. Parr (2000). "Making Rational Decisions Using Adaptive Utility Elicitation". In: *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence, AAAI/IAAI*. Ed. by H. A. Kautz and B. W. Porter. AAAI Press / The MIT Press, pp. 363–369. ISBN: 0-262-51112-6 (cit. on p. 82).

Chaki, S., A. Gurfinkel, and O. Strichman (2009). "Decision Diagrams for Linear Arithmetic". In: *Proceedings of 9th International Conference on Formal*

Methods in Computer-Aided Design, FMCAD. IEEE, pp. 53–60. DOI: 10.1109/FMCAD.2009.5351143 (cit. on p. 95, 106, 112).

Chavira, M. and A. Darwiche (2008). "On probabilistic inference by weighted model counting". In: *Artificial Intelligence* 172.6-7, pp. 772–799 (cit. on p. 49).

Choi, A. and A. Darwiche (2013). "Dynamic Minimization of Sentential Decision Diagrams". In: *Proceedings of the 27th AAAI Conference on Artificial Intelligence*. Ed. by M. desJardins and M. L. Littman. AAAI Press (cit. on p. 22).

Christ, J., J. Hoenicke, and A. Nutz (2012). "SMTInterpol: An Interpolating SMT Solver". In: *Model Checking Software - 19th International Workshop, SPIN*. Vol. 7385. Lecture Notes in Computer Science. Springer, pp. 248–254 (cit. on p. 111).

Cimatti, A., A. Griggio, B. J. Schaafsma, and R. Sebastiani (2013). "The MathSAT5 SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS*. Vol. 7795. Lecture Notes in Computer Science. Springer, pp. 93–107 (cit. on p. 111).

Crawford, J. M., M. L. Ginsberg, E. M. Luks, and A. Roy (1996). "Symmetry-Breaking Predicates for Search Problems". In: *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning, KR*, pp. 148–159 (cit. on p. 52).

Dalvi, N. and D. Suciu (2007). "Efficient query evaluation on probabilistic databases". In: *The International Journal on Very Large Data Bases, VLDB* 16.4, pp. 523–544. DOI: 10.1007/s00778-006-0004-3 (cit. on p. 29).

Dantsin, E. (1990). "Probabilistic Logic Programs and their Semantics". In: *Proceedings of the First Russian Conference on Logic Programming*, pp. 152–164 (cit. on p. 28).

Darwiche, A. (2000). "A Differential Approach to Inference in Bayesian Networks". In: *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence, UAI*. Ed. by C. Boutilier and M. Goldszmidt. Morgan Kaufmann, pp. 123–132. ISBN: 1-55860-709-9 (cit. on p. 20, 81).

Darwiche, A. (2002). "A Logical Approach to Factoring Belief Networks". In: *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasonin, KR*, pp. 409–420 (cit. on p. 20).

Darwiche, A. (2004). "New Advances in Compiling CNF to Decomposable Negation Normal Form". In: *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI*. IOS Press, pp. 318–322. ISBN: 9781586034528 (cit. on p. 20, 21, 106).

Darwiche, A. (2009). *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press. ISBN: 978-0-521-88438-9 (cit. on p. 1, 85, 92, 93).

Darwiche, A. (2011). "SDD: A New Canonical Representation of Propositional Knowledge Bases". In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI*, pp. 819–826. DOI: 10.5591/978-1-57735-516-8/IJCAI11-143 (cit. on p. 22, 23, 24, 111).

Darwiche, A. and A. Hirth (2020). "On the Reasons Behind Decisions". In: *Proceedings of the 24th European Conference on Artificial Intelligence, ECAI*. Vol. 325. IOS Press, pp. 712–720. DOI: 10.3233/FAIA200158 (cit. on p. 111).

Darwiche, A. and P. Marquis (2002). "A Knowledge Compilation Map". In: *Journal of Artificial Intelligence Research* 17.1, pp. 229–264. ISSN: 1076-9757. DOI: 10.1613/jair.989 (cit. on p. 2, 18, 19, 20, 21, 68, 106, 108, 120).

Davis, M., G. Logemann, and D. W. Loveland (1962). "A Machine Program for Theorem-Proving". In: *Communications of the ACM* 5.7, pp. 394–397. DOI: 10.1145/368273.368557 (cit. on p. 15, 50).

Davis, M. and H. Putnam (1960). "A Computing Procedure for Quantification Theory". In: *Journal of the ACM* 7.3, pp. 201–215. DOI: 10.1145/321033.321034 (cit. on p. 15).

De Maeyer, D., J. Renkens, L. Cloots, L. De Raedt, and K. Marchal (2013). "PheNetic: network-based interpretation of unstructured gene lists in E. coli". In: *Molecular bioSystems* 9 (7), pp. 1594–1603. DOI: 10.1039/C3MB25551D (cit. on p. 47).

De Maeyer, D., B. Weytjens, L. De Raedt, and K. Marchal (2016). "Network-Based Analysis of eQTL Data to Prioritize Driver Mutations". In: *Genome Biology and Evolution* 8.3, pp. 481–494. ISSN: 1759-6653. DOI: 10.1093/gbe/evw010 (cit. on p. 47).

De Maeyer, D., B. Weytjens, J. Renkens, L. De Raedt, and K. Marchal (2015). "PheNetic: network-based interpretation of molecular profiling data". In: *Nucleic Acids Research* 43.W1, W244–W250. ISSN: 0305-1048. DOI: 10.1093/nar/gkv347 (cit. on p. 47).

De Raedt, L. (2007). "ProbLog and its Application to Link Mining in Biological Networks". In: *Mining and Learning with Graphs, MLG*. Ed. by P. Frasconi, K. Kersting, and K. Tsuda (cit. on p. 47).

De Raedt, L., K. Kersting, S. Natarajan, and D. Poole (2016). "Statistical Relational Artificial Intelligence: Logic, Probability, and Computation". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 10.2, pp. 1–189. DOI: 10.2200/S00692ED1V01Y201601AIM032 (cit. on p. 68).

De Raedt, L. and A. Kimmig (2015). "Probabilistic (logic) programming concepts". In: *Machine Learning* 100.1, pp. 5–47. DOI: 10.1007/s10994-015-5494-z (cit. on p. 28, 29).

De Raedt, L., A. Kimmig, and H. Toivonen (2007). "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery". In: *Proceedings of the 20th International Joint on Artificial Intelligence, IJCAI*. Vol. 7, pp. 2462–2467 (cit. on p. 1, 27, 29, 31).

De Salvo Braz, R., C. O'Reilly, V. Gogate, and R. Dechter (2016). "Probabilistic Inference Modulo Theories". In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 3591–3599 (cit. on p. 113).

De Smet, L., P. Zuidberg Dos Martires, R. Manhaeve, G. Marra, A. Kimmig, and L. De Readt (2023). "Neural Probabilistic Logic Programming in Discrete-Continuous Domains". In: *Proceedings of the 39th Conference on Uncertainty in Artificial Intelligence, UAI*. Ed. by R. J. Evans and I. Shpitser. Vol. 216. PMLR, pp. 529–538 (cit. on p. 37, 38).

Dechter, R. (1999). "Bucket Elimination: A Unifying Framework for Reasoning". In: *Artificial Intelligence* 113.1-2, pp. 41–85 (cit. on p. 81, 84).

Dechter, R. (2013). *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers. DOI: 10.2200/S00529ED1V01Y201308AIM023 (cit. on p. 85, 91, 92, 94, 95, 96, 97, 100, 101, 104).

Dechter, R. and R. Mateescu (2007). "AND/OR Search Spaces for Graphical Models". In: *Artificial Intelligence* 171.2-3, pp. 73–106. ISSN: 0004-3702. DOI: 10.1016/j.artint.2006.11.003 (cit. on p. 81).

Derkinderen, V., J. Bekker, and P. Smet (2023a). "Optimizing workforce allocation under uncertain activity duration". In: *Computers & Industrial Engineering* 179, p. 109228. DOI: 10.1016/j.cie.2023.109228 (cit. on p. 175).

Derkinderen, V., J. Bekker, and P. Smet (2023b). *Replication Data for: Optimizing Workforce Allocation under Uncertain Activity Duration*. Version V1. URL: https://doi.org/10.48804/YHMU7R (cit. on p. 176).

Derkinderen, V. and L. De Raedt (2020). "Algebraic Circuits for Decision Theoretic Inference and Learning". In: *Proceedings of the 24th European Conference on Artificial Intelligence, ECAI*. Vol. 325. IOS Press, pp. 2569–2576. DOI: 10.3233/FAIA200392 (cit. on p. 9, 13, 18, 38, 41, 47, 67, 176).

Derkinderen, V., E. Heylen, P. Zuidberg Dos Martires, S. Kolb, and L. De Raedt (2020). "Ordering Variables for Weighted Model Integration". In: *Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence, UAI*. Ed. by R. P. Adams and V. Gogate. Vol. 124. AUAI Press, pp. 879–888 (cit. on p. 9, 83, 175).

Derkinderen, V., R. Manhaeve, P. Zuidberg Dos Martires, and L. De Raedt (2023c). "Semirings for Probabilistic and Neural-Symbolic Logic Programming". Accepted with minor revision in International Journal of Approximate Reasoning (cit. on p. 8, 22, 27, 175).

Derkinderen, V., P. Zuidberg Dos Martires, S. Kolb, and P. Morettin (2023d). "Top-Down Knowledge Compilation for Counting Modulo Theories". In: *CoRR* abs/2306.04541. accepted at Workshop on Counting and Sampling at SAT 2023. DOI: 10.48550/arXiv.2306.04541 (cit. on p. 10, 105, 176).

Devriendt, J., B. Bogaerts, M. Bruynooghe, and M. Denecker (2016). "Improved Static Symmetry Breaking for SAT". In: *Theory and Applications of Satisfiability Testing - SAT*. Vol. 9710. Springer, pp. 104–122 (cit. on p. 52).

Devriendt, J., B. Bogaerts, B. D. Cat, M. Denecker, and C. Mears (2012). "Symmetry Propagation: Improved Dynamic Symmetry Breaking in SAT". In: *IEEE 24th International Conference on Tools with Artificial Intelligence ICTAI*. IEEE Computer Society, pp. 49–56 (cit. on p. 123).

Doherty, P. and A. Szalas (2022). "A landscape and implementation framework for probabilistic rough sets using ProbLog". In: *Information Sciences* 593, pp. 546–576. DOI: 10.1016/j.ins.2021.12.062 (cit. on p. 47).

Dolan, S. (2013). "Fun with Semirings: A functional pearl on the abuse of linear algebra". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP*. ACM, pp. 101–110. DOI: 10.1145/2500365.2500613 (cit. on p. 47).

Dries, A., A. Kimmig, J. Davis, V. Belle, and L. De Raedt (2017). "Solving Probability Problems in Natural Language". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 3981–3987. DOI: 10.24963/ijcai.2017/556 (cit. on p. 47).

Dutertre, B. (2014). "Yices 2.2". In: *Computer Aided Verification - 26th International Conference, CAV*. Vol. 8559. Springer, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49 (cit. on p. 110, 111).

Eisner, J. (2002). "Parameter Estimation for Probabilistic Finite-State Transducers". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 1–8. DOI: 10.3115/1073083.1073085 (cit. on p. 3, 71).

Eisner, J. and N. W. Filardo (2010). "Dyna: Extending Datalog for Modern AI". In: *Datalog Reloaded - First International Workshop, Datalog 2010. Revised Selected Papers*. Ed. by O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers. Vol. 6702. Springer, pp. 181–220. DOI: 10.1007/978-3-642-24206-9_11 (cit. on p. 29, 30).

Feldstein, J. and V. Belle (2021). "Lifted Reasoning Meets Weighted Model Integration". In: *Proceedings of the 37th Conference on Uncertainty in Artificial Intelligence, UAI*. Vol. 161. AUAI Press, pp. 322–332 (cit. on p. 113).

Fichte, J. K., M. Hecher, and F. Hamiti (2020). "The Model Counting Competition 2020". In: *CoRR* abs/2012.01323 (cit. on p. 57).

Fierens, D., G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt (2015). "Inference and learning in probabilistic logic programs using weighted Boolean formulas". In: *Theory and Practice of Logic Programming* 15.3, pp. 358–401. DOI: 10.1017/S1471068414000076 (cit. on p. 2, 8, 42, 68, 125).

Flach, P. (1994). *Simply Logical: Intelligent Reasoning by Example*. John Wiley. ISBN: 0471941522 (cit. on p. 1, 28, 31).

Fuhr, N. (1995). "Probabilistic Datalog - A Logic For Powerful Retrieval Methods". In: *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 282–290. DOI: 10.1145/215206.215372 (cit. on p. 29).

Fuhr, N. and T. Rölleke (1997). "A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems". In: *ACM Transactions on Information Systems, TOIS* 15.1, pp. 32–66. DOI: 10.1145/239041.239045 (cit. on p. 29).

Gelenbe, E. and G. Hebrail (1986). "A Probability Model of Uncertainty in Data Bases". In: *Second International Conference on Data Engineering*. IEEE, pp. 328–333. DOI: 10.1109/ICDE.1986.7266237 (cit. on p. 29).

Gomes, C. P. and D. Shmoys (2002). "Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem". In: *proceedings of the Computational Symposium on Graph Coloring and Generalizations*, pp. 22–39 (cit. on p. 51, 124).

Goodman, N. D., J. B. Tenenbaum, and T. P. Contributors (2016). *Probabilistic Models of Cognition*. http://probmods.org/v2. Accessed: 2023-7-12 (cit. on p. 28).

Grohe, M., B. L. Kaminski, J.-P. Katoen, and P. Lindner (2022). "Generative Datalog with Continuous Distributions". In: *Journal of the ACM* 69.6, 46:1–46:52. DOI: 10.1145/3559102 (cit. on p. 29).

Groß, A., B. Kracher, J. Kraus, S. Kühlwein, A. Pfister, S. Wiese, K. Luckert, O. Pötz, T. Joos, D. Van Daele, L. De Raedt, M. Kühl, and H. Kestler (2019). "Representing dynamic biological networks with multi-scale probabilistic models". In: *Communications Biology* 2.1, p. 21. DOI: 10.1038/s42003-018-0268-3 (cit. on p. 47).

Gutmann, B., I. Thon, A. Kimmig, M. Bruynooghe, and L. De Raedt (2011a). "The Magic of Logical Inference in Probabilistic Programming". In: *Theory and Practice of Logic Programming* 11.4-5, pp. 663–680 (cit. on p. 29).

Gutmann, B., A. Kimmig, K. Kersting, and L. De Raedt (2008). "Parameter Learning in Probabilistic Databases: A Least Squares Approach". In: *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML/PKDD*, pp. 473–488. DOI: 10.1007/978-3-540-87479-9_49 (cit. on p. 77).

Gutmann, B., I. Thon, and L. De Raedt (2011b). "Learning the Parameters of Probabilistic Logic Programs from Interpretations". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML/PKDD*. Ed. by D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis. Vol. 6911. Springer, pp. 581–596. ISBN: 978-3-642-23779-9. DOI: 10.1007/978-3-642-23780-5_47 (cit. on p. 77).

Heule, M. J. H., M. Järvisalo, and M. Suda (2019). “SAT Competition 2018”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 11.1, pp. 133–154. DOI: 10.3233/SAT190120 (cit. on p. 61).

Hitzler, P. and M. K. Sarker, eds. (2021). *Neuro-Symbolic Artificial Intelligence: The State of the Art*. Vol. 342. Frontiers in Artificial Intelligence and Applications. IOS Press. ISBN: 978-1-64368-244-0. DOI: 10.3233/FAIA342 (cit. on p. 120).

Hochreiter, S. (2022). “Toward a Broad AI”. In: *Communications of the ACM* 65.4, pp. 56–57. ISSN: 0001-0782. DOI: 10.1145/3512715 (cit. on p. 120).

Hocquette, C., S. Dumancic, and A. Cropper (2023). “Learning Logic Programs by Discovering Higher-Order Abstractions”. In: *CoRR* abs/2308.08334 (cit. on p. 118).

Holtzen, S., T. D. Millstein, and G. Van den Broeck (2019). “Generating and Sampling Orbits for Lifted Probabilistic Inference”. In: *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence, UAI*. Vol. 115, pp. 985–994 (cit. on p. 52).

Hommersom, A. and M. L. P. Bueno (2016). “Toward Computing Conflict-Based Diagnoses in Probabilistic Logic Programming”. In: *Proceedings of the 3rd International Workshop on Probabilistic Logic Programming, PLP@ILP*. Vol. 1661. CEUR Workshop Proceedings, pp. 29–38 (cit. on p. 47).

Huang, J., Z. Li, B. Chen, K. Samel, M. Naik, L. Song, and X. Si (2021). “Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning”. In: *Advances in Neural Information Processing Systems, NeurIPS* 34, pp. 25134–25145 (cit. on p. 46, 120).

Huang, J. and A. Darwiche (2004). “Using DPLL for Efficient OBDD Construction”. In: *Theory and Applications of Satisfiability Testing - SAT* (cit. on p. 112).

Huang, J. and A. Darwiche (2005). “DPLL with a Trace: From SAT to Knowledge Compilation”. In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 156–162 (cit. on p. 20, 106).

Hung, N. D. (2017). “Inference and Learning in Probabilistic Argumentation”. In: *Multi-disciplinary Trends in Artificial Intelligence - 11th International Workshop, MIWAI*. Vol. 10607. Springer, pp. 3–17. DOI: 10.1007/978-3-319-69456-6_1 (cit. on p. 47).

Hyvärinen, A. E. J., M. Marescotti, L. Alt, and N. Sharygina (2016). “OpenSMT2: An SMT Solver for Multi-core and Cloud Computing”. In: *Theory and Applications of Satisfiability Testing - SAT*. Vol. 9710. Springer, pp. 547–553. DOI: 10.1007/978-3-319-40970-2_35 (cit. on p. 111).

Imbert, J.-L. (1990). “About Redundant Inequalities Generated by Fourier’s Algorithm”. In: *Proceedings of the Fourth International Conference on Artificial Intelligence: Methodology, Systems, Applications, AIMSA*. Elsevier, pp. 117–127 (cit. on p. 94).

Ivrii, A., S. Malik, K. S. Meel, and M. Y. Vardi (2016). "On computing Minimal Independent Support and its applications to sampling and counting". In: *Constraints* 21.1, pp. 41–58. DOI: 10.1007/s10601-015-9204-z (cit. on p. 62).

Karp, R. M. (1972). "Reducibility Among Combinatorial Problems". In: *Complexity of Computer Computations*. The IBM Research Symposia Series. Plenum Press, New York, pp. 85–103 (cit. on p. 14).

Kask, K., A. Gelfand, L. Otten, and R. Dechter (2011). "Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical Models". In: *Proceedings of the 25th AAAI Conference on Artificial Intelligence* (cit. on p. 95, 104).

Kersting, K. and L. De Raedt (2000). "Bayesian Logic Programs". In: *Proceedings of the 10th International Conference on Inductive Logic Programming, Work-in-progress reports*, pp. 1–18 (cit. on p. 28).

Kiesel, R. and T. Eiter (2023). "Knowledge Compilation and More with SharpSAT-TD". In: *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR*. Ed. by P. Marquis, T. C. Son, and G. Kern-Isberner, pp. 406–416. DOI: 10.24963/kr.2023/40 (cit. on p. 21).

Kimmig, A. and F. Costa (2012). "Link and Node Prediction in Metabolic Networks with Probabilistic Logic". In: *Bisociative Knowledge Discovery: An Introduction to Concept, Algorithms, Tools, and Applications*. Vol. 7250. Springer, pp. 407–426. ISBN: 978-3-642-31830-6. DOI: 10.1007/978-3-642-31830-6_29 (cit. on p. 47).

Kimmig, A., G. Van den Broeck, and L. De Raedt (2011). "An Algebraic Prolog for Reasoning about Possible Worlds". In: *Proceedings of the 25th AAAI Conference on Artificial Intelligence* (cit. on p. 8, 29, 37, 38, 44, 68, 76).

Kimmig, A., G. Van den Broeck, and L. De Raedt (2017). "Algebraic model counting". In: *Journal of Applied Logic* 22, pp. 46–62. DOI: 10.1016/j.jal.2016.11.031 (cit. on p. 3, 13, 14, 20, 24, 30, 32, 68).

Kitching, M. and F. Bacchus (2007). "Symmetric Component Caching". In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 118–124 (cit. on p. 52, 118).

Kjærulff, U. (1990). "Triangulation of graphs–algorithms giving small total state space". In: *Research report* (cit. on p. 85).

Kolb, S. (2019). "Learn + Solve: Learning and Solving Constrained Hybrid Inference Problems". PhD thesis. KU Leuven (cit. on p. 88, 145, 147).

Kolb, S., M. Mladenov, S. Sanner, V. Belle, and K. Kersting (2018). "Efficient Symbolic Integration for Probabilistic Inference". In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI* (cit. on p. 85, 87).

Kolb, S., P. Morettin, P. Z. D. Martires, F. Sommavilla, A. Passerini, R. Sebastiani, and L. D. Raedt (2019a). "The pywmi Framework and Toolbox for Probabilistic Inference using Weighted Model Integration". In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 6530–6532. DOI: 10.24963/ijcai.2019/946 (cit. on p. 85).

Kolb, S., P. Zuidberg Dos Martires, and L. De Raedt (2019b). "How to Exploit Structure while Solving Weighted Model Integration Problems". In: *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence, UAI*. Vol. 115. AUAI Press, pp. 744–754 (cit. on p. 7, 9, 84, 85, 87, 88, 91, 92, 102, 109, 147).

Koller, D. and N. Friedman (2009). *Probabilistic Graphical Models - Principles and Techniques*. MIT Press. ISBN: 978-0-262-01319-2 (cit. on p. 68, 95).

Korb, K. B. and A. E. Nicholson (2010). *Bayesian artificial intelligence*. CRC press (cit. on p. 76).

Koriche, F., J. Lagniez, P. Marquis, and S. Thomas (2015). "Compiling Constraint Networks into Multivalued Decomposable Decision Graphs". In: *IJCAI*. AAAI Press, pp. 332–338 (cit. on p. 106, 112, 113).

Körner, P., M. Leuschel, J. Barbosa, V. S. Costa, V. Dahl, M. V. Hermenegildo, J. F. Morales, J. Wielemaker, D. Diaz, and S. Abreu (2022). "Fifty Years of Prolog and Beyond". In: *Theory and Practice of Logic Programming* 22.6, pp. 776–858. DOI: 10.1017/S1471068422000102 (cit. on p. 1).

Kuiter, E., S. Krieter, C. Sundermann, T. Thüm, and G. Saake (2023). "Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses". In: *Software Engineering*. Vol. P-332. LNI. Gesellschaft für Informatik e.V., pp. 83–84 (cit. on p. 15).

Lagniez, J. and P. Marquis (2017). "An Improved Decision-DNNF Compiler". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 667–673. DOI: 10.24963/ijcai.2017/93 (cit. on p. 15, 21, 50, 66, 117).

Lagniez, J.-M. and P. Marquis (2021). "About Caching in D4 2.0". In: *Workshop on Counting and Sampling* (cit. on p. 51, 59).

Lakshmanan, L. V., N. Leone, R. Ross, and V. S. Subrahmanian (1997). "ProbView: A Flexible Probabilistic Database System". In: *ACM Transactions on Database Systems, TODS* 22.3, pp. 419–469. DOI: 10.1145/261124.261131 (cit. on p. 29).

Latour, A. L. D., B. Babaki, A. Dries, A. Kimmig, G. Van den Broeck, and S. Nijssen (2017). "Combining Stochastic Constraint Optimization and Probabilistic Programming - From Knowledge Compilation to Constraint Solving". In: *Principles and Practice of Constraint Programming - 23rd International Conference, CP*. Vol. 10416. Springer, pp. 495–511. DOI: 10.1007/978-3-319-66158-2_32 (cit. on p. 47, 71).

Lauria, M., J. Elffers, J. Nordström, and M. Vinyals (2017). "CNFgen: A Generator of Crafted Benchmarks". In: *Theory and Applications of Satisfiability Testing - SAT*. Vol. 10491. Springer, pp. 464–473 (cit. on p. 51, 124).

Lauritzen, S. L. and D. J. Spiegelhalter (1988). "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 50.2, pp. 157–194 (cit. on p. 76).

Lee, J., R. Marinescu, A. T. Ihler, and R. Dechter (2019). "A Weighted Mini-Bucket Bound for Solving Influence Diagram". In: *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence, UAI*. Ed. by A. Globerson and R. Silva. AUAI Press, p. 432 (cit. on p. 82).

Lemire, D. and O. Kaser (2016). "Faster 64-bit universal hashing using carry-less multiplications". In: *Journal of Cryptographic Engineering* 6.3, pp. 171–185. DOI: 10.1007/s13389-015-0110-5 (cit. on p. 57).

Li, Z. and J. Eisner (2009). "First- and Second-order Expectation Semirings with Applications to Minimum-risk Training on Translation Forests". In: *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, EMNLP*. Singapore: Association for Computational Linguistics, pp. 40–51. ISBN: 978-1-932432-59-6 (cit. on p. 81, 116).

Lloyd, J. W. (2012). *Foundations of Logic Programming*. Springer Science & Business Media (cit. on p. 28).

Ma, F., S. Liu, and J. Zhang (2009). "Volume Computation for Boolean Combination of Linear Arithmetic Constraints". In: *22nd International Conference on Automated Deduction, CADE*. Vol. 5663. Springer, pp. 453–468. DOI: 10.1007/978-3-642-02959-2_33 (cit. on p. 113).

Majercik, S. M. and M. L. Littman (1998). "Using Caching to Solve Larger Probabilistic Planning Problems". In: *Proceedings of the 15th National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI/IAAI*. Menlo Park, CA, USA: American Association for Artificial Intelligence, pp. 954–959. ISBN: 0-262-51098-7 (cit. on p. 81).

Manhaeve, R., S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt (2018). "DeepProbLog: Neural Probabilistic Logic Programming". In: *Advances in Neural Information Processing Systems, NeurIPS*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, pp. 3753–3763 (cit. on p. 8, 29, 78, 120).

Manhaeve, R., S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2021a). "Neural probabilistic logic programming in DeepProbLog". In: *Artificial Intelligence* 298, p. 103504 (cit. on p. 33, 34, 42, 120).

Manhaeve, R., G. Marra, T. Demeester, S. Dumancic, A. Kimmig, and L. D. Raedt (2021b). "Neuro-Symbolic AI = Neural + Logical + Probabilistic AI". In: *Neuro-Symbolic Artificial Intelligence: The State of the Art*. Ed. by

P. Hitzler and M. K. Sarker. Vol. 342. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 173–191. DOI: 10.3233/FAIA210354 (cit. on p. 120).

Mantadelis, T. and S. Bistarelli (2020). "Probabilistic abstract argumentation frameworks, a possible world view". In: *International Journal of Approximate Reasoning* 119, pp. 204–219 (cit. on p. 47).

Marinescu, R. (2009). "A New Approach to Influence Diagrams Evaluation". In: *Research and Development in Intelligent Systems XXVI, Incorporating Applications and Innovations in Intelligent Systems XVII, SGAI*. Ed. by M. Bramer, R. Ellis, and M. Petridis. Springer, pp. 107–120. ISBN: 978-1-84882-983-1. DOI: 10.1007/978-1-84882-983-1_8 (cit. on p. 82).

Marquis, P. (2008). *Knowledge Compilation: A Sightseeing Tour*. In Tutorial notes, ECAI'08, 2008. available on-line (cit. on p. 68).

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. URL: https://www.tensorflow.org/ (cit. on p. 75, 77).

McAreavey, K., K. Bauters, W. Liu, and J. Hong (2017). "The Event Calculus in Probabilistic Logic Programming with Annotated Disjunctions". In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS*. ACM, pp. 105–113 (cit. on p. 47).

McCarthy, J. (1959). *Programs with common sense* (cit. on p. 1).

McCarthy, J., M. Minsky, N. Rochester, and C. E. Shannon (Aug. 1955). *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. URL: https://raysolomonoff.com/dartmouth/boxa/dart564props.pdf (cit. on p. 1).

McKay, B. D. and A. Piperno (2014). "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60, pp. 94–112 (cit. on p. 55, 60, 116).

Mekuria, D. N., P. Sernani, N. Falcionelli, and A. F. Dragoni (2019). "A Probabilistic Multi-Agent System Architecture for Reasoning in Smart Homes". In: *IEEE International Symposium on INnovations in Intelligent SysTems and Applications, INISTA*, pp. 1–6. DOI: 10.1109/INISTA.2019.8778306 (cit. on p. 47).

Melibari, M., P. Poupart, and P. Doshi (2016). "Sum-Product-Max Networks for Tractable Decision Making". In: *Proceedings of the 25th International*

Joint Conference on Artificial Intelligence, IJCAI, pp. 1846–1852 (cit. on p. 81, 82).

Miosic, I. and P. Zuidberg Dos Martires (2021). "Measure Theoretic Weighted Model Integration". In: *CoRR* abs/2103.13901 (cit. on p. 36, 38).

Moldovan, B., L. Antanas, and M. Hoffmann (2012a). "Opening Doors: An Initial SRL Approach". In: *Inductive Logic Programming - 22nd International Conference, ILP*. Vol. 7842. Springer, pp. 178–192. DOI: 10.1007/978-3-642-38812-5_13 (cit. on p. 47).

Moldovan, B., P. Moreno, D. Nitti, J. Santos-Victor, and L. De Raedt (2018). "Relational affordances for multiple-object manipulation". In: *Autonomous Robots* 42.1, pp. 19–44. DOI: 10.1007/s10514-017-9637-x (cit. on p. 47).

Moldovan, B., P. Moreno, M. van Otterlo, J. Santos-Victor, and L. De Raedt (2012b). "Learning relational affordance models for robots in multi-object manipulation tasks". In: *IEEE International Conference on Robotics and Automation, ICRA*, pp. 4373–4378. DOI: 10.1109/ICRA.2012.6225042 (cit. on p. 47).

Moldovan, B., M. van Otterlo, L. De Raedt, P. Moreno, and J. Santos-Victor (2011). "Statistical Relational Learning of Object Affordances for Robotic Manipulation". In: *Latest Advances in Inductive Logic Programming, ILP, Late Breaking Papers*. Imperial College Press / World Scientific, pp. 95–103. DOI: 10.1142/9781783265091_0012 (cit. on p. 47).

Møller, J. B., J. Lichtenberg, H. R. Andersen, and H. Hulgaard (1999). "Difference Decision Diagrams". In: *Computer Science Logic, 13th International Workshop, CSL*. Vol. 1683. Springer, pp. 111–125. DOI: 10.1007/3-540-48168-0_9 (cit. on p. 106, 112).

Morettin, P., A. Passerini, and R. Sebastiani (2019). "Advanced SMT techniques for weighted model integration". In: *Artificial Intelligence* 275 (cit. on p. 25).

Morrison, D. R., S. H. Jacobson, J. J. Sauppe, and E. C. Sewell (2016). "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning". In: *Discret. Optim.* 19, pp. 79–102 (cit. on p. 14).

Moura, L. M. de and N. S. Bjørner (2008). "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*. Vol. 4963. Springer, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 107, 111).

Muise, C. J., S. A. McIlraith, J. C. Beck, and E. I. Hsu (2012). "Dsharp: Fast d-DNNF Compilation with sharpSAT". In: *Canadian Conference on AI*. Ed. by L. Kosseim and D. Inkpen. Vol. 7310. Springer, pp. 356–361. ISBN: 978-3-642-30352-4. DOI: 10.1007/978-3-642-30353-1_36 (cit. on p. 21).

Ng, A. Y. and S. J. Russell (2000). "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the 17th International Conference on Machine Learning ICML*. Morgan Kaufmann, pp. 663–670 (cit. on p. 82).

Ng, R. and V. S. Subrahmanian (1992). "Probabilistic Logic Programming". In: *Information and Computation* 101.2, pp. 150–201. DOI: 10.1016/0890-5401(92)90061-J (cit. on p. 28).

Niepert, M. (2012). "Markov Chains on Orbits of Permutation Groups". In: *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence,UAI*. AUAI Press, pp. 624–633 (cit. on p. 52).

Nieuwenhuis, R., A. Oliveras, and C. Tinelli (2006). "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)". In: *Journal of the ACM* 53.6, pp. 937–977. ISSN: 0004-5411. DOI: 10.1145/1217856.1217859 (cit. on p. 106, 107, 110).

Nilsson, N. J. (1986). "Probabilistic logic". In: *Artificial intelligence* 28.1, pp. 71–87 (cit. on p. 28).

Nitti, D., V. Belle, and L. De Raedt (2015). "Planning in Discrete and Continuous Markov Decision Processes by Probabilistic Programming". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML/PKDD*. Vol. 9285. Springer, pp. 327–342. DOI: 10.1007/978-3-319-23525-7_20 (cit. on p. 47).

Nitti, D., V. Belle, T. D. Laet, and L. De Raedt (2017). "Planning in hybrid relational MDPs". In: *Machine Learning* 106.12, pp. 1905–1932. DOI: 10.1007/s10994-017-5669-x (cit. on p. 47).

Nitti, D., T. De Laet, and L. De Raedt (2014). "Relational Object Tracking and Learning". In: *IEEE International Conference on Robotics and Automation, ICRA*. IEEE, pp. 935–942. DOI: 10.1109/ICRA.2014.6906966 (cit. on p. 47).

Niveau, A. (2012). "Compilation de connaissances pour la décision en ligne : application à la conduite de systèmes autonomes. (Knowledge compilation for online decision-making : application to the control of autonomous systems)". PhD thesis. Paul Sabatier University, Toulouse, France (cit. on p. 106, 112).

Olteanu, D., J. Huang, and C. Koch (2009). "SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases". In: *Proceedings of the 25th International Conference on Data Engineering, ICDE*. IEEE. IEEE Computer Society, pp. 640–651. DOI: 10.1109/ICDE.2009.123 (cit. on p. 29).

Orsini, F., P. Frasconi, and L. De Raedt (2017). "kProbLog: an algebraic Prolog for machine learning". In: *Machine Learning* 106.12, pp. 1933–1969. DOI: 10.1007/s10994-017-5668-y (cit. on p. 29, 30, 38).

Oztok, U., A. Choi, and A. Darwiche (2016). "Solving $\mathrm{PP}^{\mathrm{PP}}$-Complete Problems Using Knowledge Compilation". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the 15th International Conference, KR*, pp. 94–103 (cit. on p. 69).

Oztok, U. and A. Darwiche (2014). "On Compiling CNF into Decision-DNNF". In: *Principles and Practice of Constraint Programming - 20th International*

Conference, CP. Ed. by B. O'Sullivan. Vol. 8656. Springer, pp. 42–57. DOI: 10.1007/978-3-319-10428-7_7 (cit. on p. 21).

Oztok, U. and A. Darwiche (2015). "A Top-Down Compiler for Sentential Decision Diagrams". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 3141–3148 (cit. on p. 21, 24, 50).

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems, NeurIPS*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 8024–8035 (cit. on p. 75, 77).

Pearl, J. (1982). "Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach". In: *Proceedings of the National Conference on Artificial Intelligence, AAAI*, pp. 133–136 (cit. on p. 84).

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier (cit. on p. 28).

Persson, A., P. Zuidberg Dos Martires, L. De Raedt, and A. Loutfi (2020). "Semantic Relational Object Tracking". In: *IEEE Transactions on Cognitive and Developmental Systems* 12.1, pp. 84–97. DOI: 10.1109/TCDS.2019.2915763 (cit. on p. 47).

Pipatsrisawat, T. and A. Darwiche (2008). "New Compilation Languages Based on Structured Decomposability". In: *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*. Vol. 8, pp. 517–522 (cit. on p. 24, 111).

Pipatsrisawat, T. and A. Darwiche (2010). "A Lower Bound on the Size of Decomposable Negation Normal Form". In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence* (cit. on p. 24).

Poole, D. (1993). "Probabilistic Horn Abduction and Bayesian Networks". In: *Artificial Intelligence* 64.1, pp. 81–129. DOI: 10.1016/0004-3702(93)90061-F (cit. on p. 28).

Poole, D. (1997). "The Independent Choice Logic for Modelling Multiple Agents Under Uncertainty". In: *Artificial intelligence* 94.1-2, pp. 7–56. DOI: 10.1016/S0004-3702(97)00027-1 (cit. on p. 28).

Popov, M., T. Balyo, M. Iser, and T. Ostertag (2023). "Construction of Decision Diagrams for Product Configuration". In: *ConfWS*. Vol. 3509. CEUR Workshop Proceedings, pp. 108–117 (cit. on p. 121).

Pralet, C., G. Verfaillie, and T. Schiex (2007). "An Algebraic Graphical Model for Decision with Uncertainties, Feasibilities, and Utilities". In: *J. Artif. Intell. Res.* 29, pp. 421–489 (cit. on p. 72).

Rice, M. and S. Kulhari (2008). "A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction". In: *University of California, Technical Report*, p. 130 (cit. on p. 111).

Riguzzi, F. (2018). *Foundations of Probabilistic Logic Programming*. Gistrup, Denmark: River Publishers. ISBN: 9788770220187 (cit. on p. 28, 29).

Roig Vilamala, M., T. Xing, H. Taylor, L. Garcia, M. Srivastava, L. Kaplan, A. Preece, A. Kimmig, and F. Cerutti (2023). "DeepProbCEP: A neuro-symbolic approach for complex event processing in adversarial settings". In: *Expert Systems with Applications* 215, p. 119376. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2022.119376 (cit. on p. 47).

Rothkopf, C. A. and C. Dimitrakakis (2011). "Preference Elicitation and Inverse Reinforcement Learning". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML/PKDD*. Ed. by D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis. Berlin, Heidelberg, pp. 34–48. ISBN: 978-3-642-23808-6 (cit. on p. 82).

Russell, S. J. and P. Norvig (2010). *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education. ISBN: 978-0-13-207148-2 (cit. on p. 1).

Sabharwal, A. (2009). "SymChaff: exploiting symmetry in a structure-aware satisfiability solver". In: *Constraints* 14.4, pp. 478–505. DOI: 10.1007/s10601-008-9060-1 (cit. on p. 52).

Salmon, R. and P. Poupart (2019). "On the Relationship Between Satisfiability and Markov Decision Processes". In: *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence, UAI*. Ed. by A. Globerson and R. Silva. Vol. 115. AUAI Press, pp. 1105–1115 (cit. on p. 52).

Sang, T., F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi (2004). "Combining Component Caching and Clause Learning for Effective Model Counting". In: *Theory and Applications of Satisfiability Testing - SAT* (cit. on p. 16, 17, 18, 50, 51, 53, 58, 59).

Sang, T., P. Beame, and H. A. Kautz (2005). "Heuristics for Fast Exact Model Counting". In: *Theory and Applications of Satisfiability Testing - SAT*. Vol. 3569. Springer, pp. 226–240 (cit. on p. 15, 51, 53, 60).

Sanner, S., K. V. Delgado, and L. N. de Barros (2011). "Symbolic Dynamic Programming for Discrete and Continuous State MDPs". In: *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI*. AUAI Press, pp. 643–652 (cit. on p. 106, 112, 145).

Sashittal, P. and M. El-Kebir (2020). "Sampling and summarizing transmission trees with multi-strain infections". In: *Bioinformatics* 36.Supplement-1, pp. i362–i370. DOI: 10.1093/bioinformatics/btaa438 (cit. on p. 49).

Sato, T. (1995). "A Statistical Learning Method for Logic Programs with Distribution Semantics". In: *Proceedings of the 12th International Conference*

on Logic Programming, ICLP. Ed. by L. Sterling. MIT Press, pp. 715–729 (cit. on p. 28).

Sato, T. and Y. Kameya (1997). "PRISM: A Language for Symbolic-Statistical Modeling". In: *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI*. Vol. 97, pp. 1330–1339 (cit. on p. 29).

Schrag, R. J. B. R. (1997). "Using CSP Look-Back Techniques to Solve Real-World SAT Instances". In: *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference, AAAI/IAAI*. AAAI Press / The MIT Press, pp. 203–208 (cit. on p. 16, 107).

Schweitzer, P. (2009). "Problems of Unknown Complexity: Graph isomorphism and Ramsey theoretic numbers". PhD thesis. Saarland University (cit. on p. 54).

Scutari, M. (n.d.). *BNLearn: Bayesian Network Repository*. `http://www.bnlearn.com/bnrepository/`. Accessed: 2019-11-10 (cit. on p. 76).

Scutari, M. and J.-B. Denis (2014). *Bayesian networks: with examples in R*. Chapman and Hall/CRC (cit. on p. 76).

Shah, N., W. Meert, and M. Verhelst (2023). *Efficient Execution of Irregular Dataflow Graphs: Hardware/Software Co-optimization for Probabilistic AI and Sparse Linear Algebra*. Springer. ISBN: 978-3-031-33136-7. DOI: `10.1007/978-3-031-33136-7` (cit. on p. 121).

Sharma, S., S. Roy, M. Soos, and K. S. Meel (2019). "GANAK: A Scalable Probabilistic Exact Model Counter". In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 1169–1176. DOI: `10.24963/ijcai.2019/163` (cit. on p. 15, 51, 53, 57, 60, 117).

Shih, A., G. Van den Broeck, P. Beame, and A. Amarilli (2019). "Smoothing Structured Decomposable Circuits". In: *Advances in Neural Information Processing Systems, NeurIPS*, pp. 11412–11422 (cit. on p. 20).

Silva, J. P. M. and K. A. Sakallah (1996). "GRASP - a new search algorithm for satisfiability". In: *International Conference on Computer-Aided Design, ICCAD*. IEEE, pp. 220–227. DOI: `10.1109/ICCAD.1996.569607` (cit. on p. 16).

Skarlatidis, A., A. Artikis, J. Filipou, and G. Paliouras (2015). "A probabilistic logic programming event calculus". In: *Theory and Practice of Logic Programming* 15.2, pp. 213–245. DOI: `10.1017/S1471068413000690` (cit. on p. 47).

Smith, G., R. P. A. Petrick, and V. Belle (2021). "Intent Recognition in Smart Homes with ProbLog". In: *PerCom Workshops*. IEEE, pp. 430–431 (cit. on p. 47).

Somenzi, F. (1997). "CUDD: CU decision diagram package". In: *Public Software, University of Colorado* (cit. on p. 22).

Soos, M. and K. S. Meel (2019). "BIRD: Engineering an Efficient CNF-XOR SAT Solver and Its Applications to Approximate Model Counting". In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 1592–1599. DOI: 10.1609/aaai.v33i01.33011592 (cit. on p. 50).

Spallitta, G., G. Masina, P. Morettin, A. Passerini, and R. Sebastiani (2022). "SMT-based Weighted Model Integration with Structure Awareness". In: *Proceedings of the 38th Conference on Uncertainty in Artificial Intelligence, UAI*. PMLR, pp. 1876–1885 (cit. on p. 121).

Stockmeyer, L. J. (1983). "The Complexity of Approximate Counting". In: *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, STOC*. ACM, pp. 118–126. DOI: 10.1145/800061.808740 (cit. on p. 50).

Sundermann, C., T. Thüm, and I. Schaefer (2020). "Evaluating #SAT solvers on industrial feature models". In: *VaMoS*. ACM, 3:1–3:9 (cit. on p. 121).

Suryadi, D. and P. J. Gmytrasiewicz (1999). "Learning Models of Other Agents Using Influence Diagrams". In: *Proceedings of the 7th International Conference on User Modeling*. Secaucus, NJ, USA, pp. 223–232. ISBN: 3-211-83151-7 (cit. on p. 82).

Suster, S., P. Fivez, P. Totis, A. Kimmig, J. Davis, L. De Raedt, and W. Daelemans (2021). "Mapping probability word problems to executable representations". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP*. Association for Computational Linguistics, pp. 3627–3640. DOI: 10.18653/v1/2021.emnlp-main.294 (cit. on p. 47).

Sztyler, T., G. Civitarese, and H. Stuckenschmidt (2018). "Modeling and Reasoning with ProbLog: An Application in Recognizing Complex Activities". In: *PerCom Workshops*. IEEE Computer Society, pp. 259–264 (cit. on p. 47).

Tang, Y., K. Hatano, and E. Takimoto (2023). "Boosting-Based Construction of BDDs for Linear Threshold Functions and Its Application to Verification of Neural Networks". In: *DS*. Vol. 14276. Springer, pp. 477–491 (cit. on p. 121).

Tarim, A., S. Manandhar, and T. Walsh (2006). "Stochastic Constraint Programming: A Scenario-Based Approach". In: *Constraints An Int. J.* 11.1, pp. 53–80 (cit. on p. 71).

Thon, I., N. Landwehr, and L. De Raedt (2008). "A Simple Model for Sequences of Relational State Descriptions". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML/PKDD*. Ed. by W. Daelemans, B. Goethals, and K. Morik, pp. 506–521. ISBN: 978-3-540-87481-2. DOI: 10.1007/978-3-540-87481-2_33 (cit. on p. 47).

Thon, I., N. Landwehr, and L. De Raedt (2011). "Stochastic relational processes: Efficient inference and applications". In: *Machine Learning* 82.2, pp. 239–272. DOI: 10.1007/s10994-010-5213-8 (cit. on p. 47).

Thüm, T. (2020). "A BDD for Linux?: the knowledge compilation challenge for variability". In: *SPLC (A)*. ACM, 16:1–16:6 (cit. on p. 121).

Thurley, M. (2006). "sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP". In: *Theory and Applications of Satisfiability Testing - SAT*. Vol. 4121. Springer, pp. 424–429. DOI: 10.1007/11814948_38 (cit. on p. 50, 51, 53, 59).

Totis, P., A. Kimmig, and L. De Raedt (2021). "SMProbLog: Stable Model Semantics in ProbLog and its Applications in Argumentation". In: *CoRR* abs/2110.01990 (cit. on p. 47).

Tseitin, G. S. (1983). "On the Complexity of Derivation in Propositional Calculus". In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by J. H. Siekmann and G. Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28 (cit. on p. 15, 111).

Vaezipoor, P., G. Lederman, Y. Wu, C. J. Maddison, R. B. Grosse, S. A. Seshia, and F. Bacchus (2021). "Learning Branching Heuristics for Propositional Model Counting". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 12427–12435 (cit. on p. 15).

Valiant, L. G. (1979a). "The complexity of computing the permanent". In: *Theoretical Computer Science* 8.2, pp. 189–201 (cit. on p. 14, 18).

Valiant, L. G. (1979b). "The Complexity of Enumeration and Reliability Problems". In: *SIAM Journal on Computing* 8.3, pp. 410–421. DOI: 10.1137/0208032 (cit. on p. 50).

van Bremen, T., V. Derkinderen, S. Sharma, S. Roy, and K. S. Meel (2021). "Symmetric Component Caching for Model Counting on Combinatorial Instances". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 3922–3930 (cit. on p. 9, 12, 14, 49, 175).

van Bremen, T., A. Dries, and J. C. Jung (2019). "Ontology-Mediated Queries over Probabilistic Data via Probabilistic Logic Programming". In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM*. ACM, pp. 2437–2440. DOI: 10.1145/3357384.3358168 (cit. on p. 47).

van Bremen, T., A. Dries, and J. C. Jung (2020). "onto2problog: A Probabilistic Ontology-Mediated Querying System using Probabilistic Logic Programming". In: *Künstliche Intelligenz* 34.4, pp. 501–507. DOI: 10.1007/s13218-020-00670-x (cit. on p. 47).

Van den Broeck, G., D. Suciu, et al. (2017). "Query Processing on Probabilistic Data: A Survey". In: *Foundations and Trends® in Databases* 7.3-4, pp. 197–341. DOI: 10.1561/1900000052 (cit. on p. 29).

Van den Broeck, G., I. Thon, M. v. Otterlo, and L. De Raedt (2010). "DTPROBLOG: A Decision-theoretic Probabilistic Prolog". In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 1217–1222 (cit. on p. 46, 47, 68, 81).

Veiga, T., M. Silva, R. Ventura, and P. U. Lima (2019). "A Hierarchical Approach to Active Semantic Mapping Using Probabilistic Logic and Information Reward POMDPs". In: *Proceedings of the 29th International Conference on Automated Planning and Scheduling, ICAPS*, pp. 773–781 (cit. on p. 47).

Vennekens, J., S. Verbaeten, and M. Bruynooghe (2004). "Logic programs with annotated disjunctions". In: *Proceedings of the 20th International Conference on Logic Programming, ICLP*. Springer, pp. 431–445 (cit. on p. 29).

Venturato, G., V. Derkinderen, P. Zuidberg Dos Martires, and L. De Raedt (2022). "Towards Tractable Dynamic Decision Making With Circuits". In: *5th Workshop on Tractable Probabilistic Modeling at UAI 2022* (cit. on p. 47, 82, 176).

Venturato, G., V. Derkinderen, P. Zuidberg Dos Martires, and L. De Raedt (2024). "Inference and Learning in Dynamic Decision Networks Using Knowledge Compilation". In: *Accepted in Proceedings of the 38th AAAI Conference on Artificial Intelligence*. AAAI Press (cit. on p. 175).

Verreet, V., V. Derkinderen, P. Zuidberg Dos Martires, and L. De Raedt (2022a). "Inference and Learning with Model Uncertainty in Probabilistic Logic Programs". In: *Proceedings of the 36th AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 10060–10069 (cit. on p. 38, 46, 175).

Verreet, V., V. Derkinderen, P. Zuidberg Dos Martires, and L. De Raedt (2022b). "Inference and Learning with Model Uncertainty in Probabilistic Logic Programs". In: *International Conference on Logic Programming, ICLP (Recently Published Paper Track)* (cit. on p. 176).

Vlasselaer, J. and W. Meert (2012). "Statistical relational learning for prognostics". In: *Proceedings of the 21st Belgian-Dutch Conference on Machine Learning*, pp. 45–50 (cit. on p. 47).

Vlasselaer, J., G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2015). "Anytime Inference in Probabilistic Logic Programs with T_P-Compilation". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI*. Vol. 2015, pp. 1852–1858 (cit. on p. 29).

Vlasselaer, J., G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2016). "T_P-Compilation for inference in probabilistic logic programs". In: *International Journal of Approximate Reasoning* 78, pp. 15–32. DOI: 10.1016/j.ijar.2016.06.009 (cit. on p. 29).

Walsh, T. (2002). "Stochastic Constraint Programming". In: *ECAI*. IOS Press, pp. 111–115 (cit. on p. 71).

Wang, W., M. Usman, A. Almaawi, K. Wang, K. S. Meel, and S. Khurshid (2020). "A Study of Symmetry Breaking Predicates and Model Counting". In: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS*. Vol. 12078. Springer, pp. 115–134. DOI: 10.1007/978-3-030-45190-5_7 (cit. on p. 51, 52, 125).

Wang, Y. (2015). "ProbLog Program Based Ontology Matching". In: *Knowledge Science, Engineering and Management - 8th International Conference, KSEM*. Vol. 9403. Springer, pp. 778–783. DOI: 10.1007/978-3-319-25159-2_72 (cit. on p. 47).

Wang, Z., S. Vijayakumar, K. Lu, V. Ganesh, S. Jha, and M. Fredrikson (2023). "Advances in Neural Information Processing Systems, NeurIPS". In: (cit. on p. 121).

Weitkämper, F., B. Sarbu, and K. Sun (2021). "Modelling Infectious Disease Dynamics with Probabilistic Logic Programming". In: *Workshops co-located with the 37th International Conference on Logic Programming, ICLP*. Vol. 2970. CEUR Workshop Proceedings (cit. on p. 47).

Xing, T., M. R. Vilamala, L. Garcia, F. Cerutti, L. M. Kaplan, A. D. Preece, and M. B. Srivastava (2019). "DeepCEP: Deep Complex Event Processing Using Distributed Multimodal Information". In: *IEEE International Conference on Smart Computing, SMARTCOMP*. IEEE, pp. 87–92. DOI: 10.1109/SMARTCOMP.2019.00034 (cit. on p. 47).

Xu, J., Z. Zhang, T. Friedman, Y. Liang, and G. V. den Broeck (2018). "A Semantic Loss Function for Deep Learning with Symbolic Knowledge". In: *ICML*. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 5498–5507 (cit. on p. 120).

Yang, S. (1991). *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*. Tech. rep. MCNC Technical Report (cit. on p. 51).

Yang, W.-C., G. Marra, G. Rens, and L. De Raedt (2023). "Safe Reinforcement Learning via Probabilistic Logic Shields". In: *Proceedings of the 17th International Workshop on Neural-Symbolic Learning and Reasoning, NeSy*. Vol. 3432. CEUR Workshop Proceedings, pp. 428–429 (cit. on p. 47).

Yang, Z., A. Ishay, and J. Lee (2020). "NeurASP: Embracing Neural Networks into Answer Set Programming". In: *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 1755–1762. DOI: 10.24963/ijcai.2020/243 (cit. on p. 29, 46).

Zeng, Z. and G. Van den Broeck (2019). "Efficient Search-Based Weighted Model Integration". In: *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence, UAI*. Vol. 115, pp. 175–185 (cit. on p. 85, 102, 103).

Zhang, N. L. and D. Poole (1994). "A simple approach to Bayesian network computations". In: *Proceedings of the 10th Canadian Conference on Artificial Intelligence, CSCSI*, pp. 171–178 (cit. on p. 84).

Zuidberg Dos Martires, P. (2020). "From Atoms to Possible Worlds: Probabilistic Inference in the Discrete-Continuous Domain". PhD thesis. KU Leuven (cit. on p. 4).

Zuidberg Dos Martires, P. (2021). "Neural Semirings". In: *Proceedings of the 15th International Workshop on Neural-Symbolic Learning and Reasoning, NeSy*. Vol. 2986. CEUR Workshop Proceedings, pp. 94–103 (cit. on p. 46).

Zuidberg Dos Martires, P., L. De Raedt, and A. Kimmig (2023). "Declarative Probabilistic Logic Programming in Discrete-Continuous Domains". In: *CoRR* abs/2302.10674. DOI: 10.48550/arXiv.2302.10674 (cit. on p. 8, 34, 36, 37, 38).

Zuidberg Dos Martires, P., V. Derkinderen, R. Manhaeve, W. Meert, A. Kimmig, and L. De Raedt (2019a). "Transforming probabilistic programs into algebraic circuits for inference and learning". In: *Program Transformations for ML Workshop at NeurIPS 2019* (cit. on p. 75, 176).

Zuidberg Dos Martires, P., A. Dries, and L. De Raedt (2019b). "Exact and Approximate Weighted Model Integration with Probability Density Functions Using Knowledge Compilation". In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, pp. 7825–7833. DOI: 10.1609/aaai.v33i01.33017825 (cit. on p. 36, 38, 68, 87, 145).

Zuidberg Dos Martires, P., N. Kumar, A. Persson, A. Loutfi, and L. De Raedt (2020). "Symbolic Learning and Reasoning With Noisy Data for Probabilistic Anchoring". In: *Frontiers Robotics AI* 7, p. 100 (cit. on p. 47).

Curriculum Vitae

Vincent Derkinderen obtained his Bachelor of Informatics degree in 2016 at KU Leuven. Two years later and at the same university, he completed his Master of Engineering degree in the field of Computer Science, graduating magna cum laude. His specialization was Artificial Intelligence, and the title of his master thesis was *"Subgraph Search in Arithmetic Circuits for Efficient Hardware Design"*.

He officially started his doctoral studies in October 2018, under the supervision of Prof. dr. Luc De Raedt at the DTAI (Declarative Languages and Artificial Intelligence) lab in Leuven. Shortly afterwards, he received a PhD fellowship from the Research Foundation – Flanders (FWO) through the strategic basic research grant 1SA5520N, for the time period of November 2019 to 2023, for which he is very grateful. During this period, he has served as a workflow chair for the IJCAI-ECAI 2022 conference, has served as a program committee member for several conferences, including the UAI conference where he won a top reviewer award in 2023, and has reviewed for the Journal of Machine Learning Research (JMLR). In December 2023, he will defend his doctoral dissertation titled *"Knowledge Compilation and Counting: an Algebraic Journey"*

List of publications

Journal articles

V. Derkinderen, J. Bekker, and P. Smet (2023a). "Optimizing workforce allocation under uncertain activity duration". In: *Computers & Industrial Engineering* 179, p. 109228. DOI: 10.1016/j.cie.2023.109228

V. Derkinderen, R. Manhaeve, P. Zuidberg Dos Martires, and L. De Raedt (2023c). "Semirings for Probabilistic and Neural-Symbolic Logic Programming". Accepted with minor revision in International Journal of Approximate Reasoning

Conference proceedings

G. Venturato, V. Derkinderen, P. Zuidberg Dos Martires, and L. De Raedt (2024). "Inference and Learning in Dynamic Decision Networks Using Knowledge Compilation". In: *Accepted in Proceedings of the 38th AAAI Conference on Artificial Intelligence*. AAAI Press

V. Verreet, V. Derkinderen, P. Zuidberg Dos Martires, and L. De Raedt (2022a). "Inference and Learning with Model Uncertainty in Probabilistic Logic Programs". In: *Proceedings of the 36th AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 10060–10069

T. van Bremen, V. Derkinderen, S. Sharma, S. Roy, and K. S. Meel (2021). "Symmetric Component Caching for Model Counting on Combinatorial Instances". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence*. AAAI Press, pp. 3922–3930

V. Derkinderen, E. Heylen, P. Zuidberg Dos Martires, S. Kolb, and L. De Raedt (2020). "Ordering Variables for Weighted Model Integration". In: *Proceedings*

of the 36th Conference on Uncertainty in Artificial Intelligence, UAI. ed. by R. P. Adams and V. Gogate. Vol. 124. AUAI Press, pp. 879–888

V. Derkinderen and L. De Raedt (2020). "Algebraic Circuits for Decision Theoretic Inference and Learning". In: *Proceedings of the 24th European Conference on Artificial Intelligence, ECAI*. vol. 325. IOS Press, pp. 2569–2576. DOI: 10.3233/FAIA200392

Workshop papers & extended abstracts

V. Derkinderen, P. Zuidberg Dos Martires, S. Kolb, and P. Morettin (2023d). "Top-Down Knowledge Compilation for Counting Modulo Theories". In: *CoRR* abs/2306.04541. accepted at Workshop on Counting and Sampling at SAT 2023. DOI: 10.48550/arXiv.2306.04541

G. Venturato, V. Derkinderen, P. Zuidberg Dos Martires, and L. De Raedt (2022). "Towards Tractable Dynamic Decision Making With Circuits". In: *5th Workshop on Tractable Probabilistic Modeling at UAI 2022*

V. Verreet, V. Derkinderen, P. Zuidberg Dos Martires, and L. De Raedt (2022b). "Inference and Learning with Model Uncertainty in Probabilistic Logic Programs". In: *International Conference on Logic Programming, ICLP (Recently Published Paper Track)*

B. van den Berg, T. van Bremen, V. Derkinderen, A. Kimmig, T. Schrijvers, and L. De Raedt (2021). "From Probabilistic NetKAT to ProbLog: New Algorithms for Inference and Learning in Probabilistic Networks". In: *International Conference on Probabilistic Programming, Extended Abstracts*

P. Zuidberg Dos Martires, V. Derkinderen, R. Manhaeve, W. Meert, A. Kimmig, and L. De Raedt (2019a). "Transforming probabilistic programs into algebraic circuits for inference and learning". In: *Program Transformations for ML Workshop at NeurIPS 2019*

Published data

V. Derkinderen, J. Bekker, and P. Smet (2023b). *Replication Data for: Optimizing Workforce Allocation under Uncertain Activity Duration.* Version V1. URL: https://doi.org/10.48804/YHMU7R

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DECLARATIVE LANGUAGES AND ARTIFICIAL INTELLIGENCE (DTAI)
Celestijnenlaan 200A box 2402
B-3001 Leuven