Generalising Weighted Model Counting

Paulius Dilkas

Doctor of Philosophy

Artificial Intelligence Applications Institute

School of Informatics

University of Edinburgh

2022

Abstract

Given a formula in propositional or (finite-domain) first-order logic and some non-negative weights, weighted model counting (WMC) is a function problem that asks to compute the sum of the weights of the models of the formula. Originally used as a flexible way of performing probabilistic inference on graphical models, WMC has found many applications across artificial intelligence (AI), machine learning, and other domains. Areas of AI that rely on WMC include explainable AI, neural-symbolic AI, probabilistic programming, and statistical relational AI. WMC also has applications in bioinformatics, data mining, natural language processing, prognostics, and robotics.

In this work, we are interested in revisiting the foundations of WMC and considering generalisations of some of the key definitions in the interest of conceptual clarity and practical efficiency. We begin by developing a measure-theoretic perspective on WMC, which suggests a new and more general way of defining the weights of an instance. This new representation can be as succinct as standard WMC but can also expand as needed to represent less-structured probability distributions. We demonstrate the performance benefits of the new format by developing a novel WMC encoding for Bayesian networks. We then show how existing WMC encodings for Bayesian networks can be transformed into this more general format and what conditions ensure that the transformation is correct (i.e., preserves the answer). Combining the strengths of the more flexible representation with the tricks used in existing encodings yields further efficiency improvements in Bayesian network probabilistic inference.

Next, we turn our attention to the first-order setting. Here, we argue that the capabilities of practical model counting algorithms are severely limited by their inability to perform arbitrary recursive computations. To enable arbitrary recursion, we relax the restrictions that typically accompany domain recursion and generalise circuits (used to express a solution to a model counting problem) to graphs that are allowed to have cycles. These improvements enable us to find efficient solutions to counting fundamental structures such as injections and bijections that were previously unsolvable by any available algorithm.

The second strand of this work is concerned with synthetic data generation. Testing algorithms across a wide range of problem instances is crucial to ensure the validity of any claim about one algorithm's superiority over another. However, benchmarks are often limited and fail to reveal differences among the algorithms. First, we show how random instances of probabilistic logic programs (that typically use WMC algorithms for inference) can be generated using constraint programming. We also introduce

a new constraint to control the independence structure of the underlying probability distribution and provide a combinatorial argument for the correctness of the constraint model. This model allows us to, for the first time, experimentally investigate inference algorithms on more than just a handful of instances. Second, we introduce a random model for WMC instances with a parameter that influences primal treewidth—the parameter most commonly used to characterise the difficulty of an instance. We show that the easy-hard-easy pattern with respect to clause density is different for algorithms based on dynamic programming and algebraic decision diagrams than for all other solvers. We also demonstrate that all WMC algorithms scale exponentially with respect to primal treewidth, although at differing rates.

Lay Summary

Given a task to compute a number (e.g., some probability), not all approaches are equally efficient. For example, multiplying two integers is likely to be faster than simulating multiplication via repeated addition. Moreover, this computational task is given to us in some kind of format, and this format determines whether an efficient solution is easy to find. Weighted model counting (WMC) is an approach to computing sums of products efficiently by using logic to describe what needs to be computed. WMC is heavily used in artificial intelligence, machine learning, robotics, and many other fields.

This thesis approaches WMC from two fronts. First, we focus on improving WMC algorithms by generalising some of the key ideas and making them more flexible and powerful. To this end, we revisit the foundations of WMC and develop new input formats that connect logic with numbers (which need to be added and/or multiplied) in a more flexible way. We also improve the ability of a WMC algorithm to solve subproblems by recognizing them as variations of subproblems encountered before. Second, we improve our understanding of the differences among WMC algorithms by generating a variety of problem instances and testing the algorithms on them. For instances written in a more complex format, we find that all WMC algorithms behave extremely similarly, as the computational bottleneck seems to happen before WMC even takes place. On the other hand, for instances written in a simpler format, we show important differences in the performance characteristics of WMC algorithms depending on several key parameters that are used to describe these instances.

Acknowledgements

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Paulius Dilkas*)

Contents

Chapter 1

Introduction

Probabilistic methods are central to artificial intelligence [Russell and Norvig, 2020], data science [Provost and Fawcett, 2013], statistics, and machine learning [Bishop, 2007, Koller and Friedman, 2009]. A fundamental task when working with probabilities is to compute some desired probability from a collection of other known probabilities, i.e., *probabilistic inference*. This thesis is about a particular approach to probabilistic inference (and other similar computational tasks) and its use in performing inference on structured representations of probability distributions such as Bayesian networks and probabilistic logic programs. More generally, we look at different ways of describing arithmetic computations pertinent to probabilistic inference and how algorithms interpret those descriptions, leading to solutions of varying complexity. We begin with an example that showcases how one can compute a probability in various ways depending on how one chooses to reason about it.

Example 1.1. Suppose we have a biased coin that has a probability $0 \leq p \leq 1$ of landing heads. What is the probability that it lands heads *at least once* if we toss it *three times*? More formally, we have three independent Bernoulli random variables X_1, X_2, and X_3 such that $X_i \sim \text{Bernoulli}(p)$ for all i, and we want to compute

$$P \coloneqq \Pr(X_1 = 1 \cup X_2 = 1 \cup X_3 = 1).$$

The conceptually simplest way of calculating the value of P is by adding seven terms, each of which is a product of three factors, i.e., either p or $1-p$. This way, we get

$$P = ppp + pp(1-p) + \cdots + (1-p)(1-p)p. \tag{1.1}$$

One can compute the probability of any event in such a way, although the number

of arithmetic operations in Equation (1.1) scales exponentially with the number of variables.

It is more computationally efficient to reason as follows. If $X_1 = 1$, then all combinations of values of X_2 and X_3 are in the event whose probability we are trying to compute. If $X_1 = 0$, then we can similarly reason about the value of X_3 being immaterial if $X_2 = 1$. This line of reasoning gives us the following way to calculate the probability of interest:

$$P = p \times 1 \times 1 + (1-p)(p \times 1 + (1-p)p). \tag{1.2}$$

Even more efficiently, one can recognize that the only sequence of coin toss results *not* in the event $X_1 = 1 \cup X_2 = 1 \cup X_3 = 1$ is $X_1 = 0$, $X_2 = 0$, and $X_3 = 0$. Thus, the value of P can be computed as

$$P = 1 - (1-p)^3. \tag{1.3}$$

The first of these three approaches hints at the central problem of this thesis. Our goal is to efficiently compute a sum-of-products expression such as the one in Equation (1.1). Of course, the difficulty of this problem partially depends on how each problem instance is formulated, i.e., the input format. The main input format that we concern ourselves with is based on propositional logic—this variation of the problem is known as *weighted model counting* (WMC) [Chavira and Darwiche, 2008]. Equation (1.2) is an example of the kind of efficiency improvements that can be achieved by WMC.

Using logic to encode such computational problems may seem like a curious choice for a reader familiar with probability theory but not logic-based algorithms. However, propositional logic has long played an important role in efficiently solving decision, optimisation, and counting problems [Biere et al., 2009]—WMC is just an extension for function problems. Moreover, WMC has established itself as the state-of-the-art approach to probabilistic inference across many representations such as probabilistic programming languages [Riguzzi et al., 2017] and graphical models [Agrawal et al., 2021].

WMC has been extended in many ways, e.g., to support first-order logic and continuous variables. The former extension is known as *(symmetric) weighted first-order model counting* (WFOMC) [Van den Broeck et al., 2011]. WFOMC algorithms capitalise on mathematical operations besides multiplication and addition and thus can compute P from Example 1.1 as in Equation (1.3). The latter extension is called *weighted model integration* (WMI) [Belle et al., 2015]. In WMI, constraints on continuous variables are described using a fragment of first-order logic known as *linear arithmetic over the rationals* (LRA), i.e., inequalities with addition. The two extensions combined into one

Problem	Sum/Integral (over)	Product (over)
WMC	models of a propositional theory	literals
PBP, SP		arbitrary
WMI	models of a propositional LRA theory	literals
WFOMC	models of a first-order theory	predicates
WFOMI	models of a first-order LRA theory	
SumProd	instantiations of discrete variables	functions
Algebraic path	paths in a graph	edges in a path
Permanent	permutations	elements of a matrix

Table 1.1: An assortment of problems that require one to compute a quantity defined as a sum of products.

are known as *(symmetric) weighted first-order model integration* (WFOMI) [Feldstein and Belle, 2021].

Instead of performing addition and multiplication on numbers, one can do so on elements of an arbitrary (commutative) semiring. This extension of WMC is known as *semiring programming* (SP) [Belle and De Raedt, 2020]. Another important generalisation offered by SP is flexibility in how the numbers that are to be multiplied and added (i.e., the *weights*) can be defined. In this thesis, we do something similar within the constraints of a modern WMC algorithm and call our generalisation *pseudo-Boolean projection* (PBP).

While WMC and its extensions use logic-based input formats, other sum-of-products problems have been studied before. For instance, the *SumProd* problem, which generalises problems such as probabilistic inference in Bayesian networks and propositional model counting, is defined in terms of discrete variables and functions [Bacchus et al., 2009, Dechter, 1999]. In this case, the sum is over all possible instantiations of the variables, and the product is over the values of the functions. Another similar problem is the *algebraic path problem* where the sum is over all paths in a graph from one node to another, and the product is over the weights of the edges in the path [Baras and Theodorakopoulos, 2010]. This problem generalises many graph problems such as shortest and longest path and has many uses in routing and network reliability analysis. Lastly, even famous problems in algebraic complexity theory such as computing the determinant or the permanent of a matrix are examples of this sum-of-products computational paradigm [Bürgisser et al., 1997, Valiant, 1979]. See Table 1.1 for a summary

of all of the discussed problems.

WMC is a rapidly growing area of research. Publications describing novel WMC algorithms continue to appear each year [Dudek et al., 2020b, Korhonen and Järvisalo, 2021]. Furthermore, a competition[1] (as well as a workshop) for model counting and extensions thereof started running annually in 2020 [Fichte et al., 2021]. Given all of this, it is all the more important to

- develop WMC algorithms with good empirical performance,
- understand the comparative strengths and weaknesses of different approaches,
- and optimise the encoding process that transforms problems from the application (e.g., probabilistic inference) domain to a representation accepted by the algorithm.

In this thesis, we contribute to all three of these objectives in a variety of ways.

1.1 Approach, Contributions, and Outline

Our main conceptual tool on this quest is generalisation. While the term *generalisation* can be defined in many ways, we use it to mean that x is a generalisation of y if x can do/express/capture everything that y can, and more. Many important results in science and mathematics are generalisations, e.g., the Lebesgue integral generalises the Riemann integral, and Einstein's general theory of relativity generalises Newton's law of universal gravitation. An example of generalisation closer to home is the emergence of solvers for, e.g., Boolean satisfiability (SAT), constraint programming, integer programming, and linear programming, that can be used to solve many decision and optimisation problems. While designing algorithms for specific problems remains a valuable enterprise, there is indisputable value in having a range of tools with broader applicability.

We establish the following generalisations. In Chapters 3 and 4 we generalise the definition of WMC (to PBP) to take full advantage of the capabilities of recent developments in WMC algorithms. In Chapter 5, we generalise some of the procedures and data structures used by a WFOMC algorithm to make it applicable to a wider range of problem instances. The main idea behind this work is that, in many cases, efficiency improvements can be achieved by developing algorithms that can handle richer data structures. Another important idea in this work, particularly Chapters 6 and 7, is that

[1] `https://mccompetition.org/`

empirical testing of algorithms on a wide range of random instances can help reveal fundamental differences in the behaviour of said algorithms. In what follows, we review the contributions and the structure of this thesis in more detail.

Encoding a probabilistic inference problem as an instance of WMC typically necessitates adding extra literals and clauses. This is partly so because the predominant definition of WMC assigns weights to models based on weights on literals, and this severely restricts what probability distributions can be represented. In Chapter 3, we develop a *measure-theoretic perspective on WMC* and propose a way to encode conditional weights on literals analogously to conditional probabilities. This representation can be as succinct as standard WMC with weights on literals but can also expand as needed to represent probability distributions with less structure. To demonstrate the performance benefits of conditional weights over the addition of extra literals, we develop a *new WMC encoding* for Bayesian networks and adapt a recent WMC algorithm ADDMC [Dudek et al., 2020a] to the new format. Our experiments show that the new encoding significantly improves the performance of the algorithm on most benchmark instances. Chapter 3 is published as:

> P. Dilkas and V. Belle. Weighted model counting with conditional weights for Bayesian networks. In C. P. de Campos, M. H. Maathuis, and E. Quaeghebeur, editors, *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021, Virtual Event, 27-30 July 2021*, volume 161 of *Proceedings of Machine Learning Research*, pages 386–396. AUAI Press, 2021b. URL `https://proceedings.mlr.press/v161/dilkas21a.html`

Chapter 4 builds on Chapter 3 and further considers WMC in its full generality, leading to the definition of PBP. Here we present an *algorithm that transforms WMC instances into PBP instances* while eliminating around 43 % of variables on average across various Bayesian network encodings. Moreover, we identify *sufficient conditions* for such a variable removal to be possible. Our experiments show significant improvement in WMC-based Bayesian network inference. Chapter 4 is published as:

> P. Dilkas and V. Belle. Weighted model counting without parameter variables. In C. Li and F. Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2021a. doi: 10.1007/978-3-030-80223-3_10

In Chapter 5, our attention shifts to another version of WMC, namely, WFOMC as well as its unweighted variant FOMC. Despite being around for more than a decade,

practical (W)FOMC algorithms are still unable to compute functions as simple as a factorial. In this chapter, we argue that the capabilities of FOMC algorithms are severely limited by their inability to express arbitrary recursive computations. To *enable arbitrary recursion*, we relax the restrictions that typically accompany domain recursion and generalise circuits used to express a solution to an FOMC problem to graphs that may contain cycles. To this end, we enhance the most well-established WFOMC algorithm FORCLIFT [Van den Broeck et al., 2011] with new compilation rules and an algorithm to check whether a recursive call is feasible. These improvements allow us to *automatically find efficient solutions* to counting fundamental structures such as injections and bijections.

In Chapters 6 and 7, we transition to the other strand of this work, i.e., random instance generation. Testing algorithms across a wide range of problem instances is crucial to ensure the validity of any claim about one algorithm's superiority over another. However, when it comes to inference algorithms for probabilistic logic programs, experimental evaluations are limited to only a few programs. Existing methods to generate random logic programs are limited to propositional programs and often impose stringent syntactic restrictions. In Chapter 6, we present a *novel approach to generating random logic programs and random probabilistic logic programs* using constraint programming, introducing a *new constraint to control the independence structure of the underlying probability distribution*. We also provide a combinatorial argument for the correctness of the model, show how the model scales with parameter values, and use the model to compare probabilistic inference algorithms across a range of synthetic problems. Our model allows inference algorithm developers to evaluate and compare the algorithms across a wide range of instances, providing a detailed picture of their comparative strengths and weaknesses. Chapter 6 is published as:

> P. Dilkas and V. Belle. Generating random logic programs using constraint programming. In H. Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 828–845. Springer, 2020. doi: 10.1007/978-3-030-58475-7_48

In recent experiments, WMC algorithms are shown to perform similarly overall but with significant differences on specific subsets of benchmarks. A good understanding of the differences in the performance of algorithms requires identifying key characteristics that favour some algorithms over others. In Chapter 7, we introduce a *random model* for WMC instances with a parameter that influences primal treewidth—the parameter most

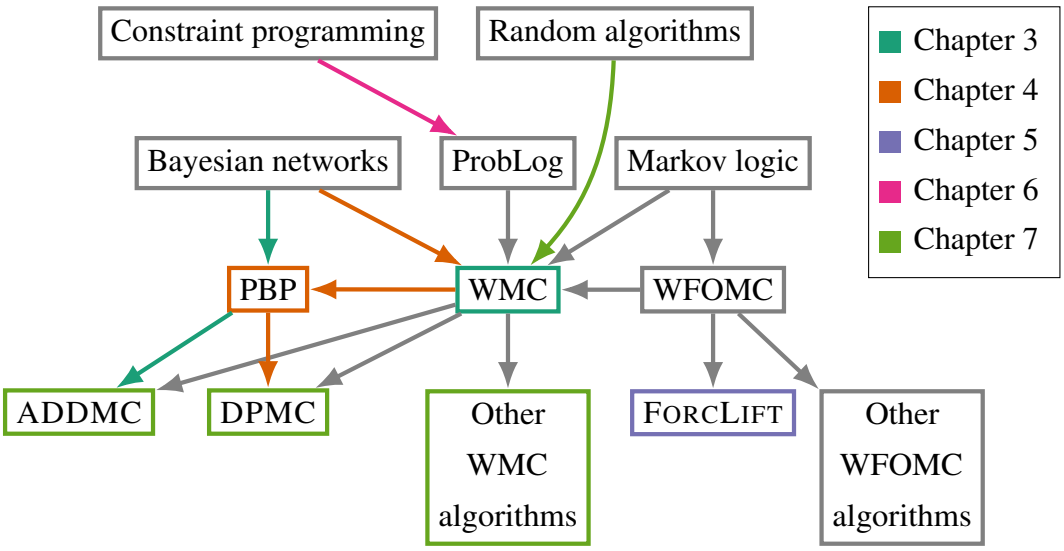


Figure 1.1: Concepts relevant to the thesis. The first row contains two approaches to generating random problem instances. The second row contains some representations of probability distributions. The third row contains encodings, i.e., computational problems that encode probabilistic inference tasks. The last row contains WMC and WFOMC algorithms. Each chapter is assigned a colour that indicates which concepts and interactions between concepts the chapter is about.

commonly used to characterise the difficulty of an instance. We then use this model to experimentally compare the performance of WMC algorithms C2D [Darwiche, 2004], CACHET [Sang et al., 2004], D4 [Lagniez and Marquis, 2017], DPMC [Dudek et al., 2020b], and MINIC2D [Oztok and Darwiche, 2015] on random instances. We show that the *easy-hard-easy pattern is different* for algorithms based on dynamic programming and algebraic decision diagrams (ADDs) than for all other solvers. We also show how all WMC algorithms scale exponentially with respect to primal treewidth and how this scalability varies across algorithms and densities. Finally, we demonstrate how the performance of ADD-based algorithms changes depending on how much determinism or redundancy there is in the numerical values of weights.

We end the introduction with a visual description of the topics covered in this thesis. Figure 1.1 lists some of the key concepts of this work and shows how each chapter of the thesis relates to these concepts and interactions between them. Chapter 2 covers a selection of topics related to this work and refers the reader to suitable literature for further information. Chapter 3 examines the definition of WMC more closely and suggests a way to bypass it, leading to a more succinct encoding of Bayesian network probabilistic inference compatible with the ADDMC algorithm. Then Chapter 4

describes WMC encodings for Bayesian networks, defines PBP, shows how to transform WMC instances to PBP instances, and how the DPMC algorithm benefits from this new format. In Chapter 5, we expand the capabilities of the WFOMC algorithm FORCLIFT to new (previously unsolvable) instances. Chapter 6 describes a constraint model that can generate random (probabilistic) logic programs in the ProbLog [De Raedt et al., 2007] language—a well known use-case of WMC. Chapter 7 develops an algorithm for generating random WMC instances and uses it showcase some important differences in the behaviour of WMC algorithms. Finally, Chapter 8 summarises our results and provides a perspective for potential future work.

Chapter 2

Background

This chapter provides a brief overview of the concepts and topics pertinent to the rest of the thesis. We start in Section 2.1 with a description of propositional logic and the kinds of computational problems that use a logic-based input format or are closely tied to logic in some other way. Then, Section 2.2 introduces two declarative programming paradigms that can be used to describe various computational problems: logic programming and constraint programming. Next, Section 2.3 covers various ways to represent probability distributions. We divide these representations into those based on graphs (i.e., probabilistic graphical models) and those based on text (i.e., probabilistic programming languages). Likewise, Section 2.4 covers various representations of Boolean and pseudo-Boolean functions. These representations (and algorithms that compile into them) are crucial in many WMC and probabilistic inference algorithms. We end the chapter with Section 2.5 which provides an overview of the applications of WMC and its impact on areas such as bioinformatics, natural language processing, and robotics.

2.1 Propositional Logic

In this section, we briefly introduce the fundamentals of propositional logic and describe some logic-based computational problems. We refer the reader to the book by Ben-Ari [2012] for a more detailed introduction to logic and its role in computer science.

An *atomic proposition* (also known as *atom* and *Boolean/logical/propositional variable*) is a variable with two possible (truth) values: `true` and `false`. We usually refer to atoms as *variables*. A *formula* is any well-formed expression that connects variables using the following Boolean/logical operators (and parentheses): negation

($\neg$), disjunction ($\vee$), conjunction ($\wedge$), (material) implication ($\Rightarrow$), and equivalence (i.e., material biconditional) ($\Leftrightarrow$). A *literal* is either a variable or its negation, respectively called *positive* and *negative* literal. A *clause* is a disjunction of literals.[1] A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, and it is in k-CNF if every clause has exactly k literals. Many other normal forms and ways to represent propositional formulas are covered in Section 2.4.

An *interpretation* (also known as a *variable assignment*) of a formula ϕ is a map from the variables of ϕ to the set $\{\texttt{true}, \texttt{false}\}$. A *model* is an interpretation under which ϕ evaluates to `true`. A formula is

satisfiable if it has at least one model,

unsatisfiable (i.e., a *contradiction*) if it has no models, and a

tautology (i.e., *valid*) if all interpretations are models.

We denote tautologies and contradictions as $\top$ and $\bot$, respectively, and often use them interchangeably with the truth values `true` and `false`. Two formulas ϕ and ψ over the same set of variables are *equivalent* (denoted $\phi \equiv \psi$) if they have equal sets of models.

Throughout the thesis, we use set-theoretic notation for many concepts in logic such as clauses and formulas in CNF (e.g., we write $c \in \phi$ to mean that clause c is one of the clauses of formula ϕ). However, this does not automatically mean that no duplicates are allowed—whether or not that is the case is clarified on a case-by-case basis.

Example 2.1. Formula $\phi := (\neg a \vee b) \wedge a$ has two variables a and b, is in CNF, and contains two clauses. The first clause $\neg a \vee b$ has a negative literal $\neg a$ and a positive literal b. Since ϕ has two variables, it also has four interpretations. Interpretation $\{a \mapsto \texttt{true}, b \mapsto \texttt{true}\}$ is a model, so ϕ is satisfiable. An equivalent set-theoretic representation of ϕ is $\{\{\neg a, b\}, \{a\}\}$.

The *primal graph* of a CNF formula is a graph that has a node for every variable, and there is an edge between two variables if they coappear in some clause. The *treewidth* of a graph G measures how similar G is to a tree and is defined as the smallest *width* of any *tree decomposition* of G [Robertson and Seymour, 1984]. The *primal treewidth* of a formula is the treewidth of its primal graph. Primal treewidth (and treewidth more generally) is a parameter frequently used to describe the parameterised complexity of algorithms [Bliem et al., 2017, Downey and Fellows, 2013, Fichte et al., 2020].

[1] The word *clause* has a different meaning in Section 2.2.1 and Chapters 5 and 6.

2.1.1 Logic-Based Computational Problems

We begin with a description of SAT and some of its extensions. Given a propositional formula[2], SAT asks whether the formula is satisfiable. SAT (also known as *propositional/Boolean satisfiability*) is the first problem shown to be NP-complete [Cook, 1971, Levin, 1973]. Motivated by many real-life problems that were found to be reducible to SAT, research in SAT solving produced algorithms that can efficiently tackle large instances despite the exponential worst-case time complexity [Biere et al., 2009].

Instead of satisfying all clauses, one can attempt to find an interpretation that satisfies the maximum number of clauses—this problem is called MaxSAT [Bacchus et al., 2021, Li and Manyà, 2009]. It is an NP-hard optimisation problem that (in its most general form) attaches a (potentially infinite) cost for failing to satisfy each clause and seeks to minimise total cost.

#SAT, or *(propositional) model counting*, asks to count the number of models of a formula [Gomes et al., 2009]. #SAT is the canonical #P-complete problem with many applications in areas such as planning and probabilistic reasoning. #∃SAT, or *projected model counting*, selects a subset of variables called *priority variables* [Aziz et al., 2015]. The task is then to count the number of assignments of values to priority variables that can be extended to models. The extension of #SAT most relevant to our work is called *weighted model counting* (WMC). Given a propositional formula ϕ and a *weight function* w from the literals of ϕ to non-negative real numbers, WMC asks to compute

$$\mathrm{WMC}(\phi) = \sum_{\omega \models \phi} \prod_{\omega \models l} w(l),$$

where the summation is over all models ω of ϕ, and the product is over all literals of ω [Chavira and Darwiche, 2008]. Lastly, both #SAT and WMC have been extended to first-order logic [Van den Broeck et al., 2011]—this is the topic of Chapter 5.

Example 2.2. The model count of the formula in Example 2.1 is equal to one. With a weight function $w := \{a \mapsto 0.7, \neg a \mapsto 0.2, b \mapsto 0.8, \neg b \mapsto 0.7\}$, the WMC of the same formula is $0.7 \times 0.8 = 0.56$.

Example 2.3. With the same weight function w as in Example 2.2, the WMC of formula $a \vee b$ is $w(a)w(b) + w(a)w(\neg b) + w(\neg a)w(b) = 0.7 \times 0.8 + 0.7 \times 0.7 + 0.2 \times 0.8 = 1.21$, and the model count of this formula is 3.

[2]Unless stated otherwise, formulas for SAT and other similar problems are assumed to be in CNF.

WMC has been extended in many ways to support, e.g., continuous variables [Belle et al., 2015], infinite domains [Belle, 2017a], and function symbols [Belle, 2017b]. In particular, the extension to first-order logic, known as *(symmetric) weighted first-order model counting* (WFOMC) [Gogate and Domingos, 2016, Van den Broeck et al., 2011] is the focus of Chapter 5. There is also recent work providing support for both continuous variables and first-order logic [Feldstein and Belle, 2021]. Finally, replacing real numbers with addition and multiplication with an arbitrary commutative semiring allows WMC to subsume a variety of other problems such as most probable explanation, shortest path, and gradient computation [Belle and De Raedt, 2020, Kimmig et al., 2017].

There are a number of other computational problems that similarly use logical or algebraic constructs to encode problems from various domains. First, a propositional formula with prepended quantifiers for all of its variables is known as a *quantified Boolean formula* [Kleine Büning and Bubeck, 2009]. One can then ask whether the formula is true or false. *Satisfiability module theories* considers SAT in the context of a background theory [Barrett et al., 2009]. These theories can describe the properties of integer arithmetic, sets, trees, strings, and many commonly-used abstract data structures. *Pseudo-Boolean* solvers consider decision and optimisation problems that can be expressed as linear inequalities over Boolean variables [Roussel and Manquinho, 2009]. *Integer (linear) programming* instances encode integer optimisation problems under inequality constraints of a certain linear-algebraic form [Wolsey, 2020]. Finally, *constraint programming* is a powerful paradigm for solving combinatorial search and optimisation problems with a much more expressive syntax [Rossi et al., 2006]—we discuss constraint programming in more detail in Section 2.2.2.

2.2 Declarative Programming

In contrast to imperative programming, in a declarative programming language, one describes *what* is to be computed but not *how*. Here we describe two declarative programming paradigms pertinent to our work: logic programming and constraint programming.

2.2.1 Logic Programming

In this subsection, we give a brief introduction to logic programming. Specifically, we focus on Prolog—the most popular logic programming language to date. We do not, however, attempt to cover all (or even most) of the capabilities of Prolog but rather focus on the main concepts and ideas relevant to our work in Chapter 6. Note that different descriptions of logic programming often use different (and mutually inconsistent) terminologies. Here we prioritise names and definitions that are sufficiently general for our needs and reasonably consistent with the terminology used in logic. For more details on logic programming and Prolog, we refer the reader to some of the numerous books on the subject [Bratko, 2012, Nilsson and Maluszynski, 1990].

A *logic program* is a finite sequence[3] of clauses. A *clause* consists of a head and a body. If a clause has an empty body, it is a *fact*, otherwise it is a *rule*. The Prolog syntax for a fact and a rule is `h.` and `h :- b.`, respectively, where `h` is the head and `b` is the body, although we often write `h` $\leftarrow$ `b` instead.

The *head* of a clause is an atom. An *atom* (i.e., atomic formula) has the form $p(t_1,\ldots,t_n)$, where p is a *predicate (symbol)*, and $(t_i)_{i=1}^n$ are terms. Here, $n \in \mathbb{N}_0$ is the *arity* of P. When the arity is equal to zero, the atom is also known as a *propositional variable*. Some built-in predicates such as equality can be written in infix notation and without parentheses, i.e., as $a = b$ instead of $=(a,b)$. A *term* is either a *(logical) variable* (i.e., a string that begins with a capital letter) or a *constant* (i.e., any other string). If an atom contains only constants, it is a *ground* atom.

The *body* of a clause is a formula.[4] A *formula* is any well-formed expression that connects atoms using conjunction, disjunction, and negation (as well as parentheses). Prolog syntax for these operators is different from the standard notation used in logic: we write '`,`' instead of $\wedge$, '`;`' instead of $\vee$, and '`\+`' instead of $\neg$. Just like with the syntax for clauses, in most cases we continue to use logic-based syntax for convenience.

Finally, a *query* is a formula to be evaluated. If the query has no variables, the evaluation returns either `true` or `false`. Otherwise, the logic programming engine tries to replace the variables of the query with constants such that the resulting formula is a logical consequence of the program. If successful, an example of such a mapping is returned; if not, the engine returns `false`.

[3] Although it is common to define logic programs as sets, the order is important for efficiency and can be the difference between finite and infinite running time.

[4] In the literature, it is common to define clause bodies as conjunctions, but here we present a more general definition, given that such a generalisation is widely supported by the relevant software.

Example 2.4. Consider the following logic program.

```
parent(sky, will).
parent(will, zoe).
ancestor(X, Z) :- parent(X, Z); (parent(X, Y), ancestor(Y, Z)).
```

In our alternative logic-based notation, the last clause could also be written as

$$\texttt{ancestor(X, Z)} \leftarrow \texttt{parent(X, Z)} \vee \big(\texttt{parent(X, Y)} \wedge \texttt{ancestor(Y, Z)}\big).$$

This program has three clauses. The first two clauses are facts whereas the last clause is a rule. The program uses two predicates (`parent` and `ancestor`), three constants (`sky`, `will`, and `zoe`), and the last clauses uses three variables (`X`, `Y`, and `Z`). Both predicates are of arity 2.

Clause-by-clause, this program can be interpreted as:

- Sky is a parent of Will.
- Will is a parent of Zoe.
- `X` is an ancestor of `Z` if `X` is a parent of `Z` or there is a `Y` such that `X` is a parent of `Y`, and `Y` is an ancestor of `Z`.

The query `ancestor(sky, zoe)` returns `true` since Sky is a parent of a parent of Zoe, and thus an ancestor. The query `ancestor(X, sky)` returns `false` because we know nothing about the ancestors of Sky. Lastly, the query `ancestor(sky, X)` could return either $\{\,\texttt{X} \mapsto \texttt{will}\,\}$ or $\{\,\texttt{X} \mapsto \texttt{zoe}\,\}$ as both Will and Zoe have Sky as an ancestor.

2.2.2 Constraint Programming

Constraint models are successfully used to tackle search problems in many domains such as bioinformatics, configuration, networks, planning, scheduling, and vehicle routing [Rossi et al., 2006]. Here we briefly describe what a constraint satisfaction problem (CSP) is, how an algorithm might attempt to solve it, and how one can help the algorithm search efficiently.

Definition 2.1. A *CSP* is a triple (X, D, C), where

- $X = (x_i)_{i=1}^n$ is an n-tuple of variables,
- $D = (D_i)_{i=1}^n$ is an n-tuple of (typically, finite) domains such that $x_i \in D_i$,

- and C is a set of constraints.

A *constraint* is a pair (S,R), where $S \subseteq X$ is the *scope* of the constraint, and $R \subseteq \prod_{x_i \in S} D_i$ is a relation specifying allowed combinations of values. Constraints can be specified either *intensionally* (i.e., by describing a formula that must be satisfied) or *extensionally* (i.e., by listing all tuples). A *solution* to the CSP is an n-tuple $(a_i)_{i=1}^n$ such that $a_i \in D_i$ and the relevant a_i's are in the relations of all the constraints in C.

Example 2.5 (n queens)**.** Imagine an $n \times n$ chess board. How can one place n queens on the board so that no two queens threaten each other (i.e., are not on the same column, row, or diagonal)? This is the famous *n queens problem*—a common example in the constraint programming literature. The solution we describe here is adapted from a constraint modelling tutorial [Stuckey et al., 2022].

First, note that each column (i.e, *file*) must have exactly one queen. Let $(q_i)_{i=1}^n$ be variables with domains $q_i \in \{1,\ldots,n\}$, where we use $q_i = j$ to denote that the i^{th} column queen is on row (i.e., *rank*) j. Then the entire problem can be described by the following three constraints.

Constraint 2.1. $\texttt{alldifferent}(\{q_i\}_{i=1}^n)$

Constraint 2.2. $\texttt{alldifferent}(\{q_i + i \mid i = 1,\ldots,n\})$

Constraint 2.3. $\texttt{alldifferent}(\{q_i - i \mid i = 1,\ldots,n\})$

Here, `alldifferent` is a constraint on a set of variables (or 'derivatives' of variables) that constrains them to be all different. Constraint 2.1 requires all queens to occupy different rows, and Constraints 2.2 and 2.3 do the same for both diagonals.

Note that, given one solution to the n-queens problem, we can easily find seven others just by rotating and flipping the board in every possible way (i.e., the symmetry group of a square has order 8). Thus, there is no reason for the constraint solver to find all eight symmetrical solutions independently. Avoiding this kind of excessive effort is the goal of *symmetry breaking* constraints.

While some symmetry breaking constraints can be expressed using variables $(q_i)_{i=1}^n$, others could benefit from a different representation. Specifically, let $\mathbf{B} = (b_{ij})$ be an $n \times n$ matrix, where each $b_{ij} \in \{\texttt{true},\texttt{false}\}$ indicates whether the $(i,j)^{\text{th}}$ square contains a queen. Constraints that connect different representations of the same problem are called *channelling* constraints. In this case, the following constraint is sufficient.

Constraint 2.4 (Channelling)**.** *For all $i,j = 1,\ldots,n$, we have that $b_{ij} \iff (q_i = j)$.*

Finally, the following is an example of a symmetry breaking constraint.

Constraint 2.5 (Symmetry breaking)**.** **B** *is lexicographically smaller than or equal to* $\mathbf{B}^\top$ *(i.e., the transpose of* **B***).*

Perhaps the most canonical way of solving a CSP is by *backtracking search*. At each step, the algorithm selects a variable x_i, a value $v \in D_i$, sets

$$x_i \coloneqq v, \tag{2.1}$$

and continues this process until either all constraints are satisfied or some constraint can no longer be satisfied.

Sometimes making a *decision* (i.e., setting a variable to be equal to a value as in Equation (2.1)) leads to other variable-value combinations becoming evidently impossible. For example, after placing a queen on a1 (i.e., setting $q_1 \coloneqq 1$), Constraint 2.1 tells us that no other queen can be placed on the first row (i.e., $q_i \neq 1$ for all $i = 2, \ldots, n$). Purging such impossible values from domains is the job of *(constraint) propagation* (or *inference*) algorithms. These algorithms are designed separately for each type of constraint and vary in their complexity and efficacy (i.e., how many values they are able to remove).

Another issue that needs to be addressed on a per-constraint basis is: how do we know when a constraint is satisfied? Indeed, if all constraints are already satisfied, then it must be the case that setting all remaining variables to *any* values produces a valid solution. This problem is known as *entailment*. Entailment algorithms take a CSP with a (potentially partial) variable-value assignment and return one out of three possible values:

true if the constraint is already satisfied,

false if it is impossible to satisfy the constraint,

maybe/undefined if neither of the above is seemingly the case.

Backtracking search has important choices to make: which variable should be given a value first? Which value from a domain is most likely to lead to a solution? These questions are answered by *variable* and *value ordering heuristics*, respectively. For example, we can choose a variable with the smallest number of values remaining in its domain—this is known as the *dom*, *smallest domain first*, or *first fail* heuristic. Value ordering heuristics typically consider what the sizes of all domains would be given each

instantiation of the selected variable and choose the value that minimises either their sum or their product [van Beek, 2006]. Both kinds of heuristics can also be random, e.g., a variable or a value can be sampled from a uniform distribution. Random heuristics are typically combined with a *restart strategy* that decides how long the search should continue before assuming that a mistake must have been made and restarting the search [van Beek, 2006].

2.3 Representations of Probability Distributions

Unless specified otherwise, by *probability distribution* we mean a *discrete* probability distribution. Moreover, we are typically only interested in probability distributions with *finite support*.

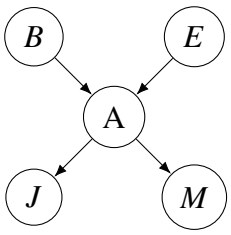
With these restrictions, one could define a probability distribution by listing all combinations of values and assigning a probability to each. However, in most realistic scenarios, the same information could be described more succinctly by taking advantage of concepts such as random variable *independence*, *conditional independence*, and *exchangeability*.

In this section, we describe some of the ways to represent a probability distribution. Section 2.3.1 is about representations based on graphs whereas Section 2.3.2 covers probabilistic programming languages.

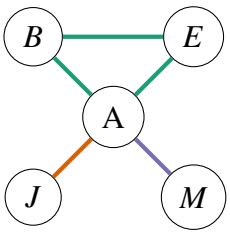
These representations also differ in their ability to reason about groups of random variables. *Propositional* models treat each random variable as a unique individual. In contrast, *relational* models work over sets of individuals and relations among them. See the book by De Raedt et al. [2016] for more detail.

2.3.1 Representations Based on Graphical Models

Perhaps the best-known representations of probability distributions are *probabilistic graphical models* (PGMs), i.e., probabilistic models that use a graph-based representation to compactly encode a probability distribution. These graphs can be either directed (as in the case of Bayesian networks) or undirected (as in the case of Markov networks). This section provides a brief overview of these two networks, although there are also other PGMs such as factor graphs [Loeliger, 2004, De Raedt et al., 2016] as well as graphical models that capture concepts other than probabilities, e.g., constraint networks, cost networks, and influence diagrams [Dechter, 2019]. For more information



(a) a Bayesian network



(b) a Markov network

Figure 2.1: Two PGMs that describe the independence structure of Example 2.6.

on PGMs, see some of the many books on the subject [Dechter, 2019, Koller and Friedman, 2009, Pearl, 1989].

Example 2.6 (A classic example)**.** Suppose you have a burglar alarm in your home. The alarm is likely (but not guaranteed) to be activated when a burglar enters, but it might also be activated by a larger earthquake or even for no apparent reason. (There might even be an earthquake at the time of a burglary!) Furthermore, suppose you have two neighbours: John and Mary. Independently, either of them might call you if they hear your alarm ringing or for some other reason. Let the following (binary) random variables denote the relevant events:

B — a burglar entering your home,

E — an earthquake happening near your home,

A — your burglar alarm activating,

J — John calling you,

M — Mary calling you.

The graph of a *Bayesian network* for this example scenario is in Figure 2.1a. This directed acyclic graph (DAG) tells us that the joint probability distribution can be factored as

$$\Pr(B,E,A,J,M) = \Pr(B) \times \Pr(E) \times \Pr(A \mid B,E) \times \Pr(J \mid A) \times \Pr(M \mid A), \qquad (2.2)$$

i.e., the probability of each random variable is conditioned on its parents in the graph. The factors in Equation (2.2) can be described using *conditional probability tables* (CPTs). CPTs assign a probability to each combination of values that the random variable and its parents can take—see Table 2.1 for an example.

b	e	a	$\Pr(A = a \mid B = b, E = e)$
false	false	false	0.999
		true	0.001
false	true	false	0.71
		true	0.29
true	false	false	0.06
		true	0.94
true	true	false	0.05
		true	0.95

Table 2.1: An example CPT for $\Pr(A \mid B, E)$ from Example 2.6.

Alternatively, the same probability distribution can be represented as an undirected PGM known as a *Markov network* (or *Markov random field*). The graph of such a network for Example 2.6 is in Figure 2.1b. Here, instead of CPTs, *potentials* are the building blocks out of which a probability distribution is constructed. A potential is a function from (some subset of) random variables to non-negative real numbers. Potentials are typically defined on the maximal cliques of the network. The edge sets of the three maximal cliques in Figure 2.1b are highlighted in different colours. Thus, the full probability distribution can be factored as

$$\Pr(B, E, A, J, M) = \frac{1}{Z} \times \psi_1(B, E, A) \times \psi_2(A, J) \times \psi_3(A, M),$$

where ψ_1, ψ_2, and ψ_3 are potentials, and Z is a normalisation constant known as the *partition function*.

What if we wanted to generalise Example 2.6 to support any number of neighbours, all of whom behave identically (i.e., have the same probabilities of calling in all circumstances)? Both Bayesian and Markov networks have been extended for such scenarios: *relational Bayesian networks* [Jaeger, 1997] can compactly describe a probability distribution over a relational structure, and *Markov logic networks* (also known as *Markov logic*) [Richardson and Domingos, 2006] extend Markov networks with support for first-order logic. The field of learning such representations from data is known as *statistical relational learning* [De Raedt et al., 2016]. The next section describes relational representations that are based on programming languages instead of graphical models.

2.3.2 Probabilistic Programming

Augmenting a programming language with probabilities is another common way to compactly represent probability distributions. Logic programming languages, in particular, have been frequently used for this purpose. Examples of probabilistic logic programming languages include the independent choice logic [Poole, 1997, 2008], PRISM [Sato and Kameya, 1997, 2008], BLOG [Milch et al., 2005], NP-BLOG [Carbonetto et al., 2005], ProbLog [De Raedt et al., 2007] and CP-logic [Vennekens et al., 2009]. Functional and imperative programming languages have also seen some use, examples of which include BUGS [Gilks et al., 1994], IBAL [Pfeffer, 2001], Church [Goodman et al., 2008], and Stan [Stan Development Team, 2022]. More information on probabilistic logic programming, probabilistic programming more generally, and statistical relational artificial intelligence can be found in the work of De Raedt et al. [2008], Gordon et al. [2014], and De Raedt et al. [2016], respectively.

Listing 2.1: A ProbLog program that computes $\Pr(B \mid J, M)$ for the scenario described in Example 2.6.

```
neighbour(john).
neighbour(marry).

0.001 :: burglary.
0.002 :: earthquake.

0.95  :: alarm :- burglary, earthquake.
0.94  :: alarm :- burglary, \+ earthquake.
0.29  :: alarm :- \+ burglary, earthquake.
0.001 :: alarm :- \+ burglary, \+ earthquake.

0.8   :: calls(X) :- alarm, neighbour(X).
0.1   :: calls(X) :- \+ alarm, neighbour(X).

evidence(calls(john)).
evidence(calls(mary)).
query(burglary).
```

Listing 2.2: A BLOG program that computes $\Pr(B \mid J, M)$ for the scenario described in Example 2.6.

```
type Neighbour;
distinct Neighbour John, Mary;

random Boolean Burglary   ~ BooleanDistrib(0.001);
random Boolean Earthquake ~ BooleanDistrib(0.002);

random Boolean Alarm ~ case[Burglary, Earthquake] in {
  [false, false] -> BooleanDistrib(0.001),
  [false, true]  -> BooleanDistrib(0.29),
  [true, false]  -> BooleanDistrib(0.94),
  [true, true]   -> BooleanDistrib(0.95)
};

random Boolean Calls(Neighbour n) ~
  if Alarm then BooleanDistrib(0.8)
  else BooleanDistrib(0.1);

obs Calls(John) = true;
obs Calls(Mary) = true;
query Burglary;
```

Listings 2.1 and 2.2 contain two probabilistic programs that encode the information in Example 2.6. In preparation for Chapter 6, let us examine the syntax and semantics of ProbLog a bit more closely. ProbLog clauses are exactly like Prolog clauses (see Section 2.2.1) but with `p ::` prepended, for some probability `p`. Without `::`, the probability associated with the clause is implicitly equal to 1. ProbLog also has keywords `evidence` and `query` that are used to define one or more (potentially conditional) probabilities of interest. Reading off the probabilities from Listing 2.1, we

can, e.g., compute the probability that John calls as

$$\begin{aligned}\Pr(j) &= \Pr(b)\Pr(e)\Pr(a \mid b,e)\Pr(j \mid a)\\ &\quad + \Pr(b)\Pr(e)\Pr(\neg a \mid b,e)\Pr(j \mid \neg a)\\ &\quad + \cdots\\ &\quad + \Pr(\neg b)\Pr(\neg e)\Pr(\neg a \mid \neg b,\neg e)\Pr(j \mid \neg a)\\ &= 0.001 \times 0.002 \times 0.95 \times 0.8 + \cdots\\ &\approx 0.102.\end{aligned}$$

More formally, the probability of a query is the sum of the probabilities of the models of the query (c.f. WMC).

2.4 Knowledge Compilation and Representation

Knowledge compilation is the process of transforming the initial representation of some data (usually based on propositional logic) to a representation that allows one to perform various operations and answer queries of interest in time polynomial in the size of this new representation. Many such representations have been proposed [Darwiche and Marquis, 2002]. Amongst them, those particularly relevant to WMC and probabilistic inference are:

- deterministic decomposable negation normal form (d-DNNF) [Darwiche, 2001b],
- sentential decision diagrams (SDDs) [Darwiche, 2011],
- (ordered) binary decision diagrams (BDDs) [Bryant, 1986],
- and algebraic decision diagrams (ADDs) [Bahar et al., 1997].

The first two items on this list are described in Sections 2.4.1 and 2.4.2, respectively, and the last two are covered in a bit more detail in Section 2.4.3. While knowledge compilation is a process (which is performed by algorithms), here our focus is on the representations themselves.

2.4.1 NNF and d-DNNF

Definition 2.2. A propositional formula ϕ is in *negation normal form* (NNF) if

- the only operators in ϕ are $\neg$, $\vee$, and $\wedge$,

- and $\neg$ is only applied to directly to variables.

Example 2.7. Formula $\neg(C \Rightarrow (\neg A \wedge B))$ can be transformed into NNF as follows:

$$\neg(C \Rightarrow (\neg A \wedge B)) \equiv \neg(\neg C \vee (\neg A \wedge B)) \equiv C \wedge (A \vee \neg B)$$

using the definition of $\Rightarrow$ and De Morgan's laws.

Definition 2.3. The d-DNNF adds decomposability and determinism to the NNF. *Decomposability* requires that, for every conjunction $\bigwedge_{i=1}^{n} \phi_i$, conjuncts ϕ_i and ϕ_j have no variables in common for all $i \neq j$ [Darwiche, 1999, 2001a]. *Determinism* requires that, for every disjunction $\bigvee_{i=1}^{n} \phi_i$, disjuncts ϕ_i and ϕ_j contradict each other (i.e., $\phi_i \wedge \phi_j \equiv \bot$) for all $i \neq j$ [Darwiche, 2001b].

Example 2.8. Formula $(A \vee \neg B) \wedge (A \vee C)$ is neither decomposable nor deterministic. It is not decomposable because $\{A,B\} \cap \{A,C\} = \{A\} \neq \emptyset$. It is not deterministic because, e.g., $A \wedge \neg B \not\equiv \bot$.

Example 2.9. Formula $C \wedge (A \vee \neg B)$ is decomposable but not deterministic. It is decomposable because $\{C\} \cap \{A,B\} = \emptyset$. It is not deterministic because $A \wedge \neg B \not\equiv \bot$.

Example 2.10. Formula $B \wedge C \wedge [\neg B \vee (A \wedge B)]$ is deterministic but not decomposable. It is deterministic because $\neg B \wedge A \wedge B \equiv \bot$. It is not decomposable because $\{B\} \cap \{A,B\} = \{B\} \neq \emptyset$.

Example 2.11. Formula $C \wedge [\neg B \vee (A \wedge B)]$ is decomposable and deterministic. It is decomposable because $\{C\} \cap \{A,B\} = \emptyset$, and $\{A\} \cap \{B\} = \emptyset$. It is deterministic because $\neg B \wedge A \wedge B \equiv \bot$.

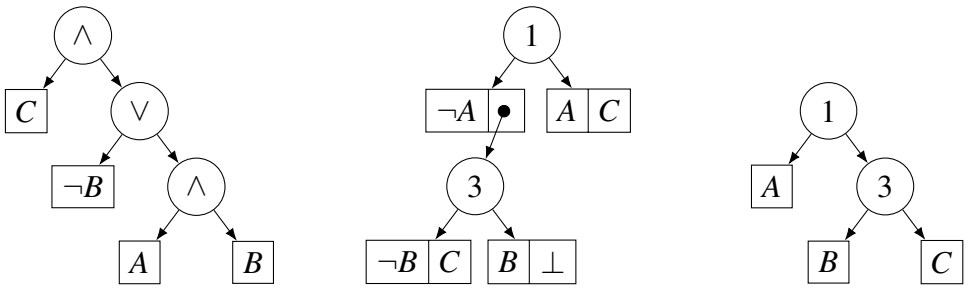
Note that the formulas in Examples 2.9 and 2.11 are equivalent, and the latter is also pictured in Figure 2.2a.

2.4.2 SDDs

To define SDDs, we first need to define vtrees.

Definition 2.4 (Pipatsrisawat and Darwiche [2008])**.** A *vtree* for a set of variables X is a full binary tree T with a bijection between X and the leaves of T.

Let $\langle\cdot\rangle$ denote the function that maps an SDD to the propositional formula that it represents.



(a) d-DNNF (b) SDD (c) vtree

Figure 2.2: A d-DNNF and an SDD representation of $C \wedge (A \vee \neg B)$, together with the corresponding vtree. The numbers 1 and 3 come from the in-order traversal of the vtree and visually connect subtrees of both the SDD and the vtree.

Definition 2.5 (Darwiche [2011])**.** Let V be a vtree for a set of variables X. Then S is an *SDD* that respects V if one of the following is true:

- $S = \bot$ ($\langle \bot \rangle := \bot$);
- $S = \top$ ($\langle \top \rangle := \top$);
- $S = x$, or $S = \neg x$, where $x \in X$ is the variable bijectively associated with the *only* node of V ($\langle x \rangle := x$, and $\langle \neg x \rangle := \neg x$);
- $S = \{ (p_i, s_i) \mid i = 1, \ldots, n \}$ for some $n \geq 1$, where *primes* $\{ p_i \}_{i=1}^n$ and *subs* $\{ s_i \}_{i=1}^n$ are SDDs such that:
  - V has more than one node,
  - each p_i respects the left subtree of V,
  - each s_i respects the right subtree fo V.
  - the primes form a *partition*, i.e.:
    - $\langle p_i \rangle \not\equiv \bot$ for all $i = 1, \ldots, n$ (i.e., the primes are *consistent*),
    - $\langle p_i \rangle \wedge \langle p_j \rangle \equiv \bot$ for all $i \neq j$ (i.e., the primes are *mutually exclusive*),
    - and $\bigvee_{i=1}^n \langle p_i \rangle \equiv \top$

  (then $\langle S \rangle := \bigvee_{i=1}^n \langle p_i \rangle \wedge \langle s_i \rangle$).

Example 2.12. Let $S = \{ (A, C), (\neg A, \{ (\neg B, C), (B, \bot) \}) \}$. Then S (as pictured in Figure 2.2b) is an SDD representation of $C \wedge (A \vee \neg B)$ that respects the vtree in Figure 2.2c.

Indeed,

$$\begin{aligned}\langle S\rangle &= (A \wedge C) \vee (\neg A \wedge [(\neg B \wedge C) \vee (B \wedge \bot)]) \\ &\equiv (A \wedge C) \vee (\neg A \wedge \neg B \wedge C) \\ &\equiv C \wedge (A \vee [\neg A \wedge \neg B]) \\ &\equiv C \wedge ([A \vee \neg A] \wedge [A \vee \neg B]) \\ &\equiv C \wedge (\top \wedge [A \vee \neg B]) \\ &\equiv C \wedge (A \vee \neg B).\end{aligned}$$

2.4.3 Other Decision Diagrams

NNF and d-DNNF are *normal forms* of propositional formulas. This means that—even though they can be represented diagrammatically as trees or circuits—a formula in (d-D)NNF (just like in CNF) is still a formula. The same applies to SDDs: while we defined them as nested sets and tuples, $\langle S\rangle$ of an SDD S is just a propositional formula with a certain structure. In contrast, the two representations we describe here—BDDs and ADDs—are defined as DAGs rather than normal forms.

Both BDDs and ADDs represent functions. BDDs represent *Boolean functions*, i.e., maps of the form $\{0,1\}^n \to \{0,1\}$ for some $n \geq 0$, where $\{0,1\}$ can be replaced by any other two-element set. A propositional formula is simply a particular representation of a Boolean function. ADDs, on the other hand, represent *pseudo-Boolean functions*, i.e., maps of the form $\{0,1\}^n \to \mathbb{R}$. Equivalently, we can write 2^X for $\{0,1\}^n$, where X is any n-element set, and 2^X denotes its powerset. The elements of X are then called *variables*. With this characterisation, pseudo-Boolean functions are also known as *set functions*.

Pseudo-Boolean functions, most commonly represented as ADDs (although a tensor-based approach has also been suggested [Dudek et al., 2019, 2020b]), have seen extensive use in value iteration for Markov decision processes [Hoey et al., 1999], both exact and approximate Bayesian network inference [Chavira and Darwiche, 2007, Gogate and Domingos, 2011], and sum-product network to Bayesian network conversion [Zhao et al., 2015]. ADDs have been extended to compactly represent additive and multiplicative patterns in the image of the function [Sanner and McAllester, 2005] and to support first-order logic [Sanner and Boutilier, 2009] and continuous variables [Sanner et al., 2011]. This last extension was also applied to weighted model integration [Belle et al., 2015, Kolb et al., 2018].

Informally, both BDDs and ADDs are like decision trees (whose leaves correspond to elements of the image of the function that is being represented) but compressed into a DAG. Below we define ADDs—the definition of BDDs simply requires replacing $\mathbb{R}$ with $\{0,1\}$. Our definition is partially based on the original definition by Bahar et al. [1997] as well as recent work by Dudek et al. [2020b] but states some details more explicitly. The definition can also be generalised to use any set instead of $\mathbb{R}$ and to represent several functions instead of just one.

Notation. Let G be a directed graph. Then $\mathcal{V}(G)$ denotes the set of nodes of G, $\mathcal{E}(G)$ denotes the set of edges of G, and $\mathcal{L}(G)$ denotes the set of sinks of G.

Definition 2.6. Given a set of variables X and a variable ordering represented as an injection $\sigma\colon X \to \mathbb{N}^+$, an *ADD* is a tuple $(G, r, \rho, \chi, \varepsilon)$ where:

- G is a rooted DAG with root $r \in \mathcal{V}(G)$ (i.e., there is a directed path from r to any other node),
- $\rho\colon \mathcal{L}(G) \to \mathbb{R}$ labels sinks with real numbers,
- $\chi\colon \mathcal{V}(G) \setminus \mathcal{L}(G) \to X$ labels other nodes with variable names,
- and $\varepsilon\colon \mathcal{E}(G) \to \{0,1\}$ labels edges.

Moreover, the following properties must be satisfied.

- Every node has outdegree either zero or two. In the latter case, the two outgoing edges $e, f \in \mathcal{E}(G)$ are such that $\varepsilon(e) \neq \varepsilon(f)$. If $e = (v,u)$, and $f = (v,w)$ for some $u, v, w \in \mathcal{V}(G)$ are such that $\varepsilon(e) = 1$, and $\varepsilon(f) = 0$, then u is the *positive successor* of v, and w is the *negative successor* of v.
- For every directed path with node sequence $v_1, v_2, \ldots, v_n$ such that $v_i \notin \mathcal{L}(G)$ for all i, we have that $\sigma(\chi(v_i)) < \sigma(\chi(v_{i+1}))$ for all $i = 1, 2, \ldots, n-1$.

We say that an ADD *has* variable $x \in X$ if there is a node $v \in \mathcal{V}(G) \setminus \mathcal{L}(G)$ such that $\chi(v) = x$.

To view an ADD as a pseudo-Boolean function, given an interpretation $\iota\colon X \to \{\texttt{true}, \texttt{false}\}$, start at the root and follow the outgoing edges until you reach a sink. If at node $v \in \mathcal{V}(G) \setminus \mathcal{L}(G)$ we have that $\iota(\chi(v)) = \texttt{true}$, then follow the outgoing edge e with $\varepsilon(e) = 1$, otherwise follow the outgoing edge f with $\varepsilon(f) = 0$. Once you reach a sink $l \in \mathcal{L}(G)$, then $\rho(l)$ is the value of the represented function at ι (where ι is interpreted as a subset of X).

(a) BDD (b) ADD

Figure 2.3: Example BDD (for the formula $C \wedge (A \vee \neg B)$) and ADD (for the function f in Example 2.13). An edge e is dashed if $\varepsilon(e) = 0$ and solid otherwise, where ε is as in Definition 2.6.

Example 2.13. Let $f\colon 2^{\{x,y\}} \to \mathbb{R}$ be a pseudo-Boolean function defined as $f(\emptyset) = f(\{x\}) = f(\{x,y\}) = 0.1$, and $f(\{y\}) = 0.3$ and $\sigma\colon \{x,y\} \to \mathbb{N}^+$ be the variable ordering function defined as $\sigma(x) = 1$, and $\sigma(y) = 2$. Then the *canonical* ADD for f under σ is pictured in Figure 2.3b[5] and can be formally defined as $(G, a, \rho, \chi, \varepsilon)$, where:

- $\mathcal{V}(G) = \{a,b,c,d\}$,
- $\mathcal{E}(G) = \{(a,b),(a,c),(b,c),(b,d)\}$,
- $\rho(c) = 0.1$, $\rho(d) = 0.3$,
- $\chi(a) = A$, $\chi(b) = B$,
- $\varepsilon((a,c)) = \varepsilon((b,d)) = 1$, and $\varepsilon((a,b)) = \varepsilon((b,c)) = 0$.

We end our discussion of BDDs and ADDs by describing some of their properties as well as operations on them.

Fact 2.1 (Bahar et al. [1997])**.** *Let X be a set of variables and $\sigma\colon X \to \mathbb{N}^+$ be an ordering function. For any subset of variables $Y \subseteq X$ and pseudo-Boolean function $f\colon 2^Y \to \mathbb{R}$, there is a unique (up to isomorphism)* canonical *ADD for f. Any ADD for f can be* reduced *to its canonical form in time linear in the number of nodes.*

Fact 2.2. *The canonical ADD for a pseudo-Boolean function $2^X \to \mathbb{R}$ has at most $2^{|X|+1}$ nodes. This upper bound is achieved when the function is injective.*

The operations pertinent to our needs are listed in Table 2.2: reduction of an ADD to its canonical form, addition/multiplication as well as scalar addition/multiplication,

[5] Also see Figure 2.3a for a BDD equivalent to the d-DNNF and the SDD in Figure 2.2.

Operation	Definition	Complexity
reduce f		$O(n)$
$r+f$, rf	$r+f\colon 2^X \to \mathbb{R}$, $(r+f)(Z) \coloneqq r+f(Z)$	$O(n)$
$f+g$, fg	$f+g\colon 2^{X\cup Y} \to \mathbb{R}$, $(f+g)(Z) \coloneqq f(Z)+g(Z)$	$O(mn)$
$f\|_{x=0}$	$f\|_{x=0}\colon 2^{X\setminus\{x\}} \to \mathbb{R}$, $f\|_{x=0}(Z) \coloneqq f(Z)$	$O(n)$
$f\|_{x=1}$	$f\|_{x=1}\colon 2^X \to \mathbb{R}$, $f\|_{x=1}(Z) \coloneqq f(Z\cup\{x\})$	$O(n)$
$\exists_x f$	$\exists_x f\colon 2^{X\setminus\{x\}} \to \mathbb{R}$, $\exists_x f(Z) \coloneqq f\|_{x=0}(Z)+f\|_{x=1}(Z)$	$O(n^2)$

Table 2.2: Operations on ADDs, their definitions, and the time complexity of the best-known algorithm for performing each operation. Let $f\colon 2^X \to \mathbb{R}$ and $g\colon 2^Y \to \mathbb{R}$ be pseudo-Boolean functions represented by ADDs with n and m nodes respectively, and let $r \in \mathbb{R}$, and $x \in X$.

two types of *restrictions*, and *projection*. For a more detailed description, we refer the reader to previous work [Bahar et al., 1997, Dudek et al., 2020a]. The linear reduction algorithm is by Somenzi [2015], the algorithms for most other operations are by Bryant [1986], and the complexity of projection comes as a corollary of the complexities of all other operations.

When referring to ADDs and their use in algorithms, we make a few simplifying assumptions. First, we assume that projection has the lowest precedence and extend the definition to allow for sets of variables. For any $W = \{w_i\}_{i=1}^k \subseteq X$, let $\exists_W f\colon 2^{X\setminus W} \to \mathbb{R}$ be defined as $\exists_W f(Z) \coloneqq \exists_{w_1}\exists_{w_2}\cdots\exists_{w_k} f(Z)$ for all $Z \subseteq X \setminus W$, where the order of w_i's is immaterial. Second, we assume that after every operation on ADDs, the resulting ADD is reduced to its canonical form and write 'the ADD for function f' to mean 'the *canonical* ADD for f'. Third, while ADDs and the functions they represent could change throughout the execution of an algorithm, we consider the set of all relevant variables X and the ordering function $\sigma\colon X \to \mathbb{N}^+$ to be fixed.

2.5 Applications of WMC

WMC is heavily used in machine learning, where recent applications of WMC include explainability [Van den Broeck et al., 2021] and computing the loss function of a neural-symbolic system [Tsamoura et al., 2021, Xu et al., 2018]. The most well-researched application of WMC, however, is inference for PGMs such as Bayesian networks

[Bart et al., 2016, Chavira and Darwiche, 2005, 2006, Darwiche, 2002, Sang et al., 2005a]. Originally, this approach was motivated by *context-specific independence*, i.e., the independence of random variables that is created by conditioning the probability distribution [Boutilier et al., 1996]. Deconstructing a graphical model to a propositional formula makes such independencies easier to exploit [Darwiche, 2002]. Moreover, WMC is used for inference in probabilistic programming languages such as ProbLog [Fierens et al., 2015, Vlasselaer et al., 2016] and Dice [Holtzen et al., 2020b].

Indeed, WMC and WFOMC play an important role in inference for statistical relational models such as ProbLog and Markov logic [De Raedt et al., 2016, Van den Broeck, 2011]. Such models are often used in natural language processing for tasks such as knowledge/information extraction [Bunescu and Mooney, 2007, Poon and Vanderwende, 2010]. In natural language processing, statistical relational models have been used to annotate articles [Verbeke et al., 2012], learn facts about the world from reading websites [Carlson et al., 2010], and solve simple probability problems described in a natural language [Dries et al., 2017]. Similarly, they have been applied to stream mining [Chandra et al., 2014], predicting criminal activity [Delaney et al., 2010], and predicting how soon a component or a machine will have to be replaced [Vlasselaer and Meert, 2012]. In robotics, statistical relational models have been used to learn object affordances [Moldovan et al., 2011, 2012, Moldovan and De Raedt, 2014] and as an expressive knowledge representation system for robot control [Jain et al., 2009]. Finally, applications in bioinformatics include the analysis of breast cancer [Côrte-Real et al., 2017, Nassif et al., 2013], genetic [Sakhanenko and Galas, 2012], and molecular profiling [De Maeyer et al., 2013] data.

Although it is hard to anticipate how the contributions of this work might affect all of these applications, efficiency improvements can enable new applications for which current WMC is not fast enough. This is especially the case for WFOMC, where said improvements usually bring classes of instances from exponential to polynomial time. At the very least, WMC-based approaches should be able to handle bigger and more complex data sets regardless of their origin.

Chapter 3

WMC with Conditional Weights for Bayesian Networks

3.1 Introduction

WMC, the way we defined it in Section 2.1.1, assigns weights to models based on weights on literals, i.e., the weight of a model is the product of the weights of all literals in it. This simplification is motivated by the fact that the number of models scales exponentially with the number of atoms, so listing the weight of every model is intractable. However, this also severely restricts what probability distributions can be represented. A common way to overcome this limitation is by adding more literals. While we show that this is always possible, we demonstrate that it can be significantly more efficient to encode weights in a more flexible format instead.

After briefly reviewing the background in Section 3.2, in Section 3.3 we describe three equivalent perspectives on the subject based on logic, set theory, and Boolean algebras. Furthermore, we describe the space of functions on Boolean algebras and various operations on those functions. Section 3.4 introduces WMC as the problem of computing the value of a measure on a Boolean algebra. We show that not all measures can be represented using literal-based WMC, but all Boolean algebras can be extended to make any measure representable in such a manner.

This new perspective allows us to not only encode any discrete probability distribution but also improve inference speed. In Section 3.5 we demonstrate this by developing a new WMC encoding for Bayesian networks that uses *conditional weights* on literals (in the spirit of conditional probabilities) that have literal-based WMC as a special case. We prove the correctness of the encoding and show how a WMC solver

ADDMC [Dudek et al., 2020a] can be adapted to the new format. ADDMC is a recently-proposed algorithm for WMC based on manipulating functions on Boolean algebras using an efficient representation for such functions known as algebraic decision diagrams (ADDs) [Bahar et al., 1997]. Our experiments in Section 3.6 focus on improving the performance of ADDMC on instances that originate from Bayesian networks. We show how our new encoding improves inference on the vast majority of benchmark instances, often by one or two orders of magnitude. We explain the performance benefits by showing how our encoding has asymptotically fewer variables and ADDs.

3.2 Related Work

Performing inference on Bayesian networks by encoding them into instances of WMC is a well-established idea with a history of more than twenty years. Five encodings have been proposed so far (we will identify them based on the initials of authors as well as publications years): `d02` [Darwiche, 2002], `sbk05` [Sang et al., 2005a], `cd05` [Chavira and Darwiche, 2005], `cd06` [Chavira and Darwiche, 2006], and `bklm16` [Bart et al., 2016][1]. Below we summarise the observed performance differences among them.

Sang et al. [2005a] claim that `sbk05` is a smaller encoding than `d02` with respect to both the number of clauses and the number of variables but provide no experimental comparison. Chavira and Darwiche [2005] compare `cd05` with `d02` by measuring the time it takes to compile either encoding into an arithmetic circuit. They show that `cd05` always compiles faster and results in a smaller arithmetic circuit (as measured by the number of edges). In their subsequent paper, the same authors perform two sets of experiments (that are relevant to this summary) [Chavira and Darwiche, 2006]. First, they compile `cd05` and `cd06` encodings into d-DNNF (i.e., deterministic decomposable negation normal form [Darwiche, 2001b]), measuring both compilation time and numbers of edges in the d-DNNF diagram. The results are mostly in favour of `cd06`. Second, they compare the inference time of `sbk05` run with CACHET [Sang et al., 2004] with the compile times of `cd05` and `cd06`, but only on five (types of) instances. In these experiments, `cd06` is always faster than `cd05`, while the comparison with `sbk05` is mixed. The performance difference between `sbk05` and `cd05` is even harder to judge: `sbk05` is better on three out of five instances and worse on the remaining two. Finally,

[1] Vomlel and Tichavský [2013] also propose an encoding, but only for networks of a particular bipartite structure and without any evaluation.

Bart et al. [2016] introduce `bklm16` and show that it has both fewer variables and fewer clauses than `cd06`. Their experiments show `bklm16` to be superior to `cd06` with respect to both compilation time and encoding size when both are compiled using C2D[2] [Darwiche, 2004] but inferior to `cd06` when `cd06` is compiled using ACE[3] (which still uses C2D but considers the structure of the Bayesian network along with its encoding). Our experiments in Section 3.6 confirm some of the findings outlined in this section while also showing that the performance of each encoding depends on the WMC algorithm in use, and smaller encodings are not necessarily faster.

3.3 Boolean Algebras, Power Sets, and Propositional Logic

In this section, we give a brief introduction to two alternative ways to think about logical constructs such as models and formulas. Let us consider a simple example of a propositional logic $\mathcal{L}$ with only two atoms a and b, and let $U = \{a, b\}$. Then 2^U, the power set of U, is the set of all models of $\mathcal{L}$, and 2^{2^U} is the set of all formulas. These sets can also be represented as Boolean algebras (e.g., using the syntax $(2^{2^U}, \wedge, \vee, \neg, \bot, \top)$) with a partial order $\leq$ that corresponds to set inclusion $\subseteq$—see Table 3.1 for examples of how various elements can be represented in both notations. Most importantly, note that the word *atom* has completely different meanings in logic and Boolean algebras. An atom in $\mathcal{L}$ is an atomic formula, i.e., an element of U, whereas an atom in a Boolean algebra is (in set-theoretic terms) a singleton set. For instance, an atom in 2^{2^U} corresponds to a model of $\mathcal{L}$, i.e., an element of 2^U. Unless referring specifically to a logic, we will use the algebraic definition of an atom and refer to logical atoms as *variables*. In the rest of the paper, for any set U, we will use set-theoretic notation for 2^U and Boolean-algebraic notation for 2^{2^U}, except for (Boolean) atoms in 2^{2^U} that are denoted as $\{x\}$ for some model $x \in 2^U$.

3.3.1 Functions on Boolean Algebras

We also consider the space of all functions from any Boolean algebra to $\mathbb{R}_{\geq 0}$ together with some operations on those functions. They will be instrumental in defining WMC as a measure in Section 3.4 and can be efficiently represented using ADDs. Furthermore,

[2] `http://reasoning.cs.ucla.edu/c2d/`
[3] `http://reasoning.cs.ucla.edu/ace/`

Name in logic	Boolean-algebraic notation	Set-theoretic notation
Atoms (elements of U)	a, b	a, b
Models (elements of 2^U)	$\neg a \wedge \neg b, a \wedge \neg b, \neg a \wedge b, a \wedge b$	$\emptyset, \{a\}, \{b\}, \{a,b\}$
Formulas (elements of 2^{2^U})	$\top$	$\{\emptyset, \{a\}, \{b\}, \{a,b\}\}$
	$\neg a \vee \neg b, a \Rightarrow b$	$\{\emptyset, \{a\}, \{b\}\}, \{\emptyset, \{b\}, \{a,b\}\}$
	$b \Rightarrow a, a \vee b$	$\{\emptyset, \{a\}, \{a,b\}\}, \{\{a\}, \{b\}, \{a,b\}\}$
	$\neg b, \neg a, a \Leftrightarrow b$	$\{\emptyset, \{a\}\}, \{\emptyset, \{b\}\}, \{\emptyset, \{a,b\}\}$
	$(a \wedge \neg b) \vee (b \wedge \neg a), a, b$	$\{\{a\}, \{b\}\}, \{\{a\}, \{a,b\}\}, \{\{b\}, \{a,b\}\}$
	$\neg a \wedge \neg b, a \wedge \neg b, \neg a \wedge b, a \wedge b$	$\{\emptyset\}, \{\{a\}\}, \{\{b\}\}, \{\{a,b\}\}$
	$\bot$	$\emptyset$

Table 3.1: Notation for a logic with two atoms. The elements in both columns are listed in the same order.

all of the operations are supported by CUDD [Somenzi, 2015]—a package used by ADDMC for ADD manipulation [Dudek et al., 2020a]. The definitions of multiplication and projection are as defined by Dudek et al. [2020a], while others are new.

Definition 3.1 (Operations on functions)**.** Let $\alpha\colon 2^X \to \mathbb{R}_{\geq 0}$ and $\beta\colon 2^Y \to \mathbb{R}_{\geq 0}$ be functions, $p \in \mathbb{R}_{\geq 0}$, and $x \in X$. We define the following operations:

Addition: $\alpha+\beta\colon 2^{X\cup Y} \to \mathbb{R}_{\geq 0}$ is such that $(\alpha+\beta)(T) = \alpha(T\cap X)+\beta(T\cap Y)$ for all $T \in 2^{X\cup Y}$.

Multiplication: $\alpha\cdot\beta\colon 2^{X\cup Y} \to \mathbb{R}_{\geq 0}$ is such that $(\alpha\cdot\beta)(T) = \alpha(T\cap X)\cdot\beta(T\cap Y)$ for all $T \in 2^{X\cup Y}$.

Scalar multiplication: $p\alpha\colon 2^X \to \mathbb{R}_{\geq 0}$ is such that $(p\alpha)(T) = p\cdot\alpha(T)$ for all $T \in 2^X$.

Complement: $\overline{\alpha}\colon 2^X \to \mathbb{R}_{\geq 0}$ is such that $\overline{\alpha}(T) = 1-\alpha(T)$ for all $T \in 2^X$.

Projection: $\exists_x\alpha\colon 2^{X\setminus\{x\}} \to \mathbb{R}_{\geq 0}$ is such that $(\exists_x\alpha)(T) = \alpha(T)+\alpha(T\cup\{x\})$ for all $T \in 2^{X\setminus\{x\}}$. For any $Z = \{z_1,\ldots,z_n\} \subseteq X$, we write $\exists_Z$ to mean $\exists_{z_1}\ldots\exists_{z_n}$.

In summary, addition, multiplication, and scalar multiplication are defined pointwise, while complement and projection interact with the algebraic structure of the domains 2^X and 2^Y. Specifically, note that both addition and multiplication are both associative and commutative. We end the discussion on function spaces by defining several special functions: unit $1\colon 2^{\emptyset} \to \mathbb{R}_{\geq 0}$ defined as $1(\emptyset) = 1$, zero $0\colon 2^{\emptyset} \to \mathbb{R}_{\geq 0}$ defined as $0(\emptyset) = 0$, and function $[a]\colon 2^{\{a\}} \to \mathbb{R}_{\geq 0}$ defined as $[a](\emptyset) = 0$, $[a](\{a\}) = 1$ for any a. Henceforth, for any function $\alpha\colon 2^X \to \mathbb{R}_{\geq 0}$ and any set T, we will write $\alpha(T)$ to mean $\alpha(T\cap X)$.

3.4 WMC as a Measure on a Boolean Algebra

In this section, we introduce an alternative definition of WMC and demonstrate how it relates to the standard one. Let U be a set. A *measure* is a function $\mu\colon 2^{2^U} \to \mathbb{R}_{\geq 0}$ such that $\mu(\bot) = 0$, and $\mu(a\vee b) = \mu(a)+\mu(b)$ for all $a,b \in 2^{2^U}$ whenever $a\wedge b = \bot$ [Gaifman, 1964, Jech, 1997]. A *weight function* is a function $\nu\colon 2^U \to \mathbb{R}_{\geq 0}$. A weight function is *factored* if $\nu = \prod_{x\in U}\nu_x$ for some functions $\nu_x\colon 2^{\{x\}} \to \mathbb{R}_{\geq 0}$, $x \in U$. We say that a weight function $\nu\colon 2^U \to \mathbb{R}_{\geq 0}$ *induces* a measure $\mu_\nu\colon 2^{2^U} \to \mathbb{R}_{\geq 0}$ if $\mu_\nu(x) = \sum_{\{u\}\leq x}\nu(u)$.

Theorem 3.1. *The function μ_ν is a measure.*

Proof. Note that $\mu_\nu(\bot) = 0$ since there are no atoms below $\bot$. Let $a, b \in 2^{2^U}$ be such that $a \wedge b = \bot$. By elementary properties of Boolean algebras, all atoms below $a \vee b$ are either below a or below b. Moreover, none of them can be below both a and b because then they would have to be below $a \wedge b = \bot$. Thus

$$\mu_\nu(a \vee b) = \sum_{\{u\} \leq a \vee b} \nu(u) = \sum_{\{u\} \leq a} \nu(u) + \sum_{\{u\} \leq b} \nu(u) = \mu_\nu(a) + \mu_\nu(b)$$

as required. □

Finally, a measure $\mu\colon 2^{2^U} \to \mathbb{R}_{\geq 0}$ is *factorable* if there exists a factored weight function $\nu\colon 2^U \to \mathbb{R}_{\geq 0}$ that induces μ. In this formulation, WMC corresponds to the process of calculating the value of $\mu_\nu(x)$ for some $x \in 2^{2^U}$ with a given definition of ν.

Example 3.1 (Relation to the classical (logic-based) view of WMC)**.** Let a and b be variables and $w\colon \{a, b, \neg a, \neg b\} \to \mathbb{R}_{\geq 0}$ be a weight function defined as $w(a) = 0.3$, $w(\neg a) = 0.7$, $w(b) = 0.2$, $w(\neg b) = 0.8$. Then formula a has two models: $\{a, b\}$ and $\{a, \neg b\}$, and its WMC is

$$\mathrm{WMC}(a) = \sum_{\omega \models a} \prod_{\omega \models l} w(l) = w(a)w(b) + w(a)w(\neg b) = 0.3. \tag{3.1}$$

Alternatively, we can define $\nu_a\colon 2^{\{a\}} \to \mathbb{R}_{\geq 0}$ as $\nu_a(\{a\}) = 0.3$, $\nu_a(\emptyset) = 0.7$ and $\nu_b\colon 2^{\{b\}} \to \mathbb{R}_{\geq 0}$ as $\nu_b(\{b\}) = 0.2$, $\nu_b(\emptyset) = 0.8$. Let μ be the measure on 2^{2^U} induced by $\nu = \nu_a \cdot \nu_b$. Then, equivalently to Equation (3.1), we can write

$$\begin{aligned} \mu(a) &= \nu(\{a, b\}) + \nu(\{a\}) \\ &= \nu_a(\{a\})\nu_b(\{b\}) + \nu_a(\{a\})\nu_b(\emptyset) = 0.3. \end{aligned}$$

Thus, one can equivalently think of WMC as summing over models of a theory or over atoms below an element of a Boolean algebra.

3.4.1 Not All Measures Are Factorable

Using this new definition of WMC, we can show that WMC with weights defined on literals is only able to capture a subset of all possible measures on a Boolean algebra. This can be demonstrated with a simple example.

Example 3.2. Let $U = \{a, b\}$ be a set of atoms and $\mu\colon 2^{2^U} \to \mathbb{R}_{\geq 0}$ a measure defined as $\mu(a \wedge b) = 0.72$, $\mu(a \wedge \neg b) = 0.18$, $\mu(\neg a \wedge b) = 0.07$, $\mu(\neg a \wedge \neg b) = 0.03$.[4] If μ could

[4]The value of μ on any other element of 2^{2^U} can be deduced from the definition of a measure.

be represented using literal-weight (factored) WMC, we would have to find two weight functions $\nu_a \colon 2^{\{a\}} \to \mathbb{R}_{\geq 0}$ and $\nu_b \colon 2^{\{b\}} \to \mathbb{R}_{\geq 0}$ such that $\nu = \nu_a \cdot \nu_b$ induces μ, i.e., ν_a and ν_b would have to satisfy this system of equations:

$$\begin{aligned} \nu_a(\{a\}) \cdot \nu_b(\{b\}) &= 0.72 \\ \nu_a(\{a\}) \cdot \nu_b(\emptyset) &= 0.18 \\ \nu_a(\emptyset) \cdot \nu_b(\{b\}) &= 0.07 \\ \nu_a(\emptyset) \cdot \nu_b(\emptyset) &= 0.03, \end{aligned}$$

which has no solutions.

Alternatively, we can let b depend on a and consider weight functions $\nu_a \colon 2^{\{a\}} \to \mathbb{R}_{\geq 0}$ and $\nu_b \colon 2^{\{a,b\}} \to \mathbb{R}_{\geq 0}$ defined as $\nu_a(\{a\}) = 0.9$, $\nu_a(\emptyset) = 0.1$, and $\nu_b(\{a,b\}) = 0.8$, $\nu_b(\{a\}) = 0.2$, $\nu_b(\{b\}) = 0.7$, $\nu_b(\emptyset) = 0.3$. One can easily check that with these definitions ν indeed induces μ.

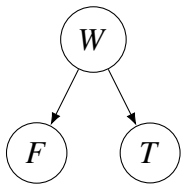
Note that in this case, we chose to interpret ν_b as $\Pr(b \mid a)$ while—with a different definition of ν_b that represents the joint probability distribution $\Pr(a,b)$—ν_b by itself could induce μ. In general, however, factoring the full weight function into several smaller functions often results in weight functions with smaller domains which leads to increased efficiency and decreased memory usage [Dudek et al., 2020a]. We can easily generalise this example further.

Theorem 3.2. *For any set U such that $|U| \geq 2$, there exists a non-factorable measure $2^{2^U} \to \mathbb{R}_{\geq 0}$.*

Since many measures of interest may not be factorable, a well-known way to encode them into instances of WMC is by adding more literals [Chavira and Darwiche, 2008]. We can use the measure-theoretic perspective on WMC to show that this is always possible, however, as ensuing sections will demonstrate, it can make the inference task much harder in practice.

Theorem 3.3. *For any set U and measure $\mu \colon 2^{2^U} \to \mathbb{R}_{\geq 0}$, there exists a set $V \supseteq U$, a factorable measure $\mu' \colon 2^{2^V} \to \mathbb{R}_{\geq 0}$, and a formula $f \in 2^{2^V}$ such that $\mu(x) = \mu'(x \wedge f)$ for all formulas $x \in 2^{2^U}$.*

Proof. Let $V = U \cup \{ f_m \mid m \in 2^U \}$, and $f = \bigwedge_{m \in 2^U} \{m\} \Leftrightarrow f_m$. We define weight function $\nu \colon 2^V \to \mathbb{R}_{\geq 0}$ as $\nu = \prod_{v \in V} \nu_v$, where $\nu_v(\{v\}) = \mu(\{m\})$ if $v = f_m$ for some $m \in 2^U$ and $\nu_v(x) = 1$ for all other $v \in V$ and $x \in 2^{\{v\}}$. Let $\mu' \colon 2^{2^V} \to \mathbb{R}_{\geq 0}$ be the



w	$\Pr(W = w)$
1	0.5
0	0.5

w	f	$\Pr(F = f \mid W = w)$
1	1	0.6
1	0	0.4
0	1	0.1
0	0	0.9

w	t	$\Pr(T = t \mid W = w)$
1	l	0.2
1	m	0.4
1	h	0.4
0	l	0.6
0	m	0.3
0	h	0.1

Figure 3.1: An example Bayesian network with its CPTs.

measure induced by ν. It is enough to show that μ and $x \mapsto \mu'(x \wedge f)$ agree on the atoms in 2^{2^U}. For any $\{a\} \in 2^{2^U}$,

$$\begin{aligned}\mu'(\{a\} \wedge f) &= \sum_{\{x\} \leq \{a\} \wedge f} \nu(x) = \nu(a \cup \{f_a\}) \\ &= \nu_{f_a}(\{f_a\}) = \mu(\{a\})\end{aligned}$$

as required. □

3.5 Encoding Bayesian Networks Using Conditional Weights

In this section, we describe a way to encode Bayesian networks into WMC without restricting oneself to factorable measures and thus having to add extra variables. We will refer to it as `cw`. A Bayesian network is a directed acyclic graph with random variables as nodes that defines a probability distribution over them. Let $\mathcal{V}$ denote this set of random variables. For any random variable $X \in \mathcal{V}$, let $\mathrm{Im}X$ denote its set of values and $\mathrm{pa}(X)$ its set of parents. The full probability distribution is then equal to $\prod_{X \in \mathcal{V}} \Pr(X \mid \mathrm{pa}(X))$. For discrete Bayesian networks (and we only consider discrete networks here), each factor of this product can be represented by a *conditional probability table* (CPT). See Figure 3.1 for an example Bayesian network that we

will refer to throughout this section. For this network, $\mathcal{V} = \{W, F, T\}$, $\text{pa}(W) = \emptyset$, $\text{pa}(F) = \text{pa}(T) = \{W\}$, $\text{Im}W = \text{Im}F = \{0,1\}$, and $\text{Im}T = \{l, m, h\}$.

Definition 3.2 (Indicator variables)**.** Let $X \in \mathcal{V}$ be a random variable. If X is binary (i.e., $|\text{Im}X| = 2$), we can arbitrary identify one of the values as 1 and the other one as 0 (i.e, $\text{Im}X \cong \{0,1\}$). Then X can be represented by a single *indicator variable* $\lambda_{X=1}$. For notational simplicity, for any set S, we write $\lambda_{X=0} \in S$ or $S = \{\lambda_{X=0}, \dots\}$ to mean $\lambda_{X=1} \notin S$.

On the other hand, if X is not binary, we represent X with $|\text{Im}X|$ indicator variables, one for each value. We let

$$\mathcal{E}(X) = \begin{cases} \{\lambda_{X=1}\} & \text{if } |\text{Im}X| = 2 \\ \{\lambda_{X=x} \mid x \in \text{Im}X\} & \text{otherwise.} \end{cases}$$

denote the set of indicator variables for X and $\mathcal{E}^*(X) = \mathcal{E}(X) \cup \bigcup_{Y \in \text{pa}(X)} \mathcal{E}(Y)$ denote the set of indicator variables for X and its parents in the Bayesian network. Finally, let $U = \bigcup_{X \in \mathcal{V}} \mathcal{E}(X)$ denote the set of all indicator variables for all random variables in the Bayesian network. For example, in the Bayesian network from Figure 3.1, $\mathcal{E}^*(T) = \{\lambda_{T=l}, \lambda_{T=m}, \lambda_{T=h}, \lambda_{W=1}\}$.

Algorithm 3.1 shows how a Bayesian network with nodes $\mathcal{V}$ can be represented as a weight function $\phi\colon 2^U \to \mathbb{R}_{\geq 0}$. The algorithm begins with the unit function and multiplies it by $\text{CPT}_X\colon 2^{\mathcal{E}^*(X)} \to \mathbb{R}_{\geq 0}$ for each random variable $X \in \mathcal{V}$. We call each such function a *conditional weight function* as it represents a conditional probability distribution. However, the distinction is primarily a semantic one: a function $2^{\{a,b\}} \to \mathbb{R}_{\geq 0}$ can represent $\Pr(a \mid b)$, $\Pr(b \mid a)$, or something else entirely, e.g., $\Pr(a \wedge b)$, $\Pr(a \vee b)$, etc.

For a binary random variable X, CPT_X is simply a sum of smaller functions, one for each row of the CPT. If X has more than two values, we also multiply CPT_X by 'clause' functions that restrict the value of $\phi(T)$ to zero whenever $|\mathcal{E}(X) \cap T| \neq 1$, i.e., we add mutual exclusivity constraints that ensure that each random variable is associated with exactly one value. Note that Chavira and Darwiche [2007] use the same ADD representation of CPTs for their compilation algorithm based on variable elimination.

Algorithm 3.1: Encoding a Bayesian network.

Data: set of random variables (i.e., nodes) $\mathcal{V}$, probability distribution Pr

Result: pseudo-Boolean function $\phi\colon 2^U \to \mathbb{R}_{\geq 0}$

$\phi \leftarrow 1$;
foreach *random variable* $X \in \mathcal{V}$ **do**
  let $\mathrm{pa}(X) = \{Y_1, \ldots, Y_n\}$;
  $\mathrm{CPT}_X \leftarrow 0$;
  if $|\operatorname{Im} X| = 2$ **then**
    forall *values* $(y_i)_{i=1}^n \in \prod_{i=1}^n \operatorname{Im} Y_i$ **do**
      $p_1 \leftarrow \Pr(X = 1 \mid y_1, \ldots, y_n)$;
      $p_0 \leftarrow \Pr(X \neq 1 \mid y_1, \ldots, y_n)$;
      $\mathrm{CPT}_X \leftarrow \mathrm{CPT}_X + p_1[\lambda_{X=1}] \cdot \prod_{i=1}^n [\lambda_{Y_i=y_i}] + p_0\overline{[\lambda_{X=1}]} \cdot \prod_{i=1}^n [\lambda_{Y_i=y_i}]$;
  else
    let $\operatorname{Im} X = \{x_1, \ldots, x_m\}$;
    forall *values* $x \in \operatorname{Im} X$ **and** $(y_i)_{i=1}^n \in \prod_{i=1}^n \operatorname{Im} Y_i$ **do**
      $p_x \leftarrow \Pr(X = x \mid y_1, \ldots, y_n)$;
      $\mathrm{CPT}_X \leftarrow \mathrm{CPT}_X + p_x[\lambda_{X=x}] \cdot \prod_{i=1}^n [\lambda_{Y_i=y_i}] + \overline{[\lambda_{X=x}]} \cdot \prod_{i=1}^n [\lambda_{Y_i=y_i}]$;
    $\mathrm{CPT}_X \leftarrow \mathrm{CPT}_X \cdot (\sum_{i=1}^m [\lambda_{X=x_i}]) \cdot \prod_{i=1}^m \prod_{j=i+1}^m (\overline{[\lambda_{X=x_i}]} + \overline{[\lambda_{X=x_j}]})$;
  $\phi \leftarrow \phi \cdot \mathrm{CPT}_X$;
return ϕ;

For the example Bayesian network in Figure 3.1, we get:

$$\begin{aligned}
\mathrm{CPT_F} &= 0.6[\lambda_{F=1}] \cdot [\lambda_{W=1}] + 0.4[\lambda_{F=0}] \cdot [\lambda_{W=1}] \\
&\quad + 0.1[\lambda_{F=1}] \cdot [\lambda_{W=0}] + 0.9[\lambda_{F=0}] \cdot [\lambda_{W=0}], \\
\mathrm{CPT_T} &= ([\lambda_{T=l}] + [\lambda_{T=m}] + [\lambda_{T=h}]) \\
&\quad \cdot (\overline{[\lambda_{T=l}]} + \overline{[\lambda_{T=m}]}) \cdot (\overline{[\lambda_{T=l}]} + \overline{[\lambda_{T=h}]}) \\
&\quad \cdot (\overline{[\lambda_{T=m}]} + \overline{[\lambda_{T=h}]}) \cdot \ldots .
\end{aligned}$$

3.5.1 Correctness

Algorithm 3.1 produces a function with a Boolean algebra as its domain. This function can be represented by an ADD [Bahar et al., 1997]. ADDMC takes an ADD $\psi\colon 2^U \to \mathbb{R}_{\geq 0}$ (expressed as a product of smaller ADDs) and returns $(\exists_U \psi)(\emptyset)$ [Dudek et al.,

2020a]. In this section, we prove that the function ϕ produced by Algorithm 3.1 can be used by ADDMC to correctly compute any marginal probability of the Bayesian network that was encoded as ϕ.[5] We begin with Lemma 3.1 which shows that any conditional weight function produces the right answer when given a valid encoding of variable-value assignments relevant to the CPT.

Lemma 3.1. *Let $X \in \mathcal{V}$ be a random variable with parents $\mathrm{pa}(X) = \{Y_1, \ldots, Y_n\}$. Then $\mathrm{CPT}_X \colon 2^{\mathcal{E}^*(X)} \to \mathbb{R}_{\geq 0}$ is such that for any $x \in \mathrm{Im}X$ and $(y_1, \ldots, y_n) \in \prod_{i=1}^n \mathrm{Im}Y_i$,*

$$\mathrm{CPT}_X(T) = \Pr(X = x \mid Y_1 = y_1, \ldots, Y_n = y_n),$$

where $T = \{\lambda_{X=x}\} \cup \{\lambda_{Y_i=y_i} \mid i = 1, \ldots, n\}$.

Proof. If X is binary, then CPT_X is a sum of $2\prod_{i=1}^n |\mathrm{Im}Y_i|$ terms, one for each possible assignment of values to variables $X, Y_1, \ldots, Y_n$. Exactly one of these terms is nonzero when applied to T, and it is equal to $\Pr(X = x \mid Y_1 = y_1, \ldots, Y_n = y_n)$ by definition.

If X is not binary, then $\left(\sum_{i=1}^m [\lambda_{X=x_i}]\right)(T) = 1$, and

$$\left(\prod_{i=1}^{m} \prod_{j=i+1}^{m} (\overline{[\lambda_{X=x_i}]} + \overline{[\lambda_{X=x_j}]})\right)(T) = 1,$$

so $\mathrm{CPT}_X(T) = \Pr(X = x \mid Y_1 = y_1, \ldots, Y_n = y_n)$ by a similar argument as before. □

Now, Lemma 3.2 shows that ϕ represents the full probability distribution of the Bayesian network, i.e., it gives the right probabilities for the right inputs and zero otherwise.

Lemma 3.2. *Let $\mathcal{V} = \{X_1, \ldots, X_n\}$. Then*

$$\phi(T) = \begin{cases} \Pr(x_1, \ldots, x_n) & \text{if } T = \{\lambda_{X_i=x_i}\}_{i=1}^n \text{ for} \\ & \text{some } (x_i)_{i=1}^n \in \prod_{i=1}^n \mathrm{Im}X_i \\ 0 & \text{otherwise,} \end{cases}$$

for all $T \in 2^U$.

Proof. If $T = \{\lambda_{X=v_X} \mid X \in \mathcal{V}\}$ for some $(v_X)_{X \in \mathcal{V}} \in \prod_{X \in \mathcal{V}} \mathrm{Im}X$, then

$$\phi(T) = \prod_{X \in \mathcal{V}} \Pr\left(X = v_X \,\middle|\, \bigwedge_{Y \in \mathrm{pa}(X)} Y = v_Y\right) = \Pr\left(\bigwedge_{X \in \mathcal{V}} X = v_X\right)$$

[5] Note that it can just as well compute *any* probability expressed using the random variables in $\mathcal{V}$.

by Lemma 3.1 and the definition of a Bayesian network. Otherwise there must be some non-binary random variable $X \in \mathcal{V}$ such that $|\mathcal{E}(X) \cap T| \neq 1$. If $\mathcal{E}(X) \cap T = \emptyset$, then $(\sum_{i=1}^{m} [\lambda_{X=x_i}])(T) = 0$, and so $\mathrm{CPT}_X(T) = 0$, and $\phi(T) = 0$. If $|\mathcal{E}(X) \cap T| > 1$, then we must have two different values $x_1, x_2 \in \mathrm{Im}\, X$ such that $\{\lambda_{X=x_1}, \lambda_{X=x_2}\} \subseteq T$ which means that $(\overline{[\lambda_{X=x_1}]} + \overline{[\lambda_{X=x_2}]})(T) = 0$, and so, again, $\mathrm{CPT}_X(T) = 0$, and $\phi(T) = 0$. □

We end with Theorem 3.4 that shows how ϕ can be combined with an encoding of a single variable-value assignment so that ADDMC [Dudek et al., 2020a] would compute its marginal probability.

Theorem 3.4. *For any $X \in \mathcal{V}$ and $x \in \mathrm{Im}\, X$,*

$$(\exists_U (\phi \cdot [\lambda_{X=x}]))(\emptyset) = \Pr(X = x).$$

Proof. Let $\mathcal{V} = \{X, Y_1, \ldots, Y_n\}$. Then

$$\begin{aligned}
(\exists_U (\phi \cdot [\lambda_{X=x}]))(\emptyset) &= \sum_{T \in 2^U} (\phi \cdot [\lambda_{X=x}])(T) \\
&= \sum_{\lambda_{X=x} \in T \in 2^U} \phi(T) \\
&= \sum_{\lambda_{X=x} \in T \in 2^U} \left(\prod_{Y \in \mathcal{V}} \mathrm{CPT}_Y \right)(T) \\
&= \sum_{(y_i)_{i=1}^n \in \prod_{i=1}^n \mathrm{Im}\, Y_i} \Pr(x, y_1, \ldots, y_n) \\
&= \Pr(X = x)
\end{aligned}$$

by:

- the proof of Theorem 1 by Dudek et al. [2020a];
- if $\lambda_{X=x} \notin T \in 2^U$, then

$$(\phi \cdot [\lambda_{X=x}])(T) = \phi(T) \cdot [\lambda_{X=x}](T \cap \{\lambda_{X=x}\}) = \phi(T) \cdot 0 = 0;$$

- Lemma 3.2;
- marginalisation of a probability distribution.

□

3.5.2 Textual Representation

Algorithm 3.1 encodes a Bayesian network into a function on a Boolean algebra, but how does it relate to the standard interpretation of a WMC encoding as a formula in conjunctive normal form (CNF) together with a collection of weights? The factors of ϕ that restrict the values of indicator variables for non-binary random variables are already expressed as a product of sums of 0/1-valued functions, i.e., a kind of CNF. Disregarding these functions, each conditional weight function CPT_X is represented by a sum with a term for every subset of $\mathcal{E}^*(X)$. To encode these terms, we introduce *extended weight clauses* to the WMC format used by CACHET [Sang et al., 2004]. For instance, here is a representation of the Bayesian network from Figure 3.1:

$$
\begin{array}{llllll}
\lambda_{T=l} & \lambda_{T=m} & \lambda_{T=h} & & 0 & \\
 & -\lambda_{T=l} & -\lambda_{T=m} & & 0 & \\
 & -\lambda_{T=l} & -\lambda_{T=h} & & 0 & \\
 & -\lambda_{T=m} & -\lambda_{T=h} & & 0 & \\
w & \lambda_{W=1} & & 0.5 & 0.5 & \\
w & \lambda_{F=1} & \lambda_{W=1} & 0.6 & 0.4 & \\
w & \lambda_{F=1} & -\lambda_{W=1} & 0.1 & 0.9 & \\
w & \lambda_{T=l} & \lambda_{W=1} & 0.2 & 1 & \\
w & \lambda_{T=m} & \lambda_{W=1} & 0.4 & 1 & \\
w & \lambda_{T=h} & \lambda_{W=1} & 0.4 & 1 & \\
w & \lambda_{T=l} & -\lambda_{W=1} & 0.6 & 1 & \\
w & \lambda_{T=m} & -\lambda_{W=1} & 0.3 & 1 & \\
w & \lambda_{T=h} & -\lambda_{W=1} & 0.1 & 1 &
\end{array}
$$

where each indicator variable is eventually replaced with a unique positive integer. Each line prefixed with a w can be split into four parts: the 'main' variable (always not negated), conditions (possibly none), and two weights. For example, the line

$$
w \quad \lambda_{T=m} \quad -\lambda_{W=1} \quad 0.3 \quad 1
$$

encodes the function $0.3[\lambda_{T=m}] \cdot \overline{[\lambda_{W=1}]} + 1\overline{[\lambda_{T=m}]} \cdot \overline{[\lambda_{W=1}]}$ and can be interpreted as defining two conditional weights: $\nu(T = m \mid W = 0) = 0.3$, and $\nu(T \neq m \mid W = 0) = 1$, the former of which corresponds to a row in the CPT of T while the latter is artificially added as part of the encoding. In our encoding of Bayesian networks, it is always the case that, in each weight clause, either both weights sum to one, or the second weight is equal to one. Finally, note that the measure induced by these weights is not

probabilistic (i.e., $\mu(\top) \neq 1$) by itself, but it becomes probabilistic when combined with the additional clauses that restrict what combinations of indicator variables can co-occur.

3.5.3 Changes to ADDMC

Here we describe two changes to ADDMC[6] [Dudek et al., 2020a] needed to adapt it to the new format. First, ADDMC constructs the primal graph of the input CNF formula as an aid for the algorithm's heuristics. We extend the usual definition of a primal graph to functions on Boolean algebras (i.e., the factors of ϕ) in the obvious way. For any pair of distinct variables $u, v \in U$, we draw an edge between them in the primal graph if there is a function $\alpha\colon 2^X \to \mathbb{R}_{\geq 0}$ that is a factor of ϕ such that $u, v \in X$. For instance, a factor such as CPT_X will enable edges between all distinct pairs of variables in $\mathcal{E}^*(X)$. Second, even though the function ϕ produced by Algorithm 3.1 is constructed to have 2^U as its domain, sometimes the domain is effectively reduced to 2^V for some $V \subset U$ by the ADD manipulation algorithms that optimise the ADD representation of a function. For a simple example, consider $\alpha : 2^{\{a\}} \to \mathbb{R}_{\geq 0}$ defined as $\alpha(\{a\}) = \alpha(\emptyset) = 0.5$. Then α can be reduced to $\alpha'\colon 2^{\emptyset} \to \mathbb{R}_{\geq 0}$ defined as $\alpha'(\emptyset) = 0.5$. To compensate for these reductions, for the original WMC format with a weight function $w\colon U \cup \{\neg u \mid u \in U\} \to \mathbb{R}_{\geq 0}$, ADDMC would multiply its computed answer by $\prod_{u \in U \setminus V} w(u) + w(\neg u)$. With the new WMC format, we instead multiply the answer by $2^{|U \setminus V|}$. Each 'excluded' variable $u \in U \setminus V$ satisfies two properties: all weights associated with u are equal to 0.5 (otherwise the corresponding CPT would depend on u, and u would not be excluded), and all other CPTs are independent of u (or they may have a trivial dependence, where the probability stays the same if u is replaced with its complement). Thus, the CPT that corresponds to u still multiplies the weight of every atom in the Boolean algebra by 0.5, but the number of atoms under consideration is halved. To correct for this, we multiply the final answer by two for every $u \in U \setminus V$.

3.6 Experimental Results

We compare the six WMC encodings for Bayesian networks when run with both ADDMC [Dudek et al., 2020a] and the WMC algorithms used in the original papers.[7]

[6] `https://github.com/vardigroup/ADDMC`

[7] Both `cd05` and `cd06` cannot be run with most WMC algorithms including ADDMC because these encodings allow for additional models that the WMC algorithm is supposed to ignore [Chavira and

We compare the encodings with respect to the total time it takes to encode a Bayesian network, compile it or run a WMC algorithm on it, and extract the (numerical) answer. Note that while all five papers that introduce other encodings include experimental comparisons of encoding size, that is not feasible with ADDMC as even instances that are fully solved in less than 0.1 s are too big to build the full ADD within reasonable time and memory limits. The experiments were run on a computing cluster with Intel Xeon Gold 6138 and Intel Xeon E5-2630 processors[8] running Scientific Linux 7 with a 32 GiB memory limit and a 1000 s timeout on both encoding and inference. For inference, we use ACE for `cd05` [Chavira and Darwiche, 2005], `cd06` [Chavira and Darwiche, 2006], and `d02` [Darwiche, 2002]; CACHET[9] [Sang et al., 2004] for `sbk05` [Sang et al., 2005a]; and C2D [Darwiche, 2004] for compilation and QUERY-DNNF [10] for answer computation for `bklm16` [Bart et al., 2016]. For encoding, we use BN2CNF [11] for `bklm16`, and ACE for all other encodings (except for `cw`, which is implemented in Python).

ACE was not used to encode evidence, as preliminary experiments revealed that the evidence-encoding implementation contains bugs that can lead to incorrect answers or a Java exception being thrown on some instances of the data set (and the source code is not publicly available). Instead, we simply list all the evidence as additional clauses in the encoding. Furthermore, to ensure that `bklm16` [Bart et al., 2016] (whether run with ADDMC [Dudek et al., 2020a] or C2D [Darwiche, 2004]) returns correct answers on most instances, we had to disable one of the improvements that `bklm16` brings over `cd06` [Chavira and Darwiche, 2006], namely, the construction of a scaling factor that 'absorbs' one probability from each CDT [Bart et al., 2016]. For realistic benchmark instances, this scaling factor can easily be below 10^{-30}, and thus would require arbitrary-precision floating-point arithmetic to be usable. Even a toy Bayesian network with seven binary independent variables with probabilities 0.1 and 0.9 is enough for BN2CNF to output precisely zero as the scaling factor. We note that this issue likely remained unnoticed because Bart et al. [2016] did not attempt to compute numerical answers in their experiments.

For each Bayesian network, we need to choose a probability to compute. Whenever a Bayesian network comes with an evidence file, we compute the probability of evidence.

Darwiche, 2005, 2006].

[8] Each instance is run on the same processor for all encodings.

[9] `https://cs.rochester.edu/u/kautz/Cachet/`

[10] `http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html`

[11] `http://www.cril.univ-artois.fr/KC/bn2cnf.html`

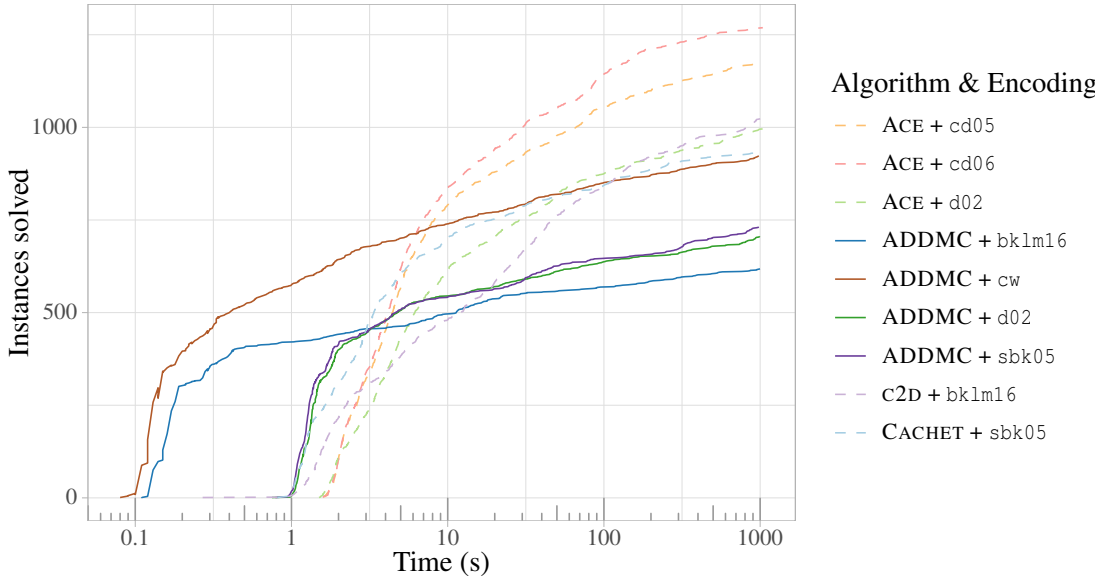


Figure 3.2: Cumulative numbers of instances solved by combinations of algorithms and encodings over time.

Otherwise, let X denote the last-mentioned random variable in the Bayesian network. If $\texttt{true} \in \mathrm{Im} X$, then we compute the marginal probability of $X = \texttt{true}$. Otherwise, we pick the value of X which is listed first and calculate its marginal probability.

For experimental data, we use the Bayesian networks available with ACE and CACHET [Sang et al., 2004], most of which happen to be binary. We classify them into the following seven categories:

- DQMR (390 instances) and
- Grid networks (450 instances) as described by Sang et al. [2005a],
- Mastermind (144 instances), and
- Random Blocks (256 instances) from the work of Chavira et al. [2006],
- remaining binary Bayesian networks (50 instances) that include Plan Recognition [Sang et al., 2005a], Friends and Smokers, Students and Professors [Chavira et al., 2006], and `tcc4f`, and
- non-binary classic Bayesian networks (176 instances) (`alarm`, `diabetes`, `hailfinder`, `mildew`, `munin1–4`, `pathfinder`, `pigs`, `water`).

Figure 3.2 shows that `cd05` [Chavira and Darwiche, 2005] and `cd06` [Chavira and Darwiche, 2006] (when run with ACE) are in the lead, while ADDMC [Dudek et al.,

Algorithm & Encoding	Unique	Fastest	Total
ACE + cd05	0	55	1169
ACE + cd06	**34**	218	**1259**
ACE + d02	0	46	993
ADDMC + bklm16	0	29	617
ADDMC + cw	14	**770**	919
ADDMC + d02	0	0	703
ADDMC + sbk05	0	0	729
C2D + bklm16	0	3	1017
CACHET + sbk05	13	229	928

Table 3.2: The numbers of instances (out of 1466) solved by each combination of algorithm and encoding (uniquely, faster than others, and in total).

Encoding(s)	Variables	Clauses/ADDs
bklm16, cd05, cd06, sbk05	$O(nv^{d+1})$	$O(nv^{d+1})$
cw	$O(nv)$	$O(nv^2)$
d02	$O(nv^{d+1})$	$O(ndv^{d+1})$

Table 3.3: Asymptotic upper bounds on the numbers of variables and clauses/ADDs for each encoding.

2020a] significantly underperforms when combined with any of the previous encodings. Our encoding cw significantly improves the performance of ADDMC, making ADDMC + cw comparable to ACE + d02, C2D + bklm16, and CACHET + sbk05. Furthermore, Table 3.2 shows that, while ACE + cd06 managed to solve the most instances, ADDMC + cw was the best-performing algorithm-encoding combination on the largest number of instances. The scatter plot on the left-hand side of Figure 3.3 add to this by showing that cw is particularly promising on Grid networks and tackles all DQMR instances in less than a second. The scatter plot on the right-hand side of Figure 3.3 shows that cw is better than sbk05 [Sang et al., 2005a] (i.e., the second-best encoding for ADDMC) on the majority of instances. Seeing how, e.g., DQMR instances are trivial for ADDMC + cw but hard for ACE + cd06, and vice versa for Mastermind instances, we conclude that the best-performing algorithm-encoding combination depends significantly on (as-of-yet unknown) properties of the Bayesian networks.

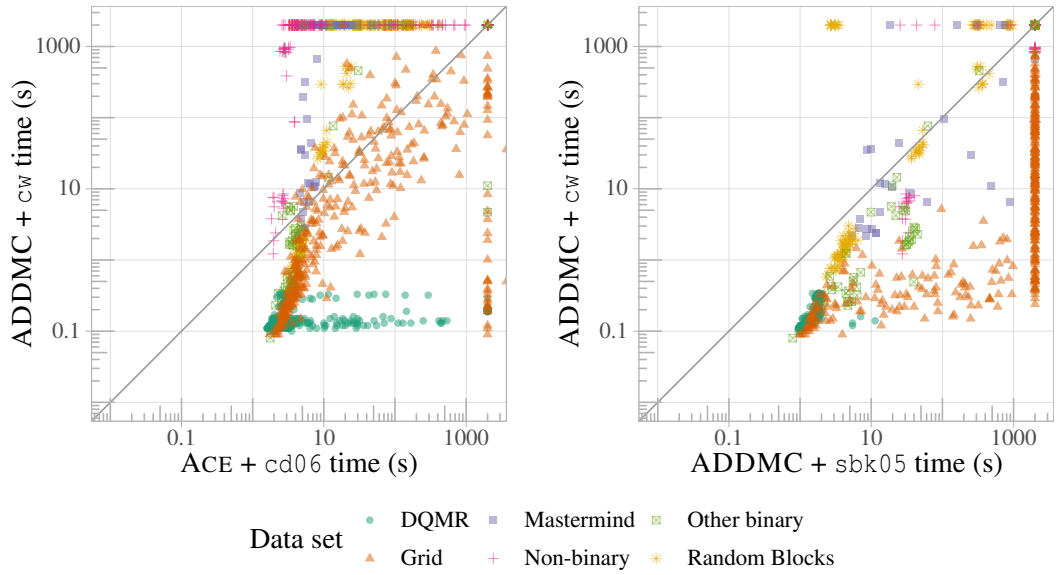


Figure 3.3: An instance-by-instance comparison between ADDMC + cw and the best overall combination of algorithm and encoding (ACE + cd06, on the left) as well as the second-best encoding for ADDMC (sbk05, on the right).

We can explain what makes ADDMC [Dudek et al., 2020a] run significantly faster with cw than with any other encoding by considering asymptotic upper bounds on the numbers of variables and ADDs based on the size and structure of the Bayesian network. Let $n = |\mathcal{V}|$ be the number of nodes in the Bayesian network, $d = \max_{X \in \mathcal{V}} |\mathrm{pa}(X)|$ the maximum in-degree (i.e., the number of parents), and $v = \max_{X \in \mathcal{V}} |\operatorname{Im} X|$ the maximum number of values per variable. Table 3.3 shows how cw has fewer variables and fewer ADDs than any other encoding. We conjecture that it is primarily the reduced number of variables that makes the ADDMC variable ordering heuristics much more effective. Note that these are upper bounds and most encodings (including cw) can be smaller in certain situations (e.g., with binary random variables or when a CPT has repeating probabilities). We equate clauses and ADDs (more specifically, factors of the function ϕ from Algorithm 3.1) here because ADDMC interprets each clause of any WMC encoding as a multiplicative factor of the ADD that represents the entire WMC instance [Dudek et al., 2020a]. For literal-weight encodings, each weight is also a factor, but that does not affect our asymptotic bounds.

3.7 Conclusions and Future Work

WMC was originally motivated by an appeal to the success of SAT solvers in efficiently tackling an NP-complete problem [Sang et al., 2005a]. ADDMC does not rely on SAT-based algorithmic techniques [Dudek et al., 2020a], and our proposed format diverges even more from the DIMACS CNF format for propositional formulas. To what extent are SAT-based methods still applicable? The answer depends significantly on the problem domain. For Bayesian networks, the rules describing that each random variable can only be associated with exactly one value were still encoded as clauses. As has been noted previously [Chavira and Darwiche, 2006], rows in CPTs with probabilities equal to zero or one can be represented as clauses as well. Therefore, our work can be seen as proposing a middle ground between #SAT and probabilistic inference.

While we chose ADDMC [Dudek et al., 2020a] as the WMC algorithm and Bayesian networks as a canonical example of a probabilistic inference task, these are only examples meant to illustrate the broader idea that choosing a more expressive representation of weights can outperform increasing the size of the problem to keep the weights simple. Indeed, in this work, we have provided a new theoretical perspective on the expressive power of WMC and illustrated the empirical benefits of that perspective. Perhaps the same idea could be adapted to other inference problem domains such as probabilistic programs [Fierens et al., 2015, Holtzen et al., 2020a] as well as to search-based solvers such as CACHET [Sang et al., 2004].

Chapter 4 continues to develop the perspective on WMC initiated in this chapter. There, we present a generalisation of WMC based on pseudo-Boolean functions. Support for the new format is then implemented for DPMC—an extension to ADDMC capable of performing computations using tensors (rather than ADDs) and planning using tree decompositions [Dudek et al., 2020b].

Chapter 4

WMC Without Parameter Variables

4.1 Introduction

Recall that in Chapter 3 we examined WMC from a measure-theoretic perspective and found that—due to the rigidity of the standard formulation—WMC encodings usually contain many superfluous variables. This observation inspired an encoding for Bayesian networks that alleviates the issue. In this chapter, we continue to tackle the subject of superfluous parameter variables but take the solution a few steps further.

If WMC is not the right format for probabilistic inference and other sum-of-products computations, what could be a better alternative? As many WMC inference algorithms [Darwiche, 2004, Oztok and Darwiche, 2015] work by compilation to tractable representations such as arithmetic circuits, deterministic, decomposable negation normal form [Darwiche, 2001b], and sentential decision diagrams (SDDs) [Darwiche, 2011], perhaps parameter variables could be avoided via direct compilation to a more convenient representation. Direct compilation of Bayesian networks to SDDs has been investigated [Choi et al., 2013]. However, SDDs only support weights on literals, and so are not expressive enough to avoid the issue. Gogate and Domingos [2010] propose a format based on weighted clauses and probabilistic semantics inspired by Markov networks. However, with a new representation comes the need to invent new encodings and inference algorithms. Moreover, to the best of our knowledge, neither approach [Choi et al., 2013, Gogate and Domingos, 2010] has a publicly available implementation.

In this work, we introduce a new computational problem called *pseudo-Boolean projection* (PBP)—a generalisation of the conditional weights approach proposed in Chapter 3. Recent WMC algorithms based on pseudo-Boolean function manipulation—namely, ADDMC [Dudek et al., 2020a] and DPMC [Dudek et al., 2020b]—can easily

be adapted to the new format. In contrast to our previous work, instead of inventing new encodings, we show how every WMC problem instance can be transformed to PBP and identify conditions under which this transformation can remove parameter variables. Four out of the five known WMC encodings for Bayesian networks [Bart et al., 2016, Chavira and Darwiche, 2005, 2006, Darwiche, 2002, Sang et al., 2005a] can indeed be simplified in this manner. We are able to eliminate 43 % of variables on average and up to 99 % on some instances. This transformation enables two encodings that were previously incompatible with most WMC algorithms (due to using a different formulation of WMC [Chavira and Darwiche, 2005, 2006]) to be run with ADDMC and DPMC and results in a significant performance boost for one other encoding, making it about three times faster than the state of the art. Finally, our theoretical contributions result in a convenient algebraic way of reasoning about two-valued pseudo-Boolean functions and position WMC encodings on common ground, identifying their key properties and assumptions.

4.2 Redefining WMC

Since the goal of this chapter is to generalise WMC in a way that eliminates the redundant parameter variables, in this section we redefine WMC in a way that explicitly partitions all variables into parameter and indicator variables. We also formalise a variant of WMC that has been implicitly used by Chavira and Darwiche [2005, 2006].

In this chapter, we denote interpretations (and models) of a propositional formula as subsets of the set of variables. These variables are implicitly mapped to `true` whereas all other variables are mapped to `false`. The *cardinality* of a model is then the cardinality of this set.

Example 4.1. Let $\phi = (\neg a \vee b) \wedge a$ be a propositional formula over variables a and b. Then $\{a, b\}$ (i.e., $\{a \mapsto \texttt{true}, b \mapsto \texttt{true}\}$) is a model of ϕ (written $\{a, b\} \models \phi$), and it has cardinality two.

Definition 4.1 (WMC)**.** A *WMC instance* is a tuple (ϕ, X_I, X_P, w), where X_I is the set of *indicator variables*, X_P is the set of *parameter variables* (with $X_I \cap X_P = \emptyset$), ϕ is a propositional formula in CNF over $X_I \cup X_P$, and $w\colon X_I \cup X_P \cup \{\neg x \mid x \in X_I \cup X_P\} \to \mathbb{R}$ is a *weight function* such that $w(x) = w(\neg x) = 1$ for all $x \in X_I$. The *answer* of the instance is $\sum_{Y \models \phi} \prod_{Y \models l} w(l)$.

In practice, we identify this partition of variables in one of two ways. If an encoding is generated by ACE[1], then variable types are explicitly identified in a file generated alongside the encoding. Otherwise, we take X_I to be the set of all variables x such that $w(x) = w(\neg x) = 1$. Next, we formally define another variant of WMC.

Definition 4.2. Let ϕ be a formula over a set of variables X. Then $Y \subseteq X$ is a *minimum-cardinality model* of ϕ if $Y \models \phi$ and $|Y| \leq |Z|$ for all $Z \models \phi$.

Definition 4.3 (Minimum-cardinality WMC)**.** A *minimum-cardinality WMC* instance consists of the same tuple as a WMC instance, but its *answer* is defined to be

$$\sum_{Y \models \phi, |Y|=k} \prod_{Y \models l} w(l)$$

(where $k = \min_{Y \models \phi} |Y|$) if ϕ is satisfiable, and zero otherwise.

Example 4.2. Let $\phi = (x \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee p) \wedge (\neg y \vee q) \wedge x$, $X_I = \{x, y\}$, $X_P = \{p, q\}$, $w(p) = 0.2$, $w(q) = 0.8$, and $w(\neg p) = w(\neg q) = 1$. Then ϕ has two models: $\{x, p\}$ and $\{x, p, q\}$ with weights 0.2 and $0.2 \times 0.8 = 0.16$, respectively. The WMC answer is then $0.2 + 0.16 = 0.36$, and the minimum-cardinality WMC answer is 0.2.

4.3 Bayesian Network Encodings

Recall that a *Bayesian network* is a directed acyclic graph with random variables as nodes and edges as conditional dependencies. As is common in related literature [Darwiche, 2002, Sang et al., 2005a], we assume that each variable has a finite number of values. We call a Bayesian network *binary* if every variable has two values.

WMC is a well-established technique for Bayesian network inference, particularly effective on networks where most variables have only a few possible values [Darwiche, 2002]. Many ways of encoding a Bayesian network into a WMC instance have been proposed. Darwiche [2002] was the first to suggest the `d02` encoding that, in many ways, remains the foundation behind most other encodings. He also introduced the distinction between *indicator* and *parameter variables*; the former represent variable-value pairs in the Bayesian network, while the latter are associated with probabilities in the conditional probability tables (CPTs). The encoding `sbk05` [Sang et al., 2005a] is the only encoding that deviates from this arrangement: for each variable in the Bayesian network, one

[1] ACE [Chavira and Darwiche, 2008] implements most of the Bayesian network encodings and can also be used for compilation (and thus inference). It is available at `http://reasoning.cs.ucla.edu/ace/`.

indicator variable acts simultaneously as a parameter variable. Chavira and Darwiche [2005] propose `cd05` where they shift from WMC to minimum-cardinality WMC because that allows the encoding to have fewer variables and clauses. In particular, they propose a way to use the same parameter variable to represent all probabilities in a CPT that are equal and keep only clauses that 'imply' parameter variables (i.e., omit clauses where a parameter variable implies indicator variables).[2] In their next encoding, `cd06`, Chavira and Darwiche [2006] optimise the aforementioned implication clauses, choosing the smallest sufficient selection of indicator variables. A decade later, Bart et al. [2016] present `bklm16` that improves upon `cd06` in two ways. First, they optimise the number of indicator variables used per Bayesian network variable from a linear to a logarithmic amount. Second, they introduce a scaling factor that can 'absorb' one probability per Bayesian network variable. However, for this work, we choose to disable the latter improvement since this scaling factor is often small enough to be indistinguishable from zero without the use of arbitrary precision arithmetic, making it completely unusable on realistic instances. Indeed, the reader is free to check that even a small Bayesian network with seven mutually independent binary variables, 0.1 and 0.9 probabilities each, is already big enough for the scaling factor to be exactly equal to zero (as produced by the `bklm16` encoder[3]). We suspect that this issue was not identified during the original set of experiments because the authors never looked at numerical answers.

Example 4.3. Let $\mathcal{B}$ be a Bayesian network with one variable X which has two values x_1 and x_2 with probabilities $\Pr(X = x_1) = 0.2$ and $\Pr(X = x_2) = 0.8$. Let x, y be indicator variables, and p, q be parameter variables. Then Example 4.2 is both the `cd05` and the `cd06` encoding of $\mathcal{B}$. The `bklm16` encoding is $(x \Rightarrow p) \wedge (\neg x \Rightarrow q) \wedge x$ with $w(p) = w(\neg q) = 0.2$, and $w(\neg p) = w(q) = 0.8$. And the `d02` encoding is $(\neg x \Rightarrow p) \wedge (p \Rightarrow \neg x) \wedge (x \Rightarrow q) \wedge (q \Rightarrow x) \wedge \neg x$ with $w(p) = 0.2$, $w(q) = 0.8$, and $w(\neg p) = w(\neg q) = 1$. Note how all other encodings have fewer clauses than `d02`. While `cd05` and `cd06` require minimum-cardinality WMC to make this work, `bklm16` achieves the same thing by adjusting weights.[4]

[2] Example 4.3 demonstrates what we mean by implication clauses.

[3] `http://www.cril.univ-artois.fr/kc/bn2cnf.html`

[4] Note that since `cd05` and `cd06` are minimum-cardinality WMC encodings, they are not supported by most WMC algorithms.

4.4 Pseudo-Boolean Functions

In this work, we propose a more expressive representation for WMC based on pseudo-Boolean functions. Since two-valued pseudo-Boolean functions will be used extensively henceforth, we introduce some new notation. For any propositional formula ϕ over a set of variables X and $p, q \in \mathbb{R}$, let $[\phi]_q^p \colon 2^X \to \mathbb{R}$ be the pseudo-Boolean function defined as

$$[\phi]_q^p(Y) \coloneqq \begin{cases} p & \text{if } Y \models \phi \\ q & \text{otherwise} \end{cases}$$

for any $Y \subseteq X$.

Below we list some properties of the operations on pseudo-Boolean functions that can be conveniently represented using our syntax. The proofs of all these properties follow directly from the definitions.

Proposition 4.1 (Basic properties)**.** *For any propositional formulas ϕ and ψ, and $a, b, c, d \in \mathbb{R}$,*

- $[\phi]_b^a = [\neg\phi]_a^b$;
- $c + [\phi]_b^a = [\phi]_{b+c}^{a+c}$;
- $c \cdot [\phi]_b^a = [\phi]_{bc}^{ac}$;
- $[\phi]_b^a \cdot [\phi]_d^c = [\phi]_{bd}^{ac}$;
- $[\phi]_0^1 \cdot [\psi]_0^1 = [\phi \wedge \psi]_0^1$.

And for any pair of pseudo-Boolean functions $f, g \colon 2^X \to \mathbb{R}$ and $x \in X$, $(fg)|_{x=i} = f|_{x=i} \cdot g|_{x=i}$ for $i = 0, 1$.

Remark. For convenience, we assume that the domain of a pseudo-Boolean function f shrinks whenever f is independent of some of the variables (i.e., $f|_{x=0} = f|_{x=1}$) and expand for binary operations to make the domains of both functions equal. For instance, let $[x]_0^1, [\neg x]_0^1 \colon 2^{\{x\}} \to \mathbb{R}$ and $[y]_0^1 \colon 2^{\{y\}} \to \mathbb{R}$ be pseudo-Boolean functions. Then $[x]_0^1 \cdot [\neg x]_0^1$ has $2^\emptyset$ as its domain. To multiply $[x]_0^1$ and $[y]_0^1$, we expand $[x]_0^1$ into $\left([x]_0^1\right)' \colon 2^{\{x,y\}} \to \mathbb{R}$ which is defined as $\left([x]_0^1\right)'(Z) \coloneqq [x]_0^1(Z \cap \{x\})$ for all $Z \subseteq \{x, y\}$ (and equivalently for $[y]_0^1$).

4.5 Pseudo-Boolean Projection

We introduce a new type of computational problem called *pseudo-Boolean projection* based on two-valued pseudo-Boolean functions. While the same computational framework can handle any pseudo-Boolean functions, two-valued functions are particularly convenient because DPMC [Dudek et al., 2020b] can be easily adapted to use them as input. Since we will only encounter functions of the form $[\phi]_b^a$, where ϕ is a conjunction of literals, we can represent it in text as `w` $\langle\phi\rangle$ `a b` where $\langle\phi\rangle$ is a representation of ϕ analogous to the representation of a clause in the DIMACS CNF format.

Definition 4.4 (PBP instance)**.** A PBP instance is a tuple (F, X, ω), where X is the set of variables, F is a set of two-valued pseudo-Boolean functions $2^X \to \mathbb{R}$, and $\omega \in \mathbb{R}$ is the scaling factor.[5] Its *answer* is $\omega \cdot \left(\exists_X \prod_{f \in F} f\right)(\emptyset)$.

4.5.1 From WMC to PBP

In this section, we describe an algorithm for transforming WMC instances to the PBP format while removing all parameter variables. We chose to transform existing encodings instead of creating a new one to reuse already-existing techniques for encoding each CPT to its minimal logical representation such as prime implicants and limited forms of resolution [Bart et al., 2016, Chavira and Darwiche, 2005, 2006]. The transformation algorithm works on four out of the five Bayesian network encodings: `bklm16` [Bart et al., 2016], `cd05` [Chavira and Darwiche, 2005], `cd06` [Chavira and Darwiche, 2006], and `d02` [Darwiche, 2002]. There is no obvious way to adjust it to work with `sbk05` because the roles of indicator and parameter variables overlap [Sang et al., 2005a].

The algorithm is based on several observations that will be made more precise in Section 4.5.2. First, all weights except for $\{ w(p) \mid p \in X_P \}$ are redundant as they either duplicate an already-defined weight or are equal to one. Second, each clause has at most one parameter variable. Third, if the parameter variable is negated, we can ignore the clause (this idea comes from the work of Chavira and Darwiche [2005]). Note that while we formulate our algorithm as a sequel to the WMC encoding procedure primarily because the implementations of Bayesian network WMC encodings are all closed-source, as all transformations in the algorithm are local, it can be efficiently incorporated into a WMC encoding algorithm with no slowdown.

[5] Adding scaling factor ω to the definition allows us to remove clauses that consist entirely of a single parameter variable. The idea of extracting some of the structure of the WMC instance into an external multiplicative factor was loosely inspired by the `bklm16` encoding, where it is used to subsume the most

Algorithm 4.1: WMC to PBP transformation.

Data: WMC (or minimum-cardinality WMC) instance (ϕ, X_I, X_P, w)

Result: PBP instance (F, X_I, ω)

$F \leftarrow \emptyset$;

$\omega \leftarrow 1$;

foreach *clause* $c \in \phi$ **do**

  if $c \cap X_P = \{ p \}$ *for some variable* p **and** $w(p) \neq 1$ **then**

    if $|c| = 1$ **then** $\omega \leftarrow \omega \times w(p)$;

    else $F \leftarrow F \cup \left\{ \left[\bigwedge_{l \in c \setminus \{ p \}} \neg l \right]_1^{w(p)} \right\}$;

  else if $\{ p \mid \neg p \in c \} \cap X_P = \emptyset$ **then**

    $F \leftarrow F \cup \{ [c]_0^1 \}$;

foreach *variable* $v \in X_I$ *such that* $\{ [v]_1^p, [\neg v]_1^q \} \subseteq F$ *for some* p *and* q **do**

  $F \leftarrow F \setminus \{ [v]_1^p, [\neg v]_1^q \} \cup \{ [v]_q^p \}$;

The algorithm is listed as Algorithm 4.1. The main part of the algorithm is the first loop that iterates over clauses. If a clause consists of a single parameter variable, we incorporate it into ω. If a clause is of the form $\alpha \Rightarrow p$, where $p \in X_P$, and α is a conjunction of literals over X_I, we transform it into a pseudo-Boolean function $[\alpha]_1^{w(p)}$. If a clause $c \in \phi$ has no parameter variables, we reformulate it into a pseudo-Boolean function $[c]_0^1$. Finally, clauses with negative parameter literals are omitted.

As all 'weighted' pseudo-Boolean functions produced by the first loop are of the form $[\alpha]_1^p$ (for some $p \in \mathbb{R}$ and formula α), the second loop merges two functions into one whenever α is a literal. Note that taking into account the order in which clauses are typically generated by encoding algorithms allows us to do this in linear time (i.e., the two mergeable functions will be generated one after the other).

4.5.2 Correctness Proofs

In this section, we outline key conditions that a (WMC or minimum-cardinality WMC) encoding has to satisfy for Algorithm 4.1 to output an equivalent PBP instance. We divide the correctness proof into two theorems: Theorem 4.2 for WMC encodings (i.e., `bklm16` and `d02`) and Theorem 4.3 for minimum-cardinality WMC encodings (i.e., `cd05` and `cd06`). We begin by listing some properties of pseudo-Boolean functions and

commonly occurring probability of each CPT [Bart et al., 2016].

establishing a canonical transformation from WMC to PBP.

Theorem 4.1 (Early projection [Dudek et al., 2020a,b])**.** *Let X and Y be sets of variables. For all pseudo-Boolean functions $f\colon 2^X \to \mathbb{R}$ and $g\colon 2^Y \to \mathbb{R}$, if $x \in X \setminus Y$, then $\exists_x(f \cdot g) = (\exists_x f) \cdot g$.*

Lemma 4.1. *For any pseudo-Boolean function $f\colon 2^X \to \mathbb{R}$, we have that $(\exists_X f)(\emptyset) = \sum_{Y \subseteq X} f(Y)$.*

Proof. If $X = \{x\}$, then

$$(\exists_x f)(\emptyset) = (f|_{x=1} + f|_{x=0})(\emptyset) = f|_{x=1}(\emptyset) + f|_{x=0}(\emptyset) = \sum_{Y \subseteq \{x\}} f(Y).$$

This easily extends to $|X| > 1$ by the definition of projection on sets of variables. □

Proposition 4.2. *Let (ϕ, X_I, X_P, w) be a WMC instance. Then*

$$\left(\left\{ [c]_0^1 \;\middle|\; c \in \phi \right\} \cup \left\{ [x]_{w(\neg x)}^{w(x)} \;\middle|\; x \in X_I \cup X_P \right\}, X_I \cup X_P, 1 \right) \tag{4.1}$$

is a PBP instance with the same answer (as in Definitions 4.1 and 4.4).

Proof. Let $f = \prod_{c \in \phi} [c]_0^1$, and $g = \prod_{x \in X_I \cup X_P} [x]_{w(\neg x)}^{w(x)}$. Then the WMC answer of Equation (4.1) is $(\exists_{X_I \cup X_P} fg)(\emptyset) = \sum_{Y \subseteq X_I \cup X_P} (fg)(Y) = \sum_{Y \subseteq X_I \cup X_P} f(Y)g(Y)$ by Lemma 4.1. Note that

$$f(Y) = \begin{cases} 1 & \text{if } Y \models \phi, \\ 0 & \text{otherwise,} \end{cases} \quad \text{and} \quad g(Y) = \prod_{Y \models l} w(l),$$

which means that $\sum_{Y \subseteq X_I \cup X_P} f(Y)g(Y) = \sum_{Y \models \phi} \prod_{Y \models l} w(l)$ as required. □

Theorem 4.2 (Correctness for WMC)**.** *Algorithm 4.1, when given a WMC instance (ϕ, X_I, X_P, w), returns a PBP instance with the same answer (as defined in Definitions 4.1 and 4.4), provided* either *of the two conditions is satisfied:*

1. *for all $p \in X_P$, there is a non-empty family of literals $(l_i)_{i=1}^n$ such that*
   - *(a) $w(\neg p) = 1$,*
   - *(b) $l_i \in X_I$ or $\neg l_i \in X_I$ for all $i = 1, \ldots, n$,*
   - *(c) and $\{ c \in \phi \mid p \in c \text{ or } \neg p \in c \} = \{ p \vee \bigvee_{i=1}^n \neg l_i \} \cup \{ l_i \vee \neg p \mid i = 1, \ldots, n \}$;*
2. *or for all $p \in X_P$,*
   - *(a) $w(p) + w(\neg p) = 1$,*

(b) for any clause $c \in \phi$, $|c \cap X_P| \leq 1$,

(c) there is no clause $c \in \phi$ *such that* $\neg p \in c$,

(d) if $\{p\} \in \phi$, *then there is no clause* $c \in \phi$ *such that* $c \neq \{p\}$ *and* $p \in c$,

(e) and for any $c, d \in \phi$ *such that* $c \neq d$, $p \in c$ *and* $p \in d$, $\bigwedge_{l \in c \setminus \{p\}} \neg l \wedge \bigwedge_{l \in d \setminus \{p\}} \neg l$ *is false.*

Condition 1 (for `d02`) simply states that each parameter variable is equivalent to a conjunction of indicator literals. Condition 2 is for encodings that have implications rather than equivalences associated with parameter variables (which, in this case, is `bklm16`). It ensures that each clause has at most one positive parameter literal and no negative ones, and that at most one implication clause per any parameter variable $p \in X_P$ can 'force p to be positive'.

Proof. By Proposition 4.2,

$$\left(\left\{ [c]_0^1 \;\middle|\; c \in \phi \right\} \cup \left\{ [x]_{w(\neg x)}^{w(x)} \;\middle|\; x \in X_I \cup X_P \right\}, X_I \cup X_P, 1 \right) \tag{4.2}$$

is a PBP instance with the same answer as the given WMC instance. By Definition 4.4, its answer is $\left(\exists_{X_I \cup X_P} \left(\prod_{c \in \phi} [c]_0^1\right) \prod_{x \in X_I \cup X_P} [x]_{w(\neg x)}^{w(x)}\right)(\emptyset)$. Since both Conditions 1 and 2 ensure that each clause in ϕ has at most one parameter variable, we can partition ϕ into $\phi_* := \{c \in \phi \mid \texttt{Vars}(c) \cap X_P = \emptyset\}$ and $\phi_p := \{c \in \phi \mid \texttt{Vars}(c) \cap X_P = \{p\}\}$ for all $p \in X_P$. We can then use Theorem 4.1 to reorder the answer into $\left(\exists_{X_I} \left(\prod_{x \in X_I} [x]_{w(\neg x)}^{w(x)}\right) \left(\prod_{c \in \phi_*} [c]_0^1\right) \prod_{p \in X_P} \exists_p [p]_{w(\neg p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1\right)(\emptyset)$.

Let us first consider how the unfinished WMC instance (F, X_I, ω) after the loop on lines 3–8 differs from Equation (4.2). Note that Algorithm 4.1 leaves each $c \in \phi_*$ unchanged, i.e., adds $[c]_0^1$ to F. We can then fix an arbitrary $p \in X_P$ and let F_p be the set of functions added to F as a replacement of ϕ_p. It is sufficient to show that

$$\omega \prod_{f \in F_p} f = \exists_p [p]_{w(\neg p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1. \tag{4.3}$$

Note that under Condition 1, $\bigwedge_{c \in \phi_p} c \equiv p \Leftrightarrow \bigwedge_{i=1}^n l_i$ for some family of indicator variable literals $(l_i)_{i=1}^n$. Thus, $\exists_p [p]_{w(\neg p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1 = \exists_p [p]_1^{w(p)} [p \Leftrightarrow \bigwedge_{i=1}^n l_i]_0^1$. If $w(p) = 1$, then

$$\exists_p [p]_1^{w(p)} \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i \right]_0^1 = \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i \right]_0^1 \Bigg|_{p=1} + \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i \right]_0^1 \Bigg|_{p=0}. \tag{4.4}$$

Since for any input, $\bigwedge_{i=1}^{n} l_i$ is either true or false, exactly one of the two summands in Equation (4.4) will be equal to one, and the other will be equal to zero, and so

$$\left[p \Leftrightarrow \bigwedge_{i=1}^{n} l_i\right]_0^1\Bigg|_{p=1} + \left[p \Leftrightarrow \bigwedge_{i=1}^{n} l_i\right]_0^1\Bigg|_{p=0} = 1,$$

where 1 is a pseudo-Boolean function that always returns one. On the other side of Equation (4.3), since $F_p = \emptyset$, and ω is unchanged, we get $\omega \prod_{f \in F_p} f = 1$, and so Equation (4.3) is satisfied under Condition 1 when $w(p) = 1$.

If $w(p) \neq 1$, then $F_p = \left\{ [\bigwedge_{i=1}^{n} l_i]_1^{w(p)} \right\}$, and $\omega = 1$, and so we want to show that $[\bigwedge_{i=1}^{n} l_i]_1^{w(p)} = \exists_p [p]_1^{w(p)} [p \Leftrightarrow \bigwedge_{i=1}^{n} l_i]_0^1$. Indeed,

$$\exists_p [p]_1^{w(p)} \left[p \Leftrightarrow \bigwedge_{i=1}^{n} l_i\right]_0^1 = w(p) \cdot \left[\bigwedge_{i=1}^{n} l_i\right]_0^1 + \left[\bigwedge_{i=1}^{n} l_i\right]_1^0 = \left[\bigwedge_{i=1}^{n} l_i\right]_1^{w(p)}.$$

This finishes the proof of the correctness of the first loop under Condition 1.

Now let us assume Condition 2. We still want to prove Equation (4.3). If $w(p) = 1$, then $F_p = \emptyset$, and $\omega = 1$, and so the left-hand side of Equation (4.3) is equal to one. Then the right-hand side is $\exists_p [p]_0^1 \prod_{c \in \phi_p} [c]_0^1 = \exists_p \left[p \wedge \bigwedge_{c \in \phi_p} c\right]_0^1 = \exists_p [p]_0^1 = 0 + 1 = 1$ since $p \in c$ for every clause $c \in \phi_p$.

If $w(p) \neq 1$, and $\{p\} \in \phi_p$, then, by Condition 2d, $\phi_p = \{\{p\}\}$, and Algorithm 4.1 produces $F_p = \emptyset$, and $\omega = w(p)$, and so $\exists_p [p]_{w(\neg p)}^{w(p)} [p]_0^1 = \exists_p [p]_0^{w(p)} = w(p) = \omega \prod_{f \in F_p} f$. The only remaining case is when $w(p) \neq 1$ and $\{p\} \notin \phi_p$. Then $\omega = 1$, and $F_p = \left\{ \left[\bigwedge_{l \in c \setminus \{p\}} \neg l\right]_1^{w(p)} \;\middle|\; c \in \phi_p \right\}$, so we need to show that

$$\prod_{c \in \phi_p} \left[\bigwedge_{l \in c \setminus \{p\}} \neg l\right]_1^{w(p)} = \exists_p [p]_{1-w(p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1.$$

We can rearrange the right-hand side as

$$\begin{aligned}
\exists_p [p]_{1-w(p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1 &= \exists_p [p]_{1-w(p)}^{w(p)} \left[p \vee \bigwedge_{c \in \phi_p} c \setminus \{p\}\right]_0^1 \\
&= w(p) + (1 - w(p)) \left[\bigwedge_{c \in \phi_p} c \setminus \{p\}\right]_0^1 \\
&= \left[\bigwedge_{c \in \phi_p} c \setminus \{p\}\right]_{w(p)}^1 = \left[\bigvee_{c \in \phi_p} \bigwedge_{l \in c \setminus \{p\}} \neg l\right]_1^{w(p)}.
\end{aligned}$$

By Condition 2e, $\bigwedge_{l \in c \setminus \{p\}} \neg l$ can be true for at most one $c \in \phi_p$, and so

$$\left[\bigvee_{c \in \phi_p} \bigwedge_{l \in c \setminus \{p\}} \neg l \right]_1^{w(p)} = \prod_{c \in \phi_p} \left[\bigwedge_{l \in c \setminus \{p\}} \neg l \right]_1^{w(p)}$$

which is exactly what we needed to show. This ends the proof that the first loop of Algorithm 4.1 preserves the answer under both Condition 1 and Condition 2. Finally, the loop on lines 9–10 of Algorithm 4.1 replaces $[v]_1^p[\neg v]_1^q$ with $[v]_q^p$ (for some $v \in X_I$ and $p, q \in \mathbb{R}$), but, of course, $[v]_1^p[\neg v]_1^q = [v]_1^p[v]_q^1 = [v]_q^p$, i.e., the answer is unchanged. □

Theorem 4.3 (Minimum-cardinality correctness)**.** *Let (ϕ, X_I, X_P, w) be a minimum-cardinality WMC instance that satisfies Conditions 2b–2e of Theorem 4.2 as well as the following:*

1. *for all parameter variables $p \in X_P$, $w(\neg p) = 1$.*
2. *all models of $\{ c \in \phi \mid c \cap X_P = \emptyset \}$ (as subsets of X_I) have the same cardinality;*
3. *$\min_{Z \subseteq X_P} |Z|$ such that $Y \cup Z \models \phi$ is the same for all $Y \models \{ c \in \phi \mid c \cap X_P = \emptyset \}$.*

Then Algorithm 4.1, when applied to (ϕ, X_I, X_P, w), outputs a PBP instance with the same answer (as defined in Definitions 4.3 and 4.4).

In this case, we have to add some assumptions about the cardinality of models. Condition 2 states that all models of the indicator-only part of the formula have the same cardinality. Bayesian network encodings such as `cd05` and `cd06` satisfy this condition by assigning an indicator variable to each possible variable-value pair and requiring each random variable to be paired with exactly one value. Condition 3 then says that the smallest number of parameter variables needed to turn an indicator-only model into a full model is the same for all indicator-only models. As some ideas duplicate between the proofs of Theorems 4.2 and 4.3, the following proof is slightly less explicit and assumes that $\omega = 1$.

Proof. Let (F, X_I, ω) be the tuple returned by Algorithm 4.1 and note that

$$F = \left\{ [c]_0^1 \mid c \in \phi, c \cap X_P = \emptyset \right\} \cup \left\{ \left[\bigwedge_{l \in c \setminus \{p\}} \neg l \right]_1^{w(p)} \;\middle|\; p \in X_P, p \in c \in \phi, c \neq \{p\} \right\}.$$

We split the proof into two parts. In the first part, we show that there is a bijection between minimum-cardinality models of ϕ and $Y \subseteq X_I$ such that $\left(\prod_{f \in F} f\right)(Y) \neq 0$.[6]

[6]For convenience and without loss of generality we assume that $w(p) \neq 0$ for all $p \in X_P$.

Let $Y \subseteq X_I$ and $Z \subseteq X_I \cup X_P$ be related via this bijection. Then in the second part we will show that

$$\prod_{Z \models l} w(l) = \left(\prod_{f \in F} f \right)(Y). \tag{4.5}$$

On the one hand, if $Z \subseteq X_I \cup X_P$ is a minimum-cardinality model of ϕ, then

$$\left(\prod_{f \in F} \right)(Z \cap X_I) \neq 0$$

under the given assumptions. On the other hand, if $Y \subseteq X_I$ is such that $\left(\prod_{f \in F}\right)(Y) \neq 0$, then $Y \models \{ c \in \phi \mid c \cap X_P = \emptyset \}$. Let $Y \subseteq Z \subseteq X_I \cup X_P$ be the smallest superset of Y such that $Z \models \phi$ (it exists by Condition 2c of Theorem 4.2). We need to show that Z has minimum cardinality. Let Y' and Z' be defined equivalently to Y and Z. We will show that $|Z| = |Z'|$. Note that $|Y| = |Y'|$ by Condition 2, and $|Z \setminus Y| = |Z' \setminus Y'|$ by Condition 3. Combining that with the general property that $|Z| = |Y| + |Z \setminus Y|$ finishes the first part of the proof.

For the second part, let us consider the multiplicative influence of a single parameter variable $p \in X_P$ on Equation (4.5). If the left-hand side is multiplied by $w(p)$ (i.e., $p \in Z$), then there must be some clause $c \in \phi$ such that $Z \setminus \{p\} \not\models c$. But then $Y \models \bigwedge_{l \in c \setminus \{p\}} \neg l$, and so the right-hand side is multiplied by $w(p)$ as well (exactly once because of Condition 2e of Theorem 4.2). This argument works in the other direction as well. □

4.6 Experimental Evaluation

We run a set of experiments, comparing all five original Bayesian network encodings (`bklm16`, `cd05`, `cd06`, `d02`, `sbk05`) as well as the first four with Algorithm 4.1 applied afterwards.[7] For each encoding `e`, we write `e++` to denote the combination of encoding a Bayesian network as a WMC instance using `e` and transforming it into a PBP instance using Algorithm 4.1. Along with DPMC[8], we also include WMC algorithms used in the papers that introduce each encoding: ACE for `cd05`, `cd06`, and `d02`; CACHET[9] [Sang et al., 2004] for `sbk05`; and C2D[10] [Darwiche, 2004] with QUERY-DNNF[11] for `bklm16`. ACE is also used to encode Bayesian networks into WMC instances for all

[7] Recall that `cd05` and `cd06` are incompatible with DPMC.

[8] `https://github.com/vardigroup/DPMC`

[9] `https://cs.rochester.edu/u/kautz/Cachet/`

[10] `http://reasoning.cs.ucla.edu/c2d/`

[11] `http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html`

encodings except for `bklm16` which uses another encoder mentioned previously. We focus on the following questions:

- Can parameter variable elimination improve inference speed?
- How does DPMC combined with encodings without (and with) parameter variables compare with other WMC algorithms and other encodings?
- Which instances is our approach particularly successful on (compared to other algorithms and encodings and to the same encoding before our transformation)?
- What proportion of variables is typically eliminated?
- Do some encodings benefit from this transformation more than others?

4.6.1 Setup

DPMC is run with tree decomposition-based planning and ADD-based execution—the best-performing combination in the original set of experiments [Dudek et al., 2020b]. We use a single iteration of HTD [Abseher et al., 2017] to generate approximately optimal tree decompositions—we found that this configuration is efficient enough to handle huge instances, and yet the width of the returned decomposition is unlikely to differ from optimal by more than one or two. We also enabled DPMC's greedy mode. This mode (which was not part of the original paper [Dudek et al., 2020b]) optimises the order in which ADDs are multiplied by prioritising those with small representations.

The experimental data and the protocol for choosing which probability to compute are as in Chapter 3. The experiments were run on a computing cluster with Intel Xeon E5-2630, Intel Xeon E7-4820, and Intel Xeon Gold 6138 processors with a 1000 s timeout separately on both encoding and inference, and a 32 GiB memory limit.[12]

4.6.2 Results

Figure 4.1 shows DPMC + `bklm16++` to be the best-performing combination across all time limits up to 1000 s with ACE + `cd06` and DPMC + `bklm16` not far behind. Overall, DPMC + `bklm16++` is 3.35 times faster than DPMC + `bklm16` and 2.96 times faster than ACE + `cd06`. Table 4.1 further shows that DPMC + `bklm16++` solves almost a

[12] Each instance was run on the same processor across all algorithms and encodings.

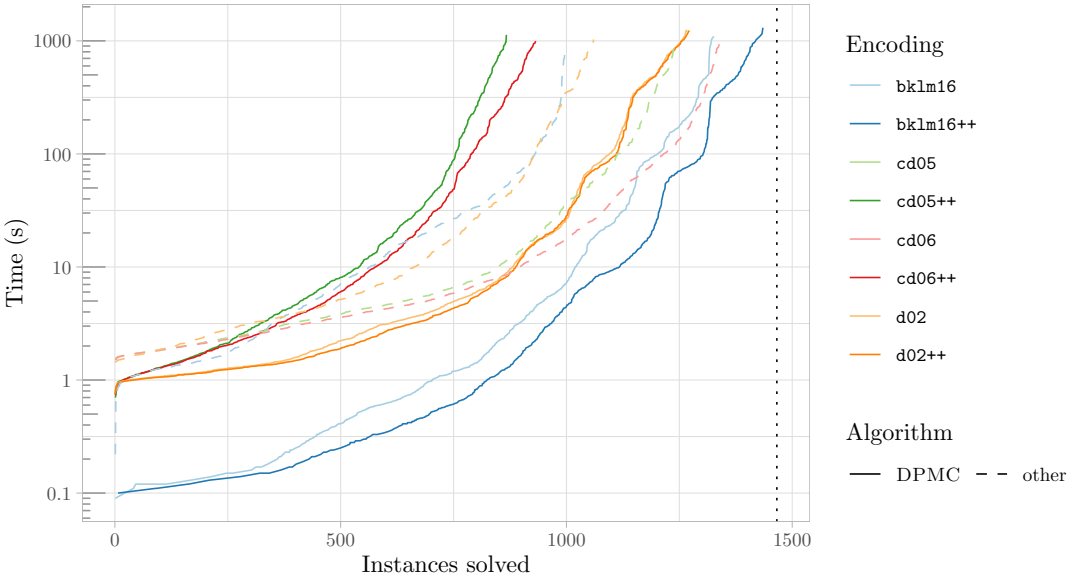


Figure 4.1: Cactus plot of all algorithm-encoding pairs. The dotted line denotes the total number of instances used.

hundred more instances than any other combination, and is the fastest in 69.1 % of them.

The scatter plots in Figure 4.2 show that how DPMC + `bklm16++` (and perhaps DPMC more generally) compares to ACE + `cd06` depends significantly on the data set: the former is a clear winner on DQMR and Grid instances, while the latter performs well on Mastermind and Random Blocks. Perhaps because the underlying WMC algorithm remains the same, the difference between DPMC + `bklm16` with and without applying Algorithm 4.1 is quite noisy, i.e, with most instances scattered around the line of equality. However, our transformation does enable DPMC to solve many instances that were previously beyond its reach.

We also record numbers of variables in each encoding before and after applying Algorithm 4.1. Figure 4.3 shows a significant reduction in the number of variables. For instance, the median number of variables in instances encoded with `bklm16` was reduced four times: from 1499 to 376. While `bklm16++` results in the overall lowest number of variables, the difference between `bklm16++` and `d02++` seems small. Indeed, the numbers of variables in these two encodings are equal for binary Bayesian networks (i.e., most of our data). Nonetheless, `bklm16++` is still much faster than `d02++` when run with DPMC.

It is also worth noting that there was no observable difference in the width of the project-join tree used by DPMC (which is equivalent to the primal treewidth of the input

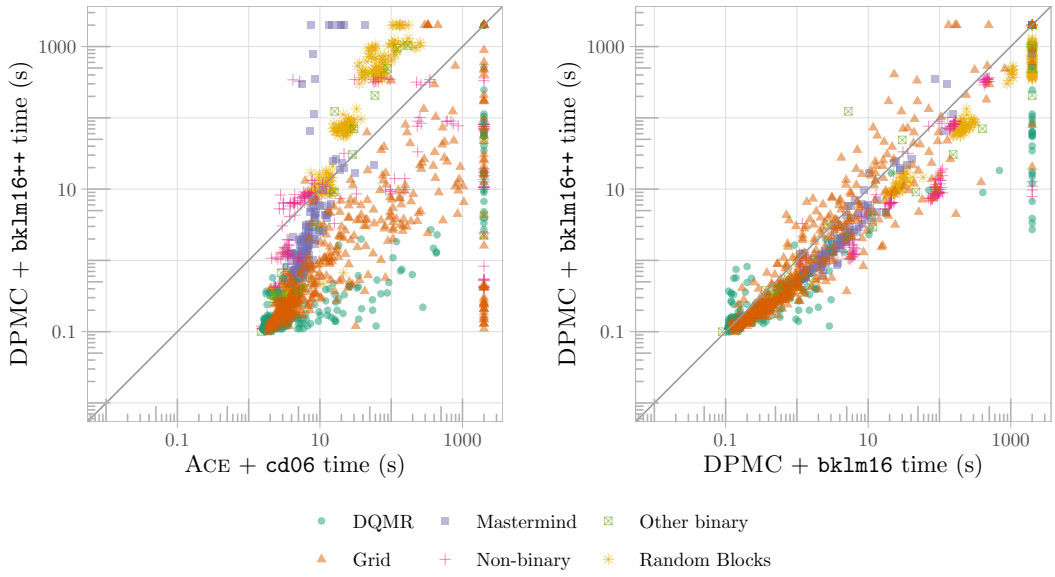


Figure 4.2: An instance-by-instance comparison between DPMC + bklm16++ (the best combination according to Figure 4.1) and the second and third best-performing combinations: ACE + cd06 and DPMC + bklm16.

formula [Dudek et al., 2020b]) before and after applying Algorithm 4.1—the observed performance improvement is more likely related to the variable ordering heuristic used by ADDs.[13]

Overall, transforming WMC instances to the PBP format allows us to significantly simplify each instance. This transformation is particularly effective on bklm16, allowing it to surpass cd06 and become the new state of the art. While there is a similarly significant reduction in the number of variables for d02, the performance of DPMC + d02 is virtually unaffected. Finally, while our transformation makes it possible to use cd05 and cd06 with DPMC, the two combinations remain inefficient.

4.7 Conclusion and Future Work

In this chapter, we showed how the number of variables in a WMC instance can be significantly reduced by transforming it into a representation based on two-valued pseudo-Boolean functions. In some cases, this led to significant improvements in inference speed, allowing DPMC + bklm16++ to overtake ACE + cd06 as the new state of the art WMC technique for Bayesian network inference. Moreover, we identified key

[13] The data on this (along with the implementation of Algorithm 4.1) is available at https://github.com/dilkas/wmc-without-parameters.

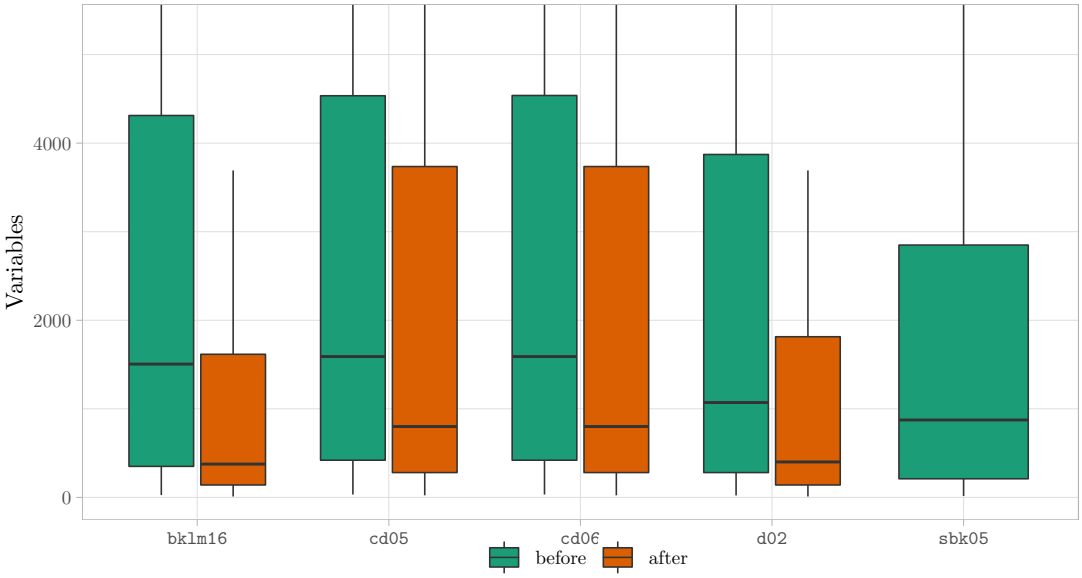


Figure 4.3: Box plots of the numbers of variables in each encoding across all benchmark instances before and after applying Algorithm 4.1. Outliers and the top parts of some whiskers are omitted.

properties of Bayesian network encodings that allow for parameter variable removal. However, these properties were rather different for each encoding, and so an interesting question for future work is whether they can be unified into a more abstract and coherent list of conditions.

Bayesian network inference was chosen as the example application of WMC because it is the first and the most studied one [Bart et al., 2016, Chavira and Darwiche, 2005, 2006, Darwiche, 2002, Sang et al., 2005a]. While the distinction between indicator and parameter variables is often not explicitly described in other WMC encodings [Fierens et al., 2015, Holtzen et al., 2020b, Xu et al., 2018], perhaps variables could still be partitioned in this way, allowing for not just faster inference with DPMC or ADDMC but also for well-established WMC encoding and inference techniques (such as in the work by Chavira and Darwiche [2005, 2006]) to be transferred to other application domains.

Similarly, could weighted first-order model counting (WFOMC) benefit from a more flexible approach to weights? The standard (‘symmetric’) definition of WFOMC assigns a pair of weights to each predicate [Van den Broeck et al., 2011]. Perhaps WFOMC could benefit from weights that depend on numerical constants in addition to predicates. This idea could also be seen as an extension of weighted first-order model integration [Feldstein and Belle, 2021] to discrete measurable spaces.

Combination	Fastest	Solved
ACE + cd05	27	1247
ACE + cd06	135	1340
ACE + d02	56	1060
DPMC + bklm16	241	1327
DPMC + bklm16++	**992**	**1435**
DPMC + cd05++	0	867
DPMC + cd06++	0	932
DPMC + d02	1	1267
DPMC + d02++	7	1272
DPMC + sbk05	31	1308
C2D + bklm16	0	997
CACHET + sbk05	49	983

Table 4.1: The numbers of instances (out of 1466) that each algorithm and encoding combination solved faster than any other combination and in total.

Lastly, we noted how the parameter equivalent to primal treewidth used by DPMC [Dudek et al., 2020b] remains virtually unchanged by our transformation despite a significant reduction in instance size. We also know that WMC (and hence WMC in the PBP format) is fixed-parameter tractable (FPT) with respect to primal treewidth (see Chapter 7 for a discussion on the parameterised complexity of WMC). WMC being FPT means that it admits *kernelisation*. In other words, any WMC instance can be transformed (in polynomial time) to an instance whose size depends only on the parameter (e.g., primal treewidth) and not on the size of the original instance [Downey and Fellows, 2013]. So, perhaps the WMC to PBP transformation presented in this chapter can be improved and formalised into a kernelisation algorithm for WMC instances within this more flexible PBP format. Kernelisation as well as the parameterised complexity angle on WMC more broadly is an underexplored area of research that we approach in more detail in Chapter 7.

Chapter 5

Recursive Solutions to FOMC

5.1 Introduction

First-order model counting (FOMC) is the problem of computing the number of models of a sentence in first-order logic (FOL) given the size(s) of its domain(s) [Beame et al., 2015]. *Symmetric weighted FOMC* (WFOMC) extends FOMC with (pairs of) weights on predicates and asks for a weighted sum across all models instead. WFOMC emerged as the dominant approach to *lifted (probabilistic) inference*. Lifted inference techniques exploit symmetries in probabilistic models by reasoning about sets rather than individuals [Kersting, 2012]. By doing so, many instances become solvable in polynomial time [Van den Broeck, 2011]. Lifted inference algorithms are typically used on probabilistic models such as probabilistic programming languages [De Raedt and Kimmig, 2015, Riguzzi et al., 2017], Markov logic networks [Van den Broeck et al., 2011, Gogate and Domingos, 2016, Richardson and Domingos, 2006], and other lifted graphical [Kimmig et al., 2015] and statistical relational [De Raedt et al., 2016] models. Lifted inference techniques for probabilistic databases, while developed somewhat independently, have also been inspired by WFOMC [Gatterbauer and Suciu, 2015, Gribkoff et al., 2014].

Traditionally in computational complexity theory, a problem is *tractable* if it can be solved in time polynomial in the size of the instance. The equivalent notion in (W)FOMC is liftability. A (W)FOMC instance is *(domain-)liftable* if it can be solved in time polynomial in the size(s) of the domain(s) [Jaeger and Van den Broeck, 2012]. Over more than a decade, more and more classes of instances were shown to be liftable [van Bremen and Kuzelka, 2021a, Kazemi et al., 2016, Kuusisto and Lutz, 2018, Kuzelka, 2021]. To begin with, we know that the class of all sentences of FOL with up to two

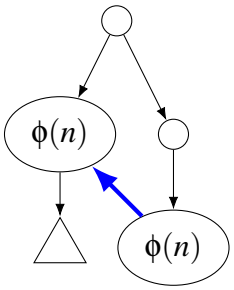
variables (denoted FO^2) is liftable [Van den Broeck, 2011]. We also know that there exists a sentence with three variables for which FOMC is $\#\mathsf{P}_1$-complete (i.e., FO^3 is not liftable) [Beame et al., 2015]. Since these two results came out, most of the research on (W)FOMC focused on developing faster solutions for the FO^2 fragment [Malhotra and Serafini, 2021, van Bremen and Kuzelka, 2021b] and defining new liftable fragments. These fragments include $\mathsf{S}^2\mathsf{FO}^2$ and $\mathsf{S}^2\mathsf{RU}$ [Kazemi et al., 2016], U_1 [Kuusisto and Lutz, 2018], FO^2 with tree axioms [van Bremen and Kuzelka, 2021a], and C^2 (i.e., the two variable fragment with counting quantifiers) [Kuzelka, 2021, Malhotra and Serafini, 2021]. On the empirical front, there are several open-source implementations of exact WFOMC algorithms: FORCLIFT [Van den Broeck et al., 2011], probabilistic theorem proving [Gogate and Domingos, 2016], and L2C [Kazemi and Poole, 2016]. Approximate counting is supported by APPROXMC3 [van Bremen and Kuzelka, 2020] as well as FORCLIFT [Van den Broeck et al., 2012] and probabilistic theorem proving [Gogate and Domingos, 2016].

However, none of the publicly available exact (W)FOMC algorithms can efficiently compute functions as simple as a factorial.[1] We claim that this shortcoming is due to the inability of these algorithms to construct recursive solutions. The topic of recursion in the context of WFOMC has been studied before but in limited ways. Barvínek et al. [2021] use WFOMC to generate numerical data which is then used to conjecture recurrence relations that explain that data. Van den Broeck [2011] introduced the idea of *domain recursion*. Intuitively, domain recursion partitions a domain of size n into a single explicitly named constant and the remaining domain of size $n-1$. However, many stringent conditions are enforced to ensure that the search for a tractable solution always terminates.

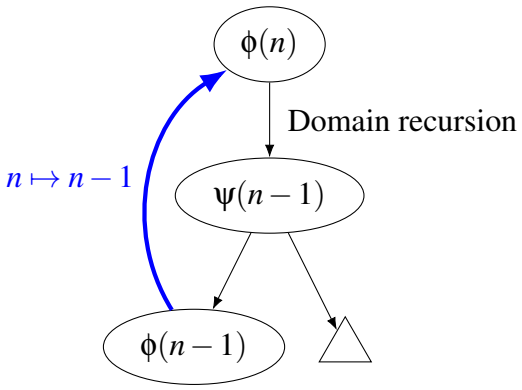
In this work, we show how to relax these restrictions in a way that results in a stronger (W)FOMC algorithm, capable of handling more instances in a lifted manner. The ideas presented in this chapter are implemented in CRANE—an extension of the arguably most well-known WFOMC algorithm FORCLIFT written in Scala. FORCLIFT works in two stages: compilation and evaluation/propagation.[2] In the first part, various *(compilation) rules* are applied to the input (or some derivative) formula, gradually constructing a circuit. In the second part, the weights of the instance (and sometimes

[1] The problem of computing the factorial can be described using two variables and counting quantifiers, so it is known to be liftable in theory [Kuzelka, 2021].

[2] There is also an intermediate stage called *smoothing* that takes place between compilation and evaluation. As our changes to smoothing are quite elementary, we do not discuss them to not distract the reader from the main contributions of this chapter.



(a) The kind of non-tree-like edges that are supported by both FORCLIFT and CRANE, i.e., those that create cycles in the undirected version of the graph but not directed cycles



(b) The kind of non-tree-like edges that are only supported by CRANE, i.e., those that create directed cycles

Figure 5.1: Non-tree-like edges in first-order knowledge compilation (highlighted in blue)

constants) are propagated through the circuit, computing the WMC.

The main conceptual difference between CRANE and FORCLIFT is that we utilise labelled directed graphs instead of circuits. The cycles in these graphs represent recursive calls. See Figure 5.1 for a conceptual illustration. An example scenario could work as follows. Suppose the original formula ϕ depends on a domain of size $n \in \mathbb{N}$. Our generalised version of domain recursion transforms ϕ into a different formula ψ that depends on a domain of size $n-1$. After some number of subsequent transformations, the algorithm identifies that a solution to ψ can be constructed in part by finding a solution to a version of ϕ where the domain of size n is replaced by a domain of size $n-1$. Recognising ϕ from before, we can add a cycle-forming edge to the graph (highlighted in blue in Figure 5.1b), which can be interpreted as function f relying on $f(n-1)$ to compute $f(n)$.

Once I finalise all of the sections and their order, write up a final introductory paragraph.

- To construct these graphs, FORCLIFT is...
- With additional compilation rules and an algorithm for checking whether a 'recursive call' is possible, FORCLIFT [Van den Broeck et al., 2011] is adapted to be able to construct recursive functions that efficiently solve counting problems that used to be beyond its reach.
- Three 'new' compilation rules (generalised domain recursion (introduce the acronym), constraint removal, and the thing that finds loops (identifying possibilities for recursion)) and a novel compilation algorithm (a hybrid of greedy and breadth-first search).
- What we do with the compilation result is different (see below)
- evaluation/interpretation: algebraic simplification, dynamic programming
- limitation: function extraction and simplification need to be automated, same for finding base cases

5.2 Preliminaries

In this section, we describe our format for FOMC instances, introduce some notation, and discuss our caching scheme, which is used to identify possibilities for a recursive call. Note that although the focus of this chapter is on unweighted model counting, FORCLIFT's [Van den Broeck et al., 2011] support for weights trivially transfers to CRANE as well.

Our representation of FOMC instances is heavily based on the format used internally by FORCLIFT, some aspects of which are described by Van den Broeck et al. [2011]. FORCLIFT is able to translate sentences in a variant of function-free many-sorted FOL with equality to this internal format. An *atom* is $p(t_1, \ldots, t_n)$ for some predicate p and terms $t_1, \ldots, t_n$. Here, $n \in \mathbb{N}_0$ is the *arity* of p. A *term* is either a constant or a variable. A *literal* is either an atom or the negation of an atom (denoted by $\neg p(t_1, \ldots, t_n)$). Let $\mathcal{D}$ be the set of all domains; note that this set expands during compilation.

Definition 5.1 (Constraint)**.** An *(inequality) constraint* is a pair (a, b), where a is a

variable, and b is either a variable or a constant.

Definition 5.2 (Clause)**.** A *clause*[3] is a triple $c = (L, C, \delta_c)$, where L is a set of literals, C is a set of constraints, and δ_c is the domain map of c. Let Vars be the function that maps clauses and sets of either literals or constraints to the set of variables contained within. In particular, $\text{Vars}(c) \coloneqq \text{Vars}(L) \cup \text{Vars}(C)$. *Domain map* $\delta_c \colon \text{Vars}(c) \to \mathcal{D}$ is a function that maps all variables in c to their domains such that (s.t.) if $(X, Y) \in C$ for some variables X and Y, then $\delta_c(X) = \delta_c(Y)$. For convenience, we sometimes write δ_c for the domain map of c without unpacking c into its three constituents.

Similarly to variables in Definition 5.2, all constants are (implicitly) mapped to domains, and each n-ary predicate is implicitly mapped to a sequence of n domains. For constant x, predicate p, and domains a and b, we write, e.g., $x \in a$ and $p \in a \times b$ to denote that x is associated with a, and p is associated with a and b (in that order).

Definition 5.3 (Formula)**.** A *formula* (called a c-theory by Van den Broeck et al. [2011]) is a set of clauses such that all constraints and atoms 'type check' with respect to domains.

Example 5.1. Let $\phi \coloneqq \{ c_1, c_2 \}$ be a formula with clauses

$$c_1 \coloneqq (\{ \neg p(X,Y), \neg p(X,Z) \}, \{ (Y,Z) \}, \{ X \mapsto a, Y \mapsto b, Z \mapsto b \}),$$
$$c_2 \coloneqq (\{ \neg p(X,Y), \neg p(Z,Y) \}, \{ (X,Z) \}, \{ X \mapsto a, Y \mapsto b, Z \mapsto a \})$$

for some predicate p, variables X, Y, Z, and domains a and b. Then $\text{Vars}(\{ (Y,Z) \}) = \{ Y, Z \}$, and $\text{Vars}(c_1) = \text{Vars}(c_2) = \{ X, Y, Z \}$. Based on the domain maps of c_1 and c_2, we can infer that $p \in a \times b$. All variables (in both clauses) that occur as the first argument to p are in a, and likewise all variables that occur as the second argument to p are in b. Therefore, ϕ 'type checks' as a valid formula.

There are two major differences between Definitions 5.1–5.3 and the corresponding concepts introduced by Van den Broeck et al. [2011]. First, we decouple variable-to-domain assignments from constraints and move them to a separate function δ_c in Definition 5.2. Formalising these assignments as a function unveils the (previously implicit) assumption that each variable must be assigned to a domain. Second, while Van den Broeck et al. [2011] allow for equality constraints and constraints of the form $X \notin d$ for some variable X and domain d, we exclude such constraints simply because they are not needed.

[3] Van den Broeck et al. [2011] refer to clauses as c-clauses.

One can read a formula as in Definition 5.3 as a sentence in a FOL. All variables in a clause are implicitly universally quantified (but note that variables are never shared among clauses), and all clauses in a formula are implicitly linked by conjunction. Thus, we can read formula ϕ from Example 5.1 as

$$(\forall X \in a.\ \forall Y \in b.\ \forall Z \in b.\ Y \neq Z \implies \neg p(X,Y) \vee \neg p(X,Z)) \wedge$$
$$(\forall X \in a.\ \forall Y \in b.\ \forall Z \in a.\ X \neq Z \implies \neg p(X,Y) \vee \neg p(Z,Y)).$$

Once domains are mapped to finite sets and constants to specific (and different) elements in those sets, a formula can be viewed as a set of conditions that the predicates (interpreted as relations) have to satisfy.[4] Hence, FOMC is the problem of counting the number of combinations of relations that satisfy these conditions.

Example 5.2. Let ϕ be as in Example 5.1 and let $|a| = |b| = 2$. There are $2^{2\times 2} = 16$ possible relations between a and b. Let us count how many of them satisfy the conditions imposed on predicate p. The empty relation does. All four relations of cardinality one do too. Finally, there are two relations of cardinality two that satisfy the conditions as well. Thus, the FOMC of ϕ (when $|a| = |b| = 2$) is 7. Incidentally, the FOMC of ϕ counts partial injections. We will continue to use the problem of counting partial injections (and the formula from Example 5.1 specifically) as the main running example throughout the chapter.

Notation for functions. We write $\to$ for functions, $\nrightarrow$ for partial functions, $\rightarrowtail\!\!\!\!\!\rightarrow$ for bijections, and $\hookrightarrow$ for set inclusion. Let id denote the identity function (on any domain). For any function f, let $\mathrm{dom}(f)$ be its domain, and $\mathrm{Im}\, f$ be its image.

Notation for lists. Let $\langle\rangle$ and $\langle x\rangle$ denote an empty list and a list with one element x, respectively. We write $\in$ for (in-order) enumeration, $+\!\!+$ for concatenation, and $|\cdot|$ for the length of a list. Let $h : t$ denote a list with first element (i.e., head) h and remaining list (i.e., tail) t. We also use list comprehensions written equivalently to set comprehensions. For example, let $L \coloneqq \langle 1\rangle$ and $M \coloneqq \langle 2\rangle$ be two lists. Then $M = \langle 2x \mid x \in L\rangle$, $L +\!\!+ M = 1 : \langle 2\rangle$, and $|M| = 1$.

Hashing. We use (integer-valued) hash functions to efficiently discard pairs of formulas that are too different for recursion to be established. The hash code of a clause

[4] If some domain is not big enough to contain all of its constants, the formula is unsatisfiable.

$c = (L, C, \delta)$ (denoted by $\#c$) combines the hash codes of the sets of constants and predicates in c, the numbers of positive and negative literals, the number of inequality constraints $|C|$, and the number of variables $|\operatorname{Vars}(c)|$. The hash code of a formula ϕ combines the hash codes of all its clauses and is denoted by $\#\phi$.

Caching. Van den Broeck et al. [2011] use a cache to check if a formula is identical to one of the formulas that have already been fully compiled. If that is the case, then the circuit already contains the subcircuit for this formula. Instead of duplicating this subcircuit, one would draw an edge that creates an undirected (but not a directed) cycle (as in Figure 5.1a). In CRANE, to facilitate recursion, we extend the caching scheme to include formulas that have been encountered but not fully compiled yet. Hence, the same procedure can now create directed cycles in the FCG. Formally, we define a *cache* to be a map from integers (e.g., hash codes) to sets of pairs of the form (ϕ, v), where ϕ is a formula, and v is an FCG node.

5.3 Methods

We begin this section by formally defining the graphs that CRANE uses as a generalisation of circuits. Then, Section 5.3.1 describes three new compilation rules, and Section 5.3.2 outlines the hybrid search algorithm that replaces the greedy search used by FORCLIFT [Van den Broeck et al., 2011].

A *first-order deterministic decomposable negation normal form computational graph* (FCG) is a (weakly connected) directed graph with a single source, node labels, and ordered outgoing edges.[5] We denote an FCG as $G = (V, s, N^+, \tau)$, where V is the set of nodes, and $s \in V$ is the unique source. Function N^+ maps each node in V to a *list* of its direct successors. Node labels consist of two parts: the *type* and the *parameters*. To avoid clutter, we leave the parameters implicit and let τ denote the node-labelling function that maps each node in V to its type. For each node $v \in V$, the length of list $N^+(v)$ (i.e., the out-degree of v) is determined by its type $\tau(v)$. Most of the types are as in previous work [Van den Broeck, 2011, Van den Broeck et al., 2011]. The type for non-tree-like edges (denoted by REF), while used before, is extended to contain the information necessary to support recursive calls. We also add three new types:

- a type for constraint removal denoted by CR,

[5] Note that imposing an ordering on outgoing edges is just a limited version of edge labelling.

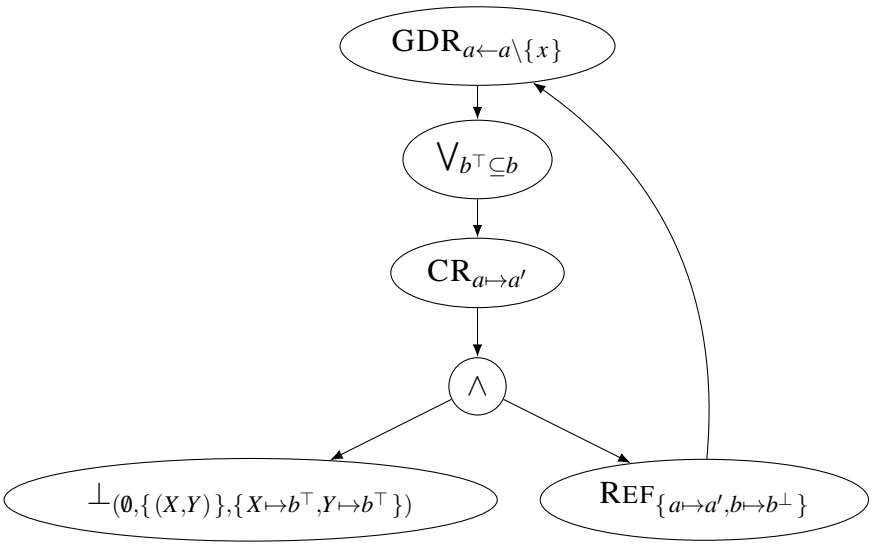


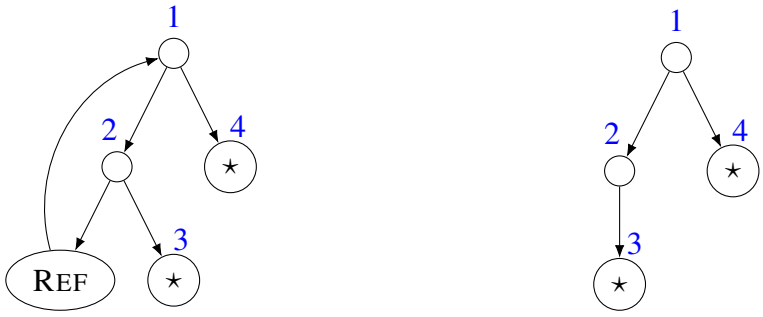
Figure 5.2: A simplified version of an FCG constructed by CRANE for the problem of counting partial injections from Example 5.1. Label $\bigvee_{b^\top\subseteq b}$ denotes set-disjunction, $\wedge$ denotes conjunction, and $\bot$ denotes a contradiction—see the work by Van den Broeck et al. [2011] for the descriptions of these node types. Here we omit nodes whose only arithmetic effect is multiplication by one. Some of these nodes play an important role in the weighted version of the problem whereas others are remnants of the interaction between compilation rules and the way in which FORCLIFT handles existential quantifiers.

- a type for GDR denoted by GDR (both with out-degree one),
- and $\star$—a placeholder type (with out-degree zero) for nodes that are going to be replaced.

When drawing an FCG, we order outgoing edges from left to right, write node labels directly on the nodes, and omit irrelevant labels and/or parameters. See Figure 5.2 for an example FCG. Its source node has out-degree 1 (i.e., $|N^+(s)| = 1$), label GDR$_{a\leftarrow a\setminus\{x\}}$, and type GDR (i.e., $\tau(s) = \text{GDR}$).

Similarly to Van den Broeck et al. [2011], we write T_p for an FCG that has a node with label T_p (i.e., type T and parameter(s) p) and $\star$'s as all of its direct successors. In particular, as an FCG, $\star$ denotes $(\{s\}, s, \{s \mapsto \langle\rangle\}, \{s \mapsto \star\})$, i.e., an FCG with just one node of type $\star$ and no edges. We write $T_p(v)$ for an FCG with one edge from a node labelled T_p to some other node v (and no other nodes or edges).

Finally, we introduce a structure that represents a solution to a (W)FOMC problem while it is still being built. A *chip* is a pair (G, L), where G is an FCG, and L is a list of formulas, such that $|L|$ is equal to the number of $\star$'s in G. The intuition behind this



(a) An example FCG (b) The underlying tree of Figure 5.3a

Figure 5.3: An FCG and its underlying tree. The integers in blue denote the pre-order traversal of the underlying tree.

definition is that L contains formulas that still need to be compiled. Once a formula is compiled, it replaced one of the $\star$'s in G. We say that an FCG is *complete* (i.e., it represents a *complete solution*) if it has no $\star$'s. Similarly, a chip is complete if its FCG is complete (or, equivalently, the list of formulas is empty). Let the *underlying tree* of G be the induced subgraph of G that omits all REF nodes.[6] Then we can define an implicit bijection between the formulas in L and the $\star$'s in G according to the order in which elements of L are listed and the pre-order traversal of the underlying tree of G. For example, if G is as in Figure 5.3a (with its underlying tree in Figure 5.3b), then $|L| = 2$. Moreover, the first element of L is associated with the $\star$ labelled 3, and the second with the one labelled 4.

5.3.1 New Compilation Rules

A *(compilation) rule* takes a formula and returns a set of chips. The cardinality of this set is the number of different ways in which the rule can be applied to the input formula. While FORCLIFT [Van den Broeck et al., 2011] heuristically chooses one of them, in an attempt to not miss a solution, CRANE returns them all. In particular, if a rule returns an empty set, then that rule is not applicable to the formula, and the algorithm continues with the next rule.

5.3.1.1 Generalised Domain Recursion

Notation. Let S be a set of constraints or literals, V a set of variables, and x either a variable or a constant. We write $S[x/V]$ to denote S with all occurrences of all variables

[6] Subsequently presented algorithms ensure that the underlying tree is guaranteed to be a tree.

in V replaced with x.[7]

The main idea behind domain recursion (both the original version by Van den Broeck [2011] and the one presented here) is as follows. Let $d \in \mathcal{D}$ be a domain. Assuming that $d \neq \emptyset$, pick some $x \in d$. Then, for every variable X associated with domain d that occurs in a literal, consider two possibilities: $X = x$ and $X \neq x$.

Example 5.3. Consider formula ϕ that consists of a single clause

$$(\{\neg p(X,Y), \neg p(X,Z)\}, \{(Y,Z)\}, \{X \mapsto a, Y \mapsto b, Z \mapsto b\}).$$

Then we can introduce constant $x \in a$ and rewrite ϕ as $\phi' = \{c_1, c_2\}$, where

$$c_1 = (\{\neg p(x,Y), \neg p(x,Z)\}, \{(Y,Z)\}, \{Y \mapsto b, Z \mapsto b\}),$$
$$c_2 = (\{\neg p(X,Y), \neg p(X,Z)\}, \{(X,x), (Y,Z)\}, \{X \mapsto a', Y \mapsto b, Z \mapsto b\}),$$

and $a' = a \setminus \{x\}$.

Van den Broeck [2011] imposes stringent preconditions on the input formula to ensure that the expanded version of the formula (as in Example 5.3) can be handled efficiently. The clauses in this expanded formula are then partitioned into three parts based on whether the transformation introduced constants or constraints or both. The aforementioned conditions ensure that these parts can be treated independently.

In contrast, GDR has only one precondition: for GDR to be applicable on domain $d \in \mathcal{D}$, there must be at least one variable with domain d that is featured in a literal (and not just in constraints). Without such variables, GDR would have no effect on the formula. GDR is also simpler in that the expanded formula is left as-is to be handled by other compilation rules. Typically, after a few more rules are applied, a combination of CR and REF nodes introduces a loop back to the GDR node, thus completing the definition of a recursive function. The GDR compilation rule is summarised as Algorithm 5.1 and explained in more detail using the example below.

Example 5.4. Let $\phi := \{c_1, c_2\}$ be the formula from Example 5.1 with clauses

$$c_1 = (\{\neg p(X,Y), \neg p(X,Z)\}, \{(Y,Z)\}, \{X \mapsto a, Y \mapsto b, Z \mapsto b\}),$$
$$c_2 = (\{\neg p(X,Y), \neg p(Z,Y)\}, \{(X,Z)\}, \{X \mapsto a, Y \mapsto b, Z \mapsto a\}).$$

While GDR is possible on both domains, here we illustrate how it works on a. Having chosen a domain, the algorithm iterates over the clauses of ϕ. Suppose line 5 picks

[7] Note that if (X,Y) is a two-variable constraint, substituting a constant c for X would result in (c,Y), which would have to be rewritten as (Y,c) to fit the definition of a constraint.

Algorithm 5.1: The compilation rule for GDR nodes.

Input: formula ϕ

Output: set of chips S

1 $S \leftarrow \emptyset$;

2 **foreach** *domain* $d \in \mathcal{D}$ *s.t. there is* $c \in \phi$ *and* $v \in \mathrm{Vars}(L_c)$ *s.t.* $\delta_c(v) = d$ **do**

3 $\quad \phi' \leftarrow \emptyset$;

4 $\quad x \leftarrow$ a new constant in domain d;

5 $\quad$ **foreach** *clause* $c = (L, C, \delta) \in \phi$ **do**

6 $\quad\quad V \leftarrow \{ v \in \mathrm{Vars}(L) \mid \delta(v) = d \}$;

7 $\quad\quad$ **foreach** *subset* $W \subseteq V$ *s.t.* $W^2 \cap C = \emptyset$ **and** $W \cap \{ v \in \mathrm{Vars}(C) \mid (v,y) \in C$ *for some constant* $y \} = \emptyset$ **do**

$\quad\quad\quad$ /* δ' restricts δ to the new set of variables */

8 $\quad\quad\quad \phi' \leftarrow \phi' \cup \{ (L[x/W], C[x/W] \cup \{ (v,x) \mid (v \in V \setminus W) \}, \delta') \}$;

9 $\quad S \leftarrow S \cup \{ (\mathrm{GDR}_{d \leftarrow d \setminus \{x\}}, \langle \phi' \rangle) \}$;

$c = c_1$ as the first clause. Then, set V is constructed to contain all variables with domain $d = a$ that occur in the literals of clause c. In this case, $V = \{ X \}$.

Line 7 iterates over all subsets $W \subseteq V$ of variables that can be replaced by a constant without resulting in formulas that are evidently unsatisfiable. We impose two restrictions on W. First, $W^2 \cap C = \emptyset$ ensures that there are no pairs of variables in W that are constrained to be distinct, since that would result in a $x \neq x$ constraint after substitution. Similarly, we want to avoid variables in W that have inequality constraints with constants: after substitution, such constraints would transform into inequality constraints between two constants. In this case, both subsets of V satisfy these conditions, and line 8 generates two clauses for the output formula:

$$(\{ \neg p(X,Y), \neg p(X,Z) \}, \{ (Y,Z), (X,x) \}, \{ X \mapsto a, Y \mapsto b, Z \mapsto b \}),$$

from $W = \emptyset$ and

$$(\{ \neg p(x,Y), \neg p(x,Z) \}, \{ (Y,Z) \}, \{ Y \mapsto b, Z \mapsto b \})$$

from $W = V$.

When line 5 picks $c = c_2$, then $V = \{ X, Z \}$. The subset $W = V$ fails to satisfy the conditions on line 7 because of the $X \neq Z$ constraint. The other three subsets of V all generate clauses for ϕ'. Indeed, $W = \emptyset$ generates

$$(\{ \neg p(X,Y), \neg p(Z,Y) \}, \{ (X,Z), (X,x), (Z,x) \}, \{ X \mapsto a, Y \mapsto b, Z \mapsto a \}),$$

Algorithm 5.2: The compilation rule for CR nodes.

Input: formula ϕ, set of domains $\mathcal{D}$

Output: set of chips S

1 $S \leftarrow \emptyset$;

2 **foreach** *domain* $d \in \mathcal{D}$ *and element* $x \in d$ *s.t.* x *does not occur in any literal of any clause of* ϕ **and** *for each clause* $c = (L, C, \delta_c) \in \phi$ *and variable* $v \in \text{Vars}(c)$, *either* $\delta_c(v) \neq d$ *or* $(v, x) \in C$ **do**

3 add a new domain d' to $\mathcal{D}$;

4 $\phi' \leftarrow \emptyset$;

5 **foreach** *clause* $(L, C, \delta) \in \phi$ **do**

6 $C' \leftarrow \{ (a, b) \in C \mid b \neq x \}$;

7 $\delta' \leftarrow v \mapsto \begin{cases} d' & \text{if} \delta(v) = d \\ \delta(v) & \text{otherwise;} \end{cases}$

8 $\phi' \leftarrow \phi' \cup \{ (L, C', \delta') \}$

9 $S \leftarrow S \cup \{ (\text{CR}_{d \mapsto d'}, \langle \phi' \rangle) \}$;

$W = \{X\}$ generates

$$(\{\neg p(x,Y), \neg p(Z,Y)\}, \{(Z,x)\}, \{Y \mapsto b, Z \mapsto a\}),$$

and $W = \{Z\}$ generates

$$(\{\neg p(X,Y), \neg p(x,Y)\}, \{(X,x)\}, \{X \mapsto a, Y \mapsto b\}).$$

5.3.1.2 Constraint Removal

Recall that GDR on a domain d creates constraints of the form $X_i \neq x$ for some constant $x \in d$ and family of variables $X_i \in d$. Once certain conditions are satisfied, Algorithm 5.2 can eliminate these constraints and replace d with a new domain d', which can be interpreted as $d \setminus \{x\}$. These conditions (on line 2 of the algorithm) are that a constraint of the form $X \neq e$ exists for all variables $X \in d$ across all clauses, and such constraints are the only place where e occurs. The algorithm then proceeds to construct the new formula by removing constraints (on line 6) and constructing a new domain map δ' that replaces d with d' (on line 7).

Example 5.5. Let $\phi = \{c_1, c_2, c_3\}$ be a formula with clauses

$$c_1 = (\emptyset, \{(Y,X)\}, \{X \mapsto b^\top, Y \mapsto b^\top\}),$$
$$c_2 = (\{\neg p(X,Y), \neg p(X,Z)\}, \{(X,x),(Y,Z)\}, \{X \mapsto a, Y \mapsto b^\bot, Z \mapsto b^\bot\}),$$
$$c_3 = (\{\neg p(X,Y), \neg p(Z,Y)\}, \{(X,x),(Z,X),(Z,x)\}, \{X \mapsto a, Y \mapsto b^\bot, Z \mapsto a\}).$$

Domain a and with its element $x \in a$ satisfy the preconditions for constraint removal. The rule introduces a new domain a' and transforms ϕ to $\phi' = (c'_1, c'_2, c'_3)$, where

$$c'_1 = c_1$$
$$c'_2 = (\{\neg p(X,Y), \neg p(X,Z)\}, \{(Y,Z)\}, \{X \mapsto a', Y \mapsto b^\bot, Z \mapsto b^\bot\})$$
$$c'_3 = (\{\neg p(X,Y), \neg p(Z,Y)\}, \{(Z,X)\}, \{X \mapsto a', Y \mapsto b^\bot, Z \mapsto a'\}).$$

5.3.1.3 Identifying Opportunities for Recursion

Notation. First, for partial functions $\alpha, \beta \colon A \nrightarrow B$ s.t. $\alpha|_{\mathrm{dom}(\alpha)\cap\mathrm{dom}(\beta)} = \beta|_{\mathrm{dom}(\alpha)\cap\mathrm{dom}(\beta)}$, we write $\alpha \cup \beta$ for the unique partial function s.t. $\alpha\cup\beta|_{\mathrm{dom}(\alpha)} = \alpha$, and $\alpha\cup\beta|_{\mathrm{dom}(\beta)} = \beta$. Second, let Doms be a function that maps any clause or formula to the set of domains used within. Specifically, $\mathrm{Doms}(c) \coloneqq \mathrm{Im}\,\delta_c$ for any clause c, and $\mathrm{Doms}(\phi) \coloneqq \bigcup_{c\in\phi} \mathrm{Doms}(c)$ for any formula ϕ. Third, for any clause $c = (L, C, \delta_c)$, bijection $\beta \colon \mathrm{Vars}(c) \rightarrowtail\!\!\!\!\!\twoheadrightarrow V$ (for some set of variables V), and function $\gamma \colon \mathrm{Doms}(c) \to \mathcal{D}$, let $c[\beta,\gamma] = d$ be the clause c with all occurrences of any variable $v \in \mathrm{Vars}(c)$ in L and C replaced with $\beta(v)$ (so $\mathrm{Vars}(d) = V$) and $\delta_d \colon V \to \mathcal{D}$ defined as $\delta_d \coloneqq \gamma \circ \delta_c \circ \beta^{-1}$. In other words, δ_d is the unique function that makes

$$\begin{array}{ccc} \mathrm{Vars}(c) & \overset{\beta}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} & V = \mathrm{Vars}(d) \\ {\scriptstyle \delta_c}\downarrow & & \downarrow{\scriptstyle \exists!\delta_d} \\ \mathrm{Doms}(c) & \xrightarrow[\gamma]{} & \mathcal{D} \end{array}$$

commute. For example, if clause

$$c_1 \coloneqq (\{\neg p(X,Y), \neg p(X,Z)\}, \{(Y,Z)\}, \{X \mapsto a, Y \mapsto b, Z \mapsto b\})$$

is as in Example 5.1, then

$$c_1[\{X \mapsto A, Y \mapsto B, Z \mapsto C\}, \{a \mapsto b, b \mapsto c\}] = (\{\neg p(A,B), \neg p(A,C)\}, \{(B,C)\}, \{A \mapsto b, B \mapsto c, C \mapsto c\}).$$

Algorithm 5.3 describes the compilation rule for creating REF nodes. For every formula ψ s.t. $\#\psi = \#\phi$ that we have encountered so far, function `identifyRecursion` is called to check whether a recursive call is actually feasible. If it is, the function returns a (total) map $\rho\colon \mathrm{Doms}(\psi) \to \mathrm{Doms}(\phi)$ that shows how ψ can be transformed into ϕ by replacing each domain $d \in \mathrm{Doms}(\psi)$ with $\rho(d) \in \mathrm{Doms}(\phi)$. Otherwise, `identifyRecursion` returns `null` to signify that ϕ and ψ are too different for recursion to work. This happens if ϕ and ψ (or their subformulas explored in recursive calls) are structurally different (i.e., the numbers of clauses or the hash codes fail to match) or if a clause of ψ cannot be paired with a sufficiently similar clause of ϕ.

Function `identifyRecursion` iterates over pairs of clauses of ϕ and ψ that have the same hash codes. It uses function `generateMaps` to check whether the two clauses are sufficiently similar. If so, the function calls itself on the remaining clauses until the map $\rho\colon \mathrm{Doms}(\psi) \nrightarrow \mathrm{Doms}(\phi)$ becomes total, and all clauses are successfully coupled.

Given two clauses $c \in \psi$ and $d \in \phi$, `generateMaps` considers all possible bijections[8] $\beta\colon \mathrm{Vars}(c) \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathrm{Vars}(d)$ and calls `constructDomainMap`, which then attempts to construct a map $\gamma\colon \mathrm{Doms}(c) \to \mathrm{Doms}(d)$ consistent with both β and (the as yet partial map) $\rho\colon \mathrm{Doms}(\psi) \nrightarrow \mathrm{Doms}(\phi)$. The **yield** keyword in `generateMaps` works as in programming languages such as C#, JavaScript, and Python, and lazily returns a sequence of values, computing each element of the sequence as needed.

Diagrammatically, `constructDomainMap` attempts to find a $\gamma\colon \mathrm{Doms}(c) \to \mathrm{Doms}(d)$ s.t.

$$\begin{array}{ccc} V = \mathrm{Vars}(c) & \overset{\beta}{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} & \mathrm{Vars}(d) \\ \Big\downarrow{\scriptstyle \delta_c} & & \Big\downarrow{\scriptstyle \delta_d} \\ \mathrm{Doms}(c) & \overset{\gamma}{\dashrightarrow} & \mathrm{Doms}(d) \\ \Big\downarrow & & \Big\downarrow \\ \mathrm{Doms}(\psi) & \underset{\rho}{\longrightarrow\!\!\!\!\!\!\mid} & \mathrm{Doms}(\phi). \end{array} \tag{5.1}$$

commutes (and returns `null` if such a function does not exist). Indeed, for every variable in $V = \mathrm{Vars}(c)$, the function returns `null` if either the top rectangle from V to $\mathrm{Doms}(d)$ or the outer rectangle from V to $\mathrm{Doms}(\phi)$ fails to commute. These checks also ensure that $\rho \cup \gamma$ is possible on line 11 of the algorithm, i.e., $\rho|_{\mathrm{dom}(\rho)\cap\mathrm{dom}(\gamma)} = \gamma|_{\mathrm{dom}(\rho)\cap\mathrm{dom}(\gamma)}$.

[8] Although the number of bijections between two sets of cardinality n is $n!$, this part of the algorithm is unlikely to cause performance issues for two reasons. First, in practice, n is usually at most two or three. Second, due to the manner in which formulas are modified by compilation rules, if any bijection results in a successfully identified recursive relationship, it is almost always the identity bijection.

Example 5.6. As in Example 5.1, let $\psi := \{c_1, c_2\}$ be a formula with clauses

$$c_1 = (\{\neg p(X,Y), \neg p(X,Z)\}, \{(Y,Z)\}, \{X \mapsto a, Y \mapsto b, Z \mapsto b\}),$$
$$c_2 = (\{\neg p(X,Y), \neg p(Z,Y)\}, \{(X,Z)\}, \{X \mapsto a, Y \mapsto b, Z \mapsto a\}).$$

Let formula $\phi := \psi[\mathrm{id}, \{a \mapsto a', b \mapsto b^\perp\}]$ be just like ψ but with different domains. In other words, $\phi = \{d_1, d_2\}$, where

$$d_1 := (\{\neg p(X,Y), \neg p(X,Z)\}, \{(Y,Z)\}, \{X \mapsto a', Y \mapsto b^\perp, Z \mapsto b^\perp\}),$$
$$d_2 := (\{\neg p(X,Y), \neg p(Z,Y)\}, \{(X,Z)\}, \{X \mapsto a', Y \mapsto b^\perp, Z \mapsto a'\}).$$

Note that $\#\phi = \#\psi$ and assume that $(\psi, v) \in C(\#\phi)$ for some node v. We shall see how Algorithm 5.3 identifies that the FCG for ψ can be reused for ϕ as well.

Since both formulas are non-empty, the algorithm proceeds with the for-loops on lines 8–10. Suppose $c = c_1$ and $d = d_1$ get picked. Since both clauses have three variables, in the worst case, function `generateMaps` would have $3! = 6$ bijections to check. Suppose the identity bijection is picked first. Then `constructDomainMap` is called with the following parameters:

- $V = \{X, Y, Z\}$,
- $\delta_c = \{X \mapsto a, Y \mapsto b, Z \mapsto b\}$,
- $\delta_d = \{X \mapsto a', Y \mapsto b^\perp, Z \mapsto b^\perp\}$,
- $\beta = \mathrm{id} = \{X \mapsto X, Y \mapsto Y, Z \mapsto Z\}$,
- $\rho = \emptyset$.

Since $\delta_c(Y) = \delta_c(Z)$, and $\delta_d(Y) = \delta_d(Z)$, `constructDomainMap` returns $\gamma = \{a \mapsto a', b \mapsto b^\perp\}$. Thus, `generateMaps` yields its first pair of maps (β, γ) to line 10. Furthermore, this pair satisfies $c[\beta, \gamma] = d$. On line 11, a recursive call to `identifyRecursion` $(\{c_2\}, \{d_2\}, \gamma)$ is made.

In this subproblem where both formulas are left with a single clause each, again we have two non-empty formulas with equal hash codes. Thus `generateMaps` is called with $c = c_2$, $d = d_2$, and $\rho = \{a \mapsto a', b \mapsto b^\perp\}$. Suppose line 15 picks the identity bijection gain. Then `constructDomainMap` is called with the following parameters:

- $V = \{X, Y, Z\}$,
- $\delta_c = \{X \mapsto a, Y \mapsto b, Z \mapsto a\}$,

- $\delta_d = \{X \mapsto a', Y \mapsto b^{\perp}, Z \mapsto a'\}$,
- $\beta = \{X \mapsto X, Y \mapsto Y, Z \mapsto Z\}$,
- $\rho = \{a \mapsto a', b \mapsto b^{\perp}\}$.

Since β and ρ commute (as in Diagram 5.1), and there are no new domains in $\text{Doms}(c)$ and $\text{Doms}(d)$, γ exists and is equal to ρ. Again, the returned pair (β, γ) satisfies $c[\beta, \gamma] = d$. Line 11 calls `identifyRecursion(∅, ∅, ρ)`, which immediately returns $\rho = \{a \mapsto a', b \mapsto b^{\perp}\}$ as the final answer. Therefore, one can indeed reuse an FCG for ψ to compute the model count of ϕ.

5.3.2 Compilation as Search

Given a formula ϕ, we want to find an FCG that encodes a way to compute the model count of ϕ. While FORCLIFT [Van den Broeck et al., 2011] uses greedy search[9], CRANE has a new search algorithm—a combination of greedy and breadth-first search.

We split all compilation rules into *greedy* and *non-greedy*. Greedy rules represent indisputable choices in the compilation process. They are applied to each encountered formula as soon and as many times as possible (in a predefined order). Most rules are greedy, i.e., those that produce a sink node with no leftover formula, those that simplify the formula without changing the FCG, and those that split the formula into parts that can be solved independently. The constraint removal rule described in Section 5.3.1.2 is greedy. On the other hand, non-greedy rules signify uncertain choices that we may want to retract. They also correspond to edges in the implicit search tree; thus, the first solution found by the search algorithm always has the fewest applications of non-greedy rules. These rules include the GDR and REF rules described in Sections 5.3.1.1 and 5.3.1.3, respectively, and some rules from previous work [Van den Broeck et al., 2011] such as atom counting, inclusion-exclusion, independent partial grounding, and shattering.

Search can be conceptualised as traversing a directed graph composed of states and actions that lead from one state to the next. We define a *state* as a triple (G, L, C), where (G, L) is a chip, and C is a cache. The actions that can be taken in such a state are applications of compilation rules that remove the first formula from L and potentially add something to G, L, and C.

[9]The algorithm is not described in any paper on FORCLIFT but can be found in its source code at `https://dtai.cs.kuleuven.be/drupal/wfomc`

The search algorithm is described as Algorithms 5.4–5.7, with the main procedure in Algorithm 5.4. Since for most formulas we are able to find several FCGs (of various complexities), the algorithm maintains set S of found solutions, i.e., complete FCGs. We begin by applying all suitable greedy rules on line 2. If greedy rules are enough to find a complete FCG, the algorithm stops. Otherwise, lines 5 and 6 set up a queue for breadth-first search. The algorithm continues to take a state from the queue, call `applyAllRules` on it, and place the resulting states back on the queue while filtering out complete FCGs and adding them to S instead. Since GDR can be applied to almost all formulas, the search is infinite. In our implementation, we stop searching when one of the following conditions is satisfied: (*a*) the desired number of solutions is found, (*b*) the search tree reaches a certain height, or (*c*) the algorithm times out.

Function `applyAllRules` (see Algorithm 5.5) takes a state and generates a sequence of new states created by applying one non-greedy rule followed by all applicable greedy rules. We assume that the input state contains at least one formula (otherwise it would be a complete solution) and that all applicable greedy rules have already been applied. The algorithm iterates over all non-greedy rules and all chips generated by these rules when applied to ϕ. If the FCG is $\star$, then $|L''| = 1$, and L'' contains a modified version of ϕ—in this case, we rerun `applyAllRules` on the same state but with the updated formula. Otherwise, we update the cache, call another function to apply greedy rules, and merge (G'', L'') with a copy of the input state. In doing so, ϕ is replaced by L'', preserving the implicit bijection between the ordering of the list and the structure of $\star$'s in the FCG.

Algorithm 5.6 defines two functions that work together to handle the application of greedy rules. Function `applyGreedyRules` takes a formula and returns the maximal chip that can be constructed by the application of greedy rules (and updates the cache whenever the application of a rule results in a new node). The implementation of this function is simplified by three assumptions about greedy rules. First, we assume that greedy rules can be applied in any order. Second, we assume that the set of chips returned by a greedy rule has at most one element. Third, if a rule returns an empty FCG, i.e., $G = \star$, then L has exactly one formula, i.e., the rule simply transforms the input formula. In such a case, we continue the application of greedy rules to the new formula. Otherwise, line 5 updates the cache and calls the second function in Algorithm 5.6, `applyGreedyRulesToFormulas`, on the new chip (G, L) that has $|L|$ new formulas that could benefit from greedy rules. Finally, if none of the greedy rules are applicable, `applyGreedyRules` returns the same formula ϕ formatted as a

state. Function `applyGreedyRulesToFormulas` takes a state and updates it by running `applyGreedyRules` on all formulas in L and incorporating the resulting chips as direct successors of the source node s. Hence, we assume that the input FCG (which comes directly from applying a single greedy rule) has just one non-$\star$ node, s, and $N^+(s)$ contains exactly $|L|$ $\star$'s.

Finally, Algorithm 5.7 describes two helper functions: `updCache` for updating the cache and `mergeFcgs` for merging two FCGs. Note that REF nodes are not placed in the cache because the relation identified by the compilation rule for REF (see Algorithm 5.3) is transitive. In other words, instead of calling a function $f(\mathbf{n}) \coloneqq g(\mathbf{m})$ (where $\mathbf{n}$ and $\mathbf{m}$ are integer parameter vectors, and $\mathbf{m}$ is constructed from $\mathbf{n}$), we can always directly call function g. Function `mergeFcgs` finds a $\star$ in G and replaces it with G'. Note that the order in which the nodes of G are visited must be the pre-order traversal of the underlying tree of G. Hence, the algorithm skips REF nodes and, for each directed edge, considers the source before considering the target. Parameter r, initially set to the source of G, keeps track of the root of the subtree that needs to be explored. Recursive calls return `null` if there are no $\star$'s in the subtree rooted at r. However, we only call `mergeFcgs` with G's that have at least one $\star$, so the return value of the initial call to the function is never `null`.

5.4 How to Interpret an FCG

When FORCLIFT [Van den Broeck et al., 2011] compiles a WFOMC instance into a circuit, each gate type encodes an arithmetic operation on its inputs and parameters. These operations are then immediately performed while traversing the circuit and using domain sizes and weights as the initial inputs. With CRANE, the interpretation of an FCG is a collection of functions. Each function has (some) domain sizes as parameters and may contain recursive calls to other functions, including itself. While there may be any number of subsidiary functions, there is always one main function that can be called with the sizes of the domains of the input formula as arguments. Henceforth, this function is always called f, and it is defined by the source node.

The interpretation of a node is decided by its type. Here we describe the interpretations of new (or significantly changed) types and refer the reader to previous work [Van den Broeck et al., 2011] for information on other types. Both CR and GDR node do not contribute anything to the definitions of functions—the interpretation of such a node is simply the interpretation of its only direct successor. Obviously, $\star$ nodes also have no

interpretation, although for a different reason: incomplete FCGs are not meant to be interpreted. The interpretation of a REF node is a function call. The direct successor of the REF node (say, v) then must introduce a function. The parameters of this function are the sizes of all domains used by nodes reachable from v.

Example 5.7. Let us use the FCG from Figure 5.2 as an example. The input formula (i.e., the formula in Example 5.1) has two domains: a and b. Thus, the interpretation of the FCG is a function $f\colon \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$. Let $m \coloneqq |a|$, and $n \coloneqq |b|$. The node labelled $\bigvee_{b^\top \subseteq b}$ tells us that $f(m,n) = \sum_{l=0}^{n} \binom{n}{l} \square$, where $\square$ is the interpretation of the remaining subgraph, and l iterates over all possible sizes of $b^\top$. It also creates two subdomains $b^\top, b^\bot \subseteq b$ that partition b, i.e., as the size of $b^\top$ increases, the size of $b^\bot$ correspondingly decreases. Nodes labelled $\wedge$ correspond to multiplication. Therefore, $f(m,n) = \sum_{l=0}^{n} \binom{n}{l} \diamondsuit \times \heartsuit$, where $\diamondsuit$ is the interpretation of the contradiction (i.e., $\bot$) node, and $\heartsuit$ is the interpretation of the REF node.

A contradiction node with clause c as a parameter is interpreted as one if the clause has groundings and zero otherwise. In this case, $c = (\emptyset, \{(X,Y)\}, \{X \mapsto b^\top, Y \mapsto b^\top\})$, which can be read as $\forall X,Y \in b^\top.\ X \neq Y \implies \bot$, i.e., $\forall X,Y \in b^\top, X = Y$. This latter sentence is true if and only if $|b^\top| < 2$. Therefore, we can use the Iverson bracket notation to write

$$\diamondsuit = [l < 2] \coloneqq \begin{cases} 1 & \text{if } l < 2 \\ 0 & \text{otherwise.} \end{cases}$$

It remains to interpret the REF node. Parameter $\{a \mapsto a', b \mapsto b^\bot\}$ tells us that the interpretation of the REF node should be the same as that of the source node, but with domains a and b replaced with a' and $b^\bot$, respectively. Domain a' was created by a constraint removal rule applied on a, so $|a'| = m - 1$. Now $b^\bot = b \setminus b^\top$, and $|b^\top| = l$, so $|b^\bot| = n - l$. Thus, the interpretation of the REF node is a recursive call to $f(m-1, n-l)$. Therefore,

$$f(m,n) = \sum_{l=0}^{n} \binom{n}{l} [l < 2] f(m-1, n-l) = f(m-1,n) + n f(m-1,n-1). \tag{5.2}$$

In order to use this recursive function to compute the model count of the input formula for any domain sizes, one just needs to find the base cases $f(0,n)$ and $f(m,0)$ for all $m,n \in \mathbb{N}_0$.

5.5 Empirical Results

In this section, we compare CRANE and FORCLIFT [Van den Broeck et al., 2011] on their ability to count various kinds of functions. (Other WFOMC algorithms such as L2C [Kazemi and Poole, 2016] and probabilistic theorem proving [Gogate and Domingos, 2016] are unable to solve any of the instances that FORCLIFT fails on.) We begin by describing how such functions can be expressed in FOL. FORCLIFT then translates these sentences in FOL to formulas as defined in Definition 5.3.

Let $p \in a \times b$ be a predicate. To restrict all relations representable by p to just functions from a to b, in FOL one might write

$$\forall X \in a.\ \forall Y \in b.\ \forall Z \in b.\ p(X,Y) \wedge p(X,Z) \implies Y = Z$$

and

$$\forall X \in a.\ \exists Y \in b.\ p(X,Y). \tag{5.3}$$

The former sentence says that one element of a can map to at *most* one element of b, and the latter sentence says that each element of a must map to at *least* one element of b. One can then add

$$\forall W \in a.\ \forall X \in a.\ \forall Y \in b.\ p(W,Y) \wedge p(X,Y) \implies W = X$$

to restrict p to injections or

$$\forall Y \in b.\ \exists X \in a.\ p(X,Y)$$

to ensure surjectivity or remove Equation (5.3) to consider partial functions. Lastly, one can replace all occurrences of b with a to model endofunctions (i.e., functions with the same domain and codomain) instead.

In our experiments, we consider all sixteen combinations of these properties, i.e., injectivity, surjectivity, partiality, and endo-. FORCLIFT is always run until it terminates. CRANE is run until either five solutions are found or the search tree reaches height 6.[10] If successful, FORCLIFT generates a circuit, and CRANE generates one or more (complete) FCGs. In both cases, we manually convert the resulting graphs into definitions of functions as described in Section 5.4. We then assess the complexity of each solution and pick the best if CRANE returns several solutions of varying complexities. When assessing the complexity of each such definition, we make two assumptions. First,

[10]The search tree has a high branching factor, so exploring all nodes at depth 5 takes at most a few seconds whereas doing the same for depth 6 can be computationally infeasible in some cases.

we can compute the binomial coefficient $\binom{n}{k}$ in $\Theta(nk)$ time. Second, techniques such as dynamic programming and memoization are used to avoid recomputing the same binomial coefficient or function call multiple times.

The experimental results are summarised in Table 5.1. The best-known asymptotic complexity for computing total surjections is by Earnest [2018]. All other best-known complexity results are inferred from the formulas and programs on the on-line encyclopedia of integer sequences [OEIS Foundation Inc., 2022]. On instances that could already be solved by FORCLIFT, the two algorithms perform equally well. However, CRANE is also able to solve all but one instances that FORCLIFT fails on in at most cubic time.

Let us examine the case of counting partial (non-endomorphic) injections more closely. The FCG in Figure 5.2 and Example 5.7 counts partial injections and is responsible for the *mn* entry in the table. For a complete solution, Equation (5.2) must be combined with the base case $f(0,n) = 1$ for all $n \in \mathbb{N}_0$. In other words, the base case says that the empty partial map is the only partial injection with an empty domain, regardless of the codomain. Finally, note that $f(m,n)$ can be evaluated in $\Theta(mn)$ time by a dynamic programming algorithm that computes $f(i,j)$ for all $i = 0,\ldots,m$ and $j = 0,\ldots,n$.

5.6 Conclusion and Future Work

In this chapter, we showed how a state-of-the-art (W)FOMC algorithm can be empowered by generalising domain recursion and adding support for cycles in the graph that encodes a solution. To construct such graphs, CRANE supplements FORCLIFT [Van den Broeck et al., 2011] with three new compilation rules and a hybrid search algorithm. In Section 5.5, we saw examples of instances that become liftable not just in theory [Kuzelka, 2021] but with an implemented algorithm as well. However, our experiments covered only a small set of instances since some parts of CRANE are yet to be fully automated. Here we focused on various function-counting problems—it remains to be seen what other types of instances become liftable as a result.

The most important direction for future work is in fully automating this new way of computing the (W)FOMC of a formula. First, we need an algorithm that transforms FCGs into definitions of functions. Formalising this process would also allow us to prove the correctness of the new compilation rules in constructing FCGs that indeed compute the right WMC. Second, these definitions must be simplified before they can

be used, perhaps by a computer algebra system. Third, most importantly, we need a way to find the base cases for the recursive definitions provided by CRANE. What makes this problem non-trivial is that the number of base cases is not constant for functions of arity greater than one (i.e., formulas that mention more than one domain). On the other hand, assuming that a domain is empty can greatly simplify a problem. Fourth, since the first solution found by CRANE is not always optimal in terms of its complexity, an automated way to determine the asymptotic complexity of a solution would be helpful as well. Achieving these goals would make CRANE capable of automatically constructing efficient ways to compute a function (e.g., a sequence) of interest. In addition to the potential impact to areas of artificial intelligence (AI) such as statistical relational AI [De Raedt et al., 2016], CRANE could be beneficial to research in combinatorics as well [Barvínek et al., 2021].

Algorithm 5.3: The compilation rule for REF nodes.

Input: formula ϕ, cache C

Output: a set of chips

forall *pairs of formulas and nodes* $(\psi, v) \in C(\#\phi)$ **do**
  $\rho \leftarrow$ `identifyRecursion`(ϕ, ψ);
  if $\rho \neq$ `null` **then return** $\{(\text{REF}_\rho(v), \langle\rangle)\}$;
return $\emptyset$;
Function `identifyRecursion`(*formula* ϕ, *formula* ψ, *map* $\rho = \emptyset$)**:**
  if $|\phi| \neq |\psi|$ **or** $\#\phi \neq \#\psi$ **then return** `null`;
  if $\phi = \emptyset$ **then return** ρ;
  foreach *clause* $c \in \psi$ **do**
    foreach *clause* $d \in \phi$ *s.t.* $\#d = \#c$ **do**
      forall $(\beta, \gamma) \in$ `generateMaps`(c, d, ρ) *s.t.* $c[\beta, \gamma] = d$ **do**
        $\rho' \leftarrow$ `identifyRecursion`$(\phi \setminus \{d\}, \psi \setminus \{c\}, \rho \cup \gamma)$;
        if $\rho' \neq$ `null` **then return** ρ';
    return `null`;
Function `generateMaps`(*clause c, clause d, map* ρ)**:**
  foreach *bijection* $\beta\colon \text{Vars}(c) \rightarrowtail\!\!\!\!\!\rightarrow \text{Vars}(d)$ **do**
    $\gamma \leftarrow$ `constructDomainMap`$(\text{Vars}(c), \delta_c, \delta_d, \beta, \rho)$;
    if $\gamma \neq$ `null` **then yield** (β, γ);
Function `constructDomainMap`(*set of variables* V, *maps* δ_c, δ_d, β, ρ)**:**
  $\gamma \leftarrow \emptyset$;
  foreach *variable* $v \in V$ **do**
    if $\delta_c(v) \in \text{dom}(\rho)$ **and** $\rho(\delta_c(v)) \neq \delta_d(\beta(v))$ **then return** `null`;
    if $\delta_c(v) \notin \text{dom}(\gamma)$ **then** $\gamma \leftarrow \gamma \cup \{\delta_c(v) \mapsto \delta_d(\beta(v))\}$;
    else if $\gamma(\delta_c(v)) \neq \delta_d(\beta(v))$ **then return** `null`;
  return γ;

Algorithm 5.4: The (main part of the) search algorithm.

Input: a formula ϕ_0

Result: all found FCGs for ϕ_0 are in set S

```
1  S ← ∅;
2  (G_0, L_0, C_0) ← applyGreedyRules(φ_0, ∅);
3  if L_0 = ⟨⟩ then S ← {G_0};
4  else
5      q ← a empty queue of states;
6      q.put((G_0, L_0, C_0));
7      while not q.empty() do
8          foreach state (G, L, C) ∈ applyAllRules(q.get()) do
9              if L = ⟨⟩ then S ← S ∪ {G};
10             else q.put((G, L, C));
```

Algorithm 5.5: The function for applying non-greedy rules.

```
1  Function applyAllRules(state s = (G, L, C)):
2      (G', L', C') ← a copy of s;
3      φ : T ← L;        /* separate the first formula from the rest */
4      foreach non-greedy rule r do
5          foreach chip (G'', L'') ∈ r(φ) do
6              if G'' = ⋆ then yield applyAllRules((G', L'' +# T, C'));
7              else
8                  C' ← updCache(C', φ, G'');
9                  (G'', L'', C') ← applyGreedyRulesToFormulas(G'', L'', C');
10                 yield (mergeFcgs(G', G''), C', L'' +# T);
11     (G', L', C') ← a copy of s;
```

Algorithm 5.6: Helper functions that apply greedy rules to a) a single formula and b) all uncompiled formulas in a state.

```
1  Function applyGreedyRules(formula ϕ, cache C):
2      foreach greedy rule r s.t. r(ϕ) ≠ ∅ do
3          (G, L) ← the only chip in r(ϕ);
4          if G = ⋆ then return applyGreedyRules(the formula in L, C);
5          return applyGreedyRulesToFormulas(G, L, updCache(C, ϕ, G));
6      return (⋆, ⟨ϕ⟩, C);
7  Function applyGreedyRulesToFormulas((V, s, N⁺, τ), list L, cache C):
8      if L = ⟨⟩ then return ((V, s, N⁺, τ), L, C);
9      L′ ← ⟨⟩;
10     foreach formula ϕ ∈ L do
11         (G′, L″, C) ← applyGreedyRules(ϕ, C);
12         L′ ← L′ ++ L″;
13         if G′ = (V′, s′, N′, τ′) ≠ ⋆ then
14             (V, N⁺, τ) ← (V ∪ V′, N⁺ ∪ N′, τ ∪ τ′);
15             replace the corresponding ⋆ in N⁺(s) with s′;
16     return ((V, s, N⁺, τ), L′, C);
```

Algorithm 5.7: Helper functions for updating a cache and merging FCGs.

```
1  Function updCache(cache C, formula φ, FCG (V, s, N⁺, τ)):
2      if τ(s) = REF then return C;
3      if #φ ∉ dom(C) then return C ∪ { #φ ↦ (φ, s) };
4      if there is no (φ′, v) ∈ C(#φ) s.t. v = s then C(#φ) ← ⟨(φ, s)⟩ +⊕ C(#φ);
5      return C;
6  Function mergeFcgs(G = (V, s, N⁺, τ), G′ = (V′, s′, N′, τ′), r = s):
7      if G = ⋆ then return G′;
8      if τ(r) = REF then return null;
9      foreach t ∈ N⁺(r) do
10         if τ(t) = ⋆ then
11             replace t with s′ in N⁺(r);
12             return (V ∪ V′, s, N⁺ ∪ N′, τ ∪ τ′);
13         G″ ← mergeFcgs(G, G′, t);
14         if G″ ≠ null then return G″;
15     return null;
```

Function Class			Asymptotic Complexity of Counting		
Partial	Endo-	Class	Best Known	With FORCLIFT	With CRANE
✓/✗	✓/✗	Functions	$\log m$	m	m
✗	✗	Surjections	$n \log m$	$m^3 + n^3$	$m^3 + n^3$
✗	✓		$m \log m$	m^3	m^3
✓	✗		Same as injections from b to a		
✓	✓		Same as endo-injections		
✗	✗	Injections	m	—	mn
✗	✓		m	—	m^3
✓	✗		$\min\{m,n\}^2$	—	mn
✓	✓		m^2	—	—
✗	✗	Bijections	m	—	m
✗	✓		Same as (partial) (endo-)injections		
✓	✓/✗				

Table 5.1: The worst-case complexity of counting various types of functions. Here, m is the size of domain a, and n is the size of domain b. All asymptotic complexities are in $\Theta(\cdot)$. A dash means that no complete solution was found.

Chapter 6

Generating Random Logic Programs Using Constraint Programming

6.1 Introduction

Probabilistic logic programming languages such as the independent choice logic [Poole, 1997], PRISM [Sato and Kameya, 1997], and ProbLog [De Raedt et al., 2007] are promising frameworks for codifying complex statistical relational models whose inference procedures often rely on WMC [Fierens et al., 2011, 2015, Vlasselaer et al., 2016]. However, if one were to survey the literature, one often finds that an inference algorithm is only tested on a small number (1–4) of data sets [Bruynooghe et al., 2010, Kimmig et al., 2011, Vlasselaer et al., 2015], originating from areas such as social networks, citation patterns, and biological data. But how confident can we be that an algorithm works well if it is only tested on a few problems?

About thirty years ago, SAT solving technology was dealing with a similar lack of clarity [Selman et al., 1996]. This changed with the study of generating random SAT instances against different input parameters (e.g., clause length and the total number of variables) to better understand the behaviour of algorithms and their ability to solve random synthetic problems. Unfortunately, when it comes to generating random logic programs, all approaches so far focused exclusively on propositional programs [Amendola et al., 2017, 2020, Wang et al., 2015, Zhao and Lin, 2003], often with severely limiting conditions such as two-literal clauses [Namasivayam, 2009, Namasivayam and Truszczynski, 2009] or clauses of the form $a \leftarrow \neg b$ [Wen et al., 2016].

In this work (Sections 6.3–6.5), we introduce a constraint-based representation

for logic programs based on simple parameters that describe the program's size, what predicates and constants it uses, etc. This representation takes the form of a *constraint satisfaction problem* (CSP), i.e., a set of discrete variables and restrictions on what values they can take. Every solution to this problem (as output by a constraint solver) directly translates into a logic program. One can either output all (sufficiently small) programs that satisfy the given conditions or use random value ordering heuristics and restarts to generate random programs. For sampling from a uniform distribution, the CSP can be transformed into a belief network [Dechter et al., 2002]. In fact, the same model can generate both probabilistic programs in the syntax of ProbLog [De Raedt et al., 2007] and non-probabilistic Prolog programs. To the best of our knowledge, this is the first work that

- addresses the problem of generating random logic programs in its full generality (i.e., including first-order clauses with variables), and
- compares and evaluates inference algorithms for probabilistic logic programs on more than a handful of instances.

A major advantage of a constraint-based approach is the ability to add additional constraints as needed, and to do that efficiently (compared to generate-and-test approaches). As an example of this, in Section 6.7 we develop a custom constraint that, given two predicates p and q, ensures that any ground atom with predicate p is independent of any ground atom with predicate q. In this way, we can easily regulate the independence structure of the underlying probability distribution. In Section 6.6 we also present a combinatorial argument for correctness that counts the number of programs that the model produces for various parameter values. We end the chapter with two experimental results in Section 6.9: one investigating how the constraint model scales when tasked with producing more complex programs, and one showing how the model can be used to evaluate and compare probabilistic inference algorithms.

6.2 Preliminaries

All constraint variables that we use in this chapter are integer or set variables, however, if an integer refers to a logical construct (e.g., a logical variable or a constant), we will make no distinction between the two. We say that a constraint variable is *(fully) determined* if its domain (at the time) has exactly one value. We let $\square$ denote the absent/disabled value of an optional variable [Mears et al., 2014]. We write $\mathtt{a}[b] \in c$ to

mean that a is an array of variables of length b such that each element of a has domain c. Similarly, we write c : a[b] to denote an array a of length b such that each element of a has type c. Finally, we assume that all arrays start with index zero.

Parameters of the model. We begin by defining sets and lists of the primitives used in constructing logic programs: a list of predicates $\mathcal{P}$, a list of their corresponding arities $\mathcal{A}$ (so $|\mathcal{A}| = |\mathcal{P}|$), a set of variables $\mathcal{V}$, and a set of constants $\mathcal{C}$. Either $\mathcal{V}$ or $\mathcal{C}$ can be empty, but we assume that $|\mathcal{C}| + |\mathcal{V}| > 0$. Similarly, the model supports zero-arity predicates but requires at least one predicate to have non-zero arity. For notational convenience, we also set $\mathcal{M}_{\mathcal{A}} = \max \mathcal{A}$. Next, we need a measure of how complex a body of a clause can be. As we represent each body by a tree (see Section 6.4), we set $\mathcal{M}_{\mathcal{N}} \geq 1$ to be the maximum number of nodes in the tree representation of any clause. We also set $\mathcal{M}_{\mathcal{C}}$ to be the maximum number of clauses in a program. We must have that $\mathcal{M}_{\mathcal{C}} \geq |\mathcal{P}|$ because we require each predicate to have at least one clause that defines it. The model supports enforcing predicate independence (see Section 6.7), so a set of independent pairs of predicates is another parameter. Since this model can generate probabilistic as well as non-probabilistic programs, each clause is paired with a probability which is randomly selected from a given list—our last parameter. For generating non-probabilistic programs, one can set this list to $\langle 1 \rangle$. Finally, we define $\mathcal{T} = \{\neg, \wedge, \vee, \top\}$ as the set of tokens that (together with atoms) form a clause. All decision variables of the model can now be divided into $2 \times \mathcal{M}_{\mathcal{C}}$ separate groups, treating the body and the head of each clause separately. We say that the variables are contained in two arrays: Body : bodies[$\mathcal{M}_{\mathcal{C}}$] and Head : heads[$\mathcal{M}_{\mathcal{C}}$].

6.3 Heads of Clauses

In Sections 6.3 and 6.4, we define clauses (i.e., their heads and bodies) using constraint programming terms. We define the *head* of a clause as a predicate $\in \mathcal{P} \cup \{\square\}$ and arguments[$\mathcal{M}_{\mathcal{A}}$] $\in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$. Here, we use $\square$ to denote either a disabled clause that we choose not to use or disabled arguments if the arity of the predicate is less than $\mathcal{M}_{\mathcal{A}}$. The reason why we need a separate value for the latter (i.e., why it is not enough to fix disabled arguments to a single already-existing value) will become clear in Section 6.5. This predicate variable has a corresponding arity that depends on the predicate. We can define arity $\in [0, \mathcal{M}_{\mathcal{A}}]$ as the arity of the predicate if predicate $\in \mathcal{P}$ and zero otherwise using the table constraint [Mairy et al., 2015]. This

constraint uses a set of pairs of the form (p,a), where p ranges over all possible values of the `predicate`, and a is either the arity of predicate p or zero. Having defined arity, we can now fix the superfluous arguments.

Constraint 6.1. *For* $i = 0,\ldots,\mathcal{M}_{\mathcal{A}}-1$, $\texttt{arguments}[i] = \square \iff i \geq \texttt{arity}$.

We also add a constraint that each predicate should get at least one clause.

Constraint 6.2. *Let* $P = \{\,, h.\texttt{predicate} \mid h \in \texttt{heads}\}$ *be a multiset. Then*

$$\texttt{nValues}(P) = \begin{cases} |\mathcal{P}| & \text{if } \texttt{count}(\square, P) = 0 \\ |\mathcal{P}|+1 & \text{otherwise,} \end{cases}$$

where $\texttt{nValues}(P)$ *counts the number of unique values in P, and* $\texttt{count}(\square, P)$ *counts how many times* $\square$ *appears in P.*

Finally, we want to disable duplicate clauses but with one exception: there may be more than one disabled clause, i.e., a clause with head $\texttt{predicate} = \square$. Assuming a lexicographic order over entire clauses such that $\square > p$ for all $p \in \mathcal{P}$ and the head predicate is the 'first digit' of this representation, the following constraint disables duplicates as well as orders the clauses.

Constraint 6.3. *For* $i = 1,\ldots,\mathcal{M}_{\mathcal{C}}-1$, *if* $\texttt{heads}[i].\texttt{predicate} \neq \square$, *then*

$$(\texttt{heads}[i-1], \texttt{bodies}[i-1]) < (\texttt{heads}[i], \texttt{bodies}[i]).$$

6.4 Bodies of Clauses

As was briefly mentioned before, the *body* of a clause is represented by a tree. It has two parts. First, there is the $\texttt{structure}[\mathcal{M}_{\mathcal{N}}] \in [0, \mathcal{M}_{\mathcal{N}}-1]$ array that encodes the structure of the tree using the following two rules: $\texttt{structure}[i] = i$ means that the i^{th} node is a root, and $\texttt{structure}[i] = j$ (for $j \neq i$) means that the i^{th} node's parent is node j. The second part is the array $\texttt{Node} : \texttt{values}[\mathcal{M}_{\mathcal{N}}]$ such that $\texttt{values}[i]$ holds the value of the i^{th} node, i.e., a representation of the atom or logical operator.

We can use the `tree` constraint [Fages and Lorca, 2011] to forbid cycles in the `structure` array and simultaneously define $\texttt{numTrees} \in \{1,\ldots,\mathcal{M}_{\mathcal{N}}\}$ to count the number of trees. We will view the tree rooted at the zeroth node as the main tree and restrict all other trees to single nodes. For this to work, we need to make sure that the zeroth node is indeed a root, i.e., fix $\texttt{structure}[0] = 0$. For convenience, we also

define $\texttt{numNodes} \in \{1,\dots,\mathcal{M}_\mathcal{N}\}$ to count the number of nodes in the main tree. We define it as $\texttt{numNodes} = \mathcal{M}_\mathcal{N} - \texttt{numTrees} + 1$.

Example 6.1. Let $\mathcal{M}_\mathcal{N} = 8$. Then $\neg p(X) \vee (q(X) \wedge p(X))$ can be encoded as:

$$\texttt{structure} = [0, 0, 0, \quad 1, \quad 2, \quad 2, \ 6, 7], \quad \texttt{numNodes} = 6,$$
$$\texttt{values} = [\vee, \neg, \wedge, p(X), q(X), p(X), \top, \top], \quad \texttt{numTrees} = 3.$$

Here, $\top$ is the value we use for the remaining one-node trees. The elements of the `values` array are nodes. A *node* has a $\texttt{name} \in \mathcal{T} \cup \mathcal{P}$ and $\texttt{arguments}[\mathcal{M}_\mathcal{A}] \in \mathcal{V} \cup \mathcal{C} \cup \{\square\}$. The node's `arity` can then be defined in the same way as in Section 6.3. Furthermore, we can use Constraint 6.1 to again disable the extra arguments.

Example 6.2. Let $\mathcal{M}_\mathcal{A} = 2$, $X \in \mathcal{V}$, and let p be a predicate with arity 1. Then the node representing atom $p(X)$ has: $\texttt{name} = p$, $\texttt{arguments} = [X, \square]$, $\texttt{arity} = 1$.

We need to constrain the forest represented by the `structure` array together with its `values` to eliminate symmetries and adhere to our desired format. First, we can recognize that the order of the elements in the `structure` array does not matter, i.e., the structure is only defined by how the elements link to each other, so we can add a constraint for sorting the `structure` array. Next, since we already have a variable that counts the number of nodes in the main tree, we can fix the structure and the values of the remaining trees to some constant values.

Constraint 6.4. *For* $i = 1,\dots,\mathcal{M}_\mathcal{N} - 1$*, if* $i < \texttt{numNodes}$*, then*

$$\texttt{structure}[i] = i, \quad \textit{and} \quad \texttt{values}[i].\texttt{name} = \top,$$

else $\texttt{structure}[i] < i$.

The second part of this constraint states that every node in the main tree except the zeroth node cannot be a root and must have its parent located to the left of itself. Next, we classify all nodes into three classes: predicate (or empty) nodes, negation nodes, and conjunction/disjunction nodes based on the number of children (zero, one, and two, respectively).

Constraint 6.5. *For* $i = 0,\dots,\mathcal{M}_\mathcal{N} - 1$*, let* C_i *be the number of times* i *appears in the* `structure` *array with index greater than* i*. Then*

$$C_i = 0 \iff \texttt{values}[i].\texttt{name} \in \mathcal{P} \cup \{\top\},$$
$$C_i = 1 \iff \texttt{values}[i].\texttt{name} = \neg,$$
$$C_i > 1 \iff \texttt{values}[i].\texttt{name} \in \{\wedge, \vee\}.$$

The value $\top$ serves a twofold purpose: it is used as the fixed value for nodes outside the main tree, and, when located at the zeroth node, it can represent a clause with an empty body. Thus, we can say that only root nodes can have $\top$ as the value.

Constraint 6.6. *For* $i = 0, \ldots, \mathcal{M}_{\mathcal{N}} - 1$,

$$\texttt{structure}[i] \neq i \implies \texttt{values}[i].\texttt{name} \neq \top.$$

Finally, we add a way to disable a clause by setting its head predicate to $\square$.

Constraint 6.7. *For* $i = 0, \ldots, \mathcal{M}_{\mathcal{C}} - 1$, *if* $\texttt{heads}[i].\texttt{predicate} = \square$, *then*

$$\texttt{bodies}[i].\texttt{numNodes} = 1, \quad \textit{and} \quad \texttt{bodies}[i].\texttt{values}[0].\texttt{name} = \top.$$

6.5 Variable Symmetry Breaking

Ideally, we want to avoid generating programs that are equivalent in the sense that they produce the same answers to all queries. Even more importantly, we want to avoid generating multiple internal representations that ultimately result in the same program. This is the purpose of *symmetry-breaking constraints*, another important benefit of which is that the constraint solving task becomes easier [Walsh, 2006]. Given any clause, we can permute the variables in that clause without changing the meaning of the clause or the entire program. Thus, we want to fix the order of variables. Informally, we can say that variable X goes before variable Y if the first occurrence of X in either the head or the body of the clause is before the first occurrence of Y. Note that the constraints described in this section only make sense if $|\mathcal{V}| > 1$ and that all definitions and constraints here are on a per-clause basis.

Definition 6.1. Let $N = \mathcal{M}_{\mathcal{A}} \times (\mathcal{M}_{\mathcal{N}} + 1)$, and let $\texttt{terms}[N] \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$ be a flattened array of all arguments in a particular clause. Then we can use a channeling constraint to define $\texttt{occ}[|\mathcal{C}| + |\mathcal{V}| + 1]$ as an array of subsets of $\{0, \ldots, N-1\}$ such that for all $i = 0, \ldots, N-1$, and $t \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$,

$$i \in \texttt{occ}[t] \quad \iff \quad \texttt{terms}[i] = t.$$

Next, we introduce an array that holds the first occurrence of each variable.

Definition 6.2. Let $\texttt{intros}[|\mathcal{V}|] \in \{0, \ldots, N\}$ be such that for $v \in \mathcal{V}$,

$$\texttt{intros}[v] = \begin{cases} 1 + \min \texttt{occ}[v] & \text{if } \texttt{occ}[v] \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Here, a value of zero means that the variable does not occur in the clause (this choice is motivated by subsequent constraints). As a consequence, all other indices are shifted by one. Having set this up, we can now eliminate variable symmetries simply by sorting `intros`. In other words, we constrain the model so that the variable listed first (in whatever order $\mathcal{V}$ is presented in) has to occur first in our representation of a clause.

Example 6.3. Let $\mathcal{C} = \emptyset$, $\mathcal{V} = \{X, Y, Z\}$, $\mathcal{M}_{\mathcal{A}} = 2$, $\mathcal{M}_{\mathcal{N}} = 3$, and consider the clause $\mathsf{sibling}(X,Y) \leftarrow \mathsf{parent}(X,Z) \wedge \mathsf{parent}(Y,Z)$. Then

$$\begin{aligned} \texttt{terms} &= [X, Y, \square, \square, X, Z, Y, Z], \\ \texttt{occ} &= [\{0,4\}, \{1,6\}, \{5,7\}, \{2,3\}], \\ \texttt{intros} &= [0, 1, 5], \end{aligned}$$

where the $\square$'s correspond to the conjunction node.

We end the section with several redundant constraints that make the CSP easier to solve. First, we can state that the positions occupied by different terms must be different.

Constraint 6.8. *For* $u \neq v \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$, $\texttt{occ}[u] \cap \texttt{occ}[v] = \emptyset$.

The reason why we use zero to represent an unused variable is so that we could now use the 'all different except zero' constraint for the `intros` array. We can also add another link between `intros` and `occ` that essentially says that the smallest element of a set is an element of the set.

Constraint 6.9. *For* $v \in \mathcal{V}$, $\texttt{intros}[v] \neq 0 \iff \texttt{intros}[v] - 1 \in \texttt{occ}[v]$.

Finally, we define an auxiliary set variable to act as a set of possible values that `intros` can take. Let $\texttt{potentials} \subseteq \{0, \ldots, N\}$ be such that for $v \in \mathcal{V}$, $\texttt{intros}[v] \in \texttt{potentials}$. Using this new variable, we can add a constraint saying that non-predicate nodes in the tree representation of a clause cannot have variables as arguments.

Constraint 6.10. *For* $i = 0, \ldots, \mathcal{M}_{\mathcal{N}} - 1$, *let*

$$S = \{\, \mathcal{M}_{\mathcal{A}} \times (i+1) + j + 1 \mid j = 0, \ldots, \mathcal{M}_{\mathcal{A}} - 1 \,\}.$$

If $\texttt{values}[i].\texttt{name} \notin \mathcal{P}$, *then* $\texttt{potentials} \cap S = \emptyset$.

6.6 Counting Programs

To demonstrate the correctness of the model, this section derives combinatorial expressions for counting the number of programs with up to $\mathcal{M}_\mathcal{C}$ clauses and up to $\mathcal{M}_\mathcal{N}$ nodes per clause, and arbitrary $\mathcal{P}$, $\mathcal{A}$, $\mathcal{V}$, and $\mathcal{C}$. Being able to establish two ways to generate the same sequence of numbers (i.e., numbers of programs with certain properties and parameters) allows us to gain confidence that the constraint model accurately matches our intentions. For this section, we introduce the term *total arity* of a body of a clause to refer to the sum total of arities of all predicates in the body.

We will first consider clauses with *gaps*, i.e., without taking variables and constants into account. Let $T(n,a)$ denote the number of possible clause bodies with n nodes and total arity a. Then $T(1,a)$ is the number of predicates in $\mathcal{P}$ with arity a, and the following recursive definition can be applied for $n > 1$:

$$T(n,a) = T(n-1,a) + 2 \sum_{\substack{c_1+\cdots+c_k=n-1,\\ 2\le k\le \frac{a}{\min \mathcal{A}},\\ c_i\ge 1 \text{ for all } i}} \sum_{\substack{d_1+\cdots+d_k=a,\\ d_i\ge \min \mathcal{A} \text{ for all } i}} \prod_{i=1}^{k} T(c_i,d_i).$$

The first term here represents negation, i.e., negating a formula consumes one node but otherwise leaves the task unchanged. If the first operation is not a negation, then it must be either conjunction or disjunction (hence the coefficient '2'). In the first sum, k represents the number of children of the root node, and each c_i is the number of nodes dedicated to child i. Thus, the first sum iterates over all possible ways to partition the remaining $n-1$ nodes. Similarly, the second sum considers every possible way to partition the total arity a across the k children nodes. We can then count the number of possible clause bodies with total arity a (and any number of nodes) as

$$C(a) = \begin{cases} 1 & \text{if } a = 0 \\ \sum_{n=1}^{\mathcal{M}_\mathcal{N}} T(n,a) & \text{otherwise.} \end{cases}$$

The number of ways to select n terms is

$$P(n) = |\mathcal{C}|^n + \sum_{\substack{1\le k\le |\mathcal{V}|,\\ 0=s_0<s_1<\cdots<s_k<s_{k+1}=n+1}} \prod_{i=0}^{k} (|\mathcal{C}|+i)^{s_{i+1}-s_i-1}.$$

The first term is the number of ways to select n constants. The parameter k is the number of variables used in the clause, and $s_1,\ldots,s_k$ mark the first occurrence of each variable. For each gap between any two introductions (or before the first introduction, or after the

last introduction), we have $s_{i+1} - s_i - 1$ spaces to be filled with any of the $|\mathcal{C}|$ constants or any of the i already-introduced variables.

Let us order the elements of $\mathcal{P}$, and let a_i be the arity of the i^{th} predicate. The number of programs is then:

$$\sum_{\substack{\sum_{i=1}^{|\mathcal{P}|} h_i = n, \\ |\mathcal{P}| \leq n \leq \mathcal{M}_C, \\ h_i \geq 1 \text{ for all } i}} \prod_{i=1}^{|\mathcal{P}|} \binom{\sum_{a=0}^{\mathcal{M}_A \times \mathcal{M}_N} C(a) P(a + a_i)}{h_i}, \tag{6.1}$$

Here, we sum over all ways to distribute $|\mathcal{P}| \leq n \leq \mathcal{M}_C$ clauses among $|\mathcal{P}|$ predicates so that each predicate gets at least one clause. For each predicate, we can then count the number of ways to select its clauses out of all possible clauses. The number of possible clauses can be computed by considering each possible arity a, and multiplying the number of 'unfinished' clauses $C(a)$ by the number of ways to select the required $a + a_i$ terms in the body and the head of the clause. Finally, we compare the numbers produced by Equation (6.1) with the numbers of programs generated by our model in 1032 different scenarios, thus showing that the combinatorial description developed in this section matches the model's behaviour.

6.7 Stratification and Independence

Stratification is a condition necessary for probabilistic logic programs [Mantadelis and Rocha, 2017] and often enforced on logic programs [Bidoit, 1991] that helps to ensure a unique answer to every query. This is achieved by restricting the use of negation so that any program $\mathscr{P}$ can be partitioned into a sequence of programs $\mathscr{P} = \bigsqcup_{i=1}^{n} \mathscr{P}_i$ such that, for all i, the negative literals in $\mathscr{P}_i$ can only refer to predicates defined in $\mathscr{P}_j$ for $j \leq i$ [Bidoit, 1991].

Independence, on the other hand, is defined on a pair of predicates (say, $p, q \in \mathcal{P}$) and can be interpreted in two ways. First, if p and q are independent, then any ground atom of p is independent of any ground atom of q in the underlying probability distribution of the probabilistic program. Second, the part of the program needed to fully define p is disjoint from the part of the program needed to define q.

These two seemingly disparate concepts can be defined using the same building block, i.e., a predicate dependency graph. Let $\mathscr{P}$ be a probabilistic logic program with its set of predicates $\mathcal{P}$. Its *(predicate) dependency graph* is a directed graph $G_{\mathscr{P}}$ with elements of $\mathcal{P}$ as nodes and an edge between $p, q \in \mathcal{P}$ if there is a clause in $\mathscr{P}$ with q

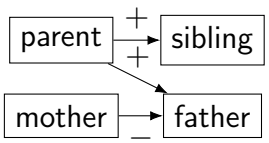


Figure 6.1: The predicate dependency graph of the program from Example 6.4. Positive edges are labelled with '+', and negative edges with '−'.

as the head and p mentioned in the body. We say that the edge is *negative* if there exists a clause with q as the head and at least one instance of p at the body such that the path from the root to the p node in the tree representation of the clause passes through at least one negation node; otherwise, it is *positive*. We say that $\mathscr{P}$ (or $G_{\mathscr{P}}$) has a *negative cycle* if $G_{\mathscr{P}}$ has a cycle with at least one negative edge. A program $\mathscr{P}$ is *stratified* if $G_{\mathscr{P}}$ has no negative cycles.[1] Thus a simple entailment algorithm for stratification can be constructed by selecting all clauses, all predicates of which are fully determined, and looking for negative cycles in the dependency graph constructed based on those clauses using an algorithm such as Bellman-Ford.

For any predicate $p \in \mathcal{P}$, the set of *dependencies* of p is the smallest set D_p such that $p \in D_p$, and, for every $q \in D_p$, all direct predecessors of q in $G_{\mathscr{P}}$ are in D_p. Two predicates p and q are *independent* if $D_p \cap D_q = \emptyset$.

Example 6.4. Consider the following (fragment of a) program:

$$\begin{aligned} \mathsf{sibling}(X,Y) &\leftarrow \mathsf{parent}(X,Z) \wedge \mathsf{parent}(Y,Z), \\ \mathsf{father}(X,Y) &\leftarrow \mathsf{parent}(X,Y) \wedge \neg\mathsf{mother}(X,Y). \end{aligned} \tag{6.2}$$

Its predicate dependency graph is in Figure 6.1. Because of the negation in Clause 6.2, the edge from mother to father is negative, while the other two edges are positive. The dependencies of each predicate are:

$$\begin{aligned} D_{\mathsf{parent}} &= \{\,\mathsf{parent}\,\}, \quad & D_{\mathsf{sibling}} &= \{\,\mathsf{sibling},\mathsf{parent}\,\}, \\ D_{\mathsf{mother}} &= \{\,\mathsf{mother}\,\}, \quad & D_{\mathsf{father}} &= \{\,\mathsf{father},\mathsf{mother},\mathsf{parent}\,\}. \end{aligned}$$

Hence, we have two pairs of independent predicates, i.e., mother is independent of parent and sibling.

Since the definition of independence relies on the dependency graph, we can represent this graph as an adjacency matrix constructed as part of the model. Let $\mathbf{A}$

[1] This definition is an extension of a well-known result for logic programs [Balbin et al., 1991] to probabilistic logic programs with arbitrary complex clause bodies.

Edges	Name	Notation
0	Determined	$\Delta(r)$
1	Almost determined	$\Gamma(r,s,t)$
>1	Undetermined	$\Upsilon(r)$

Table 6.1: Types of (potential) dependencies of a predicate p based on the number of undetermined edges on the path from the dependency to p.

Algorithm 6.1: Entailment for independence.

Data: predicates p_1, p_2

1 $D \leftarrow \{ (d_1, d_2) \in \texttt{deps}(p_1,\ 1) \times \texttt{deps}(p_2,\ 1) \mid d_1.\text{predicate} = d_2.\text{predicate} \}$;

2 **if** $D = \emptyset$ **then return** `true`;

3 **if** $\exists(\Delta\_, \Delta\_) \in D$ **then return** `false` **else return** `undefined`;

be a $|\mathcal{P}| \times |\mathcal{P}|$ binary matrix defined element-wise by stating that $\mathbf{A}[i][j] = 0$ if and only if, for all $k = 0, \ldots, \mathcal{M}_C - 1$, either $\texttt{heads}[k].\texttt{predicate} \neq j$ or $i \notin \{ a.\texttt{name} \mid a \in \texttt{bodies}[k].\texttt{values} \}$.

Given a partially-solved model with its predicate dependency graph, let us pick an arbitrary path from q to p (for some $p, q \in \mathcal{P}$) that consists of determined edges that are denoted by 1 in $\mathbf{A}$ and potential/undetermined edges that are denoted by $\{0, 1\}$. Each such path characterises a *(potential) dependency* q for p. We classify all such dependencies into three classes depending on the number of undetermined edges on the path. These classes are outlined in Table 6.1, where r represents the dependency predicate q, and, in the case of Γ, $(s,t) \in \mathcal{P}^2$ is the one undetermined edge on the path. For a dependency d—regardless of its exact type—we will refer to its predicate r as $d.\texttt{predicate}$. In describing the algorithms, we will use '_' to replace any of r, s, t in situations where the name is unimportant.

Each entailment algorithm returns one out of three values: `true` if the constraint is guaranteed to hold, `false` if the constraint is violated, and `undefined` if whether the constraint will be satisfied or not depends on the future decisions made by the solver. Algorithm 6.1 outlines a simple entailment algorithm for the independence of two predicates p_1 and p_2. First, we separately calculate all dependencies of p_1 and p_2 and look at the set D of dependencies that p_1 and p_2 have in common. If there are none, then the predicates are clearly independent. If they have a dependency in common that is already fully determined (Δ) for both predicates, then they cannot be independent.

Algorithm 6.2: Propagation for independence.

Data: predicates p_1, p_2; adjacency matrix $\mathbf{A}$

1 **for** $(d_1, d_2) \in$ `deps` $(p_1, 0) \times$ `deps` $(p_2, 0)$ *s.t.* d_1.predicate $= d_2$.predicate **do**
2 **if** d_1 **is** $\Delta(\_)$ **and** d_2 **is** $\Delta(\_)$ **then** `fail()`;
3 **if** $\{d_1, d_2\} = \{\Delta(\_), \Gamma(\_, s, t)\}$ **then** $\mathbf{A}[s][t]$.`removeValue(1)`;

Algorithm 6.3: Dependencies of a predicate.

Data: adjacency matrix $\mathbf{A}$

1 **Function** `deps` (*predicate p, flag* allDeps) **:**
2 $D \leftarrow \{\Delta(p)\}$;
3 **while** `true` **do**
4 $D' \leftarrow \emptyset$;
5 **foreach** *dependency* $d \in D$ **and** *predicate* $q \in \mathcal{P}$ **do**
6 edge $\leftarrow \mathbf{A}[q][d.\text{predicate}] = \{1\}$;
7 **if** edge **and** d **is** $\Delta(\_)$ **then** $D' \leftarrow D' \cup \{\Delta(q)\}$;
8 **else if** edge **and** d **is** $\Gamma(\_, s, t)$ **then** $D' \leftarrow D' \cup \{\Gamma(q, s, t)\}$;
9 **else if** $|\mathbf{A}[q][d.\text{predicate}]| > 1$ **and** d **is** $\Delta(r)$ **then**
10 $D' \leftarrow D' \cup \{\Gamma(q, q, r)\}$;
11 **else if** $|\mathbf{A}[q][d.\text{predicate}]| > 1$ **and** allDeps **then** $D' \leftarrow D' \cup \{\Upsilon(q)\}$;
12 **if** $D' = D$ **then return** D **else** $D \leftarrow D'$;

Otherwise, we return `undefined`.

Propagation algorithms have two goals: causing a contradiction (failing) in situations where the corresponding entailment algorithm would return `false`, and eliminating values from domains of variables that are guaranteed to cause a contradiction. Algorithm 6.2 does the former on line 2. Furthermore, for any dependency shared between predicates p_1 and p_2, if it is determined (Δ) for one predicate and almost determined (Γ) for another, then the edge that prevents the Γ from becoming a Δ cannot exist—line 3 handles this possibility.

The function `deps` in Algorithm 6.3 calculates D_p for any predicate p. It has two versions: `deps`$(p, 1)$ returns all dependencies, while `deps`$(p, 0)$ returns only determined and almost-determined dependencies. It starts by establishing the predicate p itself as a dependency and continues to add dependencies of dependencies until the set D stabilises. For each dependency $d \in D$, we look at the in-links of d in the predicate dependency

graph. If the edge from some predicate q to d.`predicate` is fully determined and d is determined, then q is another determined dependency of p. If the edge is determined but d is almost determined, then q is an almost-determined dependency. The same outcome applies if d is fully determined but the edge is undetermined. Finally, if we are interested in collecting all dependencies regardless of their status, then q is a dependency of p as long as the edge from q to d.`predicate` is possible. Note that if there are multiple paths in the dependency graph from q to p, Algorithm 6.3 could include q once for each possible type (Δ, Υ, and Γ), but Algorithms 6.1 and 6.2 would still work as intended.

Example 6.5. Consider this partially determined (fragment of a) program:

$$\square(X,Y) \leftarrow \mathsf{parent}(X,Z) \wedge \mathsf{parent}(Y,Z),$$
$$\mathsf{father}(X,Y) \leftarrow \mathsf{parent}(X,Y) \wedge \neg\mathsf{mother}(X,Y),$$

where $\square$ indicates an unknown predicate with domain

$$D_{\square} = \{\, \mathsf{father}, \mathsf{mother}, \mathsf{parent}, \mathsf{sibling} \,\}.$$

The predicate dependency graph is pictured in Figure 6.2. Suppose we have a constraint that mother and parent must be independent. The lists of potential dependencies for both predicates are:

$$D_{\mathsf{mother}} = \{\, \Delta(\mathsf{mother}), \Gamma(\mathsf{parent}, \mathsf{parent}, \mathsf{mother}) \,\},$$
$$D_{\mathsf{parent}} = \{\, \Delta(\mathsf{parent}) \,\}.$$

An entailment check at this stage would produce `undefined`, but propagation replaces the boxed value in Figure 6.2a with zero, eliminating the potential edge from parent to mother. This also eliminates mother from $D_{\square}$, and this is enough to make Algorithm 6.1 return `true`.

6.8 Example Programs

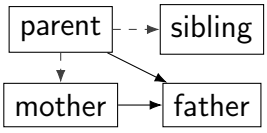
Integrate this into the chapter. See ?? for examples of programs generated using various sets of parameter values. Make sure that the initial overview mentions this section. Modify the surrounding introductions/conclusions from both sides.

In this appendix, we provide examples of probabilistic logic programs generated by the constraint model from Chapter 6. In all cases, we use

$$\{\, 0.1, 0.2, \ldots, 0.9, 1, 1, 1, 1, 1 \,\}$$

$$\begin{array}{l} \text{father} \\ \text{mother} \\ \text{parent} \\ \text{sibling} \end{array} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & \boxed{\{\,0,1\,\}} & \{0,1\} & \{0,1\} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

(a) The adjacency matrix of the graph. The boxed value is the decision variable that will be propagated by Algorithm 6.2.



(b) A drawing of the graph. Dashed edges are undetermined—they may or may not exist.

Figure 6.2: The predicate dependency graph of Example 6.5.

as the multiset of probabilities. Each clause is written on a separate line and ends with a full stop. The head and the body of each clause are separated with `:-` (instead of $\leftarrow$). The probability of each clause is prepended to the clause, using `::` as a separator. Probabilities equal to one and empty bodies of clauses can be omitted. Conjunction, disjunction, and negation are denoted by commas, semicolons, and '`\+`', respectively. Parentheses are used to demonstrate precedence, although many of them are redundant.

By setting $\mathcal{P} = [\mathtt{p}]$, $\mathcal{A} = [1]$, $\mathcal{V} = \{\mathtt{X}\}$, $\mathcal{C} = \emptyset$, $\mathcal{M}_{\mathcal{N}} = 4$, and $\mathcal{M}_{\mathcal{C}} = 1$, we get fifteen one-line programs, six of which are without negative cycles (as highlighted below). Only the last program has no cycles at all.

1. `0.5 :: p(X) :- (\+(p(X))), (p(X)).`
2. `0.8 :: p(X) :- (\+(p(X))); (p(X)).`
3. `0.8 :: p(X) :- (p(X)); (p(X)).`
4. `0.7 :: p(X) :- (p(X)), (p(X)).`
5. `0.6 :: p(X) :- (p(X)), (\+(p(X))).`
6. `p(X) :- (p(X)); (\+(p(X))).`
7. `0.1 :: p(X) :- (p(X)); (p(X)); (p(X)).`
8. `0.8 :: p(X) :- (p(X)), (p(X)), (p(X)).`
9. `p(X) :- \+(p(X)).`
10. `0.1 :: p(X) :- \+(\+(p(X))).`

11. `p(X) :- \+((p(X)); (p(X))).`

12. `0.4 :: p(X) :- \+((p(X)), (p(X))).`

13. `0.4 :: p(X) :- \+(\+(\+(p(X)))).`

14. `0.7 :: p(X) :- p(X).`

15. `p(X).`

Note that:

- A program such as Program 14, because of its cyclic definition, defines a predicate that has probability zero across all constants. This can more easily be seen as solving equation $0.7x = x$.
- Programs 10 and 14 are not equivalent (i.e., double negation does not cancel out) because Program 10 has a negative cycle and is thus considered to be ill-defined.

To demonstrate variable symmetry reduction in action, we set $\mathcal{P} = [\mathtt{p}]$, $\mathcal{A} = [3]$, $\mathcal{V} = \{\mathtt{X},\mathtt{Y},\mathtt{Z}\}$, $\mathcal{C} = \emptyset$, $\mathcal{M}_{\mathcal{N}} = 1$, $\mathcal{M}_{\mathcal{C}} = 1$, and forbid all cycles. This gives us the following five programs:

- `0.8 :: p(Z, Z, Z).`
- `p(Y, Y, Z).`
- `p(Y, Z, Z).`
- `p(Y, Z, Y).`
- `0.1 :: p(X, Y, Z).`

This is one of many possible programs with $\mathcal{P} = [\mathtt{p},\mathtt{q},\mathtt{r}]$, $\mathcal{A} = [1,2,3]$, $\mathcal{V} = \{\mathtt{X},\mathtt{Y},\mathtt{Z}\}$, $\mathcal{C} = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$, $\mathcal{M}_{\mathcal{N}} = 5$, $\mathcal{M}_{\mathcal{C}} = 5$, and without negative cycles:

```
p(b) :- \+((q(a, b)), (q(X, Y)), (q(Z, X))).
0.4 :: q(X, X) :- \+(r(Y, Z, a)).
q(X, a) :- r(Y, Y, Z).
q(X, a) :- r(Y, b, Z).
r(Y, b, Z).
```

Finally, we set $\mathcal{P} = [\mathtt{p}, \mathtt{q}, \mathtt{r}]$, $\mathcal{A} = [1,1,1]$, $\mathcal{V} = \emptyset$, $\mathcal{C} = \{\mathtt{a}\}$, $\mathcal{M}_{\mathcal{N}} = 3$, $\mathcal{M}_{\mathcal{C}} = 3$, forbid negative cycles, and constrain predicates `p` and `q` to be independent. The resulting search space contains thousands of programs such as:

- ```
  0.5 :: p(a) :- (p(a)); (p(a)).
  0.2 :: q(a) :- (q(a)), (q(a)).
  0.4 :: r(a) :- \+(q(a)).
  ```

- ```
  p(a) :- p(a).
  0.5 :: q(a) :- (r(a)); (q(a)).
  r(a) :- (r(a)); (r(a)).
  ```

- ```
  p(a) :- (p(a)); (p(a)).
  0.6 :: q(a) :- q(a).
  0.7 :: r(a) :- \+(q(a)).
  ```

6.9 Experimental Results

We now present the results of two experiments: in Section 6.9.1 we examine the scalability of our constraint model with respect to its parameters and in Section 6.9.2 we demonstrate how the model can be used to compare inference algorithms and describe their behaviour across a wide range of programs. The experiments were run on a system with Intel Core i5-8250U processor and 8 GB of RAM. The constraint model was implemented in Java 8 with Choco 4.10.2 [Prud'homme et al., 2017]. All inference algorithms are implemented in ProbLog 2.1.0.39 and were run using Python 3.8.2 with PySDD 0.2.10 and PyEDA 0.28.0. For both sets of experiments, we generate programs without negative cycles and use a 60 s timeout.

6.9.1 Empirical Performance of the Model

Along with constraints, variables, and their domains, two more design decisions are needed to complete the model: heuristics and restarts. By trial and error, the variable ordering heuristic was devised to eliminate sources of *thrashing*, i.e., situations where a contradiction is being 'fixed' by making changes that have no hope of fixing the contradiction. Thus, we partition all decision variables into an ordered list of groups and require the values of all variables from one group to be determined before moving to the next group. Within each group, we use the 'first fail' variable ordering heuristic.
  ```

The first group consists of all head predicates. Afterwards, we handle all remaining decision variables from the first clause before proceeding to the next. The decision variables within each clause are divided into

- the `structure` array,
- body predicates,
- head arguments,
- (if $|\mathcal{V}| > 1$) the `intros` array,
- body arguments.

For instance, in the clause from Example 6.3, all visible parts of the clause would be decided in this order:

$$\overset{1}{\mathsf{sibling}}(\overset{3}{X},\overset{3}{Y}) \leftarrow \overset{2}{\mathsf{parent}}(\overset{4}{X},\overset{4}{Z}) \overset{2}{\wedge} \overset{2}{\mathsf{parent}}(\overset{4}{Y},\overset{4}{Z}).$$

We also employ a geometric restart policy, restarting after $10, 10 \times 1.1, 10 \times 1.1^2, \ldots$ contradictions.[2] We ran 399 360 experiments, investigating the model's efficiency and gaining insight into what parameter values make the CSP harder. For $|\mathcal{P}|$, $|\mathcal{V}|$, $|\mathcal{C}|$, $\mathcal{M}_{\mathcal{N}}$, and $\mathcal{M}_{\mathcal{C}} - |\mathcal{P}|$ (i.e., the number of clauses in addition to the mandatory $|\mathcal{P}|$ clauses), we assign all combinations of 1, 2, 4, 8. $\mathcal{M}_{\mathcal{A}}$ is assigned to values 1–4. For each $|\mathcal{P}|$, we also iterate over all possible numbers of independent pairs of predicates, ranging from 0 up to $\binom{|\mathcal{P}|}{2}$. For each combination of the above-mentioned parameters, we pick ten random ways to assign arities to predicates (such that $\mathcal{M}_{\mathcal{A}}$ occurs at least once) and ten random combinations of independent pairs.

The majority (97.7 %) of runs finished in under 1 s, while four instances timed out: all with $|\mathcal{P}| = \mathcal{M}_{\mathcal{C}} - |\mathcal{P}| = \mathcal{M}_{\mathcal{N}} = 8$ and the remaining parameters all different. This suggests that—regardless of parameter values—most of the time a solution can be identified instantaneously while occasionally a series of wrong decisions can lead the solver into a part of the search space with no solutions.

In Figure 6.3, we plot how the mean number of nodes in the binary search tree grows as a function of each parameter (the plot for the median is very similar). The growth of each curve suggests how the model scales with higher values of the parameter. From this plot, it is clear that $\mathcal{M}_{\mathcal{N}}$ is the limiting factor. This is because some tree structures

[2] Restarts help overcome early mistakes in the search process but can be disabled if one wants to find all solutions, in which case search is complete regardless of the variable ordering heuristic.

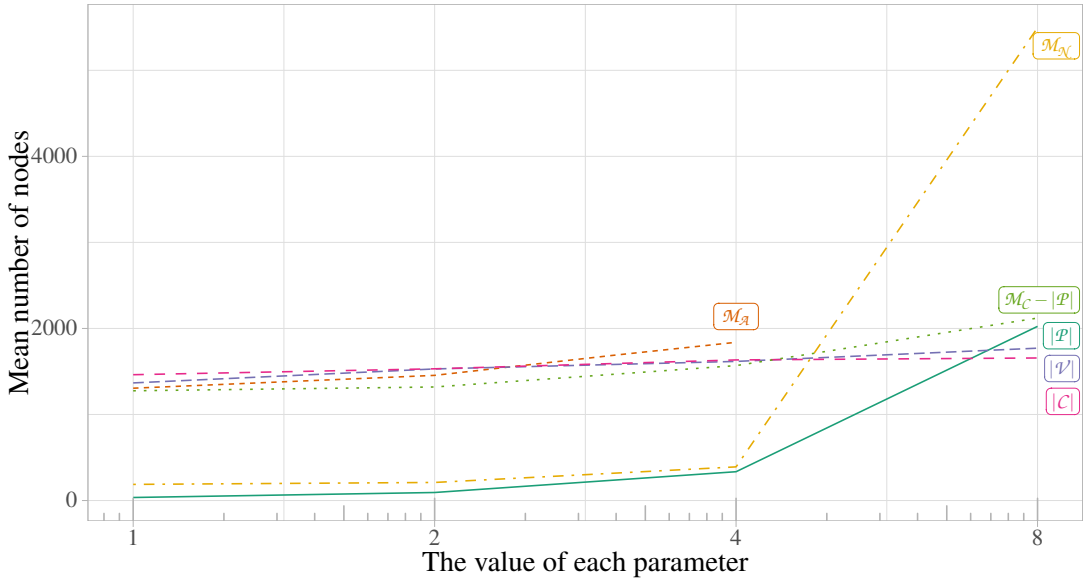


Figure 6.3: The mean number of nodes in the binary search tree for each value of each experimental parameter. Note that the horizontal axis is on a $\log_2$ scale.

can be impossible to fill with predicates without creating either a negative cycle or a forbidden dependency, and such trees become more common as the number of nodes increases. Likewise, a higher number of predicates complicates the situation as well.

6.9.2 Experimental Comparison of Inference Algorithms

For this experiment, we consider clauses of two types: *rules* are clauses such that the head atom has at least one variable, and *facts* are clauses with empty bodies and no variables.[3] We use our constraint model to generate the rules according to the following parameter values: $|\mathcal{P}|, |\mathcal{V}|, \mathcal{M}_\mathcal{N} \in \{2,4,8\}$, $\mathcal{M}_\mathcal{A} \in \{1,2,3\}$, $\mathcal{M}_\mathcal{C} = |\mathcal{P}|$, $\mathcal{C} = \emptyset$. These values are (approximately) representative of many standard benchmarking instances which often have 2–8 predicates of arity one or two, 0–8 rules, and a larger database of facts [Fierens et al., 2015]. Just like before, we explore all possible numbers of independent predicate pairs. We also add a constraint that forbids empty bodies. For both rules and facts, probabilities are uniformly sampled from $\{0.1, 0.2, \ldots, 0.9\}$. Furthermore, all rules are probabilistic, while we vary the proportion of probabilistic facts among 25 %, 50 %, and 75 %. For generating facts, we consider $|\mathcal{C}| \in \{100, 200, 400\}$ and vary the number of facts among 10^3, 10^4, and 10^5 but with one exception: the number of facts is not allowed to exceed 75 % of all possible facts with the given values

[3] Note that these definitions are slightly different from the definitions in Chapter 2.

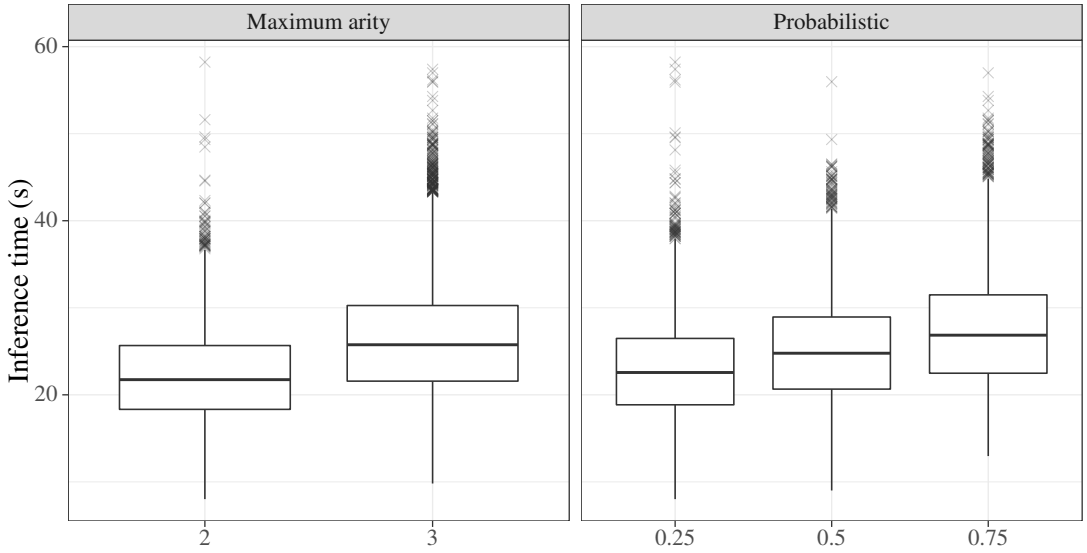


Figure 6.4: Inference time for different values of $\mathcal{M}_{\mathcal{A}}$ and proportions of probabilistic facts that are probabilistic. The total number of facts is fixed at 10^5.

of $\mathcal{P}$, $\mathcal{A}$, and $\mathcal{C}$. Facts are generated using a simple procedure that randomly selects a predicate, combines it with the right number of constants, and checks whether the generated atom is already included or not. We randomly select configurations from the description above and generate ten programs with a complete restart of the constraint solver before the generation of each program, including choosing different arities and independent pairs. Finally, we set the query of each program to a random fact not explicitly included in the program and consider six natively supported algorithms and knowledge compilation techniques: binary decision diagrams (BDDs) [Bryant, 1986], negation normal form (NNF), deterministic decomposable NNF (d-DNNF) [Darwiche and Marquis, 2002], K-Best [De Raedt et al., 2007], and two encodings based on sentential decision diagrams [Darwiche, 2011], one of which encodes the entire program (SDDX), while the other one encodes only the part of the program relevant to the query (SDD).[4]

Out of 11 310 generated problem instances, about 35 % were discarded because one or more algorithms were not able to ground the instance unambiguously. The first observation (pictured in Figure 6.5) is that the algorithms are remarkably similar, i.e., the differences in performance are small and consistent across all parameter values

[4]Forward SDDs (FSDDs) and forward BDDs (FBDDs) [Tsamoura et al., 2020, Vlasselaer et al., 2015] are omitted because the former uses too much memory and the implementation of the latter seems to be broken at the time of writing.

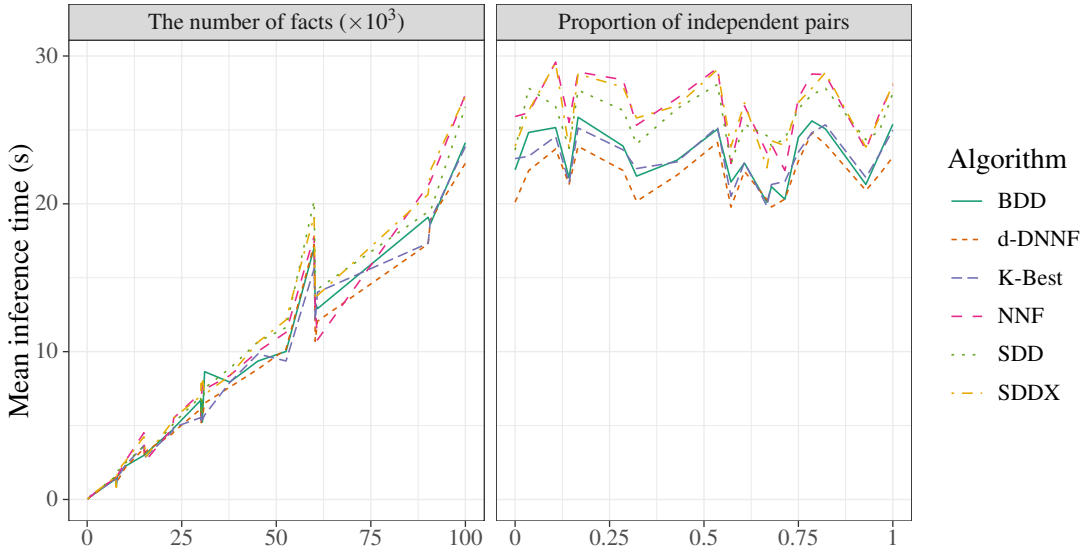


Figure 6.5: Mean inference time for a range of ProbLog inference algorithms as a function of the total number of facts in the program and the proportion of independent pairs of predicates. For the second plot, the number of facts is fixed at 10^5.

(including parameters not shown in the figure). Unsurprisingly, the most important predictor of inference time is the number of facts. However, after fixing the number of facts to a constant value, we can still observe that inference becomes harder with higher arity predicates as well as when facts are mostly probabilistic (see Figure 6.4). Finally, according to Figure 6.5, the independence structure of a program does not affect inference time, i.e., state-of-the-art inference algorithms—although they are supposed to [Fierens et al., 2011]—do not exploit situations where separate parts of a program can be handled independently.

6.10 Conclusion

We described a constraint model for generating both logic programs and probabilistic logic programs. The model avoids unnecessary symmetries, is reasonably efficient and supports additional constraints such as predicate independence. Our experimental results provide the first comparison of inference algorithms for probabilistic logic programming languages that generalises over programs, i.e., is not restricted to just a few programs and data sets. On the one hand, the experimental results revealed a weakness shared by all of the inference algorithms, i.e., the inability to ignore the part of a program that is easily seen to be irrelevant to the given query. On the other hand, all of

the algorithms behaved identically. Chapter 7 continues the quest to better understand the differences among WMC algorithms by generating WMC instances with varying structural properties.

We end the chapter by outlining two directions for future work. First, the experimental evaluation in Section 6.9.1 revealed scalability issues, particularly concerning the length/complexity of clauses. However, this particular issue is likely to resolve itself if the format of a clause is restricted to a conjunction of literals. Second, random instance generation typically focuses on either realistic instances or sampling from a simple and well-defined probability distribution. Our approach can be used to achieve the former, but it is an open question how it could accommodate the latter.

Chapter 7

Generating Random WMC Instances

7.1 Introduction

The experimental comparison of WMC algorithms on random probabilistic logic programs in Chapter 6 failed to discern any differences among the algorithms. However, both Chapters 3 and 4 and recent work by others [Dudek et al., 2020a,b, Lagniez and Marquis, 2017], show most WMC algorithms performing very similarly overall but with overwhelming differences when run on specific subsets of data. Examples of such segregating data sets include bipartite Bayesian networks by Sang et al. [2005a] and relational Bayesian networks by Chavira et al. [2006] that encode reachability in graphs under node deletion. So far, such performance differences remain unexplained. However, knowledge about the nature of these differences can inform our choices and aid in further algorithmic developments. Moreover, identifying performance predictors of algorithms is often an important step in developing a portfolio approach to the problem [Xu et al., 2008]. Lastly, if new algorithms are always tested on the same set of benchmarks, eventually they may become somewhat fitted to the particular characteristics of those instances, leading to algorithms that may perform worse when run on new types of data [Hossain et al., 2010].

Both theoretical and experimental analysis of SAT (and, to a lesser extent, #SAT) algorithms on random instances is a rich area of research spanning almost forty years. Variations of some of the first random models ever proposed [Franco and Paull, 1983, Purdom Jr. and Brown, 1983] continue to be instrumental up to this day for, e.g., establishing the location of the threshold between satisfiable and unsatisfiable instances [Achlioptas and Moore, 2002] and efficiently approximating #SAT [Galanis et al., 2020]. Other random models consider non-uniform variable frequencies [Ansótegui

et al., 2009], fixing the number of times each variable occurs both positively and negatively [Coja-Oghlan and Wormald, 2018], and adding other constraints such as cardinality and 'exclusive or' [Pote et al., 2019]. In contrast, only one WMC algorithm so far has been analysed using random instances [Sang et al., 2004, 2005b]. The goal of this chapter is to explain some of the differences between WMC algorithms via an experimental study that uses random instances.

Experimental work investigating how SAT algorithms behave on random instances is typically centred around parameters that describe each instance independently of its size. The most well-known parameter is the ratio of clauses to variables (i.e., *(clause) density*). Early work in the area showed random 3-SAT instances to be at their hardest when density is around 4.25 [Mitchell et al., 1992]. Later work revealed that the interaction between density and empirical hardness is much more solver-dependent [Coarfa et al., 2003]. Many other parameters such as heterogeneity, locality, and modularity have emerged from attempts to generate random instances similar to industry benchmarks for SAT [Ansótegui et al., 2009, Bläsius et al., 2019, Giráldez-Cru and Levy, 2016, 2017].

What parameter(s) are most appropriate to study WMC? Theoretical upper bounds on the performance of various WMC algorithms typically include a factor exponential in the primal treewidth of the input formula (or a closely related notion) [Bacchus et al., 2009, Darwiche, 2001a, 2004, Sang et al., 2004]. However—as we show in Section 7.3—instances generated by a standard random model for k-CNF formulas fail to exhibit enough variance in primal treewidth for us to infer its effect on the behaviour of the algorithms. Therefore, we present an extension of this model with a parameter that influences primal treewidth. The performance of WMC algorithms that use data structures called *algebraic decision diagrams* (ADDs) [Bahar et al., 1997] is also known to depend on the numerical values of weights [Dudek et al., 2020a,b]. Thus, our random model also includes two parameters that control redundancies in these values. We also investigate the effect of redundant weight values (e.g., having weights set to zero and one or having the same weight repeat many times) on the running times of the algorithms.

In addition to introducing a new random model for WMC instances, the contributions of this chapter include several key experimental findings about the behaviour of WMC algorithms—namely, C2D[1] [Darwiche, 2004], CACHET[2] [Sang et al., 2004], D4[3]

[1] http://reasoning.cs.ucla.edu/c2d/
[2] https://cs.rochester.edu/u/kautz/Cachet/
[3] https://www.cril.univ-artois.fr/KC/d4.html

[Lagniez and Marquis, 2017], DPMC[4] [Dudek et al., 2020b], and MINIC2D[5] [Oztok and Darwiche, 2015]—on random instances. First, we show that the easy-hard-easy pattern with respect to (w.r.t.) density is different for dynamic programming algorithms than it is for all other algorithms. Second, we present statistical evidence that all the algorithms scale exponentially w.r.t. primal treewidth and estimate how the base of that exponential changes w.r.t. density. Third, we show how the performance of ADD-based algorithms gradually improves w.r.t the proportion of weights that have repeating values and sharply improves w.r.t. the proportion of weights set to zero and one.

7.2 Background on WMC Algorithms

In this section, we briefly review the three major approaches to WMC—search, knowledge compilation, and dynamic programming—and their corresponding algorithms. The main search-based WMC algorithm CACHET [Sang et al., 2004] is based on a conflict-driven clause learning SAT solver [Moskewicz et al., 2001], which is then extended with a component caching scheme and adapted to counting.

Recall that knowledge compilation refers to transformations of propositional formulas into more restrictive formats that make various operations (such as model counting) tractable in the size of the representation [Darwiche and Marquis, 2002]. C2D [Darwiche, 2004], D4 [Lagniez and Marquis, 2017], and MINIC2D [Oztok and Darwiche, 2015] are all algorithms of this type. C2D compiles to deterministic decomposable negation normal form (d-DNNF) [Darwiche, 2001b]. Similarly, D4 compiles to decision-DNNF (also known as decomposable decision graphs) [Fargier and Marquis, 2006]. The only difference between d-DNNF and decision-DNNF is that decision-DNNF has if-then-else constructions instead of disjunctions [Lagniez and Marquis, 2017]. Finally, MINIC2D compiles to decision-SDDs—a subset of sentential decision diagrams (SDDs) that form a subset of d-DNNF [Darwiche, 2011].

All of the algorithms mentioned above execute in exactly the same way regardless of whether computing WMC or #SAT. Two recent WMC algorithms instead use data structures whose size (and thus the runtime of the algorithm) depends on the numerical values of weights. These data structures are representations of pseudo-Boolean functions, i.e., functions of the form $f\colon 2^X \to \mathbb{R}_{\geq 0}$, where X is a set, and 2^X denotes its powerset. ADDMC is the first such algorithm [Dudek et al., 2020a]. It uses ADDs

[4] https://github.com/vardigroup/dpmc
[5] http://reasoning.cs.ucla.edu/minic2d/

to represent pseudo-Boolean functions, combining and simplifying them in a bottom-up dynamic programming fashion. Since the size of an ADD for f depends on the cardinality of the range of f [Bahar et al., 1997], the performance of the algorithm is sensitive to the numerical values of weights, e.g., to how frequently they repeat. DPMC extends ADDMC in two ways [Dudek et al., 2020b]. First, DPMC allows for the order and nesting of operations on ADDs to be determined from an approximately-minimal-width tree decomposition rather than by heuristics.[6] Second, tensors are offered as an alternative to ADDs.

In all known parameterised complexities of WMC algorithms, the exponential factor is a function of primal treewidth or a closely related parameter. Interestingly, C2D is specifically designed to handle high primal treewidth (which the author refers to as *connectivity* [Darwiche, 1999]) and improves upon an earlier algorithm that has $O(mw2^w)$ time complexity, where m is the number of clauses, and w is the width of the decomposition tree which is known to be at most primal treewidth [Darwiche, 2001a, 2004]. While the complexity of CACHET was not analysed directly, the algorithm is based on component caching which is known to have a $2^{O(w)}n^{O(1)}$ time complexity, where n is the number of variables, and w is the branchwidth of the underlying hypergraph [Bacchus et al., 2009, Sang et al., 2004], which is known to be within a constant factor of primal treewidth [Robertson and Seymour, 1991]. Similarly, the complexity of DPMC is not described in the paper, although the authors define a notion of width w that is at most primal treewidth plus one and estimate the running time of the (execution part of the) algorithm to be proportional to 2^w [Dudek et al., 2020b].

7.3 Random Formulas with Varying Primal Treewidth

Notation. For any graph G, we write $\mathcal{V}(G)$ for its set of nodes and $\mathcal{E}(G)$ for its set of edges. Let S be a finite set. We write $\mathcal{U}S$ for the discrete uniform probability distribution on S. We represent any other probability distribution as a pair (S, p) where $p\colon S \to [0,1]$ is a probability mass function. For any probability distribution $\mathcal{P}$, we write $x \leftsquigarrow \mathcal{P}$ to denote the act of sampling x from $\mathcal{P}$. For instance, $x \leftsquigarrow (\{1,2\}, \{1 \mapsto 0.1, 2 \mapsto 0.9\})$ means that x becomes equal to 1 with probability 0.1 or to 2 with probability 0.9.

Our random model is based on the following parameters:

- the number of variables $\nu \in \mathbb{N}^+$,

[6] There is also a recent line of work in using tree decompositions to guide the heuristics of search-based model counters [Korhonen and Järvisalo, 2021].

- density $\mu \in \mathbb{R}_{>0}$,
- clause width $\kappa \in \mathbb{N}^+$ (for k-CNF formulas, $\kappa = k$),
- a parameter $\rho \in [0,1]$ that influences the primal treewidth of the formula,
- the proportion $\delta \in [0,1]$ of variables x such that $w(x) = 1$ and $w(\neg x) = 0$ or $w(x) = 0$ and $w(\neg x) = 1$,
- and the proportion $\varepsilon \in [0, 1-\delta]$ of variables x such that $w(x) = w(\neg x) = 0.5$.

The first three parameters are the standard parameters used to generate random k-CNF formulas with $\nu\mu$ clauses (up to rounding). We expect to observe (possibly different) values of μ that maximize the running time of each algorithm for fixed values of ν and κ. Parameters δ and ε control the numerical values of weights and are part of the model because the running time of DPMC [Dudek et al., 2020b]—and other algorithms based on ADDs—depends on these values. Weights such as zero and one are particularly 'simplifying' because they are respectively the additive and multiplicative identities. Having them propagate through the algorithm reduces the size of many ADDs used by DPMC, making the algorithm more efficient. Including many copies of the same weight (e.g., 0.5) can similarly simplify ADDs as well. Other WMC algorithms are indifferent to the numerical values of weights.

The process behind generating random k-CNF formulas is summarized as Algorithm 7.1. For the rest of this section, let $x_1, x_2, \ldots, x_\nu$ be the variables of the formula under construction. We simultaneously construct both formula ϕ and its primal graph G.[7] Each iteration of the first for-loop adds a clause to ϕ. This is done by constructing a set X of variables to be included in the clause, and then randomly adding either x or $\neg x$ to the clause for each $x \in X$ on line 10. Function `newVariable` randomly selects each new variable x, and lines 7–9 add x to the graph and the formula while also adding edges between x and all the other variables in the clause. To select each variable, line 13 defines set N to contain all edges with exactly one endpoint in X. The edges that will be added to G by line 8 will form a subset of N. If $N = \emptyset$, we select the variable uniformly at random (u.a.r.) from all viable candidates. Otherwise, ρ determines how much we bias the uniform distribution towards variables that would introduce the smallest number of new edges to G.

[7] The idea to directly take the primal graph into consideration while generating the formula is new—cf. random SAT instance generators based on, e.g., adversarial evolution [Hossain et al., 2010] and community structure [Giráldez-Cru and Levy, 2016].

Algorithm 7.1: Generating a random formula.

Input: $\nu, \kappa \in \mathbb{N}^+$ such that $\kappa < \nu$, $\mu \in \mathbb{R}_{>0}$, $\rho \in [0,1]$.

Output: A k-CNF formula ϕ.

$\phi \leftarrow$ empty CNF formula;
$G \leftarrow$ empty graph;
for $i \leftarrow 1$ **to** $\lfloor \nu\mu \rfloor$ **do**
  $X \leftarrow \emptyset$;
  for $j \leftarrow 1$ **to** κ **do**
    $x \leftarrow$ `newVariable(`X, G`)`;
    $\mathcal{V}(G) \leftarrow \mathcal{V}(G) \cup \{x\}$;
    $\mathcal{E}(G) \leftarrow \mathcal{E}(G) \cup \{\{x,y\} \mid y \in X\}$;
    $X \leftarrow X \cup \{x\}$;
  $\phi \leftarrow \phi \cup \{l \leftsquigarrow \mathcal{U}\{x, \neg x\} \mid x \in X\}$;
return ϕ;
Function `newVariable(`*set of variables X, primal graph G*`)`**:**
  $N \leftarrow \{e \in \mathcal{E}(G) \mid |e \cap X| = 1\}$;
  if $N = \emptyset$ **then return** $x \leftsquigarrow \mathcal{U}(\{x_1, x_2, \ldots, x_\nu\} \setminus X)$;
  return $x \leftsquigarrow \left(\{x_1, x_2, \ldots, x_\nu\} \setminus X,\ y \mapsto \frac{1-\rho}{\nu - |X|} + \rho \frac{|\{z \in X \mid \{y,z\} \in \mathcal{E}(G)\}|}{|N|}\right)$;

When $\rho = 0$, Algorithm 7.1 reduces to what has become the standard random model for k-CNF formulas. Equivalently to Franco and Paull [1983], we independently sample a fixed number of clauses, each clause has no duplicate variables, and each variable becomes either a positive or a negative literal with equal probabilities. At the other extreme, when $\rho = 1$, the first variable of a clause is still chosen u.a.r., but all other variables are chosen from those that already coappear in a clause (if possible). The probability that a variable is selected to be included in a clause scales linearly w.r.t. the proportion of edges in N that would be repeatedly added to G if the variable y was added to the clause. This is an arbitrary choice (which appears to work well, see Section 7.3.1) although alternatives (e.g., exponential scaling) could be considered. As long as $\rho < 1$, every k-CNF formula retains a positive probability of being generated by the algorithm.

To transform the generated formula into a WMC instance, we need to define weights on literals.[8] We want to partition all variables into three groups: those with weights

[8]Recall that in Chapters 3 and 4 we showed that algorithms such as DPMC and ADDMC [Dudek et al., 2020a,b] become more efficient when equipped with a different input format that assigns weights

equal to zero and one, those with weights equal to 0.5, and those with arbitrary weights, where the size of each group is determined by δ and ε. To do this, we sample a permutation $\pi \leftsquigarrow \mathcal{U}S_\nu$ (where S_ν is the permutation group on $\{1,2,\ldots,\nu\}$), and assign to each *variable* x_n a weight drawn u.a.r. from

- $\mathcal{U}\{0,1\}$ if $\pi(n) \leq \nu\delta$,
- $\mathcal{U}\{0.5\}$ if $\nu\delta < \pi(n) \leq \nu\delta + \nu\varepsilon$,
- and $\mathcal{U}\{0.01, 0.02, \ldots, 0.99\}$[9] if $\pi(n) > \nu\delta + \nu\varepsilon$.

We extend these weights to weights on *literals* by choosing the weight of each positive literal to be equal to the weight of its variable, and the weight of each negative literal to be such that $w(x) + w(\neg x) = 1$ for all variables x. This restriction is to ensure consistent answers among the algorithms.

Example 7.1. Let $\nu = 5$, $\mu = 0.6$, $\kappa = 3$, $\rho = 0.3$, $\delta = 0.4$, and $\varepsilon = 0.2$ and consider how Algorithm 7.1 generates a random instance. Since $\kappa = 3$, and $\lfloor \nu\mu \rfloor = 3$, the algorithm will generate a 3-CNF formula with three clauses.

For the first variable of the first clause, we are choosing u.a.r. from $\{x_1, x_2, \ldots, x_5\}$. Suppose the algorithm chooses x_5. Graph G then gets its first node but no edges. The second variable is chosen u.a.r. from $\{x_1, x_2, x_3, x_4\}$. Suppose the second variable is x_2. Then G gets another node and its first edge between x_2 and x_5. The third variable in the first clause is similarly chosen u.a.r. from $\{x_1, x_3, x_4\}$ because the only edge in G has both endpoints in $X = \{x_2, x_5\}$, and so $N = \emptyset$. Suppose the third variable is x_1. Graph G becomes a triangle connecting x_1, x_2, and x_5. Each of the three variables is then added to the clause as either a positive or a negative literal (with equal probabilities). Thus, the first clause becomes, e.g., $\neg x_5 \vee x_2 \vee x_1$.

The first variable of the second clause is chosen u.a.r. from $\{x_1, x_2, \ldots, x_5\}$. Suppose it is x_5 again. When the function `newVariable` tries to choose the second variable, $X = \{x_5\}$, and so $N = \{\{x_1, x_5\}, \{x_2, x_5\}\}$. The second variable is chosen from the discrete probability distribution

$$\Pr(x_1) = \Pr(x_2) = \frac{1 - 0.3}{5 - 1} + 0.3 \times \frac{1}{2} = 0.325$$

and

$$\Pr(x_3) = \Pr(x_4) = \frac{1 - 0.3}{5 - 1} = 0.175.$$

to formulas rather than literals.

[9] For convenience, we represent $(0,1)$ as 99 discrete values.

We skip the details of how all remaining variables and clauses are selected and consider the weight assignment. First, we shuffle the list of variables and get, e.g., $L = (x_4, x_3, x_2, x_1, x_5)$. This means that the first $\nu\delta = 5 \times 0.4 = 2$ variables of L get weights u.a.r. from $\{0, 1\}$, the next $\nu\varepsilon = 5 \times 0.2 = 1$ variable gets a weight of 0.5, and the remaining two variables get weights u.a.r. from $\{0.01, 0.02, \ldots, 0.99\}$. The weight function $w\colon \{x_1, x_2, \ldots, x_5, \neg x_1, \neg x_2, \ldots, \neg x_5\} \to [0, 1]$ can then be defined as, e.g., $w(x_4) = w(\neg x_3) = 0$, $w(x_3) = w(\neg x_4) = 1$, $w(x_2) = w(\neg x_2) = 0.5$, $w(x_1) = 0.23$, $w(\neg x_1) = 0.77$, $w(x_5) = 0.18$, and $w(\neg x_5) = 0.82$.

7.3.1 Validating the Model

The idea behind our model is that manipulating the value of ρ should allow us to generate instances of varying primal treewidth. Is this effect observable in practice? In addition, as WMC instances are mostly used for probabilistic inference, they tend to be satisfiable. Therefore, we want to filter out unsatisfiable instances from those generated by the model and need to ensure that the proportion of satisfiable instances remains sufficiently high. Given that higher values of ρ can result in constraints on variables being more localised and concentrated, we ask: are instances generated with higher values of ρ less likely to be satisfiable? To answer both questions, we run the following experiment.

Experiment 7.1. We fix $\nu = 100, \delta = \varepsilon = 0$, and consider random instances with $\mu = 2.5 \times \sqrt{2}^{-5}, 2.5 \times \sqrt{2}^{-4}, \ldots, 2.5 \times \sqrt{2}^{5}$, $\kappa = 2, 3, 4, 5$, and ρ going from 0 to 1 in steps of 0.01. For each combination of parameters, we generate ten instances.[10] We check if each instance is satisfiable using MiniSat[11] 2.2.0 [Eén and Sörensson, 2003] and calculate its (approximate) primal treewidth using htd[12] [Abseher et al., 2017].

Figure 7.1 shows the relationship between ρ and primal treewidth. Except for when both μ and κ are set to very low values (i.e., the formulas are small in both clause width and the number of clauses), primal treewidth decreases as ρ increases. This downward trend becomes sharper as μ increases, however, not uniformly: it splits into a roughly linear segment that approaches a horizontal line (for most values of ρ) and a sharply-decreasing segment that approaches a vertical line (when ρ is close to one). Higher values of κ seem to expedite this transition, i.e., with a higher value of κ, a lower

[10] Since one expects similar values of ρ to produce instances with similar properties, and ρ's are enumerate quite densely, generating only ten instances is sufficient.

[11] http://minisat.se/MiniSat.html

[12] https://github.com/mabseher/htd

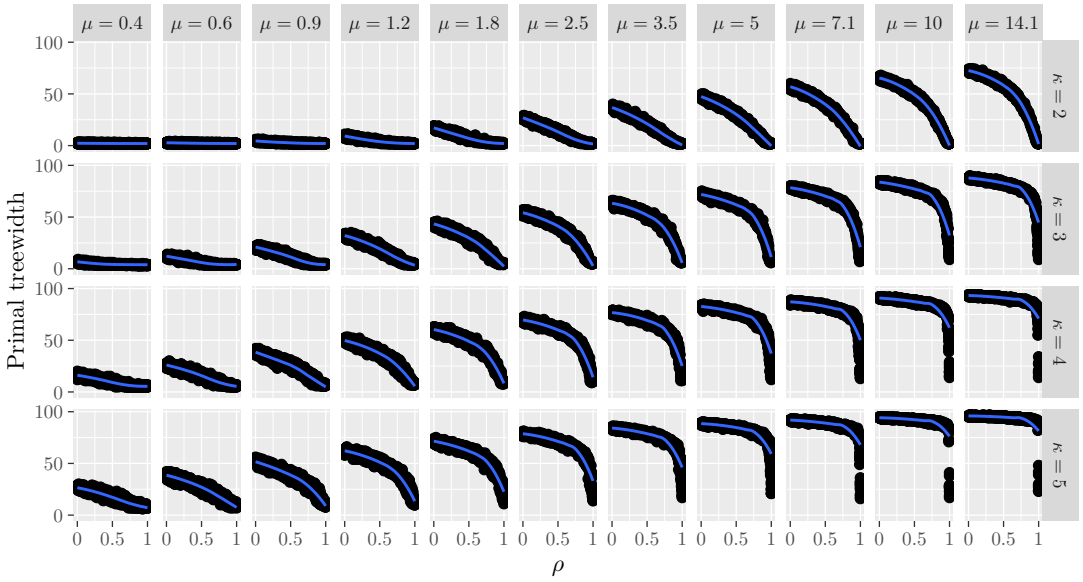


Figure 7.1: The relationship between ρ and primal treewidth for various values of μ and κ for k-CNF formulas from Experiment 7.1. Black points represent individual instances, and blue lines are smoothed means computed using locally weighted smoothing. The values of μ are rounded to one decimal place.

value of μ is sufficient for a smooth downward curve between ρ and primal treewidth to turn into a combination of a horizontal and a vertical line. While this behaviour may be troublesome when generating formulas with higher values of μ (almost all of which would be unsatisfiable), the relationship between ρ and primal treewidth is excellent for generating 3-CNF formulas close to and below the satisfiability threshold of 4.25 [Crawford and Auton, 1996].

Regarding satisfiability, the proportion of satisfiable 3-CNF formulas drops from 63.6 % when $\rho = 0$ to 50.9 % when $\rho = 1$, so—while ρ does affect satisfiability—the effect is not significant enough to influence our experimental setup in the next section.

7.4 Experimental Results

In this section, we describe three experiments that examine how the running times of WMC algorithms change w.r.t. the parameters of our random model. All experiments were run on Intel Xeon E5–2630 with Scientific Linux 7, GCC 10.2.0, Python 3.8.1, R 4.1.0, C2D 2.20 [Darwiche, 2004], CACHET 1.22 [Sang et al., 2004], HTD 1.2.0 [Abseher et al., 2017], and with no additional preprocessing. With both C2D and D4,

we use QUERY-DNNF[13] to compute the numerical answer from the compiled circuit. We omit ADDMC [Dudek et al., 2020a] from our experiments as it exceeds time and memory limits on too many instances; however, observations about the behaviour of DPMC [Dudek et al., 2020b] apply to ADDMC as well, with the addendum that the tree decomposition implicitly used by ADDMC may have a significantly higher width. DPMC is run with tree decomposition-based planning (using one iteration of HTD) and ADD-based execution—the combination that was originally found to be most effective. We restrict our attention to 3-CNF formulas, generate 100 satisfiable instances for each *combination* of parameters, and run each of the five algorithms with a 500 s time limit and an 8 GiB memory limit. While both limits are somewhat low, we prioritise large numbers of instances to increase the accuracy and reliability of our results. Unless stated otherwise, in each plot of this section, lines denote median values, and shaded regions show interquartile ranges. We run the following three experiments, setting $\nu = 70$ in all of them as we found that this produces instances of suitable difficulty.

Experiment 7.2 (Density and primal treewidth)**.** Let $\nu = 70$, μ go from 1 to 4.3 in steps of 0.3, ρ go from 0 to 0.5 in steps of 0.01, and $\delta = \varepsilon = 0$.

Experiment 7.3 (δ)**.** Let $\nu = 70$, $\mu = 2.2$[14], $\rho = 0$, δ go from 0 to 1 in steps of 0.01, and $\varepsilon = 0$.

Experiment 7.4 (ε)**.** Same as Experiment 7.3 but with $\delta = 0$ and ε going from 0 to 1 in steps of 0.01.

In each experiment, the proportion of algorithm runs that timed out never exceeded 3.8 %. While in Experiment 7.2 only 1 % of experimental runs ran out of memory, the same percentage was higher in Experiments 7.3 and 7.4—10 and 12 %, respectively. D4 [Lagniez and Marquis, 2017] and C2D are the algorithms that experienced the most issues fitting within the memory limit, accounting for 66–72 % and 28–33 % of such instances, respectively. We exclude the runs that terminated early due to running out of memory from the rest of our analysis.

In Experiment 7.2, we investigate how the running time of each algorithm depends on the density and primal treewidth by varying both μ and ρ. The results are in Figure 7.2. The first thing to note is that the peak hardness w.r.t. density occurs at around 1.9 for all

[13] http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html

[14] Experiment 7.2 shows this density to be the most challenging for DPMC.

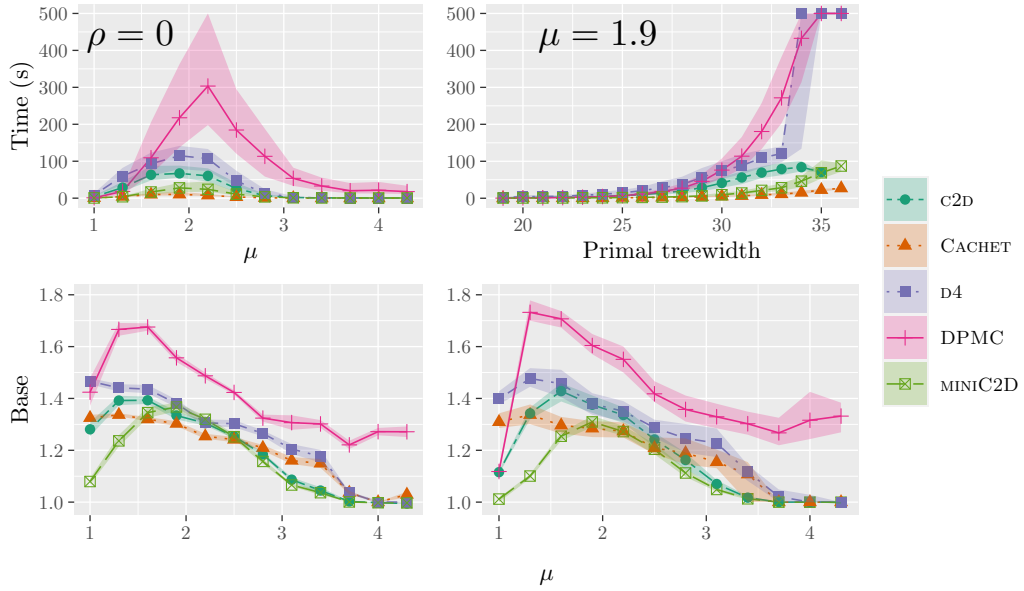


Figure 7.2: Visualisations of the data from Experiment 7.2. The top-left plot shows how the running time of each algorithm changes w.r.t. density when $\rho = 0$. The top-right plot shows changes in the running time of each algorithm w.r.t. primal treewidth with μ fixed at 1.9. The plots at the bottom show how the estimated base of the exponential relationship between primal treewidth and the runtime of each algorithm depends on μ. The bottom-left plot is for the simple linear model (with shaded regions showing standard error), and the bottom-right plot uses the estimates provided by ESA [Pushak and Hoos, 2020] (with shaded regions showing 95 % confidence intervals).

algorithms except for DPMC, which peaks at 2.2 instead.[15] This finding is consistent with previous work, which shows Cachet to peak at 1.8 [Sang et al., 2004].[16]

The other question we want to investigate using this experiment is how each algorithm scales w.r.t. primal treewidth. The top-right plot in Figure 7.2 shows this relationship for a fixed value of μ, and one can see some evidence that the running time of DPMC grows faster w.r.t. primal treewidth than the running time of the other algorithms. We use two statistical techniques to quantify this growth: a simple linear regression model and the empirical scaling analyzer (ESA) v2[17] [Pushak and Hoos, 2020]. In both cases, for each algorithm and value of μ in Experiment 7.2, we select the median runtime for all available values of primal treewidth. In the former case, we fit the model $\ln t \sim \alpha w + \beta$, where t is the median running time of the algorithm, w is the

[15] While exact values might be hard to read from the plot, they are confirmed by numerical data.

[16] For comparison, #SAT algorithms have been observed to peak at densities 1.2 and 1.5 [Bayardo Jr. and Pehoushek, 2000].

[17] `https://github.com/YashaPushak/ESA`

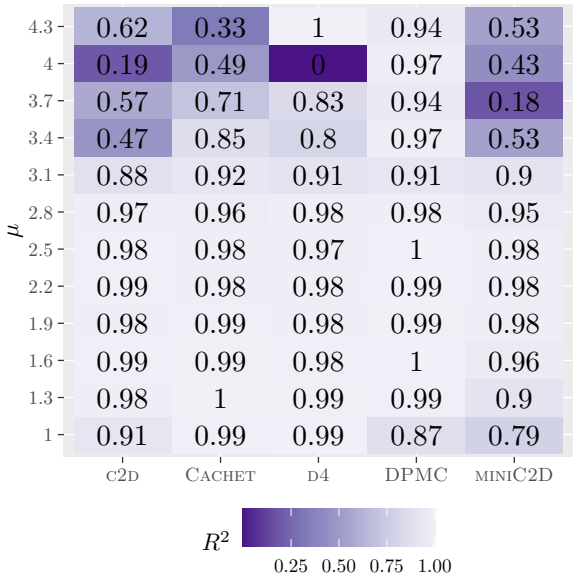


Figure 7.3: The coefficients of determination (rounded to one decimal place) of all the linear models fitted for the top-right subplot of Figure 7.2.

primal treewidth, and α and β are parameters.[18] In other words, this model attempts to express median running time as $e^{\beta}(e^{\alpha})^{w}$. In the latter case, we run ESA with 1001 bootstrap samples, a window of 101, and use the first 30 % of the data for training.

The results of both models are qualitatively the same (with the exception of DPMC run on instances with $\mu = 1$) and are displayed at the bottom of Figure 7.2. We find that, indeed, DPMC scales worse w.r.t. primal treewidth than any other algorithm across all values of μ and is the only algorithm that does not become indifferent to primal treewidth when faced with high-density formulas. A second look at the top-left subplot of Figure 7.2 suggests an explanation for the latter observation. The running times of all algorithms except for DPMC approach zero when $\mu > 3$ while the median running time of DPMC approaches a small non-zero constant instead. This observation also explains why Figure 7.3 shows that the fitted models fail to explain the data for non-ADD algorithms running on high-density instances—the running times are too small to be meaningful. In all other cases, an exponential relationship between primal treewidth and runtime fits the experimental data remarkably well.

Another thing to note is that MINIC2D [Oztok and Darwiche, 2015] is the only algorithm that exhibits a clear low-high-low pattern in the bottom subplots of Figure 7.2.

[18] Similar statistical analyses have been used to investigate polynomial-to-exponential phase transitions in SAT [Coarfa et al., 2003] and the behaviour of SAT solvers on CNF-XOR formulas [Dudek et al., 2017].

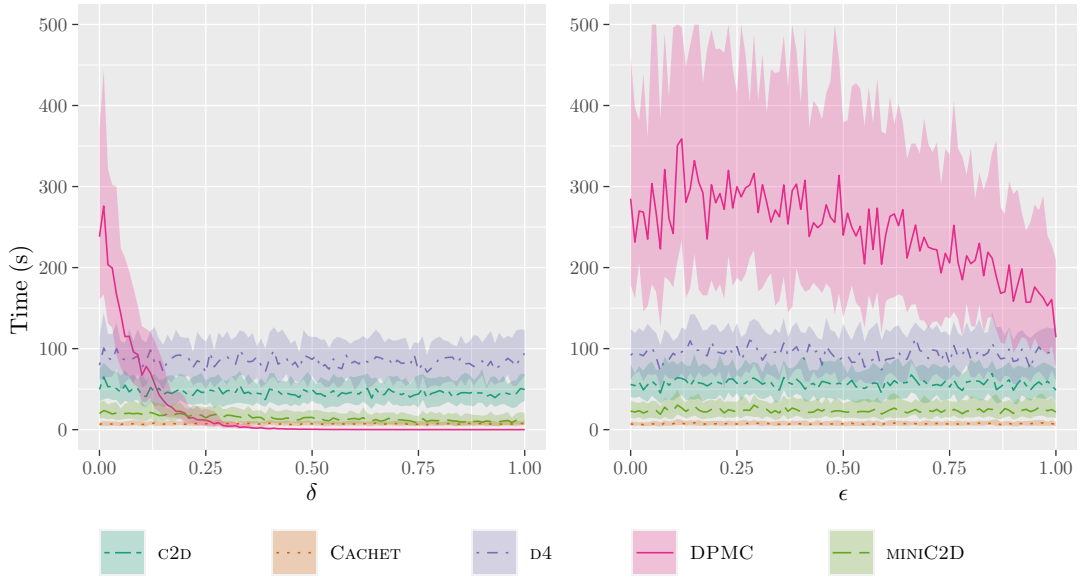


Figure 7.4: Changes in the running time of each algorithm as a result of changing δ (on the left-hand side) and ε (on the right-hand side) according to the data from Experiments 7.3 and 7.4.

To a smaller extent, the same may apply to c2d and DPMC as well, although the evidence for this is limited due to relatively large gaps between different values of μ in Experiment 7.2. In contrast, the running times of Cachet and d4 remain dependent on primal treewidth even when the density of the WMC instance is very low, suggesting that miniC2D should have an advantage on low-density high-primal-treewidth instances.

Finally, Experiments 7.3 and 7.4 investigate how changing the numerical values of weights can simplify a WMC instance. The results are in plotted Figure 7.4. As expected, the running time of all algorithms other than DPMC stay the same regardless of the value of δ or ε. The running time of DPMC, however, experiences a sharp (exponential?) decline with increasing δ. The decline w.r.t. ε is also present, although significantly less pronounced and with high variance.

How are these random instances different from real data? As a representative sample, we take the WMC encodings of Bayesian networks created using the method by Sang et al. [2005a] as found in the experimental setup[19] of the DPMC paper [Dudek et al., 2020b]. A typical WMC instance has $\nu = 200$ variables, half of which have equal weights (i.e., $\varepsilon = 0.5$), an average clause width of $\kappa = 2.6$, a density of $\mu = 2.5$, and a primal treewidth of 28. Our random instances have fewer variables and (for the most part) lower density. Another important difference is that our instances are in k-CNF

[19] https://github.com/vardigroup/DPMC/releases

whereas a typical encoding of a Bayesian network has many two-literal clauses mixed with clauses of various longer widths. Despite real instances having more variables, their primal treewidth is rather low. Perhaps this partially explains why the performance of DPMC is in line with the performance of all other algorithms on traditionally-used benchmarks [Dudek et al., 2020b] despite struggling with most of our random data.

To sum, we found that C2D and D4 are the most memory-intensive algorithms, CACHET is great on random instances in general, MINIC2D exceeds on low-density high-primal-treewidth instances, and DPMC is at its best on low-density low-primal-treewidth instances. Furthermore, a median instance with all weights equal to each other is about three times easier for DPMC than a median instance with random weights. Another important observation is about how peak hardness w.r.t. density depends on the algorithm: DPMC peaks at a higher density than all other WMC algorithms, which peak at a higher density than (some) #SAT algorithms.

7.5 Conclusions and Future Work

In this chapter, we studied the behaviour of and differences among WMC algorithms on random instances generated by a standard model for k-CNF formulas extended with parameters that control primal treewidth and literal weights. Among other things, we established statistical evidence for the existence of an exponential relationship between primal treewidth and the running time of all WMC algorithms. The running time of ADD-based algorithms was observed to peak at a higher density, scale worse w.r.t. primal treewidth, and depend negatively on repeating weight values compared to algorithms based on search or knowledge compilation. These observations can, to some degree, be extended to a closely related weighted projected model counting algorithm [Dudek et al., 2021] as well as to other applications of ADDs more generally, e.g., probabilistic inference [Chavira and Darwiche, 2007, Gogate and Domingos, 2011] and stochastic planning [Hoey et al., 1999].

One limitation of our work is that variability in primal treewidth was achieved via a parameter, and this could bias randomness in some unexpected way (although it is encouraging that there is only a slight decrease in the proportion of satisfiable instances between $\rho = 0$ and $\rho = 1$). Perhaps a theoretical investigation of the proposed model is warranted, including a characterisation of how ρ influences primal treewidth and the structure of the primal graph more generally. Since treewidth is widely used in parameterised complexity [Downey and Fellows, 2013], formally establishing a

connection with ρ could make our random model useful for a variety of other hard computational problems.

To keep the number of experiments feasible, we restricted our attention to 3-CNF formulas, although, of course, this is not very representative of real-world WMC instances. The model could be adapted to generate non-k-CNF formulas, and perhaps a more representative structure could be achieved by introducing new variables that clauses define to be equivalent to select conjunctions of literals as is done in one of the WMC encodings for Bayesian networks [Darwiche, 2002].

Chapter 8

Conclusion

In Section 8.1 we review the contributions of this thesis and in Section 8.2 we provide a perspective on how future work could develop, either directly or indirectly building on the results of our work.

8.1 Contributions

The contributions of this thesis can be divided into two parts:

- empirically-motivated contributions that make something new possible or something that already exists more efficient
- and (conceptual, theoretical, or experimental) contributions that help us understand something more fully or in a new way.

On the empirical front, most of our contributions focus on the efficiency and tractability of the propositional and first-order variants of WMC. In Chapters 3 and 4, we show how the efficiency of WMC can be improved by generalising weights from their standard definition based on literals to one capable of representing a richer subset of all possible pseudo-Boolean functions. In Chapter 5, we extend the capabilities of FORCLIFT [Van den Broeck et al., 2011] so that it is able to solve more instances in a lifted manner, e.g., instances with injective mappings. The empirical contributions of Chapters 6 and 7 are about making new things possible, i.e., introducing novel tools and methods. In Chapter 6, we developed a constraint model for (probabilistic) logic programs that can be used to generate random programs or enumerate all small programs under some given constraints. The constraints include various notions of size, the structure/complexity of a clause, and the independence of random variables. Finally,

in Chapter 7, we present a way to generate propositional formulas in CNF with varying primal treewidth. As treewidth is a well-known parameter commonly used to describe parameterised complexity results [Downey and Fellows, 2013], the same model (or a variation thereof) can be used in experimental studies of many other logic-based problems as well.

The experimental work in these last two chapters, i.e., Chapters 6 and 7, also contains important observations about WMC and probabilistic inference algorithms. First, Chapter 6 demonstrates remarkable similarities among ProbLog inference algorithms. This observation suggests that the bottleneck of ProbLog inference (at least across our random instances) might be related to logic programming more than WMC. Second, Chapter 7 reveals, among other things, that WMC algorithms based on algebraic decision diagrams (ADDs) and dynamic programming scale worse with primal treewidth and work better with instances that have fewer clauses (i.e., lower density) compared to other algorithms. Understanding such differences among algorithms is important in the development of new algorithms, algorithm portfolios, and hybrid approaches to WMC. Back in Chapter 3, we show how WMC can be seen as the problem of computing the value of a measure on some element of a Boolean algebra. This insight leads us to consider generalised weight functions that express, e.g., conditional probabilities more succinctly and can lead to improved probabilistic inference speed for Bayesian networks. In Chapter 4, we continue the work on generalising WMC and formally define the generalisation as pseudo-Boolean projection (PBP). Moreover, we show that previous work on WMC encodings is not in vain and the benefits can (in most cases) be transferred to PBP. Lastly, Chapter 5 contains two important lessons. First, 'circuits' with cycles can be more expressive than their acyclic predecessors. Second, first-order model counting (and first-order knowledge compilation in particular) can discover the definitions of recursive functions (including recurrence relations) that capture the model count of a given sentence.

In summary, we

- introduced new foundations for WMC based on measures on Boolean algebras,
- generalised WMC to PBP,
- introduced new encoding schemes and encoding transformation algorithms,
- introduced CRANE, i.e., a more powerful version of FORCLIFT that works with graphs rather than circuits,

- and provided algorithms for random instance generation.

8.2 Future Directions

In this section, we present a broad overview of how our contributions and the questions raised by this work could be taken forward and influence key areas of research in computer science, artificial intelligence, and mathematics.

8.2.1 Algorithms and Applications

In this thesis, we contributed to the development and applicability of three WMC algorithms: ADDMC [Dudek et al., 2020a], DPMC [Dudek et al., 2020b], and FORCLIFT [Van den Broeck et al., 2011]. The first two are propositional WMC algorithms based on ADDs whereas FORCLIFT is a WMC algorithm for first-order logic based on knowledge compilation. While in this work we focused exclusively on exact algorithms, all of them could be adapted to approximate instead. An approximation technique called *lifted relax, compensate and then recover* is already part of FORCLIFT [Van den Broeck et al., 2012], so it would only need to be adapted to the generalised setting of CRANE. Likewise, approximate computations using ADDs have already been studied [St-Aubin et al., 2000], so, e.g., DPMC could be extended to approximate as well.

Most weighted first-order model counting (WFOMC) algorithms try to solve each instance in a lifted manner (i.e., run in polynomial time with respect to the sizes of the domains involved) and fail if unsuccessful. FORCLIFT is an exception as it supports using a (propositional) WMC algorithm for parts of the problem that cannot be solved by other compilation rules. Can this transition to WMC be implemented more efficiently, i.e., without fully grounding the instance? Is the WFOMC algorithm better off constructing its own exponential-time solution instead of relying on a WMC algorithm (and is that even possible)? The only way WFOMC can become the standard approach to probabilistic inference in statistical relational models is by being able to gracefully handle all instances, even if it means abandoning efficiency guarantees.

Finally, ample opportunities remain to improve WMC encodings that already exist as well as connect WMC to new problem domains. In particular, we showed how PBP encodings of Bayesian networks are much smaller than the equivalent WMC encodings and can be handled more efficiently by a WMC algorithm. Designing PBP encodings for

other applications of WMC and to new problem domains could be similarly beneficial. Moreover, back in Chapter 1 we compared WMC to a range of other computational problems that ask to compute a sum of products. Establishing efficient reductions among these problems could yield new fixed-parameter tractable algorithms and/or improvements to the empirical state of the art. Similarly, adapting a WMC algorithm to a semiring other than $(\mathbb{R}_{\geq 0}, +, \cdot)$ could yield improvements to some of the related problems outlined by Kimmig et al. [2017].

8.2.2 Computational Complexity

Note that the execution of both WMC and WFOMC algorithms can be divided into two parts:

- looking for a *solution* (i.e., an arithmetic circuit/expression that computes the required sum of products) and
- performing the numerical computations that produce the final answer.

(The two are typically much more intertwined in the case of propositional WMC.) With this dichotomy in mind, one could ask: are the algorithms finding optimal solutions? How much of the total running time depends on the complexity of the solution, and how much on the algorithmic methods for finding one? Answers to these questions would highlight the weaknesses of state-of-the-art algorithms and direct the efforts of future research towards addressing these weaknesses.

On a more theoretical level, single-domain (W)FOMC problems compute sequences, many of which are well-known to mathematicians. Since there is significant interest in computing such sequences efficiently, the existence of many such sequences with no efficient formulas suggests that a tractable solution might not exist. However, we have no proof of that, i.e., no arithmetic circuit lower bounds for sequences that have been known for decades and are easy to describe in natural language and in logic. So far, the most notable hardness result states that there exists a sentence in first-order logic with three variables for which FOMC is $\#\mathsf{P}_1$-complete [Beame et al., 2015]. Having similar hardness results for sentences that are both simple and practical would be a significant advancement to the field.

8.2.3 Random Instances

In this thesis, we introduce two ways to generate random instances: one for (probabilistic) logic programs and one for propositional formulas in CNF that are then turned into WMC instances. As we provide the very first attempts at testing WMC algorithms on random data, many opportunities for improvements and future work remain.

An interesting opportunity to connect our work on random logic programs in Chapter 6 and on (W)FOMC in Chapter 5 is by adapting the constraint model to generate (W)FOMC instances instead of logic programs. This way one could systematically search for interesting instances that, e.g.,

- reveal differences in the runtime complexity of various algorithms or
- demonstrate a gap between the performance of state-of-the-art WFOMC algorithms and formulas constructed by hand.

There is also ample opportunity for theoretical contributions. For instance, one explanation for the surprising experimental results of Chapter 6 is that all of the generated instances yielded easy WMC problems, and the computational bottleneck was in the handling of the logic program before WMC. We can state this idea as a (somewhat informal) conjecture.

Conjecture 8.1. *With high probability, the WMC instance that results from a random probabilistic logic program generated by the constraint model in Chapter 6 is tractable for some WMC algorithm.*

8.2.4 Artificial Intelligence and Combinatorics

Our hope for the broader field of artificial intelligence is that—similarly to the resurgence of symbolic artificial intelligence amidst deep neural networks [Garnelo and Shanahan, 2019]—the field shifts some of its focus from numbers and probabilities to structural concepts such as functions and relations. Once a solution to, e.g., a WFOMC problem is formulated as a function f rather than the evaluation of f on some particular input values, richer ways of reasoning become available. For example, instead of asking whether a probability of some event is above/below some threshold in a particular situation, one could ask for conditions on the input values that are necessary for the probability to be sufficiently high/low. Such reasoning capabilities have clear benefits to the robustness of artificial agents and to explainability—another rapidly emerging area of research [Arya et al., 2019, Belle and Papantonis, 2021, Bueff et al., 2022].

Finally, for the benefit of both artificial intelligence and combinatorics, we would like to reiterate and expand on the notion of automatic enumerative combinatorialist by Barvínek et al. [2021]. Perhaps (W)FOMC can mature into an easy-to-use tool that can compute any function expressible in first-order logic, in many cases providing a simple solution via a combination of recursive functions. Similarly to how a constraint programmer describes the constraints and asks the solver for a solution, a combinatorialist could describe what needs to be counted in a logic-based format and receive recursive or asymptotic solutions, generating functions, etc.

Bibliography

M. Abseher, N. Musliu, and S. Woltran. htd - A free, open-source framework for (customized) tree decompositions and beyond. In D. Salvagnin and M. Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 376–386. Springer, 2017. doi: 10.1007/978-3-319-59776-8_30.

D. Achlioptas and C. Moore. The asymptotic order of the random k -SAT threshold. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*, pages 779–788. IEEE Computer Society, 2002. doi: 10.1109/SFCS.2002.1182003.

D. Agrawal, Y. Pote, and K. S. Meel. Partition function estimation: A quantitative study. In Z. Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 4276–4285. ijcai.org, 2021. doi: 10.24963/ijcai.2021/587.

G. Amendola, F. Ricca, and M. Truszczynski. Generating hard random Boolean formulas and disjunctive logic programs. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 532–538. ijcai.org, 2017. doi: 10.24963/ijcai.2017/75.

G. Amendola, F. Ricca, and M. Truszczynski. New models for generating hard random Boolean formulas and disjunctive logic programs. *Artif. Intell.*, 279, 2020. doi: 10.1016/j.artint.2019.103185.

C. Ansótegui, M. L. Bonet, and J. Levy. Towards industrial-like random SAT instances. In C. Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 387–392, 2009. URL `http://ijcai.org/Proceedings/09/Papers/072.pdf`.

V. Arya, R. K. E. Bellamy, P. Chen, A. Dhurandhar, M. Hind, S. C. Hoffman, S. Houde, Q. V. Liao, R. Luss, A. Mojsilovic, S. Mourad, P. Pedemonte, R. Raghavendra, J. T. Richards, P. Sattigeri, K. Shanmugam, M. Singh, K. R. Varshney, D. Wei, and Y. Zhang. One explanation does not fit all: A toolkit and taxonomy of AI explainability techniques. *CoRR*, abs/1909.03012, 2019.

R. A. Aziz, G. Chu, C. J. Muise, and P. J. Stuckey. #∃SAT: Projected model counting. In M. Heule and S. A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 2015. doi: 10.1007/978-3-319-24318-4_10.

F. Bacchus, S. Dalmao, and T. Pitassi. Solving #SAT and Bayesian inference with backtracking search. *J. Artif. Intell. Res.*, 34:391–442, 2009. doi: 10.1613/jair.2648.

F. Bacchus, M. Järvisalo, and R. Martins. Maximum satisfiability. In *Handbook of Satisfiability*, pages 929–991. IOS PRESS, 2021.

R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods Syst. Des.*, 10(2/3):171–206, 1997. doi: 10.1023/A:1008699807402.

I. Balbin, G. S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. *J. Log. Program.*, 11(3&4):295–344, 1991. doi: 10.1016/0743-1066(91)90030-S.

J. S. Baras and G. Theodorakopoulos. *Path Problems in Networks*. Synthesis Lectures on Communication Networks. Morgan & Claypool Publishers, 2010. doi: 10.2200/S00245ED1V01Y201001CNT003.

C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-825.

A. Bart, F. Koriche, J. Lagniez, and P. Marquis. An improved CNF encoding scheme for probabilistic inference. In G. A. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum, and F. van Harmelen, editors, *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 613–621. IOS Press, 2016. doi: 10.3233/978-1-61499-672-9-613.

J. Barvínek, T. van Bremen, Y. Wang, F. Zelezný, and O. Kuzelka. Automatic conjecturing of P-recursions using lifted inference. In N. Katzouris and A. Artikis, editors, *Inductive Logic Programming - 30th International Conference, ILP 2021, Virtual Event, October 25-27, 2021, Proceedings*, volume 13191 of *Lecture Notes in Computer Science*, pages 17–25. Springer, 2021. doi: 10.1007/978-3-030-97454-1_2.

R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In H. A. Kautz and B. W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pages 157–162. AAAI Press / The MIT Press, 2000. URL http://www.aaai.org/Library/AAAI/2000/aaai00-024.php.

P. Beame, G. Van den Broeck, E. Gribkoff, and D. Suciu. Symmetric weighted first-order model counting. In T. Milo and D. Calvanese, editors, *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 313–328. ACM, 2015. doi: 10.1145/2745754.2745760.

V. Belle. Open-universe weighted model counting. In S. P. Singh and S. Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 3701–3708. AAAI Press, 2017a. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15008.

V. Belle. Weighted model counting with function symbols. In G. Elidan, K. Kersting, and A. T. Ihler, editors, *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*. AUAI Press, 2017b. URL http://auai.org/uai2017/proceedings/papers/132.pdf.

V. Belle and L. De Raedt. Semiring programming: A semantic framework for generalized sum product problems. *Int. J. Approx. Reason.*, 126:181–201, 2020. doi: 10.1016/j.ijar.2020.08.001.

V. Belle and I. Papantonis. Principles and practice of explainable machine learning. *Frontiers Big Data*, 4:688969, 2021. doi: 10.3389/fdata.2021.688969.

V. Belle, A. Passerini, and G. Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In Q. Yang and M. J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2770–2776. AAAI Press, 2015. URL http://ijcai.org/Abstract/15/392.

M. Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition*. Springer, 2012. ISBN 978-1-4471-4128-0. doi: 10.1007/978-1-4471-4129-7.

N. Bidoit. Negation in rule-based database languages: A survey. *Theor. Comput. Sci.*, 78(1):3–83, 1991. doi: 10.1016/0304-3975(51)90003-5.

A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 2009. IOS Press. ISBN 978-1-58603-929-5.

C. M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007. ISBN 9780387310732. URL https://www.worldcat.org/oclc/71008143.

T. Bläsius, T. Friedrich, and A. M. Sutton. On the empirical time complexity of scale-free 3-SAT at the phase transition. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 2019. doi: 10.1007/978-3-030-17462-0_7.

B. Bliem, M. Moldovan, M. Morak, and S. Woltran. The impact of treewidth on ASP grounding and solving. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 852–858. ijcai.org, 2017. doi: 10.24963/ijcai.2017/118.

C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In E. Horvitz and F. V. Jensen, editors, *UAI '96: Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence, Reed College, Portland, Oregon, USA, August 1-4, 1996*, pages 115–123. Morgan Kaufmann, 1996.

I. Bratko. *Prolog Programming for Artificial Intelligence, 4th Edition*. Addison-Wesley, 2012. ISBN 978-0-3214-1746-6.

M. Bruynooghe, T. Mantadelis, A. Kimmig, B. Gutmann, J. Vennekens, G. Janssens, and L. De Raedt. ProbLog technology for inference in a probabilistic first order logic. In H. Coelho, R. Studer, and M. J. Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 719–724. IOS Press, 2010. doi: 10.3233/978-1-60750-606-5-719.

R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. doi: 10.1109/TC.1986.1676819.

A. Bueff, I. Papantonis, A. Simkute, and V. Belle. Explainability in machine learning: a pedagogical perspective. *CoRR*, abs/2202.10335, 2022.

R. Bunescu and R. Mooney. Statistical relational learning for natural language information extraction. *Statistical relational learning*, pages 535–552, 2007.

P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1997. ISBN 3-540-60582-7.

P. Carbonetto, J. Kisynski, N. de Freitas, and D. Poole. Nonparametric Bayesian logic. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*, pages 85–93. AUAI Press, 2005.

A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell. Toward an architecture for never-ending language learning. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1879`.

S. Chandra, J. Sahs, L. Khan, B. M. Thuraisingham, and C. C. Aggarwal. Stream mining using statistical relational learning. In R. Kumar, H. Toivonen, J. Pei, J. Z. Huang, and X. Wu, editors, *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 743–748. IEEE Computer Society, 2014. doi: 10.1109/ICDM.2014.144.

M. Chavira and A. Darwiche. Compiling Bayesian networks with local structure. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1306–1312. Professional Book Center, 2005. URL http://ijcai.org/Proceedings/05/Papers/0931.pdf.

M. Chavira and A. Darwiche. Encoding CNFs to empower component analysis. In A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2006. doi: 10.1007/11814948_9.

M. Chavira and A. Darwiche. Compiling Bayesian networks using variable elimination. In M. M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2443–2449, 2007. URL http://ijcai.org/Proceedings/07/Papers/393.pdf.

M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008. doi: 10.1016/j.artint.2007.11.002.

M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational Bayesian networks for exact inference. *Int. J. Approx. Reason.*, 42(1-2):4–20, 2006. doi: 10.1016/j.ijar.2005.10.001.

A. Choi, D. Kisa, and A. Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In L. C. van der Gaag, editor, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 12th European Conference, ECSQARU 2013, Utrecht, The Netherlands, July 8-10, 2013. Proceedings*, volume 7958 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 2013. doi: 10.1007/978-3-642-39091-3_11.

C. Coarfa, D. D. Demopoulos, A. S. M. Aguirre, D. Subramanian, and M. Y. Vardi. Random 3-SAT: The plot thickens. *Constraints An Int. J.*, 8(3):243–261, 2003. doi: 10.1023/A:1025671026963.

A. Coja-Oghlan and N. Wormald. The number of satisfying assignments of random regular k-SAT formulas. *Comb. Probab. Comput.*, 27(4):496–530, 2018. doi: 10.1017/S0963548318000263.

S. A. Cook. The complexity of theorem-proving procedures. In M. A. Harrison, R. B. Banerji, and J. D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. doi: 10.1145/800157.805047.

J. Côrte-Real, I. Dutra, and R. Rocha. On applying probabilistic logic programming to breast cancer data. In N. Lachiche and C. Vrain, editors, *Inductive Logic Programming - 27th International Conference, ILP 2017, Orléans, France, September 4-6, 2017, Revised Selected Papers*, volume 10759 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2017. doi: 10.1007/978-3-319-78090-0_3.

J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3-SAT. *Artif. Intell.*, 81(1-2):31–57, 1996. doi: 10.1016/0004-3702(95)00046-1.

A. Darwiche. Compiling knowledge into decomposable negation normal form. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 284–289. Morgan Kaufmann, 1999. URL `http://ijcai.org/Proceedings/99-1/Papers/042.pdf`.

A. Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001a. doi: 10.1145/502090.502091.

A. Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Appl. Non Class. Logics*, 11(1-2):11–34, 2001b. doi: 10.3166/jancl.11.11-34.

A. Darwiche. A logical approach to factoring belief networks. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M. Williams, editors, *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002*, pages 409–420. Morgan Kaufmann, 2002.

A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In R. L. de Mántaras and L. Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 328–332. IOS Press, 2004.

A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In T. Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 819–826. IJCAI/AAAI, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-143.

A. Darwiche and P. Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17: 229–264, 2002. doi: 10.1613/jair.989.

D. De Maeyer, J. Renkens, L. Cloots, L. De Raedt, and K. Marchal. PheNetic: network-based interpretation of unstructured gene lists in E. coli. *Molecular BioSystems*, 9(7): 1594–1603, 2013.

L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015. doi: 10.1007/s10994-015-5494-z.

L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In M. M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007. URL `http://ijcai.org/Proceedings/07/Papers/396.pdf`.

L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors. *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*. Springer, 2008. ISBN 978-3-540-78651-1. doi: 10.1007/978-3-540-78652-8.

L. De Raedt, K. Kersting, S. Natarajan, and D. Poole. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2016. doi: 10.2200/S00692ED1V01Y201601AIM032.

R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artif. Intell.*, 113 (1-2):41–85, 1999. doi: 10.1016/S0004-3702(99)00059-4.

R. Dechter. *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms, Second Edition*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019. doi: 10.2200/S00893ED2V01Y201901AIM041.

R. Dechter, K. Kask, E. Bin, and R. Emek. Generating random solutions for constraint satisfaction problems. In R. Dechter, M. J. Kearns, and R. S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 15–21. AAAI Press / The MIT Press, 2002. URL `http://www.aaai.org/Library/AAAI/2002/aaai02-003.php`.

B. Delaney, A. S. Fast, W. M. Campbell, C. J. Weinstein, and D. D. Jensen. The application of statistical relational learning to a database of criminal and terrorist activity. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA*, pages 409–417. SIAM, 2010. doi: 10.1137/1.9781611972801.36.

P. Dilkas and V. Belle. Generating random logic programs using constraint programming. In H. Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 828–845. Springer, 2020. doi: 10.1007/978-3-030-58475-7_48.

P. Dilkas and V. Belle. Weighted model counting without parameter variables. In C. Li and F. Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2021a. doi: 10.1007/978-3-030-80223-3_10.

P. Dilkas and V. Belle. Weighted model counting with conditional weights for Bayesian networks. In C. P. de Campos, M. H. Maathuis, and E. Quaeghebeur, editors, *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021, Virtual Event, 27-30 July 2021*, volume 161 of *Proceedings of Machine Learning Research*, pages 386–396. AUAI Press, 2021b. URL `https://proceedings.mlr.press/v161/dilkas21a.html`.

R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. ISBN 978-1-4471-5558-4. doi: 10.1007/978-1-4471-5559-1.

A. Dries, A. Kimmig, J. Davis, V. Belle, and L. De Raedt. Solving probability problems in natural language. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 3981–3987. ijcai.org, 2017. doi: 10.24963/ijcai.2017/556.

J. M. Dudek, K. S. Meel, and M. Y. Vardi. The hard problems are almost everywhere for random CNF-XOR formulas. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 600–606. ijcai.org, 2017. doi: 10.24963/ijcai.2017/84.

J. M. Dudek, L. Dueñas-Osorio, and M. Y. Vardi. Efficient contraction of large tensor networks for weighted model counting through graph decompositions. *CoRR*, abs/1908.04381, 2019.

J. M. Dudek, V. Phan, and M. Y. Vardi. ADDMC: Weighted model counting with algebraic decision diagrams. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1468–1476. AAAI Press, 2020a. URL `https://aaai.org/ojs/index.php/AAAI/article/view/5505`.

J. M. Dudek, V. H. N. Phan, and M. Y. Vardi. DPMC: weighted model counting by dynamic programming on project-join trees. In H. Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 211–230. Springer, 2020b. doi: 10.1007/978-3-030-58475-7_13.

J. M. Dudek, V. H. N. Phan, and M. Y. Vardi. ProCount: Weighted projected model counting with graded project-join trees. In C. Li and F. Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 152–170. Springer, 2021. doi: 10.1007/978-3-030-80223-3_11.

M. Earnest. An efficient way to numerically compute Stirling numbers of the second kind? Computational Science Stack Exchange, August 2018. URL `https://scicomp.stackexchange.com/q/30049`. (version: 2021-10-23).

N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi: 10.1007/978-3-540-24605-3_37.

J. Fages and X. Lorca. Revisiting the tree constraint. In J. H. Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 2011. doi: 10.1007/978-3-642-23786-7_22.

H. Fargier and P. Marquis. On the use of partially ordered decision graphs in knowledge compilation and quantified Boolean formulae. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 42–47. AAAI Press, 2006. URL http://www.aaai.org/Library/AAAI/2006/aaai06-007.php.

J. Feldstein and V. Belle. Lifted reasoning meets weighted model integration. In C. P. de Campos, M. H. Maathuis, and E. Quaeghebeur, editors, *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021, Virtual Event, 27-30 July 2021*, volume 161 of *Proceedings of Machine Learning Research*, pages 322–332. AUAI Press, 2021. URL https://proceedings.mlr.press/v161/feldstein21a.html.

J. K. Fichte, M. Hecher, and A. Pfandler. Lower bounds for qbfs of bounded treewidth. In H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 410–424. ACM, 2020. doi: 10.1145/3373718.3394756.

J. K. Fichte, M. Hecher, and F. Hamiti. The model counting competition 2020. *ACM J. Exp. Algorithmics*, 26:13:1–13:26, 2021. doi: 10.1145/3459080.

D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, and L. De Raedt. Inference in probabilistic logic programs using weighted CNF's. In F. G. Cozman and A. Pfeffer, editors, *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*, pages 211–220. AUAI Press, 2011.

D. Fierens, G. Van den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory Pract. Log. Program.*, 15(3):358–401, 2015. doi: 10.1017/S1471068414000076.

J. Franco and M. C. Paull. Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discret. Appl. Math.*, 5(1):77–87, 1983. doi: 10.1016/0166-218X(83)90017-3.

H. Gaifman. Concerning measures on Boolean algebras. *Pacific Journal of Mathematics*, 14(1):61–73, 1964.

A. Galanis, L. A. Goldberg, H. Guo, and K. Yang. Counting solutions to random CNF formulas. In A. Czumaj, A. Dawar, and E. Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11,*

2020, Saarbrücken, Germany (Virtual Conference), volume 168 of *LIPIcs*, pages 53:1–53:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ICALP.2020.53.

M. Garnelo and M. Shanahan. Reconciling deep learning with symbolic artificial intelligence: representing objects and relations. *Current Opinion in Behavioral Sciences*, 29:17–23, 2019.

W. Gatterbauer and D. Suciu. Approximate lifted inference with probabilistic databases. *Proc. VLDB Endow.*, 8(5):629–640, 2015. doi: 10.14778/2735479.2735494. URL `http://www.vldb.org/pvldb/vol8/p629-gatterbauer.pdf`.

W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 43(1):169–177, 1994.

J. Giráldez-Cru and J. Levy. Generating SAT instances with community structure. *Artif. Intell.*, 238:119–134, 2016. doi: 10.1016/j.artint.2016.06.001.

J. Giráldez-Cru and J. Levy. Locality in random SAT instances. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 638–644. ijcai.org, 2017. doi: 10.24963/ijcai.2017/89.

V. Gogate and P. M. Domingos. Formula-based probabilistic inference. In P. Grünwald and P. Spirtes, editors, *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, pages 210–219. AUAI Press, 2010.

V. Gogate and P. M. Domingos. Approximation by quantization. In F. G. Cozman and A. Pfeffer, editors, *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*, pages 247–255. AUAI Press, 2011.

V. Gogate and P. M. Domingos. Probabilistic theorem proving. *Commun. ACM*, 59(7): 107–115, 2016. doi: 10.1145/2936726.

C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-633.

N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In D. A. McAllester and P. Myllymäki, editors, *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pages 220–229. AUAI Press, 2008.

A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In J. D. Herbsleb and M. B. Dwyer, editors, *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 167–181. ACM, 2014. doi: 10.1145/2593882.2593900.

E. Gribkoff, D. Suciu, and G. Van den Broeck. Lifted probabilistic inference: A guide for the database researcher. *IEEE Data Eng. Bull.*, 37(3):6–17, 2014. URL http://sites.computer.org/debull/A14sept/p6.pdf.

J. Hoey, R. St-Aubin, A. J. Hu, and C. Boutilier. SPUDD: stochastic planning using decision diagrams. In K. B. Laskey and H. Prade, editors, *UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30 - August 1, 1999*, pages 279–288. Morgan Kaufmann, 1999.

S. Holtzen, G. Van den Broeck, and T. D. Millstein. Dice: Compiling discrete probabilistic programs for scalable inference. *CoRR*, abs/2005.09089, 2020a.

S. Holtzen, G. Van den Broeck, and T. D. Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):140:1–140:31, 2020b. doi: 10.1145/3428208.

M. M. Hossain, H. A. Abbass, C. Lokan, and S. Alam. Adversarial evolution: Phase transition in non-uniform hard satisfiability problems. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*, pages 1–8. IEEE, 2010. doi: 10.1109/CEC.2010.5586506.

M. Jaeger. Relational Bayesian networks. In D. Geiger and P. P. Shenoy, editors, *UAI '97: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, Brown University, Providence, Rhode Island, USA, August 1-3, 1997*, pages 266–273. Morgan Kaufmann, 1997.

M. Jaeger and G. Van den Broeck. Liftability of probabilistic inference: Upper and lower bounds. In *Proceedings of the 2nd international workshop on statistical relational AI*, 2012.

D. Jain, L. Mösenlechner, and M. Beetz. Equipping robot control programs with first-order probabilistic reasoning capabilities. In *2009 IEEE International Conference on Robotics and Automation, ICRA 2009, Kobe, Japan, May 12-17, 2009*, pages 3626–3631. IEEE, 2009. doi: 10.1109/ROBOT.2009.5152676.

T. Jech. *Set theory, Second Edition*. Perspectives in Mathematical Logic. Springer, 1997. ISBN 978-3-540-63048-7. URL https://doi.org/10.1145/2936726.

S. M. Kazemi and D. Poole. Knowledge compilation for lifted probabilistic inference: Compiling to a low-level language. In C. Baral, J. P. Delgrande, and F. Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, pages 561–564. AAAI Press, 2016. URL http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12861.

S. M. Kazemi, A. Kimmig, G. Van den Broeck, and D. Poole. New liftable classes for first-order probabilistic inference. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages

3117–3125, 2016. URL https://proceedings.neurips.cc/paper/2016/hash/c88d8d0a6097754525e02c2246d8d27f-Abstract.html.

K. Kersting. Lifted probabilistic inference. In L. De Raedt, C. Bessiere, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, editors, *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 33–38. IOS Press, 2012. doi: 10.3233/978-1-61499-098-7-33.

A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *TPLP*, 11(2-3): 235–262, 2011. doi: 10.1017/S1471068410000566.

A. Kimmig, L. Mihalkova, and L. Getoor. Lifted graphical models: a survey. *Mach. Learn.*, 99(1):1–45, 2015. doi: 10.1007/s10994-014-5443-2.

A. Kimmig, G. Van den Broeck, and L. De Raedt. Algebraic model counting. *J. Appl. Log.*, 22:46–62, 2017. doi: 10.1016/j.jal.2016.11.031.

H. Kleine Büning and U. Bubeck. Theory of quantified Boolean formulas. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 735–760. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-735.

S. Kolb, M. Mladenov, S. Sanner, V. Belle, and K. Kersting. Efficient symbolic integration for probabilistic inference. In J. Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 5031–5037. ijcai.org, 2018. doi: 10.24963/ijcai.2018/698.

D. Koller and N. Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009. ISBN 978-0-262-01319-2. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11886.

T. Korhonen and M. Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters (short paper). In L. D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 8:1–8:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.CP.2021.8.

A. Kuusisto and C. Lutz. Weighted model counting beyond two-variable logic. In A. Dawar and E. Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 619–628. ACM, 2018. doi: 10.1145/3209108.3209168.

O. Kuzelka. Weighted first-order model counting in the two-variable fragment with counting quantifiers. *J. Artif. Intell. Res.*, 70:1281–1307, 2021. doi: 10.1613/jair.1.12320.

J. Lagniez and P. Marquis. An improved decision-DNNF compiler. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. doi: 10.24963/ijcai.2017/93.

L. A. Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9 (3):115–116, 1973.

C. M. Li and F. Manyà. MaxSAT, hard and soft constraints. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 613–631. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-613.

H. Loeliger. An introduction to factor graphs. *IEEE Signal Process. Mag.*, 21(1):28–41, 2004. doi: 10.1109/MSP.2004.1267047.

J. Mairy, Y. Deville, and C. Lecoutre. The smart table constraint. In L. Michel, editor, *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, volume 9075 of *Lecture Notes in Computer Science*, pages 271–287. Springer, 2015. doi: 10.1007/978-3-319-18008-3_19.

S. Malhotra and L. Serafini. Weighted model counting in FO2 with cardinality constraints and counting quantifiers: A closed form formula. *CoRR*, abs/2110.05992, 2021.

T. Mantadelis and R. Rocha. Using iterative deepening for probabilistic logic inference. In Y. Lierler and W. Taha, editors, *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings*, volume 10137 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2017. doi: 10.1007/978-3-319-51676-9_14.

C. Mears, A. Schutt, P. J. Stuckey, G. Tack, K. Marriott, and M. Wallace. Modelling with option types in MiniZinc. In H. Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, volume 8451 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2014. doi: 10.1007/978-3-319-07046-9_7.

B. Milch, B. Marthi, S. J. Russell, D. A. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1352–1359. Professional Book Center, 2005. URL `http://ijcai.org/Proceedings/05/Papers/1546.pdf`.

D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distributions of SAT problems. In W. R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992*, pages 459–465. AAAI Press / The MIT Press, 1992. URL `http://www.aaai.org/Library/AAAI/1992/aaai92-071.php`.

B. Moldovan and L. De Raedt. Learning relational affordance models for two-arm robots. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 2916–2922. IEEE, 2014. doi: 10.1109/IROS.2014.6942964.

B. Moldovan, M. van Otterlo, L. De Raedt, P. Moreno, and J. Santos-Victor. Statistical relational learning of object affordances for robotic manipulation. In S. H. Muggleton and H. Watanabe, editors, *Latest Advances in Inductive Logic Programming, ILP 2011, Late Breaking Papers, Windsor Great Park, UK, July 31 - August 3, 2011*, pages 95–103. Imperial College Press / World Scientific, 2011. doi: 10.1142/9781783265091_0012.

B. Moldovan, P. Moreno, M. van Otterlo, J. Santos-Victor, and L. De Raedt. Learning relational affordance models for robots in multi-object manipulation tasks. In *IEEE International Conference on Robotics and Automation, ICRA 2012, 14-18 May, 2012, St. Paul, Minnesota, USA*, pages 4373–4378. IEEE, 2012. doi: 10.1109/ICRA.2012.6225042.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. doi: 10.1145/378239.379017.

G. Namasivayam. Study of random logic programs. In P. M. Hill and D. S. Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 555–556. Springer, 2009. doi: 10.1007/978-3-642-02846-5_61.

G. Namasivayam and M. Truszczynski. Simple random logic programs. In E. Erdem, F. Lin, and T. Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2009. doi: 10.1007/978-3-642-04238-6_20.

H. Nassif, F. Kuusisto, E. S. Burnside, D. Page, J. W. Shavlik, and V. S. Costa. Score as you lift (SAYL): A statistical relational learning approach to uplift modeling. In H. Blockeel, K. Kersting, S. Nijssen, and F. Zelezný, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III*, volume 8190 of *Lecture Notes in Computer Science*, pages 595–611. Springer, 2013. doi: 10.1007/978-3-642-40994-3_38.

U. Nilsson and J. Maluszynski. *Logic, programming and Prolog*. Wiley, 1990. ISBN 978-0-471-92625-2.

OEIS Foundation Inc. The on-line encyclopedia of integer sequences. Published electronically at `https://oeis.org`, 2022.

U. Oztok and A. Darwiche. A top-down compiler for sentential decision diagrams. In Q. Yang and M. J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3141–3148. AAAI Press, 2015. URL http://ijcai.org/Abstract/15/443.

J. Pearl. *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

A. Pfeffer. IBAL: A probabilistic rational programming language. In B. Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 733–740. Morgan Kaufmann, 2001.

K. Pipatsrisawat and A. Darwiche. New compilation languages based on structured decomposability. In D. Fox and C. P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 517–522. AAAI Press, 2008. URL http://www.aaai.org/Library/AAAI/2008/aaai08-082.php.

D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell.*, 94(1-2):7–56, 1997. doi: 10.1016/S0004-3702(97)00027-1.

D. Poole. The independent choice logic and beyond. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*, pages 222–243. Springer, 2008. doi: 10.1007/978-3-540-78652-8_8.

H. Poon and L. Vanderwende. Joint inference for knowledge extraction from biomedical literature. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 2-4, 2010, Los Angeles, California, USA*, pages 813–821. The Association for Computational Linguistics, 2010. URL https://aclanthology.org/N10-1123/.

Y. Pote, S. Joshi, and K. S. Meel. Phase transition behavior of cardinality and XOR constraints. In S. Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1162–1168. ijcai.org, 2019. doi: 10.24963/ijcai.2019/162.

F. Provost and T. Fawcett. *Data Science for Business: What you need to know about data mining and data-analytic thinking*. " O'Reilly Media, Inc.", 2013.

C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. URL http://www.choco-solver.org.

P. W. Purdom Jr. and C. A. Brown. An analysis of backtracking with search rearrangement. *SIAM J. Comput.*, 12(4):717–733, 1983. doi: 10.1137/0212049.

Y. Pushak and H. H. Hoos. Advanced statistical analysis of empirical performance scaling. In C. A. C. Coello, editor, *GECCO '20: Genetic and Evolutionary Computation Conference, Cancún Mexico, July 8-12, 2020*, pages 236–244. ACM, 2020. doi: 10.1145/3377930.3390210.

M. Richardson and P. M. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2): 107–136, 2006. doi: 10.1007/s10994-006-5833-1.

F. Riguzzi, E. Bellodi, R. Zese, G. Cota, and E. Lamma. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *Int. J. Approx. Reason.*, 80:313–333, 2017. doi: 10.1016/j.ijar.2016.10.002.

N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984. doi: 10.1016/0095-8956(84)90013-3.

N. Robertson and P. D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *J. Comb. Theory, Ser. B*, 52(2):153–190, 1991. doi: 10.1016/0095-8956(91)90061-N.

F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4. URL https://www.sciencedirect.com/science/bookseries/15746526/2.

O. Roussel and V. M. Manquinho. Pseudo-Boolean and cardinality constraints. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 695–733. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-695.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN 9780134610993. URL http://aima.cs.berkeley.edu/.

N. A. Sakhanenko and D. J. Galas. Probabilistic logic methods and some applications to biology and medicine. *J. Comput. Biol.*, 19(3):316–336, 2012. doi: 10.1089/cmb.2011.0234.

T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. URL http://www.satisfiability.org/SAT04/programme/21.pdf.

T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In M. M. Veloso and S. Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 475–482. AAAI Press / The MIT Press, 2005a. URL http://www.aaai.org/Library/AAAI/2005/aaai05-075.php.

T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2005b. doi: 10.1007/11499107_17.

S. Sanner and C. Boutilier. Practical solution techniques for first-order MDPs. *Artif. Intell.*, 173(5-6):748–788, 2009. doi: 10.1016/j.artint.2008.11.003.

S. Sanner and D. A. McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1384–1390. Professional Book Center, 2005. URL http://ijcai.org/Proceedings/05/Papers/1439.pdf.

S. Sanner, K. V. Delgado, and L. N. de Barros. Symbolic dynamic programming for discrete and continuous state MDPs. In F. G. Cozman and A. Pfeffer, editors, *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*, pages 643–652. AUAI Press, 2011.

T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 1330–1339. Morgan Kaufmann, 1997. URL http://ijcai.org/Proceedings/97-2/Papers/078.pdf.

T. Sato and Y. Kameya. New advances in logic-based probabilistic modeling by PRISM. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*, pages 118–155. Springer, 2008. doi: 10.1007/978-3-540-78652-8_5.

B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artif. Intell.*, 81(1-2):17–29, 1996. doi: 10.1016/0004-3702(95)00045-3.

F. Somenzi. CUDD: CU decision diagram package release 3.0.0. *University of Colorado at Boulder*, 2015.

R. St-Aubin, J. Hoey, and C. Boutilier. APRICODD: Approximate policy construction using decision diagrams. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*, pages 1089–1095. MIT Press, 2000. URL https://proceedings.neurips.cc/paper/2000/hash/201d7288b4c18a679e48b31c72c30ded-Abstract.html.

Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.29*, 2022. URL https://mc-stan.org.

P. J. Stuckey, K. Marriott, and G. Tack. *MiniZinc Handbook: Release 2.6.2*, March 2022.

E. Tsamoura, V. Gutiérrez-Basulto, and A. Kimmig. Beyond the grounding bottleneck: Datalog techniques for inference in probabilistic logic programs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 10284–10291. AAAI Press, 2020. URL https://aaai.org/ojs/index.php/AAAI/article/view/6591.

E. Tsamoura, T. M. Hospedales, and L. Michael. Neural-symbolic integration: A compositional perspective. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 5051–5060. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16639.

L. G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8: 189–201, 1979. doi: 10.1016/0304-3975(79)90044-6.

P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 85–134. Elsevier, 2006. doi: 10.1016/S1574-6526(06)80008-8.

T. van Bremen and O. Kuzelka. Approximate weighted first-order model counting: Exploiting fast approximate model counters and symmetry. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4252–4258. ijcai.org, 2020. doi: 10.24963/ijcai.2020/587.

T. van Bremen and O. Kuzelka. Lifted inference with tree axioms. In M. Bienvenu, G. Lakemeyer, and E. Erdem, editors, *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, pages 599–608, 2021a. doi: 10.24963/kr.2021/57.

T. van Bremen and O. Kuzelka. Faster lifting for two-variable logic using cell graphs. In C. P. de Campos, M. H. Maathuis, and E. Quaeghebeur, editors, *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021, Virtual Event, 27-30 July 2021*, volume 161 of *Proceedings of Machine Learning Research*, pages 1393–1402. AUAI Press, 2021b. URL https://proceedings.mlr.press/v161/bremen21a.html.

G. Van den Broeck. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, pages 1386–1394, 2011. URL https://proceedings.neurips.cc/paper/2011/hash/846c260d715e5b854ffad5f70a516c88-Abstract.html.

G. Van den Broeck, N. Taghipour, W. Meert, J. Davis, and L. De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In T. Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 2178–2185. IJCAI/AAAI, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-363.

G. Van den Broeck, A. Choi, and A. Darwiche. Lifted relax, compensate and then recover: From approximate to exact lifted probabilistic inference. In N. de Freitas and K. P. Murphy, editors, *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, August 14-18, 2012*, pages 131–141. AUAI Press, 2012.

G. Van den Broeck, A. Lykov, M. Schleich, and D. Suciu. On the tractability of SHAP explanations. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6505–6513. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16806.

J. Vennekens, M. Denecker, and M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory Pract. Log. Program.*, 9(3):245–308, 2009. doi: 10.1017/S1471068409003767.

M. Verbeke, V. Van Asch, R. Morante, P. Frasconi, W. Daelemans, and L. De Raedt. A statistical relational learning approach to identifying evidence based medicine categories. In J. Tsujii, J. Henderson, and M. Pasca, editors, *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 579–589. ACL, 2012. URL https://aclanthology.org/D12-1053/.

J. Vlasselaer and W. Meert. Statistical relational learning for prognostics. In *Proceedings of the 21st Belgian-Dutch Conference on Machine Learning*, pages 45–50, 2012.

J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt. Anytime inference in probabilistic logic programs with Tp-compilation. In Q. Yang and M. J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1852–1858. AAAI Press, 2015. URL http://ijcai.org/Abstract/15/263.

J. Vlasselaer, A. Kimmig, A. Dries, W. Meert, and L. De Raedt. Knowledge compilation and weighted model counting for inference in probabilistic logic programs. In A. Darwiche, editor, *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016*, volume WS-16-05 of *AAAI Technical Report*. AAAI Press, 2016. URL http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12631.

J. Vomlel and P. Tichavský. Probabilistic inference in BN2T models by weighted model counting. In M. Jaeger, T. D. Nielsen, and P. Viappiani, editors, *Twelfth Scandinavian Conference on Artificial Intelligence, SCAI 2013, Aalborg, Denmark, November 20-22, 2013*, volume 257 of *Frontiers in Artificial Intelligence and Applications*, pages 275–284. IOS Press, 2013. doi: 10.3233/978-1-61499-330-8-275.

T. Walsh. General symmetry breaking constraints. In F. Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 650–664. Springer, 2006. doi: 10.1007/11889205_46.

K. Wang, L. Wen, and K. Mu. Random logic programs: Linear model. *TPLP*, 15(6): 818–853, 2015. doi: 10.1017/S1471068414000611.

L. Wen, K. Wang, Y. Shen, and F. Lin. A model for phase transition of random answer-set programs. *ACM Trans. Comput. Log.*, 17(3):22:1–22:34, 2016. doi: 10.1145/2926791.

L. A. Wolsey. *Integer programming*. John Wiley & Sons, 2020.

J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. Van den Broeck. A semantic loss function for deep learning with symbolic knowledge. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5498–5507. PMLR, 2018. URL http://proceedings.mlr.press/v80/xu18h.html.

L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008. doi: 10.1613/jair.2490.

H. Zhao, M. Melibari, and P. Poupart. On the relationship between sum-product networks and Bayesian networks. In F. R. Bach and D. M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 116–124. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/zhaoc15.html.

Y. Zhao and F. Lin. Answer set programming phase transition: A study on randomly generated programs. In C. Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2003. doi: 10.1007/978-3-540-24599-5_17.