Probabilistic Inference in Graphical and Relational Models

Paulius Dilkas

Supervisors: Mr Vaishak Belle and Dr Ron Petrick

School of Informatics, University of Edinburgh

15th April 2021

1 Introduction

2 Background

2.1 Probabilities

2.2 Representations

RDDL, probabilistic programming, ProbLog, MLNs, Markov networks, BNs, relational BNs.

2.3 Applications

Chapter 1 of SRAI: applications of ProbLog, BNs, relational BNs

2.4 Learning

program induction, probabilistic program induction, structure learning for MLNs.

2.5 Inference

This is where my contribution lies. WMC, WMI, WFOMC, some kind of conditioning algorithms.

3 Progress To Date

WP 1 ('On the Equivalence of Constants in Relational Knowledge Bases') was abandoned. While trying to take reviewer feedback into consideration as well as update and strengthen the paper, I found an important ambiguity: when defining what constants are 'captured' and 'transferred' by a clause, I fail to specify whether each relevant 'spot' is occupied by a constant or a variable. In the former case, that makes the main theorem of the paper completely trivial. While in the latter case, the theorem becomes incorrect. I spent a couple of weeks looking for ways to transform the paper into something both correct and valuable. The best idea I could find was to use the perspective from this paper in the context of inductive logic programming; however, I did not want to explore this direction further.

WP 2 ('Generating Random Logic Programs Using Constraint Programming') was revised (see Appendix A) and published and presented in CP 2020. I also gave three more talks about it:

- at the local AIAI seminar,

- for the Formal Analysis, Theory and Algorithms research section at the University of Glasgow,
- and at the FMAI 2021 workshop.

WP 3 morphed into Appendix B and was submitted to AAAI 2021 and UAI 2021. I was also invited to the program committee for UAI 2021.

WP 4 was abandoned due to lack of contributions that could be made.

A new paper was written and submitted to SAT (see Appendix C). It is based on **WP 3**, addresses the same issue, and suggests an improved solution.

4 Future Goals

- Parameterized Complexity of Weighted Model Counting in Theory and Practice
- Weighted Pseudo-Boolean Model Counting (if I can find a good application)

References

A Published Paper (CP 2020)

Generating Random Logic Programs Using Constraint Programming

Paulius Dilkas[1] and Vaishak Belle[1,2(✉)]

[1] University of Edinburgh, Edinburgh, UK
p.dilkas@sms.ed.ac.uk, vaishak@ed.ac.uk
[2] Alan Turing Institute, London, UK

Abstract. 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.

Keywords: Constraint programming · Probabilistic logic programming · Statistical relational learning

1 Introduction

Unifying logic and probability is a long-standing challenge in artificial intelligence [24], 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 [25], ICL [22], and ProbLog [11]—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 [5, 16, 28], 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 [26]. 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 [1, 2, 30, 32], often with severely limiting conditions such as two-literal clauses [20, 21] or clauses of the form $a \leftarrow \neg b$ [31].

In this work (Sects. 3 to 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 [12]. In fact, the same model can generate both probabilistic programs in the syntax of ProbLog [11] and non-probabilistic Prolog programs. To the best of our knowledge, this is the first work that (a) addresses the problem of generating random logic programs in its full generality (i.e., including first-order clauses with variables), and (b) 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. 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. 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. 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 [9]. However, since probabilistic logic programming languages are closely related to other areas of machine learning, including (imperative) probabilistic programming [10], our results can lay the foundations for exploring broader questions on generating models and testing algorithms in machine learning.

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 ex-

pression 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[3]). A *(logic) program* is a set of clauses, and a ProbLog *program* is a set of clause-probability pairs [14].

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 one value. We let $\square$ denote the absent/disabled value of an optional variable [19]. We write $\texttt{a}[b] \in c$ to mean that $\texttt{a}$ is an array of variables of length b such that each element of $\texttt{a}$ has domain c. Similarly, we write $\texttt{c} : \texttt{a}[b]$ to denote an array $\texttt{a}$ of length b such that each element of $\texttt{a}$ has type $\texttt{c}$. Finally, we assume that all arrays start with index zero.

Parameters of the model. We begin by defining sets and lists of the primitives used in constructing logic programs: a list of predicates $\mathcal{P}$, a list of their corresponding arities $\mathcal{A}$ (so $|\mathcal{A}| = |\mathcal{P}|$), a set of variables $\mathcal{V}$, and a set of constants $\mathcal{C}$. Either $\mathcal{V}$ or $\mathcal{C}$ can be empty, but we assume that $|\mathcal{C}| + |\mathcal{V}| > 0$. Similarly, the model supports zero-arity predicates but requires at least one predicate to have non-zero arity. For notational convenience, we also set $\mathcal{M}_\mathcal{A} = \max \mathcal{A}$. Next, we need a measure of how complex a body of a clause can be. As we represent each body by a tree (see Sect. 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. 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 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

[3] 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.

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

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

Constraint 2. *Let* $P = \{h.\texttt{predicate} \mid h \in \texttt{heads}\}$ *be a multiset. Then*

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

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

Finally, we want to disable duplicate clauses but with one exception: there may be more than one disabled clause, i.e., a clause with head $\texttt{predicate} = \square$. Assuming a lexicographic order over entire clauses such that $\square > \mathsf{P}$ 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 3. *For* $i = 1, \ldots, \mathcal{M}_\mathcal{C} - 1$, *if* $\texttt{heads}[i].\texttt{predicate} \neq \square$, *then*

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

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

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

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. Furthermore, we can use Constraint 1 to again disable the extra arguments.

Example 2. Let $\mathcal{M}_\mathcal{A} = 2$, $X \in \mathcal{V}$, and let P be a predicate with arity 1. Then the node representing atom $\mathsf{P}(X)$ 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 4. *For* $i = 1, \ldots, \mathcal{M}_\mathcal{N} - 1$*, if* $i < \texttt{numNodes}$*, then*

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

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

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

Constraint 5. *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*

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

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

Constraint 6. *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 7. *For* $i = 0, \ldots, \mathcal{M}_\mathcal{C} - 1$*, if* $\texttt{heads}[i].\texttt{predicate} = \square$*, then*

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

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

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

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

Definition 2. *Let* $\texttt{intros}[|\mathcal{V}|] \in \{0, \ldots, N\}$ *be such that for* $v \in \mathcal{V}$*,*

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

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

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

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

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

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

Constraint 8. *For* $u \neq v \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$, $\texttt{occ}[u] \cap \texttt{occ}[v] = \emptyset$.

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

Constraint 9. *For* $v \in \mathcal{V}$, $\texttt{intros}[v] \neq 0 \iff \texttt{intros}[v] - 1 \in \texttt{occ}[v]$.

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

Constraint 10. *For* $i = 0, \ldots, \mathcal{M}_\mathcal{N} - 1$, *let*

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

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

6 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 $n > 1$:

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

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

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

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:

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

Here, we sum over all ways to distribute $|\mathcal{P}| \leq n \leq \mathcal{M}_\mathcal{C}$ 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 (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.

7 Stratification and Independence

Stratification is a condition necessary for probabilistic logic programs [18] and often enforced on logic programs [4] 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 $j \leq i$ [4].

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.[4] 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 4. Consider the following (fragment of a) program:

$$\begin{aligned} \mathsf{sibling}(X, Y) &\leftarrow \mathsf{parent}(X, Z) \wedge \mathsf{parent}(Y, Z), \\ \mathsf{father}(X, Y) &\leftarrow \mathsf{parent}(X, Y) \wedge \neg\mathsf{mother}(X, Y). \end{aligned} \tag{2}$$

Its predicate dependency graph is in Fig. 1. Because of the negation in (2), the edge from mother to father is negative, while the other two edges are positive. The dependencies of each predicate are:

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

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

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

[4] This definition is an extension of a well-known result for logic programs [3] to probabilistic logic programs with arbitrary complex clause bodies.

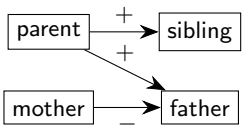


Fig. 1. The predicate dependency graph of the program from Example 4. Positive edges are labelled with '+', and negative edges with '−'.

Table 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	$\Upsilon(p)$

Algorithm 1: Entailment for independence

Data: predicates p_1, p_2
$D \leftarrow \{(d_1, d_2) \in \mathtt{deps}(p_1,\ 1) \times \mathtt{deps}(p_2,\ 1) \mid d_1.\mathsf{predicate} = d_2.\mathsf{predicate}\}$;
if $D = \emptyset$ **then return** TRUE;
if $\exists(\Delta\ \_, \Delta\ \_) \in D$ **then return** FALSE **else return** UNDEFINED;

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 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 $d.\mathsf{predicate}$. 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 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: $\mathtt{deps}(p, 1)$ returns all dependencies, while $\mathtt{deps}(p, 0)$ returns only determined and almost-determined dependencies. It starts by establishing the predicate p itself as a dependency and continues to add dependencies of depen-

Algorithm 2: Propagation for independence

```
Data: predicates p_1, p_2; adjacency matrix A
for (d_1, d_2) ∈ deps(p_1, 0) × deps(p_2, 0) s.t. d_1.predicate = d_2.predicate do
    if d_1 is Δ(_) and d_2 is Δ(_) then fail();
    if {d_1, d_2} = {Δ(_), Γ(_, s, t)} then A[s][t].removeValue(1);
```

Algorithm 3: Dependencies of a predicate

```
Data: adjacency matrix A
Function deps(p, allDeps):
    D ← {Δ(p)};
    while true do
        D' ← ∅;
        for d ∈ D and q ∈ 𝒫 do
            edge ← A[q][d.predicate] = {1};
            if edge and d is Δ(_) then D' ← D' ∪ {Δ(q)};
            else if edge and d is Γ(_, s, t) then D' ← D' ∪ {Γ(q, s, t)};
            else if |A[q][d.predicate]| > 1 and d is Δ(r) then
                D' ← D' ∪ {Γ(q, q, r)};
            else if |A[q][d.predicate]| > 1 and allDeps then D' ← D' ∪ {ϒ(q)};
        if D' = D then return D else D ← D';
```

dencies 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 $d.\mathsf{predicate}$ 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 $d.\mathsf{predicate}$ 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 5. Consider this partially determined (fragment of a) program:

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

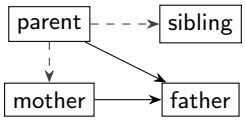
where $\square$ indicates an unknown predicate with domain

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

The predicate dependency graph is pictured in Fig. 2. Suppose we have a constraint that mother and parent must be independent. The lists of potential de-

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

(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.

Fig. 2. The predicate dependency graph of Example 5

pendencies for both predicates are:

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

An entailment check at this stage would produce UNDEFINED, but propagation replaces the boxed value in Fig. 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.

8 Experimental Results

We now present the results of two experiments: in Sect. 8.1 we examine the scalability of our constraint model with respect to its parameters and in Sect. 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 [23]. 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.

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 (a) the `structure` array, (b) body predicates, (c) head arguments, (d) (if $|\mathcal{V}| > 1$) the `intros` array, (e) body arguments. For instance, in the clause from Example 3, all visible parts of the clause would be decided in this order:

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

We also employ a geometric restart policy, restarting after $10, 10 \times 1.1, 10 \times 1.1^2, \ldots$ contradictions.[5] We ran 399 360 experiments, investigating the model's efficiency and gaining insight into what parameter values make the CSP harder. For $|\mathcal{P}|$, $|\mathcal{V}|$, $|\mathcal{C}|$, $\mathcal{M}_\mathcal{N}$, and $\mathcal{M}_\mathcal{C} - |\mathcal{P}|$ (i.e., the number of clauses in addition to the mandatory $|\mathcal{P}|$ clauses), we assign all combinations of 1, 2, 4, 8. $\mathcal{M}_\mathcal{A}$ is assigned to values 1–4. For each $|\mathcal{P}|$, we also iterate over all possible numbers of independent pairs of predicates, ranging from 0 up to $\binom{|\mathcal{P}|}{2}$. For each combination of the above-mentioned parameters, we pick ten random ways to assign arities to predicates (such that $\mathcal{M}_\mathcal{A}$ occurs at least once) and ten random combinations of independent pairs.

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

In Fig. 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.

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 [14]. 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

[5] 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.

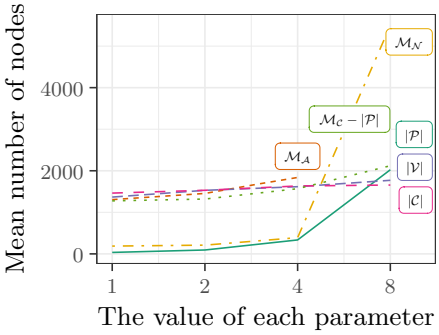


Fig. 3. The mean number of nodes in the binary search tree for each value of each experimental parameter. Note that the horizontal axis is on a $\log_2$ scale.

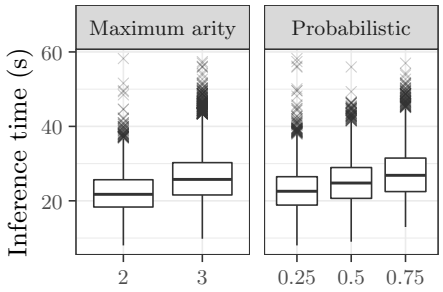


Fig. 4. Inference time for different values of $\mathcal{M}_{\mathcal{A}}$ and proportions of probabilistic facts that are probabilistic. The total number of facts is fixed at 10^5.

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) [6], negation normal form (NNF), deterministic decomposable NNF (d-DNNF) [8], K-Best [11], and two encodings based on sentential decision diagrams [7], one of which encodes the entire program (SDDX), while the other one encodes only the part of the program relevant to the query (SDD).[6]

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. 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

[6] Forward SDDs (FSDDs) and forward BDDs (FBDDs) [27, 28] are omitted because the former uses too much memory and the implementation of the latter seems to be broken at the time of writing.

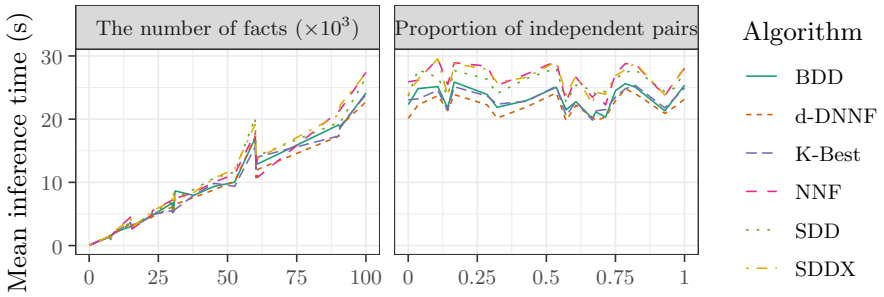


Fig. 5. Mean inference time for a range of ProbLog inference algorithms as a function of the total number of facts in the program and the proportion of independent pairs of predicates. For the second plot, the number of facts is fixed at 10^5.

when facts are mostly probabilistic (see Fig. 4). Finally, according to Fig. 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 [15]—do not exploit situations where separate parts of a program can be handled independently.

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 experimental evaluation in Sect. 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.

Acknowledgments. Paulius 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/S023208/1). Vaishak was supported by a Royal Society University Research Fellowship.

References

1. Amendola, G., Ricca, F., Truszczynski, M.: Generating hard random Boolean formulas and disjunctive logic programs. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 532–538. ijcai.org (2017). https://doi.org/10.24963/ijcai.2017/75, http://www.ijcai.org/Proceedings/2017/
2. Amendola, G., Ricca, F., Truszczynski, M.: New models for generating hard random Boolean formulas and disjunctive logic programs. Artif. Intell. **279** (2020). https://doi.org/10.1016/j.artint.2019.103185
3. Balbin, I., Port, G.S., Ramamohanarao, K., Meenakshi, K.: Efficient bottom-up computation of queries on stratified databases. J. Log. Program. **11**(3&4), 295–344 (1991). https://doi.org/10.1016/0743-1066(91)90030-S
4. Bidoit, N.: Negation in rule-based database languages: A survey. Theor. Comput. Sci. **78**(1), 3–83 (1991). https://doi.org/10.1016/0304-3975(51)90003-5
5. Bruynooghe, M., Mantadelis, T., Kimmig, A., Gutmann, B., Vennekens, J., Janssens, G., De Raedt, L.: ProbLog technology for inference in a probabilistic first order logic. In: Coelho, H., Studer, R., Wooldridge, M.J. (eds.) ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 719–724. IOS Press (2010). https://doi.org/10.3233/978-1-60750-606-5-719
6. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
7. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: Walsh, T. (ed.) IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011. pp. 819–826. IJCAI/AAAI (2011). https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-143, http://ijcai.org/proceedings/2011
8. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002). https://doi.org/10.1613/jair.989
9. De Raedt, L., Kersting, K., Natarajan, S., Poole, D.: Statistical Relational Artificial Intelligence: Logic, Probability, and Computation. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2016). https://doi.org/10.2200/S00692ED1V01Y201601AIM032
10. De Raedt, L., Kimmig, A.: Probabilistic (logic) programming concepts. Machine Learning **100**(1), 5–47 (2015). https://doi.org/10.1007/s10994-015-5494-z
11. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 2462–2467 (2007)
12. Dechter, R., Kask, K., Bin, E., Emek, R.: Generating random solutions for constraint satisfaction problems. In: Dechter, R., Kearns, M.J., Sutton, R.S. (eds.) Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada. pp. 15–21. AAAI Press / The MIT Press (2002), http://www.aaai.org/Library/AAAI/2002/aaai02-003.php
13. Fages, J., Lorca, X.: Revisiting the tree constraint. In: Lee, J.H. (ed.) Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings.

Lecture Notes in Computer Science, vol. 6876, pp. 271–285. Springer (2011). https://doi.org/10.1007/978-3-642-23786-7_22
14. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory Pract. Log. Program. **15**(3), 358–401 (2015). https://doi.org/10.1017/S1471068414000076
15. Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., De Raedt, L.: Inference in probabilistic logic programs using weighted CNF's. In: Cozman, F.G., Pfeffer, A. (eds.) UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011. pp. 211–220. AUAI Press (2011), https://dslpitt.org/uai/displayArticles.jsp?mmnu=1&smnu=1&proceeding_id=27
16. Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. TPLP **11**(2-3), 235–262 (2011). https://doi.org/10.1017/S1471068410000566
17. Mairy, J., Deville, Y., Lecoutre, C.: The smart table constraint. In: Michel, L. (ed.) Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9075, pp. 271–287. Springer (2015). https://doi.org/10.1007/978-3-319-18008-3_19
18. Mantadelis, T., Rocha, R.: Using iterative deepening for probabilistic logic inference. In: Lierler, Y., Taha, W. (eds.) Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10137, pp. 198–213. Springer (2017). https://doi.org/10.1007/978-3-319-51676-9_14
19. Mears, C., Schutt, A., Stuckey, P.J., Tack, G., Marriott, K., Wallace, M.: Modelling with option types in MiniZinc. In: Simonis, H. (ed.) Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8451, pp. 88–103. Springer (2014). https://doi.org/10.1007/978-3-319-07046-9_7
20. Namasivayam, G.: Study of random logic programs. In: Hill, P.M., Warren, D.S. (eds.) Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5649, pp. 555–556. Springer (2009). https://doi.org/10.1007/978-3-642-02846-5_61
21. Namasivayam, G., Truszczynski, M.: Simple random logic programs. In: Erdem, E., Lin, F., Schaub, T. (eds.) Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5753, pp. 223–235. Springer (2009). https://doi.org/10.1007/978-3-642-04238-6_20
22. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artif. Intell. **94**(1-2), 7–56 (1997). https://doi.org/10.1016/S0004-3702(97)00027-1
23. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017), http://www.choco-solver.org
24. Russell, S.J.: Unifying logic and probability. Commun. ACM **58**(7), 88–97 (2015). https://doi.org/10.1145/2699411
25. Sato, T., Kameya, Y.: PRISM: A language for symbolic-statistical modeling. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelli-

gence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes. pp. 1330–1339. Morgan Kaufmann (1997)
26. Selman, B., Mitchell, D.G., Levesque, H.J.: Generating hard satisfiability problems. Artif. Intell. **81**(1-2), 17–29 (1996). https://doi.org/10.1016/0004-3702(95)00045-3
27. Tsamoura, E., Gutiérrez-Basulto, V., Kimmig, A.: 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. pp. 10284–10291. AAAI Press (2020), https://aaai.org/ojs/index.php/AAAI/article/view/6591
28. Vlasselaer, J., Van den Broeck, G., Kimmig, A., Meert, W., De Raedt, L.: Anytime inference in probabilistic logic programs with Tp-compilation. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 1852–1858. AAAI Press (2015)
29. Walsh, T.: General symmetry breaking constraints. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4204, pp. 650–664. Springer (2006). https://doi.org/10.1007/11889205_46
30. Wang, K., Wen, L., Mu, K.: Random logic programs: Linear model. TPLP **15**(6), 818–853 (2015). https://doi.org/10.1017/S1471068414000611
31. Wen, L., Wang, K., Shen, Y., Lin, F.: A model for phase transition of random answer-set programs. ACM Trans. Comput. Log. **17**(3), 22:1–22:34 (2016). https://doi.org/10.1145/2926791
32. Zhao, Y., Lin, F.: Answer set programming phase transition: A study on randomly generated programs. In: Palamidessi, C. (ed.) Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2916, pp. 239–253. Springer (2003). https://doi.org/10.1007/978-3-540-24599-5_17

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)).
  ```
  ```

B Submitted Paper (AAAI 2021 and UAI 2021)

Weighted Model Counting with Conditional Weights for Bayesian Networks

Abstract

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 encoding significantly improves the performance of the algorithm on most benchmark instances. Furthermore, the same idea can be adapted to other WMC algorithms and other problem domains.

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., 2020]. 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 Section 2, in Section 3 we describe three equivalent perspectives on the subject based on logic, set theory, and Boolean algebras. Furthermore, we describe the space of functions on Boolean algebras and various operations on those functions. Section 4 introduces WMC as the problem of computing the value of a measure on a Boolean algebra. We show that not all measures can be represented using literal-based WMC, but all Boolean algebras can be extended to make any measure representable in such a manner.

This new perspective allows us to not only encode any discrete probability distribution but also improve inference speed. In Section 5 we demonstrate this by developing a new WMC encoding for Bayesian networks that uses *conditional*

Submitted to the 37th Conference on Uncertainty in Artificial Intelligence (UAI 2021). **To be used for reviewing only**.

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 Section 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.

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, Bart et al. [2016] introduce `bklm16` and show that it has both fewer variables and fewer clauses than `cd06`. Their experiments show `bklm16` to be superior to `cd06` with respect to both compilation time and encoding size when both are compiled using c2d[2] [Darwiche, 2004] but inferior to `cd06` when `cd06` is compiled using Ace[3] (which still uses c2d but considers the structure of the Bayesian network along with its encoding). Our experiments in Section 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.

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

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 $\mathscr{L}$ 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 $\mathscr{L}$, 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 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 $\mathscr{L}$ 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 $\mathscr{L}$, i.e., an element of 2^U. Unless referring specifically to a logic, we will use the algebraic definition of an atom and refer to logical atoms as *variables*. In the rest of the paper, for any set U, we will use set-theoretic notation for 2^U and Boolean-algebraic notation for 2^{2^U}, except for (Boolean) atoms in 2^{2^U} that are denoted as $\{x\}$ for some model $x \in 2^U$.

3.1 FUNCTIONS ON BOOLEAN ALGEBRAS

We also consider the space of all functions from any Boolean algebra to $\mathbb{R}_{\geq 0}$ together with some operations on those functions. They will be instrumental in defining WMC as a measure in Section 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 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})	$\top$	$\{\emptyset, \{a\}, \{b\}, \{a,b\}\}$
	$\neg a \vee \neg b, a \rightarrow b$	$\{\emptyset, \{a\}, \{b\}\}, \{\emptyset, \{a\}, \{a,b\}\}$
	$b \rightarrow a, a \vee b$	$\{\emptyset, \{b\}, \{a,b\}\}, \{\{a\}, \{b\}, \{a,b\}\}$
	$\neg b, \neg a, a \leftrightarrow b$	$\{\emptyset, \{a\}\}, \{\emptyset, \{b\}\}, \{\emptyset, \{a,b\}\}$
	$(a \wedge \neg b) \vee (b \wedge \neg a), a, b$	$\{\{a\}, \{b\}\}, \{\{a\}, \{a,b\}\}, \{\{b\}, \{a,b\}\}$
	$\neg a \wedge \neg b, a \wedge \neg b, \neg a \wedge b, a \wedge b$	$\{\emptyset\}, \{\{a\}\}, \{\{b\}\}, \{\{a,b\}\}$
	$\bot$	$\emptyset$

Definition 1 (Operations on functions)**.** Let $\alpha\colon 2^X \to \mathbb{R}_{\geq 0}$ and $\beta\colon 2^Y \to \mathbb{R}_{\geq 0}$ be functions, $p \in \mathbb{R}_{\geq 0}$, and $x \in X$. We define the following operations:

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

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

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

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

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

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

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.

Relation to the classical (logic-based) view of WMC. Let $\mathscr{L}$ be a propositional logic with two atoms a and b as in Section 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 $\mathscr{L}$ with a sole axiom a. Then Δ has two models: $\{a, b\}$ and $\{a, \neg b\}$ and its WMC [Chavira and Darwiche, 2008] is

$$\begin{aligned} \mathrm{WMC}(\Delta) &= \sum_{\omega \models \Delta} \prod_{\omega \models l} w(l) \\ &= w(a)w(b) + w(a)w(\neg b) = 0.3. \end{aligned} \tag{1}$$

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. (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.

4.1 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 1. 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 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 $x \in 2^{2^U}$.*

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 $\operatorname{im} X$ denote its set of values and $\operatorname{pa}(X)$ its set of parents. The full probability distribution is then equal to $\prod_{X \in \mathcal{V}} \Pr(X \mid \operatorname{pa}(X))$. 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. 1 for an example Bayesian network that we will refer to throughout this section. For this network, $\mathcal{V} = \{W, F, T\}$, $\operatorname{pa}(W) = \emptyset$, $\operatorname{pa}(F) = \operatorname{pa}(T) = \{W\}$, $\operatorname{im} W = \operatorname{im} F = \{0, 1\}$, and $\operatorname{im} T = \{l, m, h\}$.

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

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

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

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

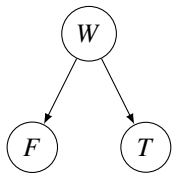
Algorithm 1 shows how a Bayesian network with vertices $\mathcal{V}$ can be represented as a weight function $\phi\colon 2^U \to \mathbb{R}_{\geq 0}$. The algorithm begins with the unit function and multiplies it by $\mathrm{CPT}_X\colon 2^{\mathscr{E}^*(X)} \to \mathbb{R}_{\geq 0}$ for each random variable $X \in \mathcal{V}$. 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 $|\mathscr{E}(X) \cap T| \neq 1$. For the Bayesian network in Fig. 1, we get:

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

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

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



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 1: An example Bayesian network with its CPTs.

Algorithm 1: Encoding a Bayesian network.

Data: vertices $\mathscr{V}$, probability distribution Pr
Result: $\phi\colon 2^U \to \mathbb{R}_{\geq 0}$
$\phi \leftarrow 1$;
for $X \in \mathscr{V}$ **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;

5.1 CORRECTNESS

Algorithm 1 produces a function with a Boolean algebra as its domain. This function can be represented by an ADD [Bahar et al., 1997]. ADDMC takes an ADD $\psi\colon 2^U \to \mathbb{R}_{\geq 0}$ (expressed as a product of smaller ADDs) and returns $(\exists_U \psi)(\emptyset)$ [Dudek et al., 2020a]. In this section, we prove that the function ϕ produced by Algorithm 1 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 $X \in \mathscr{V}$ be a random variable with parents $\mathrm{pa}(X) = \{Y_1, \ldots, Y_n\}$. Then $\mathrm{CPT}_X\colon 2^{\mathscr{E}^*(X)} \to \mathbb{R}_{\geq 0}$ is such that for any $x \in \operatorname{im} X$ and $(y_1, \ldots, y_n) \in \prod_{i=1}^n \operatorname{im} Y_i$,*

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

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 $\mathscr{V} = \{X_1, \ldots, X_n\}$. Then*

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

for all $T \in 2^U$.

We end with Theorem 4 that shows how ϕ can be combined with an encoding of a single variable-value assignment so that ADDMC would compute its marginal probability.

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

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

5.2 TEXTUAL REPRESENTATION

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

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

[Sang et al., 2004]. For instance, here is a representation of the Bayesian network from Fig. 1:

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

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.

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.

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 $\mathscr{E}^*(X)$. Second, even though the function ϕ produced by Algorithm 1 is constructed to have 2^U as its domain, sometimes the domain is effectively reduced to 2^V for some $V \subset U$ by the ADD manipulation algorithms that optimise the ADD representation of a function. For a simple example, consider $\alpha : 2^{\{a\}} \to \mathbb{R}_{\geq 0}$ defined as $\alpha(\{a\}) = \alpha(\emptyset) = 0.5$. Then α can be reduced to $\alpha'\colon 2^{\emptyset} \to \mathbb{R}_{\geq 0}$ defined as $\alpha'(\emptyset) = 0.5$. To compensate for these reductions, for the original WMC format with a weight function $w\colon U \cup \{\neg u \mid u \in U\} \to \mathbb{R}_{\geq 0}$, ADDMC would multiply its computed answer by $\prod_{u \in U \setminus V} w(u) + w(\neg u)$. With the new WMC format, we instead multiply the answer by $2^{|U \setminus V|}$. Each 'excluded' variable $u \in U \setminus V$ satisfies two properties: all weights associated with u are equal to 0.5 (otherwise the corresponding CPT would depend on u, and u would not be excluded), and all other CPTs are independent of u (or they may have a trivial dependence, where the probability stays the same if u is replaced with its complement). Thus, the CPT that corresponds to u still multiplies the weight of every atom in the Boolean algebra by 0.5, but the number of atoms under consideration is halved. To correct for this, we multiply the final answer by two for every $u \in U \setminus V$.

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

Table 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

6 EXPERIMENTAL COMPARISON

We compare the six WMC encodings for Bayesian networks when run with both ADDMC 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 lim-

[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].

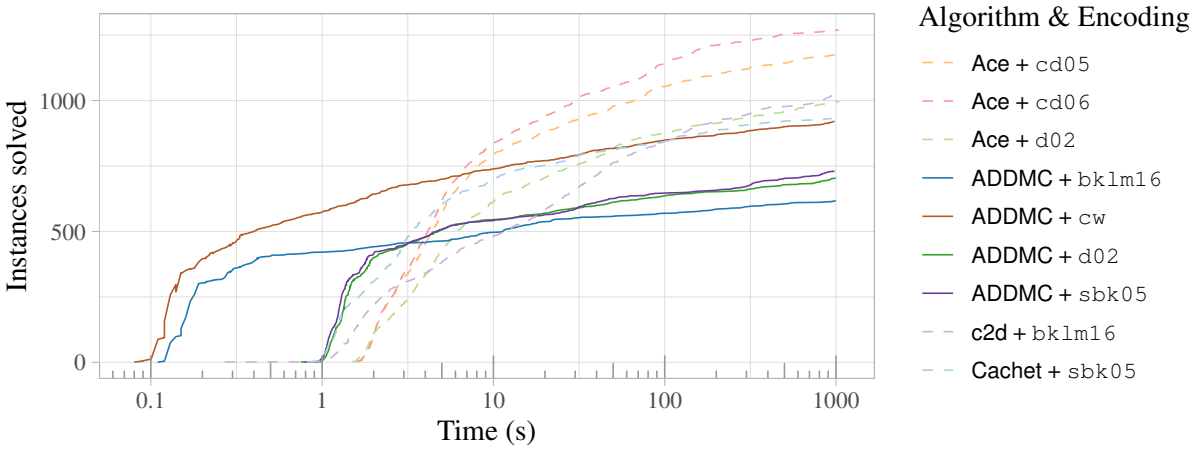


Figure 2: Cumulative numbers of instances solved by combinations of algorithms and encodings over time.

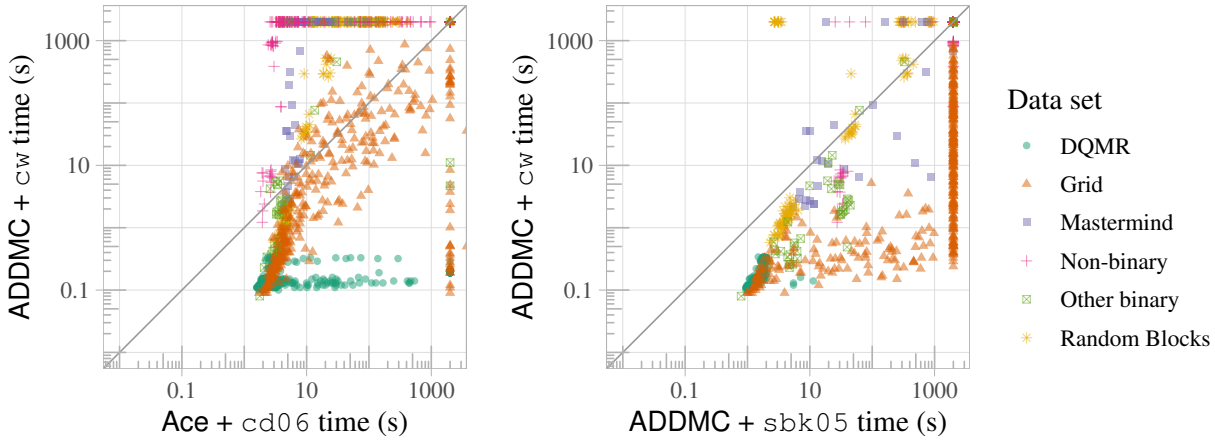


Figure 3: An instance-by-instance comparison between ADDMC + cw and the best overall combination of algorithm and encoding (Ace + cd06, on the left) as well as the second-best encoding for ADDMC (sbk05, on the right).

its. The experiments were run on a computing cluster with Intel Xeon Gold 6138 and Intel Xeon E5-2630 processors[9] running Scientific Linux 7 with a 32 GiB memory limit and a 1000 s timeout on both encoding and inference. For inference, we use Ace for cd05, cd06, and d02; Cachet[10] [Sang et al., 2004] for sbk05; and c2d [Darwiche, 2004] for compilation and query-dnnf [11] for answer computation for bklm16. 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 (whether run with ADDMC or c2d) returns correct answers on most instances, we had to disable one of the improvements that bklm16 brings over cd06, 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

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

[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 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`	$O(nv^{d+1})$	$O(ndv^{d+1})$

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, 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`). We run ADDMC with each of the five encodings once on each Bayesian network.

Figure 2 shows that `cd05` and `cd06` (when run with Ace) are in the lead, while ADDMC 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 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. 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. 3 shows that `cw` is better than `sbk05` (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 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 $n = |\mathscr{V}|$ be the number of vertices in the Bayesian network, $d = \max_{X \in \mathscr{V}} |\mathrm{pa}(X)|$ the maximum in-degree (i.e., the number of parents), and $v = \max_{X \in \mathscr{V}} |\mathrm{im}X|$ the maximum number of values per variable. Table 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 1) here because ADDMC interprets each clause of any WMC encoding as a multiplicative factor of the ADD that represents the entire WMC instance [Dudek et al., 2020a]. For literal-weight encodings, each weight is also a factor, but that does not affect our asymptotic bounds.

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 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. The same idea can be adapted to other inference problem domains such as probabilistic programs [Fierens et al., 2015, Holtzen et al., 2020] 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]. Another important direction for future work is to develop a better understanding of what properties of Bayesian networks make an instance easy for some algorithm-encoding combinations more than others.

References

R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods Syst. Des.*, 10(2/3):171–206, 1997. doi: 10.1023/A:1008699807402.

Anicet Bart, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. An improved CNF encoding scheme for probabilistic inference. In Gal A. Kaminka, Maria Fox, Paolo Bouquet, Eyke Hüllermeier, Virginia Dignum, Frank Dignum, and Frank 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. ISBN 978-1-61499-671-2. doi: 10.3233/978-1-61499-672-9-613.

Vaishak Belle. Open-universe weighted model counting. In Satinder P. Singh and Shaul 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.

Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In Yang and Wooldridge [2015], pages 2770–2776. ISBN 978-1-57735-738-4. URL http://ijcai.org/Abstract/15/392.

Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In Carla E. Brodley and Peter 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.

Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In Leslie Pack Kaelbling and Alessandro 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. ISBN 0938075934. URL http://ijcai.org/Proceedings/05/Papers/0931.pdf.

Mark Chavira and Adnan Darwiche. Encoding CNFs to empower component analysis. In Armin Biere and Carla 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. ISBN 3-540-37206-7. doi: 10.1007/11814948_9.

Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7): 772–799, 2008. doi: 10.1016/j.artint.2007.11.002.

Mark Chavira, Adnan Darwiche, and Manfred 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.

Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In Linda 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. ISBN 978-3-642-39090-6. doi: 10.1007/978-3-642-39091-3_11.

Adnan 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.

Adnan Darwiche. A logical approach to factoring belief networks. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne 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. ISBN 1-55860-554-1.

Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In Ramón López de Mántaras and Lorenza 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. ISBN 1-58603-452-9.

Adnan 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.

Jeffrey M. Dudek, Vu Phan, and Moshe Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second In-*

novative 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. ISBN 978-1-57735-823-7. URL https://aaai.org/ojs/index.php/AAAI/article/view/5505.

Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. DPMC: weighted model counting by dynamic programming on project-join trees. In Helmut 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. ISBN 978-3-030-58474-0. doi: 10.1007/978-3-030-58475-7_13.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc 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.

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

Vibhav Gogate and Pedro M. Domingos. Probabilistic theorem proving. *Commun. ACM*, 59(7):107–115, 2016. doi: 10.1145/2936726.

Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. Dice: Compiling discrete probabilistic programs for scalable inference. *CoRR*, abs/2005.09089, 2020.

Thomas Jech. *Set theory, Second Edition*. Perspectives in Mathematical Logic. Springer, 1997. ISBN 978-3-540-63048-7. URL https://doi.org/10.1145/2936726.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *J. Appl. Log.*, 22:46–62, 2017. doi: 10.1016/j.jal.2016.11.031.

Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. ISBN 978-0-9992411-0-3. doi: 10.24963/ijcai.2017/93. URL http://www.ijcai.org/Proceedings/2017/.

Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In Yang and Wooldridge [2015], pages 3141–3148. ISBN 978-1-57735-738-4. URL http://ijcai.org/Abstract/15/443.

Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann 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.

Tian Sang, Paul Beame, and Henry A. Kautz. Performing Bayesian inference by weighted model counting. In Manuela M. Veloso and Subbarao 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. ISBN 1-57735-236-X. URL http://www.aaai.org/Library/AAAI/2005/aaai05-075.php.

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

Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In Toby 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. ISBN 978-1-57735-516-8. doi: 10.5591/978-1-57735-516-8/IJCAI11-363. URL http://ijcai.org/proceedings/2011.

Jirí Vomlel and Petr Tichavský. Probabilistic inference in BN2T models by weighted model counting. In Manfred Jaeger, Thomas Dyhre Nielsen, and Paolo 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.

Wei Wei and Bart Selman. A new approach to model counting. In Fahiem Bacchus and Toby 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.

Qiang Yang and Michael J. Wooldridge, editors. *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015. AAAI Press. ISBN 978-1-57735-738-4. URL http://ijcai.org/proceedings/2015.

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

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

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

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

If X is not binary, then $\left(\sum_{i=1}^m [\lambda_{X=x_i}]\right)(T) = 1$, and $\left(\prod_{i=1}^m \prod_{j=i+1}^m (\overline{[\lambda_{X=x_i}]} + \overline{[\lambda_{X=x_j}]})\right)(T) = 1$, so $\mathrm{CPT}_X(T) = \Pr(X = x \mid Y_1 = y_1, \ldots, Y_n = y_n)$ by a similar argument as before. □

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

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

for all $T \in 2^U$.

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

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

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

Theorem 4. *For any $X \in \mathscr{V}$ and $x \in \mathrm{im}X$,*

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

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

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

by:

- the proof of Theorem 1 by Dudek et al. [2020a];
- if $\lambda_{X=x} \notin T \in 2^U$, then $(\phi \cdot [\lambda_{X=x}])(T) = \phi(T) \cdot [\lambda_{X=x}](T \cap \{\lambda_{X=x}\}) = \phi(T) \cdot 0 = 0$;
- Lemma 2;
- marginalisation of a probability distribution.

□

C Submitted Paper (SAT 2021)

Weighted Model Counting Without Parameter Variables

Paulius Dilkas[1] and Vaishak Belle[1,2]

[1] University of Edinburgh, Edinburgh, UK
p.dilkas@sms.ed.ac.uk, vaishak@ed.ac.uk
[2] Alan Turing Institute, London, UK

Abstract. 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.

Keywords: Weighted model counting · Probabilistic inference · Bayesian networks.

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 [12], has emerged as a powerful computational framework for problems in many domains, e.g., probabilistic graphical models such as Bayesian networks and Markov networks [4, 9, 10, 16, 34], neuro-symbolic artificial intelligence [39], probabilistic programs [27], and probabilistic logic programs [22]. It has been extended to support continuous variables [7], infinite domains [5], first-order logic [25, 38], and arbitrary semirings [6, 28]. 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 [4, 9, 10, 16, 34]. As the parameterised complexity of model counting (and, by extension, WMC) depends on the number of variables [2, 32], this can make WMC unnecessarily slow and could be detrimental to WMC algorithms such as ADDMC [20] that depend on variable ordering heuristics.

One approach to this problem considers weighted clauses and probabilistic semantics based on Markov networks [23]. 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 [20] and DPMC [21]. Furthermore, we identify sufficient conditions for transforming a WMC instance into our new format. As many WMC inference algorithms such as Ace, c2d [17], and miniC2D [30] work by compilation to tractable representations such as arithmetic circuits, deterministic, decomposable negation normal form [15], and sentential decision diagrams (SDDs) [18], 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 [14]. 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 [14, 23] 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 [4, 9, 10, 16, 34] 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 [9, 10]) 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.

2 Weighted Model Counting

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

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 [12], 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[3], 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 [9, 10].

Definition 2. *Let* ϕ *be a formula over a set of variables* X*. Then* $Y \subseteq X$ *is a* minimum-cardinality model *of* ϕ *if* $Y \models \phi$ *and* $|Y| \leq |Z|$ *for all* $Z \models \phi$*.*

Definition 3 (Minimum-Cardinality WMC). *A* minimum-cardinality WMC *instance consists of the same tuple as a WMC instance, but its* answer *is defined to be* $\sum_{Y \models \phi,\ |Y|=k} \prod_{Y \models l} w(l)$ *(where* $k = \min_{Y \models \phi} |Y|$*) if* ϕ *is satisfiable, and zero otherwise.*

Example 1. 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.

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 [16, 34], 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 [16]. 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

[3] Ace [12] 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/.

the year of publication. Darwiche was the first to suggest the `d02` [16] 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` [34] 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 propose `cd05` [9] where they shift from WMC to minimum-cardinality WMC because that allows the encoding to have fewer variables and clauses. In particular, they propose a way to use the same parameter variable to represent all probabilities in a CPT that are equal and keep only clauses that 'imply' parameter variables (i.e., omit clauses where a parameter variable implies indicator variables).[4] In their next encoding, `cd06` [10], the same authors optimise the aforementioned implication clauses, choosing the smallest sufficient selection of indicator variables. A decade later, Bart et al. present `bklm16` [4] 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[5]). We suspect that this issue was not identified during the original set of experiments because they never looked at numerical answers.

Example 2. 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 1 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.

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 $\{0, 1\}^n \to \mathbb{R}$ [8]. Equivalently, let X denote a set with n elements (we will refer to

[4] Example 2 demonstrates what we mean by implication clauses.
[5] http://www.cril.univ-artois.fr/kc/bn2cnf.html

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) [3] (although a tensor-based approach has also been suggested [19, 21]), have seen extensive use in value iteration for Markov decision processes [26], both exact and approximate Bayesian network inference [11, 24], and sum-product network [31] to Bayesian network conversion [40]. ADDs have been extended to compactly represent additive and multiplicative structure [37], sentences in first-order logic [35], and continuous variables [36], the last of which was also applied to weighted model integration, i.e., the WMC extension for continuous variables [7, 29].

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 [20, 21].

Definition 4 (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., $\exists_x \exists_y f = \exists_y \exists_x f$) [20, 21], 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., $\exists_x f g = \exists_x (f g)$).*

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 1. 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 $\left([x]_0^1\right)' : 2^{\{x,y\}} \to \mathbb{R}$ which is defined as $\left([x]_0^1\right)'(Z) \coloneqq [x]_0^1(Z \cap \{x\})$ for all $Z \subseteq \{x, y\}$ (and equivalently for $[y]_0^1$).

4 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, and they are easily representable in both text files and using the syntax proposed in this paper.

Definition 5 (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.*[6] *Its* answer *is* $\omega \cdot \left(\exists_X \prod_{f \in F} f\right)(\emptyset)$.

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. The algorithm works on four out of the five Bayesian network encodings: `bklm16` [4], `cd05` [9], `cd06` [10], and `d02` [16]. There is no obvious way to adjust it to work with `sbk05` because the roles of indicator and parameter (i.e., 'chance') variables overlap [34]. The algorithm is based on several observations that will be made more precise in Section 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 [9]). Note that while we formulate our algorithm as a sequel to the WMC encoding

[6] 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 [4].

Algorithm 1: WMC to PBP transformation

Data: WMC (or minimum-cardinality WMC) instance (ϕ, X_I, X_P, w)
Result: PBP instance (F, X_I, ω)

$F \leftarrow \emptyset$;
$\omega \leftarrow 1$;
foreach *clause* $c \in \phi$ **do**
  if $c \cap X_P = \{p\}$ *for some* 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 1. 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 (say, $c \in \phi$) has no parameter variables, we reformulate it into a pseudo-Boolean function $[c]_0^1$. Finally, if a clause has negative parameter literals, we skip it.

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

4.2 Correctness Proofs

In this section, we outline key properties that a (WMC or minimum-cardinality WMC) encoding has to satisfy for Algorithm 1 to output an equivalent PBP instance. We divide the correctness proof into two theorems: Theorem 2 for WMC encodings (i.e., `bklm16` and `d02`) and Theorem 3 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 1 (Early Projection [20, 21]). *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 1. *For any pseudo-Boolean function $f\colon 2^X \to \mathbb{R}$, we have that $(\exists_X f)(\emptyset) = \sum_{Y \subseteq X} f(Y)$.*

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

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

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

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

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

is a PBP instance with the same answer (as defined in Definitions 1 and 5).

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 is (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 1. Note that

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

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

Theorem 2 (Correctness for WMC). *Algorithm 1, when given a WMC instance (ϕ, X_I, X_P, w), returns a PBP instance with the same answer (as defined in Definitions 1 and 5), 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,

$$\left(\left\{[c]_0^1 \mid c \in \phi\right\} \cup \left\{[x]_{w(\neg x)}^{w(x)} \;\middle|\; x \in X_I \cup X_P\right\}, X_I \cup X_P, 1\right) \tag{2}$$

is a PBP instance with the same answer as the given WMC instance. By Definition 5, 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 $\phi_* \coloneqq \{c \in \phi \mid \texttt{Vars}(c) \cap X_P = \emptyset\}$ and $\phi_p \coloneqq \{c \in \phi \mid \texttt{Vars}(c) \cap X_P = \{p\}\}$ for all $p \in X_P$. We can then use Theorem 1 to reorder the answer into $\left(\exists_{X_I} \left(\prod_{x \in X_I} [x]_{w(\neg x)}^{w(x)}\right) \left(\prod_{c \in \phi_*} [c]_0^1\right) \prod_{p \in X_P} \exists_p [p]_{w(\neg p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1\right)(\emptyset)$.

Let us first consider how the unfinished WMC instance (F, X_I, ω) after the loop on Lines 3 to 10 differs from (2). Note that Algorithm 1 leaves each $c \in \phi_*$ unchanged, i.e., adds $[c]_0^1$ to F. We can then fix an arbitrary $p \in X_P$ and let F_p be the set of functions added to F as a replacement of ϕ_p. It is sufficient to show that

$$\omega \prod_{f \in F_p} f = \exists_p [p]_{w(\neg p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1. \tag{3}$$

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, $\exists_p [p]_{w(\neg p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1 = \exists_p [p]_1^{w(p)} \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i\right]_0^1$. If $w(p) = 1$, then

$$\exists_p [p]_1^{w(p)} \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i\right]_0^1 = \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i\right]_0^1\Bigg|_{p=1} + \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i\right]_0^1\Bigg|_{p=0}. \tag{4}$$

Since for any input, $\bigwedge_{i=1}^n l_i$ is either true or false, exactly one of the two summands in Eq. (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. (3), since $F_p = \emptyset$, and ω is unchanged, we get $\omega \prod_{f \in F_p} f = 1$, and so Eq. (3) is satisfied under Condition 1 when $w(p) = 1$.

If $w(p) \neq 1$, then $F_p = \left\{ \left[\bigwedge_{i=1}^n l_i\right]_1^{w(p)} \right\}$, and $\omega = 1$, and so we want to show that $\left[\bigwedge_{i=1}^n l_i\right]_1^{w(p)} = \exists_p [p]_1^{w(p)} \left[p \Leftrightarrow \bigwedge_{i=1}^n l_i\right]_0^1$, and 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. (3). If $w(p) = 1$, then $F_p = \emptyset$, and $\omega = 1$, and so the left-hand side of Eq. (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 1 produces $F_p = \emptyset$ and $\omega = w(p)$, and so

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

The only remaining case is when $w(p) \neq 1$ and $\{p\} \notin \phi_p$. Then $\omega = 1$, and $F_p = \left\{ \left[\bigwedge_{l \in c \setminus \{p\}} \neg l\right]_1^{w(p)} \,\middle|\, c \in \phi_p \right\}$, so we need to show that $\prod_{c \in \phi_p} \left[\bigwedge_{l \in c \setminus \{p\}} \neg l\right]_1^{w(p)} = \exists_p [p]_{1-w(p)}^{w(p)} \prod_{c \in \phi_p} [c]_0^1$. We can rearrange the right-hand side as

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

By Condition 2e, $\bigwedge_{l \in c \setminus \{p\}} \neg l$ can be true for at most one $c \in \phi_p$, and so $\left[\bigvee_{c \in \phi_p} \bigwedge_{l \in c \setminus \{p\}} \neg l\right]_1^{w(p)} = \prod_{c \in \phi_p} \left[\bigwedge_{l \in c \setminus \{p\}} \neg l\right]_1^{w(p)}$ which is exactly what we needed to show. This ends the proof that the first loop of Algorithm 1 preserves the answer under both Condition 1 and Condition 2. Finally, the loop on Lines 11 to 12 of Algorithm 1 replaces $[v]_1^p [\neg v]_1^q$ with $[v]_q^p$ (for some $v \in X_I$ and $p, q \in \mathbb{R}$), but, of course, $[v]_1^p [\neg v]_1^q = [v]_1^p [v]_q^1 = [v]_q^p$, i.e., the answer is unchanged.

Theorem 3 (Minimum-Cardinality Correctness). *Let (ϕ, X_I, X_P, w) be a minimum-cardinality WMC instance that satisfies Conditions 2b to 2e of Theorem 2 as well as the following:*

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

Then Algorithm 1, when applied to (ϕ, X_I, X_P, w), outputs a PBP instance with the same answer (as defined in Definitions 3 and 5).

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 2 and 3, the following proof is slightly less explicit and assumes that $\omega = 1$.

Proof. Let (F, X_I, ω) be the tuple returned by Algorithm 1 and note that $F = \left\{[c]_0^1 \mid c \in \phi,\, c \cap X_P = \emptyset\right\} \cup \left\{ \left[\bigwedge_{l \in c \setminus \{p\}} \neg l\right]_1^{w(p)} \;\middle|\; p \in X_P,\, p \in c \in \phi,\, c \neq \{p\} \right\}$. We split the proof into two parts. In the first part, we show that there is a bijection between minimum-cardinality models of ϕ and $Y \subseteq X_I$ such that $\left(\prod_{f \in F} f\right)(Y) \neq 0$.[7] 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

$$\prod_{Z \models l} w(l) = \left(\prod_{f \in F} f\right)(Y). \tag{5}$$

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

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

For the second part, let us consider the multiplicative influence of a single parameter variable $p \in X_P$ on Eq. (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 2). This argument works in the other direction as well.

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 1 applied afterwards.[8] For each encoding `e`, we write `e++` to denote the combination of encoding a Bayesian network as a WMC instance using `e` and transforming it into a PBP instance using Algorithm 1. Along with DPMC[9], we also include WMC algorithms used in the papers that introduce each encoding: Ace for `cd05`, `cd06`, and `d02`; Cachet[10] [33] for `sbk05`; and `c2d`[11] [17] with `query-dnnf`[12] 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?

5.1 Setup

DPMC is run with tree decomposition-based planning and ADD-based execution—the best-performing combination in the original set of experiments [21]. We use a single iteration of `htd` [1] 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 [21]) optimises the order in which pseudo-Boolean functions are multiplied by prioritising functions with small representations.

[8] Recall that `cd05` and `cd06` are incompatible with DPMC.
[9] https://github.com/vardigroup/DPMC
[10] https://cs.rochester.edu/u/kautz/Cachet/
[11] http://reasoning.cs.ucla.edu/c2d/
[12] http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html

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. [34]; – Mastermind (144 instances) and – Random Blocks (256 instances) by Chavira et al. [13]; – other binary Bayesian networks (50 instances) including Plan Recognition [34], Friends and Smokers, Students and Professors [13], and `tcc4f`; – 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 [21, 33]. 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 $\Pr(X = \mathsf{true})$, 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.[13]

5.2 Results

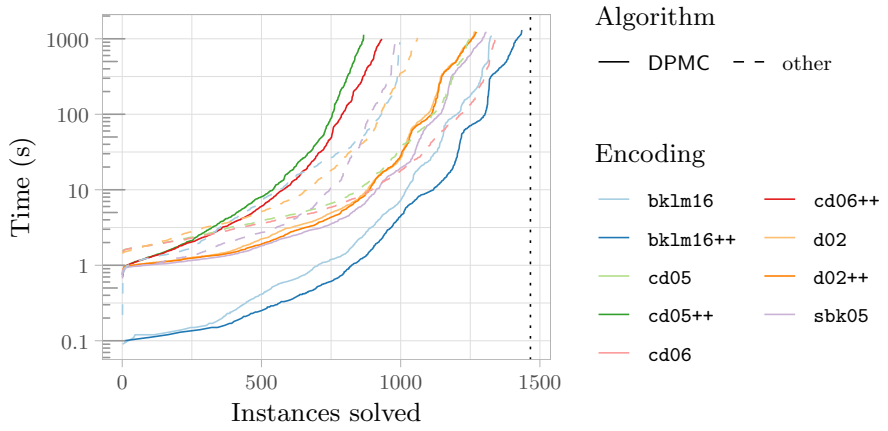


Fig. 1. Cactus plot of all algorithm-encoding pairs. The dotted line denotes the total number of instances used.

Figure 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 1 further shows that DPMC+`bklm16++`

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

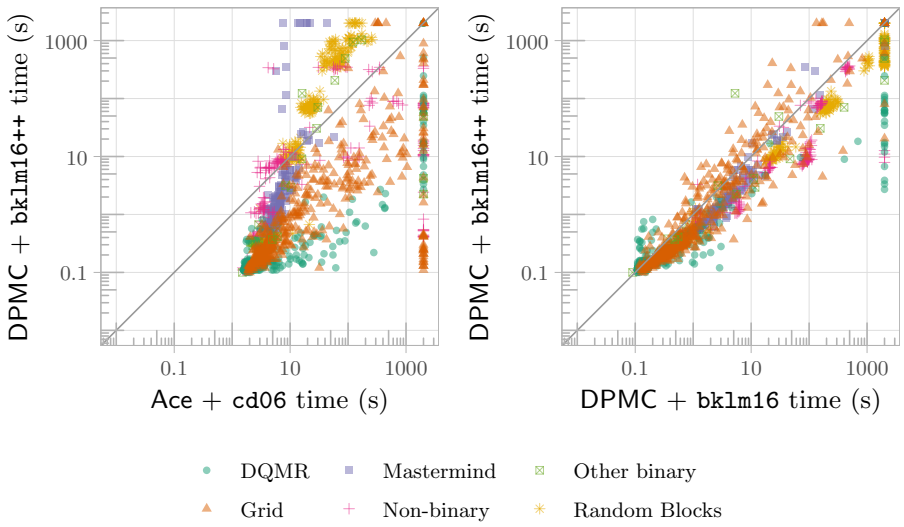


Fig. 2. An instance-by-instance comparison between DPMC + `bklm16++` (the best combination according to Fig. 1) and the second and third best-performing combinations: Ace + `cd06` and DPMC + `bklm16`.

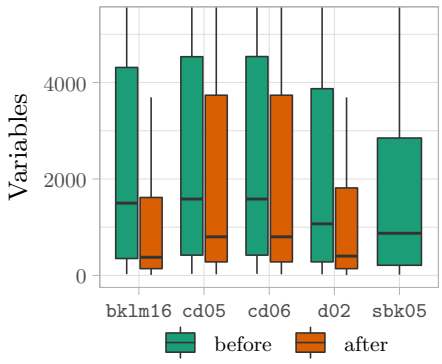


Fig. 3. Box plots of the numbers of variables in each encoding across all benchmark instances before and after applying Algorithm 1. Outliers and the top parts of some whiskers are omitted.

Table 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

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

The scatter plots in Fig. 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 1 is quite noisy, i.e, with most instances scattered around the line of equality. However, our transformation does enable DPMC to solve many instances that were previously beyond its reach.

We also record numbers of variables in each encoding before and after applying Algorithm 1. Figure 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.

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.

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 [4, 9, 10, 16, 34]. While the distinction between indicator and parameter variables is often not explicitly described in other WMC encodings [22, 27, 39], 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 [9, 10]) to be transferred to other application domains.

References

1. Abseher, M., Musliu, N., Woltran, S.: htd - A free, open-source framework for (customized) tree decompositions and beyond. In: Salvagnin, D., Lombardi, M. (eds.) Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10335, pp. 376–386. Springer (2017). https://doi.org/10.1007/978-3-319-59776-8_30
2. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and Bayesian inference. In: 44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings. pp. 340–351. IEEE Computer Society (2003). https://doi.org/10.1109/SFCS.2003.1238208
3. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. Formal Methods Syst. Des. **10**(2/3), 171–206 (1997). https://doi.org/10.1023/A:1008699807402
4. Bart, A., Koriche, F., Lagniez, J., Marquis, P.: An improved CNF encoding scheme for probabilistic inference. In: Kaminka, G.A., Fox, M., Bouquet, P., Hüllermeier, E., Dignum, V., Dignum, F., van Harmelen, F. (eds.) 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). Frontiers in Artificial Intelligence and Applications, vol. 285, pp. 613–621. IOS Press (2016). https://doi.org/10.3233/978-1-61499-672-9-613
5. Belle, V.: Open-universe weighted model counting. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA. pp. 3701–3708. AAAI Press (2017), http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15008
6. Belle, V., De Raedt, L.: Semiring programming: A semantic framework for generalized sum product problems. Int. J. Approx. Reason. **126**, 181–201 (2020). https://doi.org/10.1016/j.ijar.2020.08.001
7. Belle, V., Passerini, A., Van den Broeck, G.: Probabilistic inference in hybrid domains by weighted model integration. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 2770–2776. AAAI Press (2015), http://ijcai.org/Abstract/15/392
8. Boros, E., Hammer, P.L.: Pseudo-Boolean optimization. Discret. Appl. Math. **123**(1-3), 155–225 (2002). https://doi.org/10.1016/S0166-218X(01)00341-9
9. Chavira, M., Darwiche, A.: Compiling Bayesian networks with local structure. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005. pp. 1306–1312. Professional Book Center (2005), http://ijcai.org/Proceedings/05/Papers/0931.pdf
10. Chavira, M., Darwiche, A.: Encoding CNFs to empower component analysis. In: Biere, A., Gomes, C.P. (eds.) Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4121, pp. 61–74. Springer (2006). https://doi.org/10.1007/11814948_9
11. Chavira, M., Darwiche, A.: Compiling Bayesian networks using variable elimination. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 2443–2449 (2007), http://ijcai.org/Proceedings/07/Papers/393.pdf

12. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. **172**(6-7), 772–799 (2008). https://doi.org/10.1016/j.artint.2007.11.002
13. Chavira, M., Darwiche, A., Jaeger, M.: Compiling relational Bayesian networks for exact inference. Int. J. Approx. Reason. **42**(1-2), 4–20 (2006). https://doi.org/10.1016/j.ijar.2005.10.001
14. Choi, A., Kisa, D., Darwiche, A.: Compiling probabilistic graphical models using sentential decision diagrams. In: van der Gaag, L.C. (ed.) Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 12th European Conference, ECSQARU 2013, Utrecht, The Netherlands, July 8-10, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7958, pp. 121–132. Springer (2013). https://doi.org/10.1007/978-3-642-39091-3_11
15. Darwiche, A.: 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). https://doi.org/10.3166/jancl.11.11-34
16. Darwiche, A.: A logical approach to factoring belief networks. In: Fensel, D., Giunchiglia, F., McGuinness, D.L., Williams, M. (eds.) Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002. pp. 409–420. Morgan Kaufmann (2002)
17. Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: de Mántaras, R.L., Saitta, L. (eds.) 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. pp. 328–332. IOS Press (2004)
18. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: Walsh, T. (ed.) IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011. pp. 819–826. IJCAI/AAAI (2011). https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-143, https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-143
19. Dudek, J.M., Dueñas-Osorio, L., Vardi, M.Y.: Efficient contraction of large tensor networks for weighted model counting through graph decompositions. CoRR **abs/1908.04381** (2019)
20. Dudek, J.M., Phan, V., Vardi, M.Y.: 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. pp. 1468–1476. AAAI Press (2020), https://aaai.org/ojs/index.php/AAAI/article/view/5505
21. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: DPMC: weighted model counting by dynamic programming on project-join trees. In: Simonis, H. (ed.) Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12333, pp. 211–230. Springer (2020). https://doi.org/10.1007/978-3-030-58475-7_13
22. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory Pract. Log. Program. **15**(3), 358–401 (2015). https://doi.org/10.1017/S1471068414000076

23. Gogate, V., Domingos, P.M.: Formula-based probabilistic inference. In: Grünwald, P., Spirtes, P. (eds.) UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010. pp. 210–219. AUAI Press (2010)
24. Gogate, V., Domingos, P.M.: Approximation by quantization. In: Cozman, F.G., Pfeffer, A. (eds.) UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011. pp. 247–255. AUAI Press (2011)
25. Gogate, V., Domingos, P.M.: Probabilistic theorem proving. Commun. ACM **59**(7), 107–115 (2016). https://doi.org/10.1145/2936726
26. Hoey, J., St-Aubin, R., Hu, A.J., Boutilier, C.: SPUDD: stochastic planning using decision diagrams. In: Laskey, K.B., Prade, H. (eds.) UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30 - August 1, 1999. pp. 279–288. Morgan Kaufmann (1999)
27. Holtzen, S., Van den Broeck, G., Millstein, T.D.: Scaling exact inference for discrete probabilistic programs. Proc. ACM Program. Lang. **4**(OOPSLA), 140:1–140:31 (2020). https://doi.org/10.1145/3428208
28. Kimmig, A., Van den Broeck, G., De Raedt, L.: Algebraic model counting. J. Appl. Log. **22**, 46–62 (2017). https://doi.org/10.1016/j.jal.2016.11.031
29. Kolb, S., Mladenov, M., Sanner, S., Belle, V., Kersting, K.: Efficient symbolic integration for probabilistic inference. In: Lang, J. (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. pp. 5031–5037. ijcai.org (2018). https://doi.org/10.24963/ijcai.2018/698
30. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 3141–3148. AAAI Press (2015), http://ijcai.org/Abstract/15/443
31. Poon, H., Domingos, P.M.: Sum-product networks: A new deep architecture. In: Cozman, F.G., Pfeffer, A. (eds.) UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011. pp. 337–346. AUAI Press (2011)
32. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discrete Algorithms **8**(1), 50–64 (2010). https://doi.org/10.1016/j.jda.2009.06.002
33. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: 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), http://www.satisfiability.org/SAT04/programme/21.pdf
34. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA. pp. 475–482. AAAI Press / The MIT Press (2005), http://www.aaai.org/Library/AAAI/2005/aaai05-075.php
35. Sanner, S., Boutilier, C.: Practical solution techniques for first-order MDPs. Artif. Intell. **173**(5-6), 748–788 (2009). https://doi.org/10.1016/j.artint.2008.11.003
36. Sanner, S., Delgado, K.V., de Barros, L.N.: Symbolic dynamic programming for discrete and continuous state MDPs. In: Cozman, F.G., Pfeffer, A. (eds.) UAI

2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011. pp. 643–652. AUAI Press (2011)
37. Sanner, S., McAllester, D.A.: Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005. pp. 1384–1390. Professional Book Center (2005), http://ijcai.org/Proceedings/05/Papers/1439.pdf
38. Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., De Raedt, L.: Lifted probabilistic inference by first-order knowledge compilation. In: Walsh, T. (ed.) IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011. pp. 2178–2185. IJCAI/AAAI (2011). https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-363
39. Xu, J., Zhang, Z., Friedman, T., Liang, Y., Van den Broeck, G.: A semantic loss function for deep learning with symbolic knowledge. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5498–5507. PMLR (2018), http://proceedings.mlr.press/v80/xu18h.html
40. Zhao, H., Melibari, M., Poupart, P.: On the relationship between sum-product networks and Bayesian networks. In: Bach, F.R., Blei, D.M. (eds.) Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015. JMLR Workshop and Conference Proceedings, vol. 37, pp. 116–124. JMLR.org (2015), http://proceedings.mlr.press/v37/zhaoc15.html