Probabilistic Inference via Weighted Model Counting: Algorithms, Encodings, and Random Instances

Paulius Dilkas

Doctor of Philosophy
Artificial Intelligence Applications Institute
School of Informatics
University of Edinburgh
2022

Abstract

Given a formula ϕ in propositional or first-order logic and some weights in $\mathbb{R}_{\geq 0}$ (usually defined over propositional variables or predicates), weighted model counting (WMC) asks to compute the sum of the weights of the models of ϕ. One strand of my work shows how WMC is the process of computing the measure of an element of a Boolean algebra. This observation allows us to generalise WMC, significantly reducing the complexity of WMC instances that encode probabilistic inference in Bayesian networks. Another strand of my work is concerned with synthetic data generation. In particular, we show how random instances of probabilistic logic programs (that typically use variations of WMC algorithms for inference) can be generated using constraint programming. We also present a random model for WMC instances and show how the algorithms differ with respect to key properties of the instances, e.g., a version of treewidth. Finally, in the first-order setting, we expand the range of tractable problem instances by developing an algorithm that can define quantities and sub-quantities of interest by constructing recursive functions.

Acknowledgements

The first author was supported by the EPSRC Centre for Doctoral Training in Robotics and Autonomous Systems, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016834/1). This work has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF) (http://www.ecdf.ed.ac.uk/).

We thank the anonymous reviewers for their helpful comments.

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

- What is probabilistic inference?

1.1 Thesis Structure

1. 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. 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.

2. Weighted model counting (WMC) has emerged as the unifying inference mechanism across many (probabilistic) domains. Encoding an 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. 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 state-of-the-art WMC algorithm ADDMC to the new format. Our experiments show that the new encod-

ing significantly improves the performance of the algorithm on most benchmark instances.

3. Weighted model counting (WMC) is a powerful computational technique for a variety of problems, especially commonly used for probabilistic inference. However, the standard definition of WMC that puts weights on literals often necessitates WMC encodings to include additional variables and clauses just so each weight can be attached to a literal. This paper complements previous work by considering WMC instances in their full generality and using recent state-of-the-art WMC techniques based on pseudo-Boolean function manipulation, competitive with the more traditional WMC algorithms based on knowledge compilation and backtracking search. We present an algorithm that transforms WMC instances into a format based on pseudo-Boolean functions 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, outperforming the current state of the art.

4. TODO

5. TODO

Chapter 2

Background

TODO: outside of specific sections, have a one-paragraph introduction before and a one-paragraph summary at the end.

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 variable) is a variable with two possible (truth) values: true and false. Unless specified otherwise, we will 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 Sect. 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 $\{\text{true}, \text{false}\}$. A *model* is an interpretation under which ϕ evaluates to true. A formula is *satisfiable* if it has at least one model.

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 we assume no duplicates—whether or not that is the case is clarified on a case-by-case basis.

Example 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 $\{a \mapsto$

[1] In the context of logic programs, the word *clause* is used differently (see Sect. 2.2.1 and Chapter 3).

true, $b \mapsto$ true $\}$ is a model, so ϕ is satisfiable. An equivalent set-theoretic representation of ϕ is $\{\{\neg a, b\}, \{a\}\}$.

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 minise 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 7.

Example 2. The model count of the formula in Example 1 is equal to one. With a weight function $w \coloneqq \{a \mapsto 0.7, \neg a \mapsto 0.2, b \mapsto 0.8, \neg b \mapsto 0.7\}$, the WMC of the same formula is $0.7 \times 0.8 = 0.56$.

Example 3. With the same weight function w as in Example 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.

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

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

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 Sect. 2.2.2.

2.2 Declarative 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 3. 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. 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).

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.

[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.

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.

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

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

Constraint 2. `alldifferent`$(\{q_i + i \mid i = 1,\dots,n\})$

Constraint 3. `alldifferent`$(\{q_i - i \mid i = 1,\dots,n\})$

Here, `alldifferent` is a constraint on a set of variables (or 'derivatives' of variables) that constrains them to be all different. Constraint 1 requires all queens to occupy different rows, and Constraints 2 and 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 $b_{ij} \in \{\text{true}, \text{false}\}$ indicates whether the (i,j)-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 4 (Channelling)**.** *For all $i, j = 1,\dots,n$, we have that $b_{ij} \Leftrightarrow (q_i = j)$.*

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

Constraint 5 (Symmetry breaking)**.** $\mathbf{B}$ *is lexicographically smaller than or equal to* $\mathbf{B}^\top$ *(i.e., the transpose of* $\mathbf{B}$*).*

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 Eq. (2.1)) leads to other variable-value combinations becoming evidently impossible. For example, after placing a queen on a1 (i.e., setting $q_1 := 1$), Constraint 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.

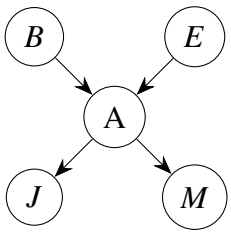
Backtracking search has important choices to make: which variable should be given a value first? Which value from a domain is most likely to lead to a solution? These questions are answered by *variable* and *value ordering heuristics*, respectively. For example, we can choose a variable with the smallest 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.

2.3 Representations of Probability Distributions

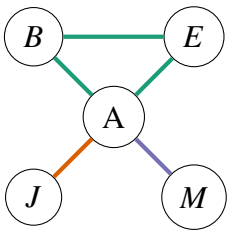
Unless specified otherwise, by *probability distribution* we mean a *discrete* probability distribution. Moreover, we are typically only interested in probability distributions with *finite support*.

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. Sect. 2.3.1 is about representations based on graphs whereas Sect. 2.3.2 covers probabilistic programming languages.



(a) a Bayesian network



(b) a Markov network

Figure 2.1: Two PGMs that describe the independence structure of Example 6

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

Example 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 Fig. 2.1a. This (directed acyclic) graph tells us that the joint probability distribution can be factored

Table 2.1: An example CPT for $\Pr(A \mid B,E)$ from Example 6

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

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 Eq. (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.

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 6 is in Fig. 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 Fig. 2.1b are highlighted in different colours. Thus, the full probability distribution can be factorised 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 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

Listing 2.1: An example BLOG program

```
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;
```

Listing 2.2: An example ProbLog program

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

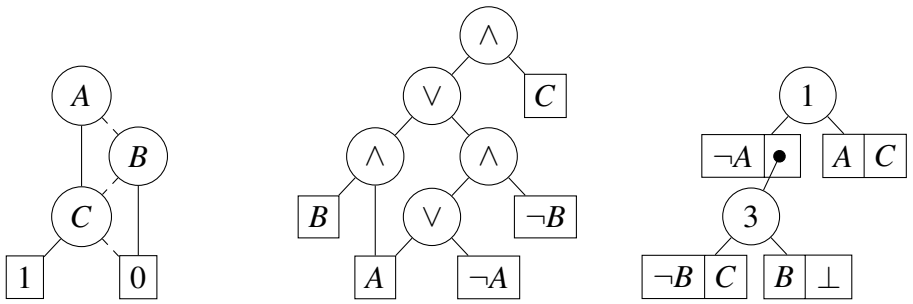
0.001 :: burglary.
0.002 :: earthquake.

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).
```

Augmenting a programming language with probabilities is another common approach to conveniently and compactly represent a probability distribution. Logic programming languages, in particular, have been frequently used for this purpose, examples of which include earlier languages such as independent choice logic [Poole, 1997] and PRISM [Sato and Kameya, 1997] as well as more recent ones such as ProbLog [De Raedt et al., 2007] (see **??**) and CP-logic [Vennekens et al., 2009]. Functional



(a) (RO)BDD (b) d-DNNF (c) SDD

Figure 2.2: Example diagrams for $C \wedge (A \vee \neg B)$

and imperative programming languages have also seen some use. Probabilistic semantics in these languages typically rely on two constructs: the ability to draw random values from probability distributions and the ability to condition a program on observations [Gordon et al., 2014]. Examples of such languages include functional languages such as Church [Goodman et al., 2008] and imperative languages like BLOG [Milch et al., 2005] (see **??**).

(and how probabilities are computed, including some details specifically about ProbLog)

2.4 Knowledge Compilation

(including lots of detail about all the data structures)

Many WMC algorithms rely on knowledge compilation, i.e., compilation of the structure of the initial representation into a form that allows one to perform various operations and answer queries of interest in time polynomial in the size of the compiled representation. Traditionally, the initial representation is a propositional formula. Many such representations have been proposed [Darwiche and Marquis, 2002]. Amongst them, the ones that are used in probabilistic inference include (reduced ordered) binary decision diagrams ((RO)BDDs) [Bryant, 1986], deterministic decomposable negation normal form (d-DNNF) [Darwiche, 2001], sentential decision diagrams (SDDs) [Darwiche, 2011], and probabilistic SDDs [Kisa et al., 2014]—some of them are pictured in Fig. 2.2. A BDD is similar to a decision tree that ends with either one or zero but generalised to a directed acyclic graph. Both d-DNNF and SDD are normal forms for propositional formulae that satisfy certain properties. Probabilistic SDDs extend SDDs with probability labels on edges. BDDs are a strict subset of SDDs that are a strict subset of d-DNNF [Darwiche, 2011].

Similarly to how BDDs represent Boolean functions, ADDs represent pseudo-Boolean functions, i.e., while (non-trivial) BDDs always have two sinks marked with one and zero, ADDs can have any number of sinks that contain, typically, real numbers [Bahar et al., 1997]. ADDs have been extended to represent the additive and multiplicative structure in sink values more compactly [Sanner and McAllester, 2005] and to support first-order logic [Sanner and Boutilier, 2009] and continuous variables [Sanner et al., 2011]. ADDs have been used to represent MDP value functions [Hoey et al., 1999] and probabilities in PGMs [Chavira and Darwiche, 2007, Gogate and Domingos,

2011].

- Boolean and Pseudo-Boolean Functions
- NNF
- d-DNNF
- SDDs
- BDDs
- ADDs (with a brief mention of AADDs, XDDs, etc.)

2.5 Applications

of WMC?

copy info from my year 2 report

- Statistical Relational Learning
- Neuro-Symbolic Artificial Intelligence
- Natural Language Processing
- Robotics

Chapter 3

Generating Random Logic Programs Using Constraint Programming

3.1 Introduction

Unifying logic and probability is a long-standing challenge in artificial intelligence [Russell, 2015], and, in that regard, statistical relational learning (SRL) has developed into an exciting area that mixes machine learning and symbolic (logical and relational) structures. In particular, probabilistic logic programs—including languages such as PRISM [Sato and Kameya, 1997], ICL [Poole, 1997], and PROBLOG [De Raedt et al., 2007]—are promising frameworks for codifying complex SRL models. With the enhanced structure, however, inference becomes more challenging. At the moment, we have no precise way of evaluating and comparing inference algorithms. Incidentally, 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 (Sects. 3.3 to 3.5), 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 Sect. 3.7 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 Sect. 3.6 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 paper with two experimental results in Sect. 3.8: 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.

Overall, our main contributions are concerned with logic programming-based languages and frameworks, which capture a major fragment of SRL [De Raedt et al., 2016]. However, since probabilistic logic programming languages are closely related to other areas of machine learning, including (imperative) probabilistic programming [De Raedt and Kimmig, 2015], our results can lay the foundations for exploring broader questions on generating models and testing algorithms in machine learning.

3.2 Preliminaries

The basic primitives of logic programs are *constants*, *(logic) variables*, and *predicates* with their *arities*. A *term* is either a variable or a constant, and an *atom* is a predicate of arity n applied to n terms. A *formula* is any well-formed expression that connects atoms using conjunction $\wedge$, disjunction $\vee$, and negation $\neg$. A *clause* is a pair of a *head* (which is an atom) and a *body* (which is a formula[1]). A *(logic) program* is a set of clauses, and a PROBLOG *program* is a set of clause-probability pairs [Fierens et al., 2015].

In the world of CSPs, we also have *(constraint) variables*, each with a *domain*, whose values are restricted using *constraints*. All constraint variables in the model 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

[1]Our model supports arbitrarily complex bodies of clauses (e.g., $\neg \mathsf{P}(X) \vee (\mathsf{Q}(X) \wedge \mathsf{P}(X))$) because PROBLOG does too. However, one can easily restrict our representation of a body to a single conjunction of literals (e.g., $\mathsf{Q}(X) \wedge \neg \mathsf{P}(X)$)) by adding a couple of additional constraints.

one value. We let $\square$ denote the absent/disabled value of an optional variable [Mears et al., 2014]. We write $\texttt{a}[b] \in c$ to mean that a is an array of variables of length b such that each element of a has domain c. Similarly, we write $\texttt{c} : \texttt{a}[b]$ 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.

3.2.0.0.1 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 Sect. 3.4), 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 Sect. 3.7), 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 $[1]$. 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: $\texttt{Body} : \texttt{bodies}[\mathcal{M}_\mathcal{C}]$ and $\texttt{Head} : \texttt{heads}[\mathcal{M}_\mathcal{C}]$.

3.3 Heads of Clauses

We define the *head* of a clause as a $\texttt{predicate} \in \mathcal{P} \cup \{\square\}$ and $\texttt{arguments}[\mathcal{M}_\mathcal{A}] \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$. 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 Sect. 3.5. This predicate variable has a corresponding arity that depends on the predicate. We can define $\texttt{arity} \in [0, \mathcal{M}_\mathcal{A}]$ as the arity of the predicate if $\texttt{predicate} \in \mathcal{P}$ 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. *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 7. *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 $\square > \mathsf{P}$ for all $\mathsf{P} \in \mathcal{P}$ and the head predicate is the 'first digit' of this representation, the following constraint disables duplicates as well as orders the clauses.

Constraint 8. *For* $i = 1, \ldots, \mathcal{M}_\mathcal{C} - 1$*, if* $\texttt{heads}[i].\texttt{predicate} \neq \square$*, then*

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

3.4 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 ith node is a root, and $\texttt{structure}[i] = j$ (for $j \neq i$) means that the ith 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 ith 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 $\texttt{numNodes} \in \{1, \ldots, \mathcal{M}_\mathcal{N}\}$ to count the number of nodes in the main tree. We define it as $\texttt{numNodes} = \mathcal{M}_\mathcal{N} - \texttt{numTrees} + 1$.

Example 7. Let $\mathcal{M}_\mathcal{N} = 8$. Then $\neg\mathsf{P}(X) \vee (\mathsf{Q}(X) \wedge \mathsf{P}(X))$ can be encoded as:

$$\begin{aligned} \texttt{structure} &= [0, 0, 0, \quad 1, \quad 2, \quad 2, \ 6, 7], \quad \texttt{numNodes} = 6, \\ \texttt{values} &= [\vee, \neg, \wedge, \mathsf{P}(X), \mathsf{Q}(X), \mathsf{P}(X), \top, \top], \quad \texttt{numTrees} = 3. \end{aligned}$$

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 Sect. 3.3. Furthermore, we can use Constraint 6 to again disable the extra arguments.

Example 8. Let $\mathcal{M}_\mathcal{A} = 2$, $X \in \mathcal{V}$, and let P be a predicate with arity 1. Then the node representing atom $\mathsf{P}(X)$ has: $\texttt{name} = \mathsf{P}$, $\texttt{arguments} = [X, \square]$, $\texttt{arity} = 1$.

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 recognise 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 9. *For* $i = 1, \ldots, \mathcal{M}_{\mathcal{N}} - 1$*, if* $i <$ `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 10. *For* $i = 0, \ldots, \mathcal{M}_{\mathcal{N}} - 1$*, let* C_i *be the number of times* i *appears in the* `structure` *array with index greater than* i*. Then*

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

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

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

3.5 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 2. 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 $t \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$,

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

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

Definition 3. 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 9. 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 13. *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 14. *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 15. *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* $\texttt{potentials} \cap S = \emptyset$.

3.6 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 ith predicate. The number of programs is then:

$$\sum_{\substack{\sum_{i=1}^{|\mathcal{P}|} h_i = n,\\ |\mathcal{P}|\le n\le \mathcal{M}_\mathcal{C},\\ h_i\ge 1 \text{ for all } i}} \; \prod_{i=1}^{|\mathcal{P}|} \binom{\sum_{a=0}^{\mathcal{M}_\mathcal{A}\times\mathcal{M}_\mathcal{N}} C(a)P(a+a_i)}{h_i}, \tag{3.1}$$

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 (3.1) 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.

3.7 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, $\mathsf{P}, \mathsf{Q} \in \mathcal{P}$) 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 $\mathsf{P}, \mathsf{Q} \in \mathcal{P}$ if there is a clause in $\mathscr{P}$ with Q 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.[2] 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 $\mathsf{P} \in \mathcal{P}$, the set of *dependencies* of P is the smallest set D_{P} such that $\mathsf{P} \in D_{\mathsf{P}}$, and, for every $\mathsf{Q} \in D_{\mathsf{P}}$, all direct predecessors of Q in $G_{\mathscr{P}}$ are in D_{P}. Two predicates P and Q are *independent* if $D_{\mathsf{P}} \cap D_{\mathsf{Q}} = \emptyset$.

Example 10. Consider the following (fragment of a) program:

$$\begin{aligned} \mathsf{sibling}(X,Y) &\leftarrow \mathsf{parent}(X,Z) \wedge \mathsf{parent}(Y,Z), \\ \mathsf{father}(X,Y) &\leftarrow \mathsf{parent}(X,Y) \wedge \neg\mathsf{mother}(X,Y). \end{aligned} \tag{3.2}$$

[2] 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.

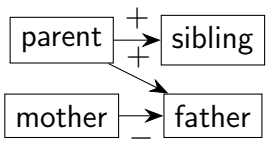


Figure 3.1: The predicate dependency graph of the program from Example 10. Positive edges are labelled with '+', and negative edges with '−'.

Table 3.1: Types of (potential) dependencies of a predicate P based on the number of undetermined edges on the path from the dependency to P

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

Its predicate dependency graph is in Fig. 3.1. Because of the negation in (3.2), the edge from mother to father is negative, while the other two edges are positive. The dependencies of each predicate are:

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

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 $\mathbf{A}$ 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 $k = 0, \ldots, \mathcal{M}_{\mathcal{C}} - 1$, 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 $\mathsf{P}, \mathsf{Q} \in \mathcal{P}$) 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 3.1, where p 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 p as d.predicate. In describing the algorithms, we will use '_' to replace any of p, 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 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

Algorithm 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;

Algorithm 2: Propagation for independence

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

1 **for** $(d_1,d_2) \in \texttt{deps}(p_1,\ 0) \times \texttt{deps}(p_2,\ 0)$ *s.t.* $d_1.\text{predicate} = d_2.\text{predicate}$ **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)`;

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. 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 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 3 calculates D_p for any predicate p. It has two versions: $\texttt{deps}(p,1)$ returns all dependencies, while $\texttt{deps}(p,0)$ 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 3 could include q once for each possible type (Δ, Υ, and Γ), but Algorithms 1 and 2 would still work as intended.

Example 11. 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

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

Algorithm 3: Dependencies of a predicate

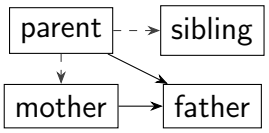
Data: adjacency matrix $\mathbf{A}$

Function `deps(`p, allDeps`)`**:**

- $D \leftarrow \{\Delta(p)\}$;
- **while** `true` **do**
  - $D' \leftarrow \emptyset$;
  - **for** $d \in D$ **and** $q \in \mathcal{P}$ **do**
    - edge $\leftarrow \mathbf{A}[q][d.\text{predicate}] = \{1\}$;
    - **if** edge **and** d **is** $\Delta(\_)$ **then** $D' \leftarrow D' \cup \{\Delta(q)\}$;
    - **else if** edge **and** d **is** $\Gamma(\_, s, t)$ **then** $D' \leftarrow D' \cup \{\Gamma(q, s, t)\}$;
    - **else if** $|\mathbf{A}[q][d.\text{predicate}]| > 1$ **and** d **is** $\Delta(r)$ **then**
      - $D' \leftarrow D' \cup \{\Gamma(q, q, r)\}$;
    - **else if** $|\mathbf{A}[q][d.\text{predicate}]| > 1$ **and** allDeps **then** $D' \leftarrow D' \cup \{\Upsilon(q)\}$;
  - **if** $D' = D$ **then return** D **else** $D \leftarrow D'$;

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

(a) The adjacency matrix of the graph. The boxed value is the decision variable that will be propagated by Algorithm 2.



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

Figure 3.2: The predicate dependency graph of Example 11

The predicate dependency graph is pictured in Fig. 3.2. Suppose we have a constraint that mother and parent must be independent. The lists of potential dependencies for both predicates are:

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

An entailment check at this stage would produce UNDEFINED, but propagation replaces the boxed value in Fig. 3.2a with zero, eliminating the potential edge from parent to mother. This also eliminates mother from $D_{\square}$, and this is enough to make Algorithm 1 return TRUE.

3.8 Experimental Results

We now present the results of two experiments: in Sect. 3.8.1 we examine the scalability of our constraint model with respect to its parameters and in Sect. 3.8.2 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.

3.8.1 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 'fail first' 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 9, 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.[3] 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.

[3] 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.

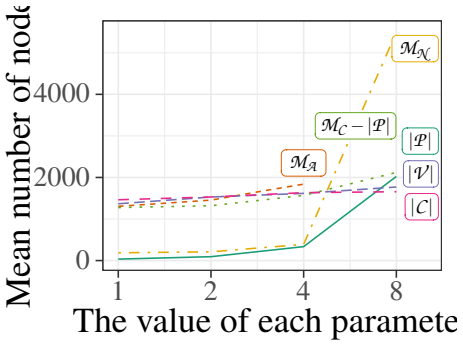


Figure 3.3: The mean number of nodes in the binary search tree for each value of each experimental parameter. Note that the horizontal axis is on a $\log_2$ scale.

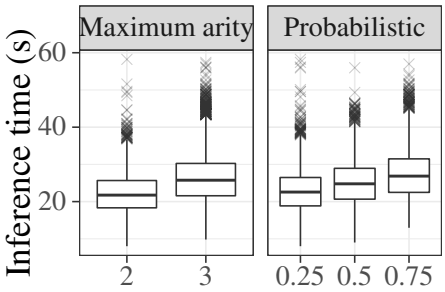


Figure 3.4: Inference time for different values of $\mathcal{M}_\mathcal{A}$ and proportions of probabilistic facts that are probabilistic. The total number of facts is fixed at 10^5.

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 Fig. 3.3, 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 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.

3.8.2 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. 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 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 Fig. 3.5) is that the algorithms are remarkably similar, i.e., the differences in performance are small and consistent across all parameter values (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 Fig. 3.4). Finally, according to Fig. 3.5, 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.

3.9 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. While the results did not reveal any significant differences among the algorithms, they did reveal a shared weakness, i.e., the inability to ignore the part of a program that is easily seen to be irrelevant to the given query.

Nonetheless, we would like to outline two directions for future work. First, the

[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 3.5: 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.

experimental evaluation in Sect. 3.8.1 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 4

Weighted Model Counting with Conditional Weights for Bayesian Networks

4.1 Introduction

Weighted model counting (WMC), i.e., an extension of model counting (#SAT) that assigns a weight to every model [Sang et al., 2005], has emerged as one of the most dominant and competitive approaches for handling inference tasks in a wide range of formalisms including Bayesian networks [Sang et al., 2005, Darwiche, 2009], probabilistic graphical models more generally [Choi et al., 2013], and probabilistic programs [Fierens et al., 2015, Holtzen et al., 2020a]. Over the last fifteen years, WMC has been extended and generalised in many ways, e.g., to handle continuous probability distributions [Belle et al., 2015], first-order probabilistic theories [Van den Broeck et al., 2011, Gogate and Domingos, 2016], and infinite domains [Belle, 2017]. Furthermore, by generalising the notion of weights to an arbitrary semiring, a range of other problems are also captured [Kimmig et al., 2017]. Exact WMC solvers typically rely on either knowledge compilation [Oztok and Darwiche, 2015, Lagniez and Marquis, 2017] or exhaustive DPLL search [Sang et al., 2005], whereas approximate solvers work by sampling [Chakraborty et al., 2014] and performing local search [Wei and Selman, 2005].

The most well-known version of WMC 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 Sect. 4.2, in Sect. 4.3 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. Sect. 4.4 introduces WMC as the problem of comput-

ing 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 Sect. 4.5 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 state-of-the-art 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]. ADDMC was already shown to be capable of solving instances other solvers fail at and being the fastest solver on the largest number of instances [Dudek et al., 2020a]. Our experiments in Sect. 4.6 focus on further 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.

4.2 Related Work

Performing inference on Bayesian networks by encoding them into instances of WMC is a well-established idea with a history of almost 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., 2005], `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. [2005] 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, 2001]), 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 Sect. 4.6 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.

While in this paper we focus on measure-theoretic foundations and propose a new encoding for Bayesian networks, a related line of work [Dilkas and Belle, 2021]—motivated by the same issue of WMC encodings having both more variables and more clauses as a consequence of having to conform to an unnecessarily restrictive format—shows how these variables and clauses can be removed from (most) already-existing Bayesian network encodings.

4.3 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 4.1 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 $x \in 2^U$.

4.3.1 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 Sect. 4.4 and can be efficiently represented using ADDs. Furthermore, 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.

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

Table 4.1: Notation for a logic with two atoms. The elements in both columns are listed in the same order.

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})	$\top$	$\{\emptyset, \{a\}, \{b\}, \{a,b\}\}$
	$\neg a \vee \neg b, a \rightarrow b$	$\{\emptyset, \{a\}, \{b\}\}, \{\emptyset, \{b\}, \{a,b\}\}$
	$b \rightarrow a, a \vee b$	$\{\emptyset, \{a\}, \{a,b\}\}, \{\{a\}, \{b\}, \{a,b\}\}$
	$\neg b, \neg a, a \leftrightarrow b$	$\{\emptyset, \{a\}\}, \{\emptyset, \{b\}\}, \{\emptyset, \{a,b\}\}$
	$(a \wedge \neg b) \vee (b \wedge \neg a), a, b$	$\{\{a\}, \{b\}\}, \{\{a\}, \{a,b\}\}, \{\{b\}, \{a,b\}\}$
	$\neg a \wedge \neg b, a \wedge \neg b, \neg a \wedge b, a \wedge b$	$\{\emptyset\}, \{\{a\}\}, \{\{b\}\}, \{\{a,b\}\}$
	$\bot$	$\emptyset$

Definition 4 (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 $\alpha(T\cap X)$.

4.4 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 1. *The function μ_ν is a measure.*

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.

4.4.0.0.1 Relation to the classical (logic-based) view of WMC. Let $\mathcal{L}$ be a propositional logic with two atoms a and b as in Sect. 4.3 and $w\colon \{a,b,\neg a,\neg b\} \to \mathbb{R}_{\geq 0}$ a weight function defined as $w(a) = 0.3$, $w(\neg a) = 0.7$, $w(b) = 0.2$, $w(\neg b) = 0.8$. Furthermore, let Δ be a theory in $\mathcal{L}$ with a sole axiom a. Then Δ has two models: $\{a,b\}$ and $\{a,\neg b\}$ and its WMC [Chavira and Darwiche, 2008] is

$$\begin{aligned}\mathrm{WMC}(\Delta) &= \sum_{\omega\models\Delta}\prod_{\omega\models l} w(l)\\ &= w(a)w(b)+w(a)w(\neg b) = 0.3.\end{aligned} \tag{4.1}$$

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 Eq. (4.1), 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.

4.4.1 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 12. 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 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.

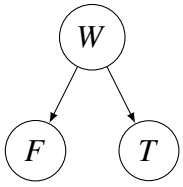
Alternatively, we can let b depend on a and consider weight functions $\nu_a \colon 2^{\{a\}} \to \mathbb{R}_{\geq 0}$ and $\nu_b \colon 2^{\{a,b\}} \to \mathbb{R}_{\geq 0}$ defined as $\nu_a(\{a\}) = 0.9, \nu_a(\emptyset) = 0.1$, and $\nu_b(\{a,b\}) = 0.8$, $\nu_b(\{a\}) = 0.2$, $\nu_b(\{b\}) = 0.7$, $\nu_b(\emptyset) = 0.3$. 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, factorising 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 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

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



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 4.1: An example Bayesian network with its CPTs.

always possible, however, as ensuing sections will demonstrate, it can make the inference task much harder in practice.[5]

Theorem 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 $x \in 2^{2^U}$.*

4.5 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 vertices 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 in this paper), each factor of this product can be represented by a CPT. See Fig. 4.1 for an example Bayesian network that we will refer to throughout this section. For this network, $\mathcal{V} = \{W, F, T\}$, $\mathrm{pa}(W) = \emptyset$, $\mathrm{pa}(F) = \mathrm{pa}(T) = \{W\}$, $\mathrm{im}W = \mathrm{im}F = \{0,1\}$, and $\mathrm{im}T = \{l, m, h\}$.

Definition 5 (Indicator variables)**.** Let $X \in \mathcal{V}$ be a random variable. If X is binary (i.e., $|\mathrm{im}X| = 2$), we can arbitrary identify one of the values as 1 and the other one as 0 (i.e, $\mathrm{im}X \cong \{0,1\}$). 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 $\lambda_{X=1} \notin S$.

[5] The proofs of this and other theoretical results can be found in the supplementary material.

Algorithm 4: Encoding a Bayesian network.

Data: vertices $\mathcal{V}$, probability distribution Pr
Result: $\phi\colon 2^U \to \mathbb{R}_{\geq 0}$

$\phi \leftarrow 1$;
for $X \in \mathcal{V}$ **do**
  let $\mathrm{pa}(X) = \{Y_1, \ldots, Y_n\}$;
  $\mathrm{CPT}_X \leftarrow 0$;
  if $|\operatorname{im} X| = 2$ **then**
    for $(y_i)_{i=1}^n \in \prod_{i=1}^n \operatorname{im} Y_i$ **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 $\operatorname{im} X = \{x_1, \ldots, x_m\}$;
    for $x \in \operatorname{im} X$ **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;

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

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

denote the set of indicator variables for X and $\mathcal{E}^*(X) = \mathcal{E}(X) \cup \bigcup_{Y \in \mathrm{pa}(X)} \mathcal{E}(Y)$ 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 Fig. 4.1, $\mathcal{E}^*(T) = \{\lambda_{T=l}, \lambda_{T=m}, \lambda_{T=h}, \lambda_{W=1}\}$.

Algorithm 4 shows how a Bayesian network with vertices $\mathcal{V}$ 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 $\mathrm{CPT}_X\colon 2^{\mathcal{E}^*(X)} \to \mathbb{R}_{\geq 0}$ 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. For the example Bayesian network in Fig. 4.1, we get:

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

4.5.1 Correctness

Algorithm 4 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 4 can be used by ADDMC to correctly compute any marginal probability of the Bayesian network that was encoded as ϕ.[6] We begin with Lemma 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 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 $T = \{\lambda_{X=x}\} \cup \{\lambda_{Y_i=y_i} \mid i = 1, \ldots, n\}$.

Now, Lemma 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 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 $T \in 2^U$.

We end with Theorem 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 4. *For any $X \in \mathcal{V}$ and $x \in \mathrm{im}X$,*

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

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

4.5.2 Textual Representation

Algorithm 4 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 Fig. 4.1:

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

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.

4.5.3 Changes to ADDMC

Here we describe two changes to ADDMC[7] [Dudek et al., 2020a] needed to adapt it to the new format.

[7] https://github.com/vardigroup/ADDMC

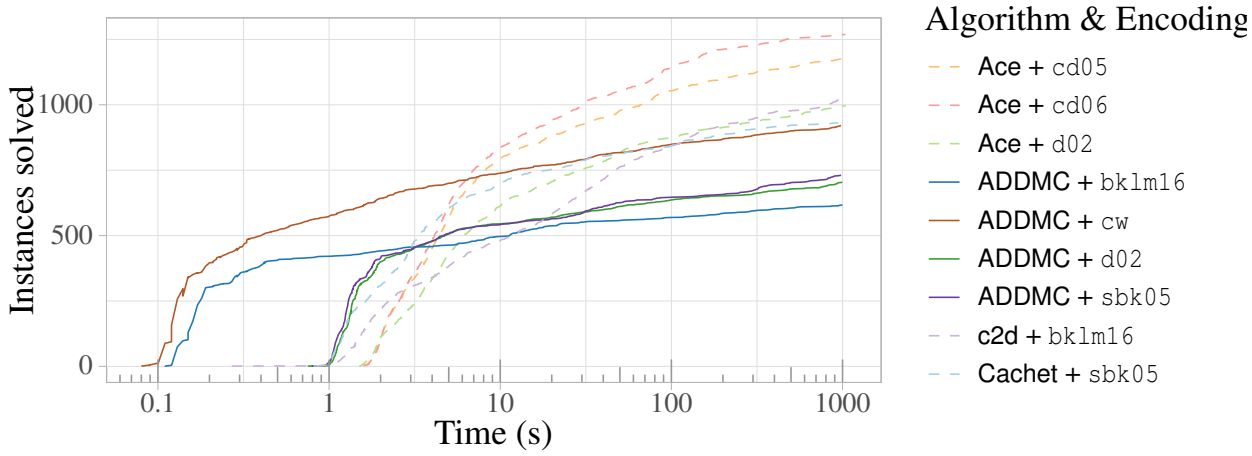


Figure 4.2: Cumulative numbers of instances solved by combinations of algorithms and encodings over time.

First, ADDMC constructs the *primal* (a.k.a. Gaifman) *graph* of the input CNF formula as an aid for the algorithm's heuristics. This graph has as vertices the variables of the formula, and there is an edge between two variables u and v if there is a clause in the formula that contains both u and v. We extend this definition to functions on Boolean algebras, i.e., the factors of ϕ. 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 4 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 $u \in U \setminus V$.

Table 4.2: The numbers of instances (out of 1466) solved by each combination of algorithm and encoding (uniquely, faster than others, and in total).

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

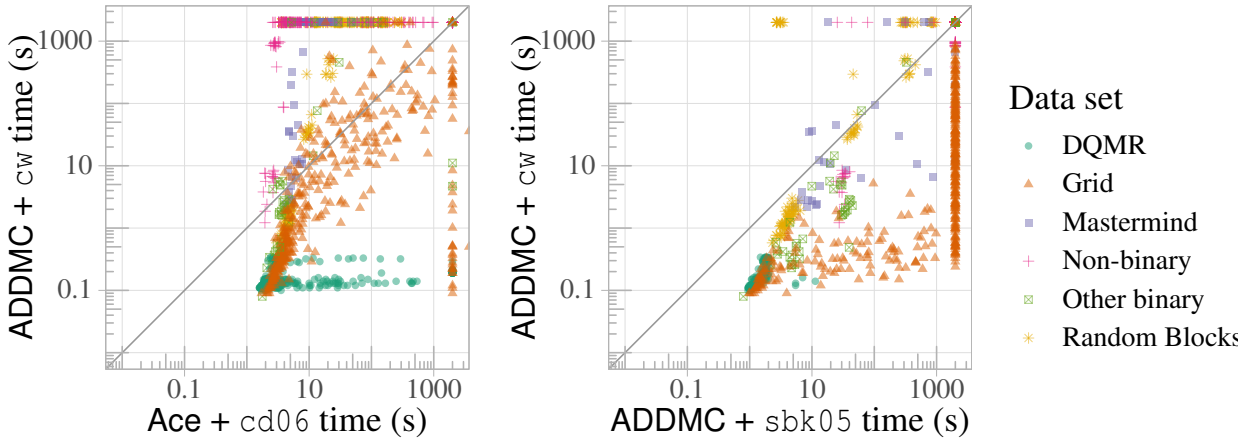


Figure 4.3: An instance-by-instance comparison between ADDMC + cw and the best overall combination of algorithm and encoding (Ace + cd06, on the left) as well as the second-best encoding for ADDMC (sbk05, on the right).

4.6 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.[8] 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[9] running Scientific Linux 7

[8] 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 Darwiche, 2005, 2006].

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

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[10] [Sang et al., 2004] for `sbk05` [Sang et al., 2005]; and c2d [Darwiche, 2004] for compilation and query-dnnf [11] for answer computation for `bklm16` [Bart et al., 2016]. For encoding, we use bn2cnf [12] 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. Otherwise, let X denote the last-mentioned vertex in the Bayesian network. If $\mathsf{true} \in \mathrm{im}\, X$, then we compute the marginal probability of $X = \mathsf{true}$. 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 and
- Grid networks as described by Sang et al. [2005],
- Mastermind, and
- Random Blocks from the work of Chavira et al. [2006],
- remaining binary Bayesian networks that include Plan Recognition [Sang et al., 2005], Friends and Smokers, Students and Professors [Chavira et al., 2006], and `tcc4f`, and
- non-binary classic Bayesian networks (`alarm`, `diabetes`, `hailfinder`, `mildew`, `munin1–4`, `pathfinder`, `pigs`, `water`).

[10] `https://cs.rochester.edu/u/kautz/Cachet/`
[11] `http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html`
[12] `http://www.cril.univ-artois.fr/KC/bn2cnf.html`

Table 4.3: Asymptotic upper bounds on the numbers of variables and clauses/ADDs for each encoding.

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

Figure 4.2 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., 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 4.2 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 Fig. 4.3 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 Fig. 4.3 shows that `cw` is better than `sbk05` [Sang et al., 2005] (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.

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 vertices in the Bayesian network, $d = \max_{X \in \mathcal{V}} |\text{pa}(X)|$ 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 4.3 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 4) 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.

4.7 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., 2005]. 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 Boolean 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] and DPMC —an extension to ADDMC that adds support for computations based on tensors (rather than ADDs) and planning based on tree decompositions [Dudek et al., 2020b].

Chapter 5

Weighted Model Counting Without Parameter Variables

5.1 Introduction

Weighted model counting (WMC), i.e., a generalisation of propositional model counting that assigns weights to literals and computes the total weight of all models of a propositional formula [Chavira and Darwiche, 2008], has emerged as a powerful computational framework for problems in many domains, e.g., probabilistic graphical models such as Bayesian networks and Markov networks [Bart et al., 2016, Chavira and Darwiche, 2005, 2006, Darwiche, 2002, Sang et al., 2005], neuro-symbolic artificial intelligence [Xu et al., 2018], probabilistic programs [Holtzen et al., 2020b], and probabilistic logic programs [Fierens et al., 2015]. It has been extended to support continuous variables [Belle et al., 2015], infinite domains [Belle, 2017], first-order logic [Gogate and Domingos, 2016, Van den Broeck et al., 2011], and arbitrary semirings [Belle and De Raedt, 2020, Kimmig et al., 2017]. However, as the definition of WMC puts weights on literals, additional variables often need to be added for the sole purpose of holding a weight [Bart et al., 2016, Chavira and Darwiche, 2005, 2006, Darwiche, 2002, Sang et al., 2005]. This can be particularly detrimental to WMC algorithms that rely on variable ordering heuristics.

One approach to this problem considers weighted clauses and probabilistic semantics based on Markov networks [Gogate and Domingos, 2010]. However, with a new representation comes the need to invent new encodings and inference algorithms. Our work is similar in spirit in that it introduces a new representation for computational problems but can reuse recent WMC algorithms based on pseudo-Boolean function manipulation, namely, ADDMC [Dudek et al., 2020a] and DPMC [Dudek et al., 2020b]. Furthermore, we identify sufficient conditions for transforming a WMC instance into our new format. 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, 2001], and sentential decision diagrams (SDDs) [Darwiche, 2011], another way to avoid parameter variables could be 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. To the best of the authors' knowledge, neither approach [Choi et al., 2013, Gogate and Domingos, 2010] has a publicly available implementation.

In this work, we introduce a way to transform WMC problems into a new format based on pseudo-Boolean functions—*pseudo-Boolean projection* (PBP). We formally show that every WMC problem instance has a corresponding PBP instance 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., 2005] 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 definition 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.

5.2 Weighted Model Counting

We begin with an overview of some notation and terminology. We use $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ to denote conjunction, disjunction, negation, material implication, and material biconditional, respectively. Throughout the paper, we use set-theoretic notation for many concepts in logic. A *clause* is a set of literals that are part of an implicit disjunction. Similarly, a *formula* in CNF is a set of clauses that are part of an implicit conjunction. We identify a *model* with a set of variables that correspond to the positive literals in the model (and all other variables are the negative literals of the model). We can then define the *cardinality* of a model as the cardinality of this set. For example, let $\phi = (\neg a \vee b) \wedge a$ be a propositional formula over variables a and b. Then an equivalent set-theoretic representation of ϕ is $\{\{\neg a, b\}, \{a\}\}$. Any subset of $\{a, b\}$ is an interpretation of ϕ, e.g., $\{a, b\}$ is a model of ϕ (written $\{a, b\} \models \phi$) of cardinality two, while $\emptyset$ is an interpretation but not a model. We can now formally define WMC.

Definition 6 (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 $\sum_{Y \models \phi} \prod_{Y \models l} w(l)$.

That is, the answer to a WMC instance is the sum of the weights of all models of ϕ, where the weight of a model is defined as the product of the weights of all (positive and negative) literals in it. Our definition of WMC is largely based on the standard definition [Chavira and Darwiche, 2008], but explicitly partitions variables into indicator and parameter variables. In practice, we identify this partition 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 a variation of the WMC problem used by some of the Bayesian network encodings [Chavira and Darwiche, 2005, 2006].

Definition 7. 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 8 (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 13. 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.

5.2.1 Bayesian Network Encodings

A *Bayesian network* is a directed acyclic graph with random variables as vertices and edges as conditional dependencies. As is common in related literature [Darwiche, 2002, Sang et al., 2005], we assume that each variable has a finite number of values. We call a Bayesian network *binary* if every variable has two values. If all variables have finite numbers of values, the probability function associated with each variable v can be represented as a *conditional probability table* (CPT), i.e., a table with a row for each combination of values that v and its parent vertices can take. Each row then also has a *probability*, i.e., a number in $[0, 1]$.

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. We will refer to them based on the initials of the authors and the year of publication. 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 CPTs. The encoding `sbk05` [Sang et al., 2005] is the only encoding that deviates from this arrangement: for each variable in the Bayesian network, one 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

[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/`.

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], the same authors 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 14. 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 13 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]

5.3 Pseudo-Boolean Functions

In this work, we propose a more expressive representation for WMC based on pseudo-Boolean functions. A *pseudo-Boolean function* is a function of the form $\{0,1\}^n \to \mathbb{R}$ [Boros and Hammer, 2002]. Equivalently, let X denote a set with n elements (we will refer to them as *variables*), and 2^X denote its powerset. Then a pseudo-Boolean function can have 2^X as its domain (then it is also known as a *set function*).

Pseudo-Boolean functions, most commonly represented as algebraic decision diagrams (ADDs) [Bahar et al., 1997] (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 [Poon and Domingos, 2011] to Bayesian network conversion [Zhao et al., 2015]. ADDs have been extended to compactly represent additive and

[2] Example 14 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.

multiplicative structure [Sanner and McAllester, 2005], sentences in first-order logic [Sanner and Boutilier, 2009], and continuous variables [Sanner et al., 2011], the last of which was also applied to weighted model integration, i.e., the WMC extension for continuous variables [Belle et al., 2015, Kolb et al., 2018].

Since two-valued pseudo-Boolean functions will be used extensively henceforth, we introduce some new notation. For any propositional formula ϕ over 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 $Y \subseteq X$. Next, we define some useful operations on pseudo-Boolean functions. The definitions of multiplication and projection are equivalent to those in previous work [Dudek et al., 2020a,b].

Definition 9 (Operations)**.** Let $f, g \colon 2^X \to \mathbb{R}$ be pseudo-Boolean functions, $x, y \in X$, $Y = \{y_i\}_{i=1}^n \subseteq X$, and $r \in \mathbb{R}$. Operations such as addition and multiplication are defined pointwise, i.e., $(f+g)(Y) \coloneqq f(Y) + g(Y)$, and likewise for multiplication. Note that properties such as associativity and commutativity are inherited from $\mathbb{R}$. By regarding a real number as a constant pseudo-Boolean function, we can reuse the same definitions to define *scalar* operations as $(r+f)(Y) = r + f(Y)$, and $(r \cdot f)(Y) = r \cdot f(Y)$.

Restrictions $f|_{x=0}, f|_{x=1} \colon 2^X \to \mathbb{R}$ of f are defined as $f|_{x=0}(Y) \coloneqq f(Y \setminus \{x\})$, and $f|_{x=1}(Y) \coloneqq f(Y \cup \{x\})$ for all $Y \subseteq X$.

Projection $\exists_x$ is an endomorphism $\exists_x \colon \mathbb{R}^{2^X} \to \mathbb{R}^{2^X}$ defined as $\exists_x f \coloneqq f|_{x=1} + f|_{x=0}$. Since projection is commutative (i.e., $\exists_x \exists_y f = \exists_y \exists_x f$) [Dudek et al., 2020a,b], we can define $\exists_Y \colon \mathbb{R}^{2^X} \to \mathbb{R}^{2^X}$ as $\exists_Y \coloneqq \exists_{y_1} \exists_{y_2} \ldots \exists_{y_n}$. Throughout the paper, projection is assumed to have the lowest precedence (e.g., $\exists_x fg = \exists_x (fg)$).

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

Proposition 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. Note that our definitions of binary operations assumed equal domains. For convenience, we can assume domains to shrink whenever a function 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 $[y]_0^1$).

5.4 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 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 10 (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 $\omega \cdot \left(\exists_X \prod_{f \in F} f\right)(\emptyset)$.

5.4.1 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., 2005].

The algorithm is based on several observations that will be made more precise in Sect. 5.4.2. 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 first appears in the `cd05` paper [Chavira and Darwiche, 2005]). Note that while we formulate our algorithm as a sequel to the WMC encoding

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

Algorithm 5: 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* 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 $v \in X_I$ *such that* $\{[v]_1^p, [\neg v]_1^q\} \subseteq F$ *for some* p *and* q **do**
  $F \leftarrow F \setminus \{[v]_1^p, [\neg v]_1^q\} \cup \{[v]_q^p\}$;

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.

The algorithm is listed as Algorithm 5. The main part of the algorithms 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).

5.4.2 Correctness Proofs

In this section, we outline key conditions that a (WMC or minimum-cardinality WMC) encoding has to satisfy for Algorithm 5 to output an equivalent PBP instance. We divide the correctness proof into two theorems: Theorem 6 for WMC encodings (i.e., `bklm16` and `d02`) and Theorem 7 for minimum-cardinality WMC encodings (i.e., `cd05` and `cd06`). We begin by listing some properties of pseudo-Boolean functions and establishing a canonical transformation from WMC to PBP.

Theorem 5 (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 3. *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 2. *Let* (ϕ, X_I, X_P, w) *be a WMC instance. Then*

$$\left(\left\{[c]_0^1 \mid c \in \phi\right\} \cup \left\{[x]_{w(\neg x)}^{w(x)} \;\middle|\; x \in X_I \cup X_P\right\}, X_I \cup X_P, 1\right) \tag{5.1}$$

is a PBP instance with the same answer (as defined in Definitions 6 and 10).

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 (5.1) 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 3. 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 6 (Correctness for WMC)**.** *Algorithm 5, when given a WMC instance* (ϕ, X_I, X_P, w)*, returns a PBP instance with the same answer (as defined in Definitions 6 and 10), 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 2,

$$\left(\left\{[c]_0^1 \mid c \in \phi\right\} \cup \left\{[x]_{w(\neg x)}^{w(x)} \;\middle|\; x \in X_I \cup X_P\right\}, X_I \cup X_P, 1\right) \tag{5.2}$$

is a PBP instance with the same answer as the given WMC instance. By Definition 10, 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 5 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 to 10 differs from (5.2). Note that Algorithm 5 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

$$\omega \prod_{f\in F_p} f = \exists_p [p]_{w(\neg p)}^{w(p)} \prod_{c\in\phi_p} [c]_0^1. \tag{5.3}$$

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

$$\exists_p [p]_1^{w(p)} \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i\right]_0^1 = \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i\right]_0^1\Bigg|_{p=1} + \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i\right]_0^1\Bigg|_{p=0}. \tag{5.4}$$

Since for any input, $\bigwedge_{i=1}^n l_i$ is either true or false, exactly one of the two summands in Eq. (5.4) 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 Eq. (5.3), since $F_p = \emptyset$, and ω is unchanged, we get $\omega \prod_{f\in F_p} f = 1$, and so Eq. (5.3) 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 Eq. (5.3). If $w(p) = 1$, then $F_p = \emptyset$, and $\omega = 1$, and so the left-hand side of Eq. (5.3) 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 5 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 5 preserves the answer under both Condition 1 and Condition 2. Finally, the loop on Lines 11 to 12 of Algorithm 5 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 7 (Minimum-Cardinality Correctness)**.** *Let (ϕ, X_I, X_P, w) be a minimum-cardinality WMC instance that satisfies Conditions 2b to 2e of Theorem 6 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 5, when applied to (ϕ, X_I, X_P, w), outputs a PBP instance with the same answer (as defined in Definitions 8 and 10).

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 6 and 7, the following proof is slightly less explicit and assumes that $\omega = 1$.

Proof. Let (F, X_I, ω) be the tuple returned by Algorithm 5 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] 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

$$\prod_{Z \models l} w(l) = \left(\prod_{f \in F} f \right)(Y). \tag{5.5}$$

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

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

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 6). 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 Eq. (5.5). 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 6). This argument works in the other direction as well. □

5.5 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 5 applied afterwards.[7] For each encoding `e`, we write `e++` to denote the combination of encoding

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

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

a Bayesian network as a WMC instance using `e` and transforming it into a PBP instance using Algorithm 5. 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 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?

5.5.1 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.

For experimental data, we use Bayesian networks available with Ace and Cachet. We split them into the following groups:

- DQMR (390 instances) and
- Grid networks (450 instances) as described by Sang et al. [2005];
- Mastermind (144 instances) and
- Random Blocks (256 instances) by Chavira et al. [2006];
- other binary Bayesian networks (50 instances) including Plan Recognition [Sang et al., 2005], Friends and Smokers, Students and Professors [Chavira et al., 2006], and `tcc4f`;

[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`

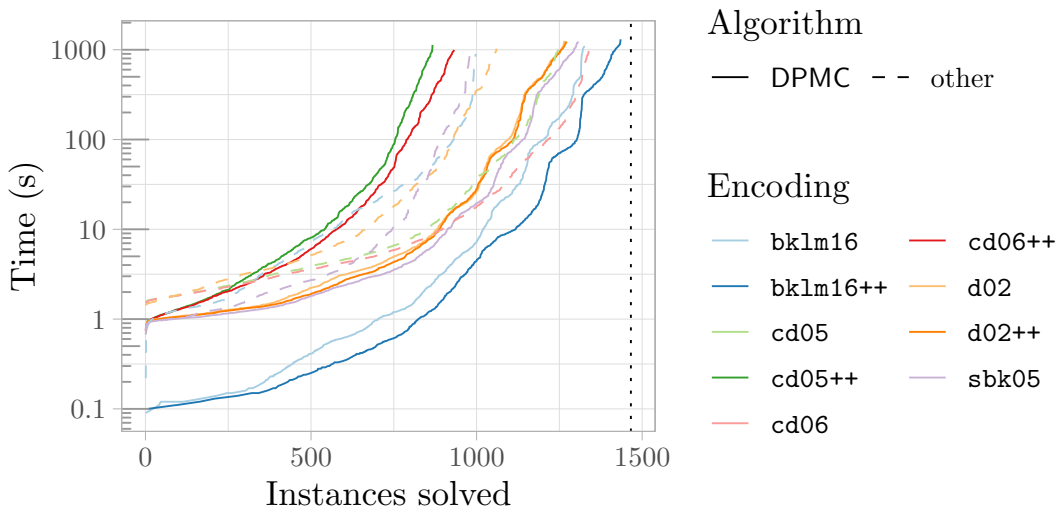


Figure 5.1: Cactus plot of all algorithm-encoding pairs. The dotted line denotes the total number of instances used.

- non-binary classic networks (176 instances): `alarm`, `diabetes`, `hailfinder`, `mildew`, `munin1–4`, `pathfinder`, `pigs`, and `water`.

To perform Bayesian network inference with DPMC (or with any other WMC algorithm not based on compilation such as Cachet), one needs to select a probability to compute [Dudek et al., 2020b, Sang et al., 2004]. If a network comes with an evidence file, we compute the probability of this evidence. Otherwise, let X be the variable last mentioned in the Bayesian network file. If true is one of the values of X, then we compute $\Pr(X = \text{true})$, otherwise we choose the first-mentioned value of X.

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]

5.5.2 Results

Figure 5.1 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 5.1 further shows that DPMC + `bklm16++` solves almost a hundred more instances than any other combination, and is the fastest in 69.1 % of them.

The scatter plots in Fig. 5.2 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 5 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.

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

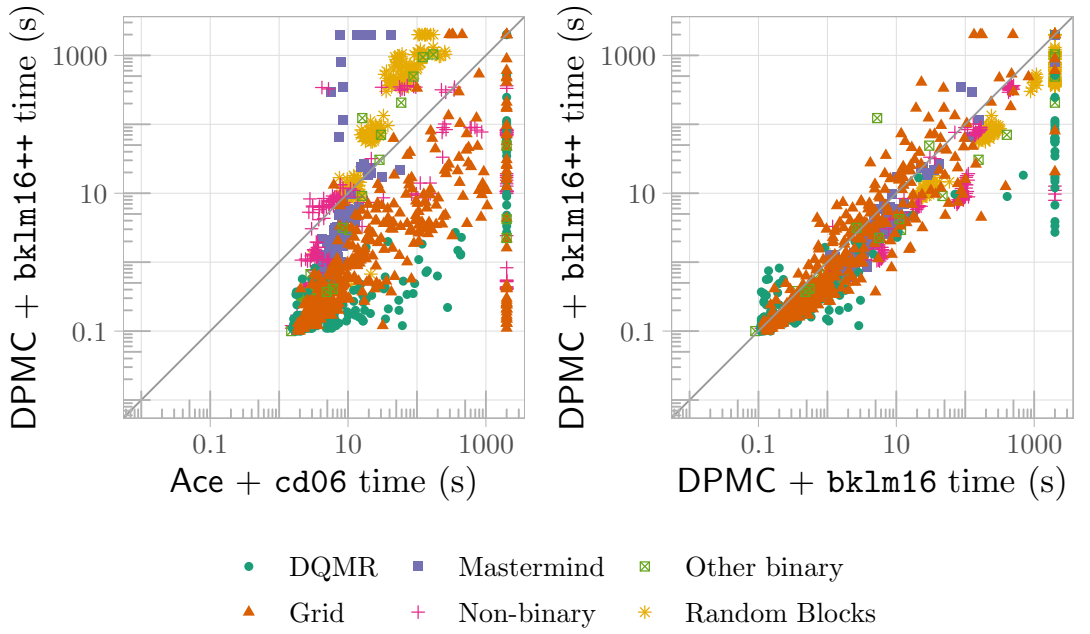


Figure 5.2: An instance-by-instance comparison between DPMC + `bklm16++` (the best combination according to Fig. 5.1) and the second and third best-performing combinations: Ace + `cd06` and DPMC + `bklm16`.

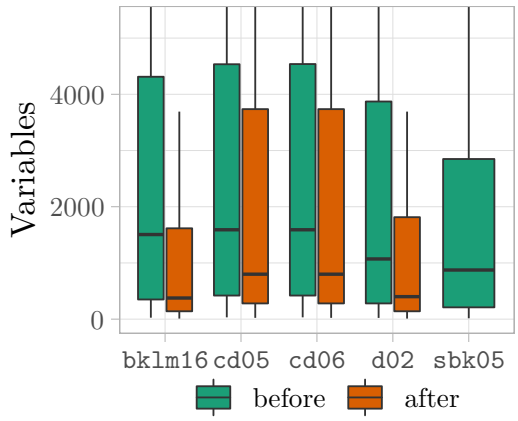


Figure 5.3: Box plots of the numbers of variables in each encoding across all benchmark instances before and after applying Algorithm 5. Outliers and the top parts of some whiskers are omitted.

Table 5.1: The numbers of instances (out of 1466) that each algorithm and encoding combination solved faster than any other combination and in total.

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

We also record numbers of variables in each encoding before and after applying Algorithm 5. Figure 5.3 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 treewidth of the primal graph of the input formula [Dudek et al., 2020b]) before and after applying Algorithm 5—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.

5.6 Conclusion

In this paper, 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 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., 2005]. 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 in some cases 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 cd05 and cd06 papers [Chavira and Darwiche, 2005, 2006]) to be transferred to other application domains.

[13] The data on this (along with the implementation of Algorithm 5) is available at https://github.com/dilkas/wmc-without-parameters.

Chapter 6

Generating Random Weighted Model Counting Instances: An Empirical Analysis with Varying Primal Treewidth

Chapter 7

Recursive Solutions to First-Order Model Counting

7.1 Introduction

7.1.0.0.1 Abstract. First-order model counting (FOMC) is a #P-complete computational problem that asks to count the models of a sentence in first-order logic. Despite being around for more than a decade, practical FOMC algorithms are still unable to compute functions as simple as a factorial. 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 (weighted) FOMC algorithm ForcLift with new compilation rules and an algorithm to check whether a recursive call is feasible. These improvements allow us to find efficient solutions to counting fundamental structures such as injections and bijections.

In a way, we're dividing the idea of domain recursion between the IDR and the Ref nodes, thus also generalising it. [TODO: mention Fig. 7.1.]

7.2 First-Order Logic

In this section, we describe a variation of function-free first-order logic with equality. For a more complete exposition of first-order logic, see the book by Brachman and Levesque [2004].

In first-order logic, an *atom* (i.e., an atomic formula) is either $t_1 = t_2$ or $P(t_1, \ldots, t_n)$ for some predicate P and terms $(t_i)_{i=1}^n$. Here, $n \in \mathbb{N}_0$ is the *arity* of P. A *term* is either a constant or a variable (we also call terms *arguments*). An atom is *ground* if all of its terms are constants.

A *formula* is a well-formed expression that connects atoms using the previously described logical operators as well as universal ($\forall x \in D.\phi$) and existential ($\exists x \in D.\phi$) quantifiers. Here, x is a variable, D is a domain, and ϕ is a formula. A variable occurrence is *bound* if it is within the scope of a quantifier for that variable, otherwise

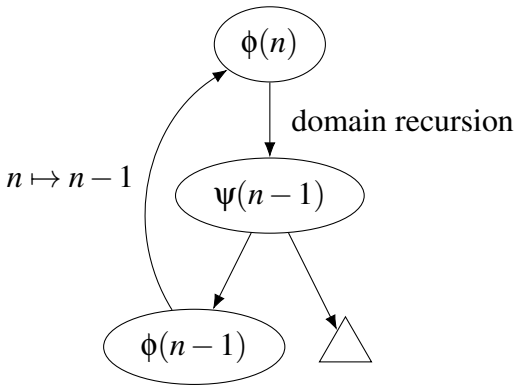


Figure 7.1: An illustration of the main idea.

it is *free*. A formula is a *sentence* if all of its variable occurrences are bound.[1]

7.2.1 How to Interpret a Sentence

Let ϕ be a sentence. Since each variable in ϕ is introduced by a quantifier, each variable is linked to a unique domain. As is done implicitly in previous work [Van den Broeck, 2013], we make the following assumption.

Assumption 1. *Each constant can be mapped to a domain, and each n-ary predicate can be mapped to a sequence of n domains such that:*

1. ...
2. ...

First, we assume that each n-ary predicate can be associated with a sequence of n domains, one for each of its terms. Second, we assume that each constant can be similarly linked to a unique domain. Third, we assume that equality is only checked between terms that have the same domain.

TODO: describe this more formally. Maybe just two rules: whenever two terms are compared, they have the same domain, the domain of a term is always the same as the k-th domain of the predicate.

- Interpretation
  - domains as sets
  - constants as specific elements of domains (different constant implies different element. If the domain is not big enough, then the formula is unsat.)
  - variables (indicated during quantification)
  - predicates are interpreted as relations
- models are defined very differently: all combinations of relations that satisfy the sentence.

[1] In the rest of this chapter, all formulas in first-order logic are assumed to be sentences.

- (multiple) (finite) domains (of discourse). This makes, e.g., satisfiability decidable and a FO formula can always be expanded to a propositional one.

more on first-order logic: [Brachman and Levesque, 2004]. The MLN paper also has a description

7.3 Preliminaries

7.3.0.0.1 Things I might need to explain.

- atom, (positive/negative) literal, constant, predicate, variable, clause, unit clause, first-order logic
- WMC, w, $\overline{\mathrm{w}}$.
- two parts: compilation and inference.
- During inference, there is a domain size map $\sigma\colon \mathcal{D} \to \mathbb{N}_0$.
- mention which rules are in Γ and which ones are in Δ (and why tryCache has to be in Δ). TODO: having notation for Δ and Γ isn't that necessary
- My formalisation of constraints is more restrictive than the original (e.g., in the thesis).
- FORCLIFT
- $\sqcup$ for both sets and functions

TODO: reorder Section 6 to introduce DR before CR before Ref.

7.3.0.0.2 Notation.

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

Most of the definitions here are adaptations/formalisations of Van den Broeck et al. [2011] and the corresponding code.

Definition 11. A *domain* is a symbol for a finite set.[2]

Definition 12. An *(inequality) constraint* is a pair (a, b), where a is a variable, and b is either a variable or a constant.

[2] In the context of functions, the domain of a function f retains its usual meaning and is denoted $\operatorname{dom}(f)$.

Definition 13. A *clause* 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 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 $\delta_c(x) = \delta_c(y)$. Note that for convenience sometimes we write δ_c for the domain map of c without unpacking c into its three constituents.

Also, let Vars be a function that maps clauses and sets of literals and inequalities to the set of variables contained within. In particular, $\text{Vars}(c) \coloneqq \text{Vars}(L) \cup \text{Vars}(C)$.

A *formula* is a set of clauses. 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. This way, one can read formulas as defined here as sentences in first-order logic.

TODO: Note: we will reuse this several times.

Example 15. Let $\phi \coloneqq \{c_1, c_2\}$ be a formula, where

$$c_1 \coloneqq (\{\neg p(X,Y), \neg p(X,Z)\}, \{(Y,Z)\}, \{X \mapsto a, Y \mapsto b, Z \mapsto b\}), \tag{7.1}$$

and

$$c_2 \coloneqq (\{\neg p(X,Y), \neg p(Z,Y)\}, \{(X,Z)\}, \{X \mapsto a, Y \mapsto b, Z \mapsto a\}), \tag{7.2}$$

for some predicate p, variables X, Y, Z, and domains a and b. Then in first-order logic ϕ could be read as

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

7.3.0.0.3 Hashing. We use hash codes to efficiently check whether a recursive relationship between two formulas is plausible. (It is plausible if the formulas are equal up to variables and domains.) The hash code of a clause $c = (L, C, \delta)$ 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 $|\text{Vars}(c)|$. The hash code of a formula ϕ combines the hash codes of all its clauses and is denoted $\#\phi$.

7.3.0.0.4 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 (a.k.a. head) h and remaining list (a.k.a. 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 $|M| = 1$.

7.4 Generalising Circuits to Labelled Graphs

A *first-order deterministic decomposable negation normal form computational graph* (FCG) is a (weakly connected) directed graph with a a single source, vertex labels, and

ordered outgoing edges.[3] We denote an FCG as $G = (V, s, N^+, \tau)$, where V is the set of vertices, and $s \in V$ is the unique source. Also, N^+ is the direct successor function that maps each vertex in V to a *list* that contains either other vertices in V or a special symbol $\star$. This symbol means that the target of the edge is yet to be determined.

Vertex labels consist of two parts: the *type* and the *parameters*. For the main definition, we leave the parameters implicit and let $\tau\colon V \to \mathcal{T}$ denote the vertex-labelling function that maps each vertex in V to its type in $\mathcal{T}$. Most of our list of types $\mathcal{T} \coloneqq \{\bigcirc, ①, ⑂, ⑄, Ⓐ, Ⓥ, \text{CR}, \text{DR}, \text{IE}, \text{REF}\}$ is as described in previous work Van den Broeck [2011], Van den Broeck et al. [2011] as well as the source code of FORCLIFT[4] but with one new type CR and two revamped types DR and REF. For each vertex $v \in V$, the length of the list $N^+(v)$ (i.e., the out-degree of v) is determined by its type $\tau(v)$.

TODO: conjunctions and disjunctions should be different from set-conjunctions and set-disjunctions.

As in previous work Van den Broeck et al. [2011], to describe individual vertices and small (sub)-FCGs, we also use notation of the form, e.g., $\text{REF}_\rho(v)$. Here, the type of the vertex (e.g., REF) is 'applied' to its direct successors (e.g., v) using either function or infix notation and provided with its parameter(s) (e.g., ρ) in the subscript. We say that 'G is an FCG for formula ϕ' if two conditions are satisfied. First, the image of N^+ contains no $\star$'s. Second, G encodes a function that maps the sizes of the domains in ϕ to the WMC of ϕ (more on this in Sect. 7.8).

7.4.0.0.1 TODO.

- provide a short explanation of the types (emphasising which ones are new/updated).
- have an example of a simple FCG. Maybe describe the figure from later and move it here.

7.5 Compilation as Search

Definition 14. A *state* (of the search for an FCG for a given formula) is a tuple (G, C, L), where:

- G is an FCG (or `null`),
- C is a compilation cache that maps integers to sets of pairs (ϕ, v), where ϕ is a formula, and v is a vertex of G (which is used to identify opportunities for recursion),
- and L is a list of formulas (that are yet to be compiled). (Note that the order is crucial!)

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

[4] `https://dtai.cs.kuleuven.be/drupal/wfomc`

Algorithm 6: The (main part of the) search algorithm

Input: a formula ϕ_0
Output: all found FCGs for ϕ_0 are in the set solutions

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

Definition 15. A (compilation) *rule* is a function that takes a formula and returns a set of (G, L) pairs, where G is a (potentially `null`) FCG, and L is a list of formulas.

TODO: Take the example from the AIAI talk.

We assume that if there is a pair $(\texttt{null}, L)$ in the set returned by a rule, then $|L| = 1$, i.e., the rule transformed the formula without creating any vertices.

TODO: explain the 'tail' part of the algorithm, i.e., that the first formula is replaced by some vertices and some formulas. And explain why we don't want to have REF vertices in the cache.

Note: At the end, `mergeFcgs` will never return `null` because there is going to be at least one $\star$ in G and the function will find it.

7.6 In-Between Compilation and Inference: Smoothing

[Insert motivation for smoothing from Section 3.4. of the ForcLift paper.] Originally, smoothing was (and still is) a two-step process. First, atoms that are still accounted for in the circuit are propagated upwards. Then, at vertices of certain types, missing atoms are detected and additional sinks are created to account for them. If left unchanged, the first step of this process would result in an infinite loop whenever a cycle is encountered. Algorithm 9 outlines how the first step can be adapted to an arbitrary directed graph.

7.7 New Compilation Rules

Let $\mathcal{D}$ be the set of all domains. Note that this set expands during the compilation.

Algorithm 7: Functions used in Algorithm 6 for applying compilation rules

Data: a set of greedy rules Γ
Data: a set of non-greedy rules Δ
Function `applyGreedyRules` (ϕ, C) **:**
  foreach *rule* $r \in \Gamma$ **do**
    if $r(\phi) \neq \emptyset$ **then**
      $(G, L) \leftarrow$ any element of $r(\phi)$;
      if $G =$ `null` **then return** `applyGreedyRules` (*the only element of* L, C) ;
      else
        $(V, s, N^+, \tau) \leftarrow G$;
        $C \leftarrow$ `updateCache` (C, ϕ, G) ;
        return `applyGreedyRulesToAllFormulas` (G, C, L) ;
  return $(\texttt{null}, C, \langle\phi\rangle)$;
Function `applyGreedyRulesToAllFormulas` ((V, s, N^+, τ), C, L) **:**
  if $L = \langle\rangle$ **then return** $((V, s, N^+, \tau), C, L)$;
  $N^+(s) \leftarrow \langle\rangle$;
  $L' \leftarrow \langle\rangle$;
  foreach *formula* $\phi \in L$ **do**
    $(G', C, L'') \leftarrow$ `applyGreedyRules` (ϕ, C) ;
    $L' \leftarrow L' +\!\!+ L''$;
    if $G' =$ `null` **then** $N^+(s) \leftarrow N^+(s) +\!\!+ \langle\star\rangle$;
    else
      $(V', s', N', \tau') \leftarrow G'$;
      $V \leftarrow V \sqcup V'$;
      $N^+ \leftarrow N^+ \sqcup N'$;
      $N^+(s) \leftarrow N^+(s) +\!\!+ \langle s'\rangle$;
      $\tau \leftarrow \tau \sqcup \tau'$;
  return $((V, s, N^+, \tau), C, L')$;
Function `applyAllRules` (s) **:**
  $(G, C, L) \leftarrow s$;
  $\phi : T \leftarrow L$;
  $(G', C', L') \leftarrow$ a copy of s;
  newStates $\leftarrow \langle\rangle$;
  foreach *rule* $r \in \Delta$ **do**
    foreach $(G'', L'') \in r(\phi)$ **do**
      if $G'' =$ `null` **then**
        newStates $\leftarrow$ newStates $+\!\!+$ `applyAllRules` $((G', C', L''))$;
      else
        $(V, s, N^+, \tau) \leftarrow G''$;
        $C' \leftarrow$ `updateCache` (C', ϕ, G'') ;
        $(G'', C', L'') \leftarrow$ `applyGreedyRulesToAllFormulas` (G'', C', L'') ;
        if $G' =$ `null` **then**
          newStates $\leftarrow$ newStates $+\!\!+ \langle(G'', C', L'' +\!\!+ T)\rangle$;
        else newStates $\leftarrow$
          newStates $+\!\!+ \langle($`mergeFcgs` $(G', G''), C', L'' +\!\!+ T)\rangle$;
    $(G', C', L') \leftarrow$ a copy of s;
  return newStates;

Algorithm 8: Helper functions used by Algorithm 7

Function `updateCache` $(C,\ \phi,\ (V,s,N^+,\tau))$**:**
  if $\tau(s) = \textsc{Ref}$ **then return** C;
  if $\#\phi \notin \mathrm{dom}(C)$ **then return** $C \cup \{\, \#\phi \mapsto (\phi, s) \,\}$;
  if *there is no* $(\phi', v) \in C(\#\phi)$ *s.t.* $v = s$ **then** $C(\#\phi) \leftarrow (\phi, s) + C(\#\phi)$;
  return C;
Function `mergeFcgs` $(G = (V,s,N^+,\tau),\ G' = (V',s',N',\tau'),\ r = s)$**:**
  if $\tau(r) = \textsc{Ref}$ **then return** `null`;
  foreach $t \in N^+(r)$ **do**
    if $t = \star$ **then**
      replace t with s' in $N^+(r)$;
      return $(V \sqcup V', s, N^+ \sqcup N', \tau \sqcup \tau')$;
    $G'' \leftarrow$ `mergeFcgs` $(G,\ G',\ t)$;
    if $G'' \neq$ `null` **then return** G'';
  return `null`;

Algorithm 9: Propagating atoms for smoothing across the FCG in a way that avoids infinite loops

Input: FCG (V,s,N^+,τ)
Input: function ι that maps vertex types in $\mathcal{T}$ to sets of atoms
Input: functions $\{\, f_t \,\}_{t \in \mathcal{T}}$ that take a list of sets of atoms and return a set of atoms
Output: function S that maps vertices in V to sets of atoms
$S \leftarrow \{\, v \mapsto \iota(\tau(v)) \mid v \in V \,\}$;
changed $\leftarrow$ `true`;
while changed **do**
  changed $\leftarrow$ `false`;
  foreach *vertex* $v \in V$ **do**
    $S' \leftarrow f_{\tau(v)}(\langle S(w) \mid w \in N^+(v) \rangle)$;
    if $S' \neq S(v)$ **then**
      changed $\leftarrow$ `true`;
      $S(v) \leftarrow S'$;

7.7.1 Identifying Possibilities for Recursion

7.7.1.1 Notation

Let Doms be a function that maps any clause or formula to the set of domains used within. Specifically, $\mathrm{Doms}(c) := \mathrm{Im}\,\delta_c$ for any clause c, and $\mathrm{Doms}(\phi) := \bigcup_{c\in\phi} \mathrm{Doms}(c)$ for any formula ϕ.

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 $\alpha \cup \beta|_{\mathrm{dom}(\beta)} = \beta$.

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 $\delta_d := \gamma \circ \delta_c \circ \beta^{-1}$. In other words, δ_d is the unique function that makes

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

commute. For example, if clause c_1 is as in Example 15, then

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

7.7.1.2 Everything Else

7.7.1.2.1 TODO

- introduce/describe Algorithm 10 and describe the cache that's being used.
- explain why $\rho \cup \gamma$ is possible
- explain what the second return statement is about and why a third one is not necessary
- explain the yield keyword
- in the example below: write down both formula using the ForcLift format

The algorithm could be improved in two ways:

- by constructing a domain map first and then using it to reduce the number of viable variable bijections.
- by similarly using the domain map ρ.

However, it is not clear that this would result in an overall performance improvement, since the number of variables in instances of interest never exceeds three and the identity bijection is typically the right one.

Algorithm 10: The compilation rule for REF

Input: formula ϕ
Output: either a singleton with the new REF vertex or $\emptyset$

foreach $(\psi, v) \in$ compilationCache(#ϕ) **do**
  $\rho \leftarrow$ `identifyRecursion`(ϕ, ψ);
  if $\rho \neq$ `null` **then return** $\{(\text{REF}_\rho(v), \emptyset)\}$;
return $\emptyset$;
Function `identifyRecursion`$(\phi, \psi, \rho = \emptyset)$**:**
  if $|\phi| \neq |\psi|$ **or** $\#\phi \neq \#\psi$ **then return** `null`;
  if $\phi = \emptyset$ **then return** ρ;
  foreach *clause* $c \in \phi$ **do**
    foreach *clause* $d \in \psi$ *s.t.* $\#d = \#c$ **do**
      foreach $(\beta, \gamma) \in$ `generateMaps`(c, d, ρ) *s.t.* $c[\beta, \gamma] = d$ **do**
        $\rho' \leftarrow$ `identifyRecursion`$(\phi \setminus \{c\}, \psi \setminus \{d\}, \rho \cup \gamma)$;
        if $\rho' \neq$ `null` **then return** ρ';
    return `null`;
Function `generateMaps`(c, d, ρ)**:**
  foreach *bijection* $\beta\colon \text{Vars}(c) \to \text{Vars}(d)$ **do**
    $\gamma \leftarrow$ `constructDomainMap`$(\text{Vars}(c), \delta_c, \delta_d, \beta, \rho)$;
    if $\gamma \neq$ `null` **then yield** (β, γ);
Function `constructDomainMap`$(V, \delta_c, \delta_d, \beta, \rho)$**:**
  $\gamma \leftarrow \emptyset$;
  foreach $v \in V$ **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;

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

$$\begin{array}{ccc} \text{Vars}(c) & \xrightarrow{\beta} & \text{Vars}(d) \\ {\scriptstyle\delta_c}\downarrow & & \downarrow{\scriptstyle\delta_d} \\ \text{Doms}(c) & \xrightarrow{\gamma} & \text{Doms}(d) \\ \downarrow & & \downarrow \\ \mathcal{D} & \xrightarrow[\rho]{} & \mathcal{D}. \end{array} \tag{7.3}$$

commutes (and returns `null` if such a function does not exist).

TODO: update this example. Fix references.

Example 16. Let ϕ be the formula from Example 15 and $\psi := \phi[\text{id}, \{a \mapsto a', b \mapsto b^\perp\}]$ (i.e., ϕ with both of its domains replaced). Since $\#\phi = \#\psi$, and both formulas are non-

empty, the algorithm proceeds with the for-loops on Lines 8 to 10. Suppose c in the algorithm refers to Eq. (7.1), and d to **??**. 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:

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

Since $\delta_i(Y) = \delta_i(Z)$ for $i \in \{c, d\}$, `constructDomainMap` returns $\gamma = \{a \mapsto a', b \mapsto b^\perp\}$. Thus, `generateMaps` yields its first pair of maps (β, γ) to Line 10. Furthermore, the pair satisfies $c[\beta, \gamma] = d$.

Since $\pi(a') = a$, and $a' \in \mathcal{C}$, `traceAncestors(`a`, `a'`)` returns `true`, which sets foundConstraintRemoval$'$ to `true` as well. When $e = b$, however, `traceAncestors(`b, $b^\perp$`)` returns `false` since $b^\perp$ is a descendant of b but not created by the constraint removal compilation rule. On Line 11, a recursive call to `identifyRecursion(`ϕ'`, `ψ'`, `γ`, true)` is made, where ϕ' and ψ' are new formulas with one clause each: Eq. (7.2) and **??**, respectively.

Again we have two non-empty formulas with equal hash codes, so `generateMaps` is called with c set to Eq. (7.2), d set to **??**, and $\rho = \{a \mapsto a', b \mapsto b^\perp\}$. Suppose Line 15 picks the identity bijection first again. 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 Eq. (7.3)), 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(`$\emptyset$`, `$\emptyset$`, `ρ`, true)`, which immediately returns $\rho = \{a \mapsto a', b \mapsto b^\perp\}$ as the final answer. This means that one can indeed use an FCG for $\text{WMC}(\phi)$ to compute $\text{WMC}(\psi)$ by replacing every mention of a with a' and every mention of b with $b^\perp$.

Algorithm 11: The compilation rule for CR

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

$S \leftarrow \emptyset$;
foreach *domain* $d \in \mathcal{D}$ *and element* $e \in d$ *s.t.* e *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** $(v, e) \in C$ **do**
  add a new domain d' to $\mathcal{D}$;
  $\phi' \leftarrow \emptyset$;
  foreach *clause* $(L, C, \delta) \in \phi$ **do**
    $C' \leftarrow \{ (x, y) \in C \mid y \neq e \}$;
    $\delta' \leftarrow \emptyset$;
    foreach *variable* $v \in \text{Vars}(L) \cup \text{Vars}(C')$ **do**
      if $\delta(v) = d$ **then** $\delta' \leftarrow \delta' \cup \{ v \mapsto d' \}$;
      else $\delta' \leftarrow \delta' \cup \{ v \mapsto \delta(v) \}$;
    $\phi' \leftarrow \phi' \cup \{ (L, C', \delta') \}$;
  $S \leftarrow S \cup \{ \text{CR}_{d \mapsto d'}, \phi' \}$;

7.7.1.3 Evaluation

$\text{WMC}(\text{REF}_\rho(v); \sigma) = \text{WMC}(v; \sigma')$ (n is the target vertex), where σ' is defined as

$$\sigma'(x) = \begin{cases} \sigma(\rho(x)) & \text{if } x \in \text{dom}(\rho) \\ \sigma(x) & \text{otherwise} \end{cases}$$

for all $x \in \mathcal{D}$.

7.7.2 Constraint Removal

TODO: describe Algorithm 11 and rewrite the example below.

Example 17. Let $\phi = \{ c_1, c_2, c_e \}$ be a formula with clauses (constants lowercase, variables uppercase)

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

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

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

Algorithm 12: The compilation rule for DR

Input: formula ϕ
Output: set S

1 $S \leftarrow \emptyset$;
2 **foreach** *domain* $d \in \mathcal{D}$ *s.t. there is a clause* $c \in \phi$ *and a variable* $v \in \mathrm{Vars}(L_c)$ *s.t.* $\delta_c(v) = d$ **do**
3 $\quad \phi' \leftarrow \emptyset$;
4 $\quad x \leftarrow$ a new constant associated with 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** $W \cap \{ v \in \mathrm{Vars}(C) \mid (v, y) \in C \text{ for some constant } y \} = \emptyset$ **do**
$\quad\quad\quad$ /* Here, δ' is the restriction of 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 $\quad S \leftarrow S \cup \{ (\mathrm{DR}_{d \leftarrow d \setminus \{x\}}, \phi') \}$;

7.7.2.0.1 Evaluation.

$$\mathrm{WMC}(\mathrm{CR}_{d \mapsto d'}(n); \sigma) = \begin{cases} \mathrm{WMC}(n; \sigma \cup \{ d' \mapsto \sigma(d) - 1 \}) & \text{if } \sigma(d) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

7.7.3 A Generalisation of Domain Recursion

7.7.3.0.1 The algorithm uses this notation for substitution. Let S be a set of constraints or literals, V a set of variables, and x either a variable or a constant. Then we write $S[x/V]$ to denote S with all occurrences of all variables in V replaced with x.[5]

TODO: Compare with the original Van den Broeck [2011].

The reason for this precondtion is the same as in the initial version of domain recursion: there must be a variable with that domain featured among the literals because it needs to be replaced by a constant. TODO: expand this.

TODO: describe Algorithm 12.

Example 18. Let $\phi = \{ c_1, c_2 \}$ be a formula, where

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

While domain recursion is possible on both domains, here we illustrate how it works on a.

Suppose Line 5 picks $c = c_1$ first. Then $V = \{ X \}$. Both subsets of V satisfy the conditions on Line 7 and generate new clauses

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

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

(from $W = \emptyset$) and

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

(from $W = V$).

When $c = c_2$, then $V = \{X, Z\}$. The subset $W = V$ fails to satisfy the first condition because of the $Z \neq X$ constraint; without this condition, the resulting clause would have an unsatisfiable constraint $x \neq x$. The other three subsets of V all generate clauses for ϕ':

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

(from $W = \emptyset$),

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

(from $W = \{X\}$), and

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

(from $W = \{Z\}$).

7.7.3.0.2 Evaluation.

$$\mathrm{WMC}(\mathrm{DR}_{d \leftarrow d \setminus \{x\}}(n); \sigma) = \begin{cases} \mathrm{WMC}(n; \sigma) & \text{if } \sigma(d) > 0 \\ 1 & \text{otherwise.} \end{cases}$$

One is picked as the multiplicative identity.

7.8 How to Evaluate an FCG

Along with the three vertex types described above, here are all the other ones. This section is mostly just taken from Van den Broeck et al. [2011] but with some changes in notation.

Definition 16. Let $\mathrm{gr}(\cdot; \sigma)$ be the function (parameterised by the domain size function σ) that takes a clause $c = (L, C, \delta)$ and returns the number of ways the variables in c can be replaced by elements of their respective domains in a way that satisfies the inequality constraints.[6] Formally, for each variable $v \in \mathrm{Vars}(c)$, let $C_v = \{w \mid (u,w) \in C \setminus \mathrm{Vars}(c)^2, u \neq v\}$ be the set of (explicitly named) constants that v is permitted to be equal to. Then

$$\mathrm{gr}(c; \sigma) := \left| \left\{ (e_v)_{v \in \mathrm{Vars}(c)} \in \prod_{v \in \mathrm{Vars}(c)} C_v \sqcup [\sigma(\delta(v)) - |C_v|] \;\middle|\; e_u \neq e_w \text{ for all } (u,w) \in C \cap \mathrm{Vars}(c)^2 \right\} \right|$$

for any clause c. (Here, $[n] := \{1, 2, \ldots, n\}$ for any non-negative integer n.)

[6] Note that the literals of the clause have no effect on gr.

TODO: how does the algorithm prevent the number in $[\cdot]$ from being negative?

TODO: explain that x, y, z refer to vertices, c refers to a clause, and describe each vertex type in a bit more detail.

tautology $\mathrm{WMC}(\textcircled{\scriptsize T}; \sigma) = 1$

contradiction $\mathrm{WMC}(\textcircled{\scriptsize $\perp$} c; \sigma) = 0^{\mathrm{gr}(c;\sigma)}$

unit clause

$$\mathrm{WMC}(\textcircled{\scriptsize 1} c; \sigma) = \begin{cases} \mathrm{w}(p)^{\mathrm{gr}(c;\sigma)} & \text{if the literal is positive} \\ \overline{\mathrm{w}}(p)^{\mathrm{gr}(c;\sigma)} & \text{otherwise,} \end{cases}$$

where p is the predicate of the literal.

smoothing $\mathrm{WMC}(\bigcirc c; \sigma) = (\mathrm{w}(p) + \overline{\mathrm{w}}(p))^{\mathrm{gr}(c;\sigma)}$, where p is the predicate of the literal.

decomposable conjunction $\mathrm{WMC}(x \textcircled{\scriptsize $\wedge$} y; \sigma) = \mathrm{WMC}(x; \sigma) \times \mathrm{WMC}(y; \sigma)$

deterministic disjunction $\mathrm{WMC}(x \textcircled{\scriptsize $\vee$} y; \sigma) = \mathrm{WMC}(x; \sigma) + \mathrm{WMC}(y; \sigma)$

decomposable set-conjunction $\mathrm{WMC}(\textcircled{\scriptsize $\wedge$}_D x; \sigma) = \mathrm{WMC}(x; \sigma)^{\sigma(D)}$

deterministic set-disjunction $\mathrm{WMC}(\textcircled{\scriptsize $\vee$}_{D \subseteq S} x; \sigma) = \sum_{d=0}^{\sigma(S)} \binom{\sigma(S)}{d} \mathrm{WMC}(x; \sigma \cup \{ D \mapsto d \})$

inclusion-exclusion $\mathrm{WMC}(\mathrm{IE}(x, y, z); \sigma) = \mathrm{WMC}(x; \sigma) + \mathrm{WMC}(y; \sigma) - \mathrm{WMC}(z; \sigma)$

7.9 Examples of Newly Domain-Liftable Formulas

7.9.0.0.1 TODO.

- Describe Fig. 7.2 and connect it to the algebraic formula.
- Explain the algebraic notation that I'm using here (e.g., that f is always the main function)
- Explain the importance of comparing domain sizes to 2. FCGs that compare the size of a domain to an integer can be constructed automatically using compilation rules, although n is upper bounded by the maximum number of variables in any clause of the input formula since there is no rule that would introduce new variables.
- Mention that it only takes a few seconds to find these solutions. Going beyond depth 6 (or sometimes even completing depth 6) is computationally infeasible with the current implementation, but depth at most 5 can be searched within at most a few seconds.
- Combine the tikz and the algebraic notation into one, so I don't need to have two versions. But how? Maybe associate a symbol with each type and only to the types that I use?

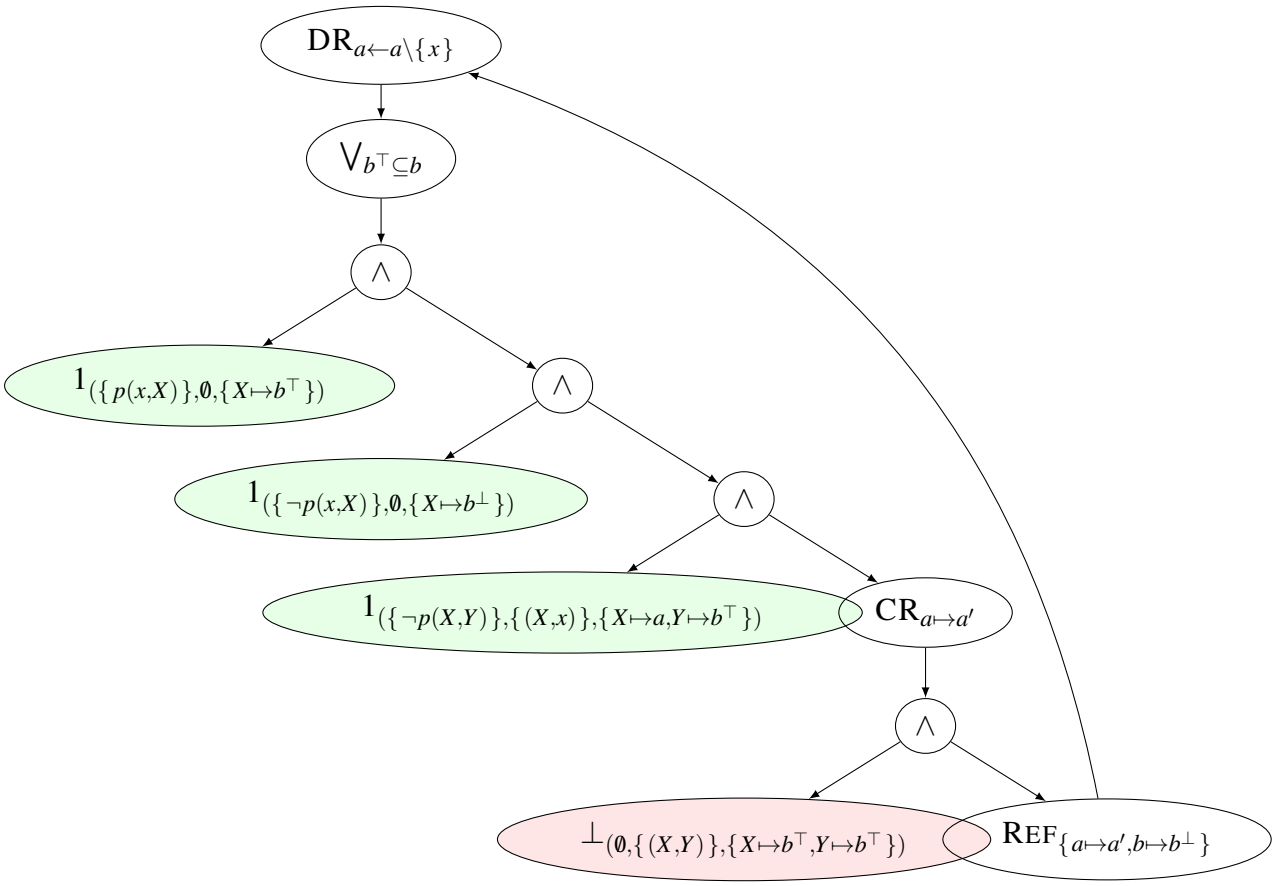


Figure 7.2: A graphical representation of an FCG for injections and partial injections between two domains. TODO: add a reference to a formula?

Function Class			Asymptotic Complexity of Counting		
Partial	Endo	Class	Best Known	With Circuits	With Graphs
✓/✗	✓/✗	Functions	$\log m$	m	m
✗	✗	Surjections	$n \log m$ Earnest [2018]	?	?
✗	✓		$n \log m$ Earnest [2018]	?	?
✓	✗		Same as injections from b to a		
✓	✓		Same as endo-injections		
✗	✗	Injections	$\log m$	-	mn
✗	✓		$\log m$	-	m^3
✓	✗		$\min\{m,n\}^2$	-	n^2
✓	✓		m^2	-	-
✗	✗	Bijections	$\log m$	-	m
✗	✓		Same as (partial) (endo-)injections		
✓	✓/✗				

Table 7.1: Here, m is the size of a, and n is the size of b. All asymptotic complexities are in $\Theta(\cdot)$. This is for unweighted counting. A hyphen means that no solution was found. Assuming all arithmetic operations to take constant time. Maybe a better solution could be found with more search. TODO: explain assumptions for the counts to not be zero. TODO: double check. TODO: maybe combine more rows/columns. TODO: transfer the citations to text.

- the exponential solutions can be computed in quadratic time with dynamic programming!
- Can't explain how formulas are translated into (my definition of) clauses without explaining Skolemization, which is out of scope.
- Note that in some cases different descriptions of the same problem lead to different solutions (with different complexities).
- Maybe we are actually guaranteed that the solution is always polynomial-time. Except... running time could be infinite?

Let p be a predicate of arity two s.t. the first argument is associated with domain a, and the second argument is associated with domain b (i.e., p represents a relation between sets a and b). Then, to restrict all relations representable by p to just functions from a to b, in first-order logic one might write

$$\forall X \in a. \forall Y \in b. \forall Z \in b. p(X,Y) \wedge p(X,Z) \implies Y = Z \tag{7.4}$$

$$\forall X \in a. \exists Y \in b. p(X,Y). \tag{7.5}$$

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

$$\forall W \in a. \forall X \in a. \forall Y \in b. p(W,Y) \wedge p(X,Y) \implies W = X \tag{7.6}$$

to restrict p to injections or

$$\forall Y \in b. \exists X \in a. p(X,Y) \tag{7.7}$$

to ensure surjectivity or remove Eq. (7.5) to consider partial functions. Lastly, one can replace all occurrences of b with a so as to model endofunctions instead.

7.9.0.0.2 Notes.

- FORCLIFT fails on all of these.
- Functions, surjections, and their partial counterparts are/were already liftable. It seems like lifting injectivity (which is a fairly general property) is the main accomplishment. (But this is just the one I noticed. There may be many others as well.)
- Here, $[\cdot]$ is the Iverson bracket.

7.9.0.0.3 Results.

- 1d bijections and 1d injections (note that it's the same problem). Depth 3 solution:

$$\begin{aligned} f(n) &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} g(n,m) \\ g(n,m) &= \sum_{l=0}^{n} \binom{n}{l} [l < 2] g(n-l, m-1) \\ &= g(n,m-1) + n g(n-1, m-1), \end{aligned}$$

  which works with base case $g(n,0) = 1$.

- 1d partial injections. 2 solutions at depth 6, but they're too complicated to check by hand. A contradiction with $X \neq x$ constraints makes things complicated.

- 2d bijections. Depth 3:

$$\begin{aligned} f(m,n) &= \sum_{l=0}^{m} \binom{m}{l} [l < 2](1 - [l < 1]) f(m-l, n-1) \\ &= m f(m-1, n-1), \end{aligned}$$

  which works with base cases $f(0,0) = 1$, $f(0,n) = 0$, $f(m,0) = 0$.

- 2d injections. Depth 2:

$$\begin{aligned} f(m,n) &= \sum_{l=0}^{m} \binom{m}{l} [l < 2] f(m-l, n-1) \\ &= f(m,n-1) + m f(m-1, n-1), \end{aligned}$$

  which works with base cases $f(0,0) = 1$ and $f(m,0) = 0$.

- 2d partial injections, depth 2. Exactly the same circuit as above but with base case $f(m,0) = 1$.

7.10 Discussion

- new rules that don't create vertices (e.g., duplicate removal, unconditional contradiction detection, etc.)
- some notes on halting
  - Search is infinite. Some rules increase the size of the formula(s), but most reduce it.
  - Inference is guaranteed to terminate if at least one domain shrinks by at least one. But note that allowing recursive calls with the same domain sizes (e.g., $f(n) = f(n) + \dots$) could be useful because these problematic terms might cancel out.
  - It's impossible for $n \leftarrow n - 1$ and `for` $n \in \dots$ to combine in a way that results in an infinite loop.
- care should be taken when cloning to preserve the validity of the cache and avoid infinite cycles (we use a separate (node $\rightarrow$ node) cache for this)

7.11 Conclusions and Future Work

7.11.0.0.1 Conclusions and observations.

- CR must be separate from DR because initially the requirement to not have the newly introduced constant in the literals is not satisfied.

7.11.0.0.2 Future work.

- Transform FCGs to definitions of (possibly recursive) functions on integers. Use a computer algebra system to simplify them.
- Design an algorithm to infer the necessary base cases. (Note that there can be an infinite amount of them when functions have more than one parameter.)
- Observation: -1 (and powers thereof) appear in every solution to a formula if and only if the formula has existential quantification. That's not very smart! By putting unit propagation into Γ, these powers are pushed to the outer layers of the solution (i.e., 'early' in the FCG). It's likely that removing this restriction would enable the algorithm to find asymptotically optimal solutions.

Chapter 8

Conclusion

8.1 Summary

8.2 Future Directions

Appendix A

Example Programs

In this appendix, we provide examples of probabilistic logic programs generated by various combinations of parameters. In all cases, we use

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

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)).
  ```
  ```

Appendix B

Proofs

Theorem 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

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

as required. □

Theorem 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 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. □

Lemma 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 $2\prod_{i=1}^{n}|\operatorname{im}Y_i|$ 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

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

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

Lemma 2. Let $\mathcal{V} = \{X_1, \ldots, X_n\}$. Then

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

for all $T \in 2^U$.

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

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

by Lemma 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 $\left(\sum_{i=1}^{m}[\lambda_{X=x_i}]\right)(T) = 0$, 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 $x_1, x_2 \in \operatorname{im}X$ 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 $\phi(T) = 0$. □

Theorem 4. For any $X \in \mathcal{V}$ and $x \in \operatorname{im}X$,

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

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

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

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 2;
- marginalisation of a probability distribution.

□

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.

G. Amendola, F. Ricca, and M. Truszczynski. Generating hard random Boolean formulas and disjunctive logic programs. In Sierra [2017], pages 532–538. ISBN 978-0-9992411-0-3. doi: 10.24963/ijcai.2017/75. URL `http://www.ijcai.org/Proceedings/2017/`.

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.

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, 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.

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.

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, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15008.

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, 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.

E. Boros and P. L. Hammer. Pseudo-Boolean optimization. *Discret. Appl. Math.*, 123 (1-3):155–225, 2002. doi: 10.1016/S0166-218X(01)00341-9.

R. J. Brachman and H. J. Levesque. *Knowledge Representation and Reasoning*. Elsevier, 2004. ISBN 978-1-55860-932-7. URL http://www.elsevier.com/wps/find/bookdescription.cws_home/702602/description.

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. ISBN 978-1-60750-605-8. 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.

S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In C. E. Brodley and P. Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 1722–1730. AAAI Press, 2014. ISBN 978-1-57735-661-5. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8364.

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.

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.

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, 2001. 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. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009. ISBN 978-0-521-88438-9. URL `http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521884389`.

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.

L. De Raedt. *Logical and relational learning*. Cognitive Technologies. Springer, 2008. ISBN 978-3-540-20040-6. doi: 10.1007/978-3-540-68856-3.

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.

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. de Salvo Braz, E. Amir, and D. Roth. A survey of first-order probabilistic models. In D. E. Holmes and L. C. Jain, editors, *Innovations in Bayesian Networks: Theory and Applications*, volume 156 of *Studies in Computational Intelligence*, pages 289–317. Springer, 2008. doi: 10.1007/978-3-540-85066-3_12.

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.

P. Dilkas and V. Belle. Weighted model counting without parameter variables. In C. M. Li and F. Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, Lecture Notes in Computer Science. Springer, 2021.

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.

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).

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. ISBN 978-3-642-23785-0. doi: 10.1007/978-3-642-23786-7_22.

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. ISBN 978-0-9749039-7-2. URL https://dslpitt.org/uai/displayArticles.jsp?mmnu=1&smnu=1&proceeding_id=27.

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.

H. Gaifman. Concerning measures on Boolean algebras. *Pacific Journal of Mathematics*, 14(1):61–73, 1964.

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. URL `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24`.

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.

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. 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. URL https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=320&proceeding_id=13.

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.

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, 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.

D. Kisa, G. Van den Broeck, A. Choi, and A. Darwiche. Probabilistic sentential decision diagrams. In C. Baral, G. De Giacomo, and T. Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*. AAAI Press, 2014. URL http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/8005.

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.

J. Lagniez and P. Marquis. An improved decision-dnnf compiler. In Sierra [2017], pages 667–673. ISBN 978-0-9992411-0-3. doi: 10.24963/ijcai.2017/93. URL http://www.ijcai.org/Proceedings/2017/.

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. ISBN 978-3-319-18007-6. doi: 10.1007/978-3-319-18008-3_19.

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. ISBN 978-3-319-51675-2. 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. ISBN 978-3-319-07045-2. 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`.

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. ISBN 978-3-642-02845-8. 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. ISBN 978-3-642-04237-9. doi: 10.1007/978-3-642-04238-6_20.

U. Nilsson and J. Maluszynski. *Logic, programming and Prolog*. Wiley, 1990. ISBN 978-0-471-92625-2.

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.

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.

H. Poon and P. M. Domingos. Sum-product networks: A new deep architecture. 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 337–346. AUAI Press, 2011.

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`.

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. 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. J. Russell. Unifying logic and probability. *Commun. ACM*, 58(7):88–97, 2015. doi: 10.1145/2699411.

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, 2005. URL `http://www.aaai.org/Library/AAAI/2005/aaai05-075.php`.

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.

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.

C. Sierra, editor. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017. ijcai.org. ISBN 978-0-9992411-0-3. URL http://www.ijcai.org/Proceedings/2017/.

F. Somenzi. CUDD: CU decision diagram package release 3.0.0. *University of Colorado at Boulder*, 2015.

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.

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.

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. *Lifted Inference and Learning in Statistical Relational Models (Eerste-orde inferentie en leren in statistische relationele modellen)*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2013. URL https://lirias.kuleuven.be/handle/123456789/373041.

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.

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.

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. ISBN 978-1-57735-738-4.

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. ISBN 978-1-61499-329-2. 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. ISBN 3-540-46267-8. 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.

W. Wei and B. Selman. A new approach to 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 324–339. Springer, 2005. ISBN 3-540-26276-8. doi: 10.1007/11499107_24.

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.

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. ISBN 3-540-20642-6. doi: 10.1007/978-3-540-24599-5_17.