TLP **15** (3): 358–401, 2015. © Cambridge University Press 2014
doi:10.1017/S1471068414000076 First published online 15 April 2014

Inference and learning in probabilistic logic programs using weighted Boolean formulas

DAAN FIERENS, GUY VAN DEN BROECK, JORIS RENKENS, DIMITAR SHTERIONOV, BERND GUTMANN, INGO THON, GERDA JANSSENS and LUC DE RAEDT

Department of Computer Science, KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium
(*e-mail:* FirstName.LastName@cs.kuleuven.be)

submitted 26 June 2012; revised 4 January 2013; accepted 28 January 2014

Abstract

Probabilistic logic programs are logic programs in which some of the facts are annotated with probabilities. This paper investigates how classical inference and learning tasks known from the graphical model community can be tackled for probabilistic logic programs. Several such tasks, such as computing the marginals, given evidence and learning from (partial) interpretations, have not really been addressed for probabilistic logic programs before. The first contribution of this paper is a suite of efficient algorithms for various inference tasks. It is based on the conversion of the program and the queries and evidence to a weighted Boolean formula. This allows us to reduce inference tasks to well-studied tasks, such as weighted model counting, which can be solved using state-of-the-art methods known from the graphical model and knowledge compilation literature. The second contribution is an algorithm for parameter estimation in the learning from interpretations setting. The algorithm employs expectation-maximization, and is built on top of the developed inference algorithms. The proposed approach is experimentally evaluated. The results show that the inference algorithms improve upon the state of the art in probabilistic logic programming, and that it is indeed possible to learn the parameters of a probabilistic logic program from interpretations.

KEYWORDS: probabilistic logic programming, probabilistic inference, parameter learning

1 Introduction

There is a lot of interest in combining probability and logic for dealing with complex relational domains. This interest has resulted in the fields of Probabilistic Logic Programming (PLP) (De Raedt *et al.* 2008) and Statistical Relational Learning (SRL) (Getoor and Taskar 2007). While the two fields essentially study the same problem, there are differences in emphasis. SRL techniques have focussed on the extension of probabilistic graphical models such as Markov or Bayesian networks with logical and relational representations as in, for instance, Markov logic (Poon and Domingos 2006). Conversely, PLP has extended logic programming (LP) languages (or Prolog) with probabilities. This has resulted in differences in representation and

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

semantics between the two approaches and, more importantly, also in differences in the inference tasks and learning settings that are supported. In graphical models and SRL, the most common inference tasks are that of computing the marginal probability of a set of random variables, given some evidence (we call this the MARG task) and finding the most likely joint state of random variables, given the evidence (the MPE task). The PLP community has mostly focussed on computing the success probability of queries without evidence. Furthermore, probabilistic logic programs are usually learned from entailment (Gutmann *et al.* 2008a; Sato and Kameya 2008), while the standard learning setting in graphical models and SRL corresponds to learning from interpretations (LFI). This paper bridges the gap between the two communities by adapting the traditional graphical model and SRL settings towards the PLP perspective. We contribute general MARG and MPE inference techniques and a LFI algorithm for PLP. In this paper we use ProbLog (De Raedt *et al.* 2007) as a PLP language, but our approach is relevant to related languages such as ICL (Poole 2008), PRISM (Sato and Kameya 2008) and LPAD/CP-logic (Vennekens *et al.* 2009) as well.

The first key contribution of this paper is a two-step approach for performing MARG and MPE inference in probabilistic logic programs. In the first step, the program is converted to an equivalent weighted Boolean (propositional) formula. This conversion is based on well-known conversions from the knowledge representation and LP literature. The MARG task then reduces to weighted model counting (WMC) on the resulting weighted formula, and the MPE task to weighted MAX-SAT. The second step then involves calling a state-of-the-art solver for WMC or MAX-SAT. In this way, we establish new links between PLP inference and standard problems such as WMC and MAX-SAT. We also identify a novel connection between PLP and Markov Logic (Poon and Domingos 2006). From a probabilistic perspective, our approach is similar to the work of Darwiche (2009) and others (Park 2002; Sang *et al.* 2005) who perform the Bayesian network inference by conversion to weighted formulas. We do the same for PLP, a much more expressive representation framework than traditional graphical models. PLP extends a programming language and allows us to concisely represent large sets of dependencies between random variables. From a logical perspective, our approach is related to Answer Set Programming (ASP), where models are often computed by translating the ASP program to a Boolean formula and applying an SAT solver (Lin and Zhao 2002). Our approach is similar in spirit, but is different in that it employs a probabilistic context.

The second key contribution of this paper is an algorithm for learning the parameters of probabilistic logic programs from data. We use the LFI setting, which is a standard setting in graphical models and SRL (although they use different terminology). This setting has also received a lot of attention in inductive LP (De Raedt 2008), but has not yet been used for *probabilistic* logic programs. Our algorithm, called LFI-ProbLog, is based on Expectation-Maximization (EM) and is built on top of the inference techniques presented in this paper.

The present paper is based on and integrates our previous papers (Fierens *et al.* 2011; Gutmann *et al.* 2011) in which inference and learning were studied and implemented separately. Historically, the LFI approach, as detailed by Gutmann *et al.* (2010, 2011), was developed first and used Binary Decision Diagrams (BDDs)

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

for inference and learning. The use of BDDs for learning in an EM style is related to the approach by Ishihata *et al.* (2008), who developed an EM algorithm for propositional BDDs and suggested that their approach can be used to perform learning from entailment for PRISM programs. Fierens *et al.* (2011) later showed that an alternative approach to inference – that is more general, efficient and principled – can be realized using WMC and compilation of d-DNNFs rather than BDDs, as in the initial ProbLog implementation (Kimmig *et al.* 2010). The present paper employs the approach by Fierens *et al.*, also for learning from interpretations in an EM style and thus integrates the two earlier approaches. The resulting techniques are integrated in a novel implementation, called ProbLog2. While the first ProbLog implementation (Kimmig *et al.* 2010) was tightly integrated in the YAP Prolog engine and employed BDDs, ProbLog2 is much closer in spirit to some ASP systems than to Prolog and employs d-DNNFs and WMC.

This paper is organized as follows. We first review the necessary background (Section 2) and introduce PLP (Section 3). Next, we state the inference tasks that we consider (Section 4). Then we introduce our two-step approach for inference (Sections 5 and 6), and introduce the new learning algorithm (Section 7). Finally, we briefly discuss the implementation of the new system (Section 8) and evaluate the entire approach by means of experiments on relational data (Section 9).

2 Background

We now review the basics of first-order logic (FOL) and LP. Readers familiar with FOL and LP can safely skip this section.

2.1 First-Order Logic

A *term* is a variable, a constant, or a functor applied to terms. An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is a predicate of arity n and t_i are terms. A *formula* is built out of atoms using universal and existential quantifiers and the usual logical connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$. An FOL theory is a set of formulas that implicitly form a conjunction. An expression is called *ground* if it does not contain variables. A ground (or propositional) theory is said to be in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals. A *literal* is an atom or its negation. Each disjunction of literals is called a *clause*. A disjunction consisting of a single literal is called a *unit clause*. Each ground theory can be written in CNF form.

The *Herbrand base* of an FOL theory is a set of all ground atoms constructed using predicates, functors and constants in the theory. A Herbrand interpretation, also called a *(possible) world*, is an assignment of a truth value to all atoms in the Herbrand base. A world or interpretation is called a *model* of the theory if it satisfies all formulas in the theory (in other words, if all formulas evaluate to be true in that world).

2.2 Logic Programming

Syntactically, a normal logic program, or briefly *logic program* (LP) is a set of rules. A *rule* (also called a *normal clause*) is a universally quantified expression of the

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

form `h :- b1,..., bn`, where h is an atom and $b_1, \ldots, b_n$ are literals. The atom h is called the *head* of the rule and $b_1, \ldots, b_n$ the *body*, representing the conjunction $b_1 \wedge \ldots \wedge b_n$. A *fact* is a rule that has *true* as its body and is written more compactly as `h`.

We use the *well-founded semantics* for LPs (Van Gelder *et al.* 1991). In the case of a negation-free LP (or *definite program*), the well-founded model is identical to a well-known *Least Herbrand Model* (LHM). The LHM is equal to the least of all the models obtained when interpreting the LP as an FOL theory of implications. The *least* model is the model that is a subset of all other models (in the sense that it makes the fewest atoms true). Intuitively, the LHM is a set of all ground atoms that are entailed by an LP. For negation-free LPs, the LHM is guaranteed to exist and be unique. For LPs with negation, we use the well-founded model. We refer to Van Gelder *et al.* (1991) for details. The ProbLog semantics requires all considered LPs to have a two-valued well-founded model (see Section 3.2). For such programs, the well-founded model is identical to the stable model (Van Gelder *et al.* 1991).

Intuitively, the reason why one considers only the *least* model of an LP is that LP semantics makes the closed world assumption (CWA). Under the CWA, everything that is not implied to be true is assumed to be false. This has implications on how to interpret rules. Given a ground LP and an atom a, the set of all rules with a in the head should be read as the *definition* of a: the atom a is defined to be true if and only if at least one of the rule bodies is true (the 'only if' is due to the CWA). This means that there is a crucial difference in semantics between LP and FOL since FOL does not make the CWA. For example, the FOL theory $\{a \leftarrow b\}$ has three models: $\{\neg a, \neg b\}$, $\{a, \neg b\}$ and $\{a, b\}$. The LP {`a :- b`} has only one model, namely, the LHM $\{\neg a, \neg b\}$ (intuitively, a and b are false because there is no rule that makes b true, and hence there is no applicable rule that makes a true either).

Because of the syntactic restrictions of LP, it is tempting to believe that FOL is more 'expressive' than LP. This is wrong because of the difference in semantics: certain concepts that can be expressed in LP cannot be expressed in FOL (see Section 3.3 for details). This motivates our interest in LP and PLP.

3 Probabilistic logic programming and ProbLog

Most PLP languages, including PRISM (Sato and Kameya 2008), ICL (Poole 2008), ProbLog (De Raedt *et al.* 2007) and LPAD (Vennekens *et al.* 2009), are based on Sato's *distribution semantics* (Sato 1995). In this paper we use ProbLog, but our approach can be used for other languages as well.

3.1 Syntax of ProbLog

A ProbLog program consists of two parts: a set of ground probabilistic facts, and a logic program, i.e. a set of rules and ('non-probabilistic') facts. A ground *probabilistic fact*, written `p::f`, is a ground fact `f` annotated with a probability `p`. We allow syntactic sugar for compactly specifying an entire set of probabilistic facts with a single statement. Concretely, we allow what we call *intensional* probabilistic

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

facts, which are statements of the form `p::f(X1,X2,..., Xn) :- body`, with `body` being a conjunction of calls to non-probabilistic facts.[1] The idea is that such a statement defines the domains of variables `X1, X2,...` and `Xn`. When defining the semantics, as well as when performing inference or learning, an intensional probabilistic fact should be replaced by its corresponding set of ground probabilistic facts, as illustrated below. An atom that unifies with a ground probabilistic fact is called a *probabilistic atom*, while an atom that unifies with the head of some rule in the logic program is called a *derived atom*. The set of probabilistic atoms must be disjoint from the set of derived atoms. Also, the rules in the program should be range-restricted: all variables in the head of a rule should also appear in a positive literal in the body of the rule.

Our running example is the program that models the well-known 'Alarm' Bayesian network.

Example 1 (*Running example*)

```
0.1::burglary.                          person(mary).
0.2::earthquake.                        person(john).
0.7::hears_alarm(X) :- person(X).
alarm :- burglary.
alarm :- earthquake.
calls(X) :- alarm, hears_alarm(X).
```

This Problog program consists of probabilistic facts and a logic program. Predicates of probabilistic atoms are `burglary/0`, `earthquake/0` and `hears_alarm/1`, and predicates of derived atoms are `person/1`, `alarm/0` and `calls/1`. Intuitively, the probabilistic facts `0.1::burglary` and `0.2::earthquake` state that there is a burglary with probability 0.1 and an earthquake with probability 0.2. The statement `0.7::hears_alarm(X) :- person(X)` is an intensional probabilistic fact and is a syntactic sugar for the following set of ground probabilistic facts:

```
0.7::hears_alarm(mary).
0.7::hears_alarm(john).
```

The rules in the program define when the alarm goes off and when a person calls as a function of the probabilistic facts.

3.2 Semantics of ProbLog

A ProbLog program specifies a probability distribution over possible worlds. To define this distribution, it is easiest to consider the grounding of the program with respect to the Herbrand base.[2] In this paper, we assume that the resulting Herbrand base is finite. For the distribution semantics in the infinite case, see Sato (1995).

[1] The notion of intensional probabilistic facts does not appear in earlier ProbLog papers but is often useful in practice.

[2] Beforehand, a preprocessing step already replaced the intensional probabilistic facts with their corresponding ground set, as illustrated before.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Table 1. *Total choices and their probabilities*

	Total choice C	P(C)
1.	$\{$ *burglary*, *earthquake*, *hears_alarm*(*john*), *hears_alarm*(*mary*) $\}$	0.0098
2.	$\{$ *burglary*, *earthquake*, *hears_alarm*(*john*) $\}$	0.0042
3.	$\{$ *burglary*, *earthquake*, *hears_alarm*(*mary*) $\}$	0.0042
4.	$\{$ *burglary*, *earthquake* $\}$	0.0018
5.	$\{$ *burglary*, *hears_alarm*(*john*), *hears_alarm*(*mary*) $\}$	0.0392
6.	$\{$ *burglary*, *hears_alarm*(*john*) $\}$	0.0168
7.	$\{$ *burglary*, *hears_alarm*(*mary*) $\}$	0.0168
8.	$\{$ *burglary* $\}$	0.0072
9.	$\{$ *earthquake*, *hears_alarm*(*john*), *hears_alarm*(*mary*) $\}$	0.0882
10.	$\{$ *earthquake*, *hears_alarm*(*john*) $\}$	0.0378
11.	$\{$ *earthquake*, *hears_alarm*(*mary*) $\}$	0.0378
12.	$\{$ *earthquake* $\}$	0.0162
13.	$\{$ *hears_alarm*(*john*), *hears_alarm*(*mary*) $\}$	0.3528
14.	$\{$ *hears_alarm*(*john*) $\}$	0.1512
15.	$\{$ *hears_alarm*(*mary*) $\}$	0.1512
16.	$\{ \}$	0.0648

Each ground probabilistic fact `p::f` gives an *atomic choice*, i.e. we can choose to include f as a fact (with probability p) or discard it (with probability $1 - p$). A *total choice* is obtained by making an atomic choice for each ground probabilistic fact. Formally, a total choice is any subset of the set of all ground probabilistic atoms. Hence, if there are n ground probabilistic atoms, then there are 2^n total choices. Moreover, we have a probability distribution over these total choices: the probability of a total choice is defined to be the product of the probabilities of atomic choices that it is composed of (we can take the product since atomic choices are seen as independent events).

Example 2 (*Total choices of the Alarm example*)
Consider the Alarm program of Example 1. The $2^4 = 16$ total choices corresponding to the four ground probabilistic atoms are given in Table 1. The first row corresponds to the total choice in which all the probabilistic atoms are true. The probability of this total choice is $0.1 \times 0.2 \times 0.7 \times 0.7 = 0.0098$. The second row corresponds to the same total choice except that *hears_alarm*(*mary*) is now false. The probability is $0.1 \times 0.2 \times 0.7 \times (1\text{-}0.7) = 0.0042$. The sum of probabilities of all 16 total choices is equal to one.

Given a particular total choice C, we obtain a logic program $C \cup R$, where R denotes the rules in the ProbLog program. We denote the well-founded model of this logic program as $WFM(C \cup R)$.[3] We call a given world ω a *model* of the ProbLog program if there indeed exists a total choice C such that $WFM(C \cup R) = \omega$. We use $MOD(L)$ to denote the set of all models of a ProbLog program L. The ProbLog semantics is only well defined for programs that are *sound* (Riguzzi and Swift 2013),

[3] Recall from Section 2.2 that for negation-free programs, the WFM is an LHM.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

i.e. programs for which each possible total choice C leads to a well-founded model that is two-valued or 'total' (Van Gelder *et al.* 1991; Riguzzi and Swift 2013).[4] Programs for which this is not the case are not considered valid ProbLog programs.

Everything is now in place to define the distribution over possible worlds: the probability of a world that is a model of the ProbLog program is equal to the probability of its total choice; the probability of a world that is not a model is 0.

Example 3 (*Models and their probabilities*)
(Continuing Example 2) The total choice $\{burglary, earthquake, hears_alarm(john)\}$, which has probability 0.1 $\times$ 0.2 $\times$ 0.7 $\times$ (1-0.7) = 0.0042, yields the following logic program.

```
burglary.                       person(mary).
earthquake.                     person(john).
hears_alarm(john).
alarm :- earthquake.
alarm :- burglary.
calls(X) :- alarm, hears_alarm(X).
```

The WFM of this program is the world $\{person(mary), person(john), burglary, earthquake, hears_alarm(john), \neg hears_alarm(mary), alarm, calls(john), \neg calls(mary)\}$. Hence, this world is a model and its probability is 0.0042. In total there are 16 models, corresponding to each of the 16 total choices shown in Table 1. Note that, out of all possible interpretations of the vocabulary, there are many that are not models of the ProbLog program. An example is any world of the form $\{burglary, \neg alarm, \ldots\}$: it is impossible that *alarm* is false while *burglary* is true. The probability of such worlds is zero.

3.3 Related languages

ProbLog is strongly related to several other languages, in particular to PLP languages such as PRISM (Sato and Kameya 2008), ICL (Poole 2008) and LPAD (Vennekens *et al.* 2009), and other languages such as Markov Logic (Poon and Domingos 2006). Table 2 shows the main features of each language and the major corresponding system.

Compared with most other PLP languages, ProbLog is more expressive with respect to the rules that are allowed in a program. This holds in particular for PRISM and ICL. Both PRISM and ICL require the rules to be acyclic (or contingently acyclic) (Poole 2008; Sato and Kameya 2008). In ProbLog we can have cyclic programs with rules such as `smokes(X) :- smokes(Y), influences(Y,X)`. This type of cyclic rules are often needed for tasks such as collective classification or social network analysis (see Section 9). In addition to acyclicity, PRISM also requires rules with unifiable heads to have mutually exclusive bodies (such that at

[4] A sufficient condition for this is that the rules in the ProbLog program are locally stratified (Van Gelder *et al.* 1991). In particular, this trivially holds for all negation-free programs.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Table 2. *Overview of features of several probabilistic logical languages and the corresponding systems (implementations). The first three features are properties of the language, the last two are properties of the system. We refer to the first ProbLog system as ProbLog1, and the system described here as ProbLog2*

Language System	ProbLog ProbLog1	ProbLog ProbLog2	PRISM PRISM	ICL AILog2	LPAD PITA	MLN Alchemy
Cyclic rules	✓	✓	–	–	✓	✓
Overlapping rule bodies	✓	✓	–	✓	✓	n/a
Inductive definitions	✓	✓	✓	✓	✓	–
Evidence on arbitrary atoms	–	✓	–	✓	–	✓
Multiple queries	–	✓	–	–	–	✓

most one of these bodies can be true simultaneously; this is the mutual exclusiveness assumption). ProbLog does not have this restriction, so rules with unifiable heads can have ‘overlapping’ bodies. For instance, the bodies of the two alarm rules in our running example are overlapping: either burglary or earthquake is sufficient for making the alarm go off, but both can also happen at the same time.

LPADs, as used in the PITA system (Riguzzi and Swift 2013), do not have these syntactic restrictions, and are hence on par with ProbLog in this respect. However, the PITA system does not support the same tasks as the new ProbLog2 system does. For instance, when computing marginal probabilities, ProbLog2 can deal with multiple queries simultaneously and can incorporate evidence, while PITA uses the more traditional PLP setting that considers one query at a time, without evidence (the *succes probability* setting, see Section 4). The same also holds for the first ProbLog system (Kimmig *et al.* 2010). Note that while evidence can in some special cases be incorporated through modelling,[5] we here focus on the general case, i.e. the ability of the system to handle evidence on any arbitrary subset of all atoms in the Herbrand base.

ProbLog2 is the first PLP system that posesses all the features considered in Table 2, i.e. that supports multiple queries and evidence while having none of the language restrictions. The experiments in this paper (Section 9) require all these features and can hence only be carried out in ProbLog2, but not in the other PLP systems.

Markov Logic (Poon and Domingos 2006) is, strictly speaking, not a PLP language as it is based on FOL instead of LP. Nevertheless, Markov Logic, of course, serves the same purpose as the above PLP languages. In terms of expressivity, Markov Logic has a drawback that it cannot express (non-ground) *inductive definitions*. An

[5] For instance, when encoding the Bayesian network in PLP, evidence on nodes at the top of the network (nodes without parents) can be incorporated by including deterministic facts in the program.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

example of an inductive definition is the definition of the notion of a path in a graph in terms of edges. This can be written in plain Prolog, and hence also in ProbLog,

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

In the knowledge representation community, it is well known that inductive definitions can naturally be represented in LP due to LP's least or well-founded model semantics (Denecker *et al.* 2001). In contrast, in FOL one cannot express non-ground inductive definitions, such as the *path* definition above (Grädel 1992). The reason is, roughly speaking, that *path* is the transitive closure of *edge*, and FOL can express that a given relation is transitive, but cannot in general specify this closure. This result carries over to the probabilistic case: we can express inductive definitions in PLP languages such as ProbLog, but not in the FOL-based languages such as Markov Logic.[6] While the non-probabilistic case has been well studied in the knowledge representation literature (Grädel 1992; Denecker *et al.* 2001), the probabilistic case has only very recently received attention (Fierens *et al.* 2012).

4 Inference tasks

In the literature on probabilistic graphical models and SRL, the two most common inference tasks are computing the marginal probability of a set of random variables, given some observations or evidence (we call this the MARG task), and finding the most similar joint state of random variables, given the evidence (known as the Most Probable Explanation (MPE) task). In PLP, the focus has been on the special case of MARG, where there is only a single query atom Q and no evidence. This task is often called Computing the *Success Probability* of Q (De Raedt *et al.* 2007). The works related to the general MARG or MPE task in the PLP literature make a number of restrictive assumptions about the given program such as acyclicity (Gutmann *et al.* 2011) and the mutual exclusiveness assumption of PRISM (Sato and Kameya 2008).There also exist approaches that transform ground probabilistic programs to the Bayesian networks and then use the standard Bayesian network inference procedures (Meert *et al.* 2009). However, these are also restricted to acyclic and already grounded programs.

Our approach for the MARG and MPE inference tasks does not suffer from such restrictions and is applicable to all ProbLog programs. We now formally define these tasks, in addition to the third, strongly related task. Let $\mathbf{At}$ be the Herbrand base, i.e. the set of all ground (probabilistic and derived) atoms in a given ProbLog program. We assume that we are given a set $\mathbf{E} \subset \mathbf{At}$ of observed atoms and a vector $\mathbf{e}$ with their observed truth values. We refer to this as the *evidence* and write $\mathbf{E} = \mathbf{e}$. Note that the evidence is essentially a partial interpretation of the atoms in the ProbLog program.

[6] This discussion applies to *non-ground* ProbLog programs and Markov Logic Networks (MLNs). In Section 5.3, we show that every *ground* ProbLog program can be converted to an equivalent ground MLN. The above implies that no such conversion exists on the non-ground (first-order) level.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

- In the **MARG** task, we are given a set $\mathbf{Q} \subset \mathbf{At}$ of atoms of interest, called *query atoms*. The task is to compute the marginal probability distribution of every such atom, given the evidence, i.e. compute $P(Q \mid \mathbf{E} = \mathbf{e})$ for each $Q \in \mathbf{Q}$.[7]
- The **EVID** or 'probability of evidence' task is to compute $P(\mathbf{E} = \mathbf{e})$. It corresponds to the likelihood of data in a learning setting and can be used as a building block for solving the MARG task (see Section 6.2).
- The **MPE** task is to find the most similar interpretation (joint state) of all non-evidence atoms, given the evidence, i.e. finding $argmax_{\mathbf{u}} P(\mathbf{U} = \mathbf{u} \mid \mathbf{E} = \mathbf{e})$, with $\mathbf{U}$ being the unobserved atoms, i.e. $\mathbf{U} = \mathbf{At} \setminus \mathbf{E}$.

As the following example illustrates, different tasks are strongly related.

Example 4 (*Inference tasks*)
Consider the ProbLog program of Example 1 and assume that we know that John calls, so $\mathbf{E} = \{calls(john)\}$ and $\mathbf{e} = \{true\}$. It can be verified that $calls(john)$ is true in six of the 16 models of the program, namely, the models of total choices 1, 2, 5, 6, 9 and 10 in Table 1. The sum of their probabilities is 0.196, so this is the probability of evidence (EVID). The MPE task boils down to finding the world with the highest probability out of the six worlds that have $calls(john) = true$. It can be verified that this is the world corresponding to the total choice nine, i.e. the choice $\{earthquake, hears_alarm(john), hears_alarm(mary)\}$. An example of the MARG task is to compute the probability that there is a burglary, i.e. $P(burglary = true \mid calls(john) = true) = \frac{P(burglary=true \wedge calls(john)=true)}{P(calls(john)=true)}$. There are four models, in which both $calls(john)$ and $burglary$ are true (models 1, 2, 5 and 6), and their sum of probabilities is 0.07. Hence, $P(burglary = true \mid calls(john) = true)$= 0.07/ 0.196 = 0.357.

Our approach to inference consists of two steps: (1) Convert the program to a weighted Boolean formula, and (2) perform inference on the resulting weighted formula. We discuss these two steps in the next sections.

5 Conversion to a weighted formula

Our conversion takes as input a ProbLog program L, evidence $\mathbf{E} = \mathbf{e}$ and a set of *query atoms* $\mathbf{Q}$, and returns a weighted Boolean (propositional) formula that contains all necessary information. The conversion is similar for each of the considered tasks (MARG, MPE or EVID). The only difference is the choice of the query set $\mathbf{Q}$. For MARG, $\mathbf{Q}$ is the set of atoms for which we want to compute marginal probabilities. For EVID and MPE, we can take $\mathbf{Q} = \emptyset$ (see Section 6.1.1).

The outline of the conversion algorithm is as follows.

1. *Ground L yielding a program L_g while taking into account $\mathbf{Q}$ and $\mathbf{E} = \mathbf{e}$* (cf. Theorem 1, Section 5.1).

[7] The common PLP task of computing the *success probability* of an atom Q is a special case of MARG with $\mathbf{Q}$ being the singleton $\{Q\}$ and $\mathbf{E} = \emptyset$.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

It is unnecessary to consider the full grounding of the program, we only need the part that is relevant to the query, given the evidence, that is, the part that captures the distribution $P(\mathbf{Q} \mid \mathbf{E} = \mathbf{e})$. We refer to the resulting program L_g as the *relevant ground program* (RGP) with respect to $\mathbf{Q}$ and $\mathbf{E} = \mathbf{e}$.

2. *Convert the ground rules in L_g to an equivalent Boolean formula φ_r* (cf. Lemma 1, Section 5.2).
   This step converts LP rules to an equivalent formula.
3. *Assert the evidence and define a weight function* (cf. Theorem 2, Section 5.3).
   To obtain the weighted formula, we first assert the evidence by defining the formula φ as the conjunction of the formula φ_r for the rules (Step 2) and for the evidence φ_e. Then we define a *weight function* for all atoms in φ.

The correctness of the algorithm is shown below; this relies on the indicated theorems and lemmas. Before describing the algorithm in detail, we illustrate it with our Alarm example.

Example 5 (*The three steps in the conversion*)
As in Example 4, we take $calls(john) = true$ as evidence. Suppose that we want to compute the marginal probability of *burglary*, so the query set $\mathbf{Q}$ is $\{burglary\}$. The RGP is as follows:

```
% ground probabilistic facts
0.1::burglary.        0.2::earthquake.        0.7::hears_alarm(john).
% ground rules
alarm :- burglary.
alarm :- earthquake.
calls(john) :- alarm, hears_alarm(john).
```

Note that *mary* does not appear in the grounding because if we have no evidence about her hearing the alarm or calling, she does not affect the probability $P(burglary \mid calls(john) = true)$.

Step 2 converts the three ground rules of the RGP to an equivalent propositional formula φ_r (see Section 5.2). This formula is the conjunction of $alarm \leftrightarrow burglary \vee earthquake$ and $calls(john) \leftrightarrow alarm \wedge hears_alarm(john)$.[8] Step 3 adds the evidence. Since we have only one evidence atom in our example (namely, $calls(john)$ is true), all we need to do is to add the positive unit clause $calls(john)$ to the formula φ_r. The resulting formula φ is $\varphi_r \wedge calls(john)$. Step 3 also defines the weight function, which assigns a weight to each literal in φ, see Section 5.3. This results in the *weighted formula*, that is, the combination of the weight function and the Boolean formula φ.

We now explain the three steps of conversion in detail.

[8] For subsequent steps, it is often convenient to write this formula in CNF. For example, some knowledge compilation systems require CNF input.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

5.1 The relevant ground program

In order to convert a ProbLog program to a Boolean formula, we first ground it. We try to find the part of the grounding that is relevant to queries **Q** and the evidence **E** = **e**. In SRL, this is also called knowledge-based model construction (Kersting and De Raedt 2001). To do this, we make use of the concept of a dependency set with respect to a ProbLog program. We first explain our algorithm and then show its correctness.

The *dependency set* of a ground atom a is the set of all ground atoms that occur in some proof of a. The dependency set of multiple atoms is the union of their dependency sets. We call a ground atom *relevant* with respect to **Q** and **E** if it occurs in the dependency set of $\mathbf{Q} \cup \mathbf{E}$. We call a ground rule relevant if it contains only relevant atoms. It is safe to restrict grounding to relevant rules only. To find relevant atoms and rules, we apply Selective Linear Definite clause (SLD) resolution to prove all atoms in $\mathbf{Q} \cup \mathbf{E}$ (this can be seen as backchaining over the rules starting from $\mathbf{Q} \cup \mathbf{E}$). We employ tabling to avoid proving the same atom twice (and to avoid going into an infinite loop if the rules are cyclic). The relevant rules are all ground rules encountered during the resolution process. As our ProbLog programs are range-restricted, all the variables in the rules used during the SLD resolution will eventually become ground, and hence also the rules themselves.

The above grounding algorithm is not optimal as it does not make use of all available information. For instance, it does not make use of exactly what the evidence is (the values **e**), but only make use of atoms that are in the evidence (the set **E**). One simple, yet sometimes very effective, optimization is to prune *inactive* rules. We call a ground rule inactive if the body of the rule contains a literal l that is false in the evidence (l can be an atom that is false in **e**, or the negation of an atom that is true in **e**). Inactive rules do not contribute to the semantics of a program. Hence, these can be omitted. In practice, we do this simultaneously with the above process: we omit inactive rules encountered during the SLD resolution.[9]

The result of this grounding algorithm is what we call the relevant ground program L_g for L with respect to **Q** and **E** = **e**. It contains all the information necessary for solving the corresponding EVID, MARG or MPE task. The advantage of this 'focussed' approach (i.e. taking into account **Q** and **E** = **e** during grounding) is that the program, and hence the weighted formula, becomes more compact, which makes subsequent inference more efficient. The disadvantage is that we need to redo the conversion to a weighted formula when the evidence and queries change. This is no problem since the conversion is fast compared to the actual inference (see Section 9).

The following theorem shows the correctness of our approach.

[9] This deals with literals that are *false* in the evidence. Conversely, when a body of a ground rule contains a literal that is *true* in the evidence, it has to be kept and the rule cannot be simplified. The reason is that the atom's presence might give rise to a positive loop, which has to be detected during the conversion of the ground program to a Boolean formula in the next step.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Theorem 1
Let L be a ProbLog program, and let L_g be the RGP for L with respect to $\mathbf{Q}$ and $\mathbf{E} = \mathbf{e}$. L and L_g specify the same distribution $P(\mathbf{Q} \mid \mathbf{E} = \mathbf{e})$.

The proofs of all theorems in this paper are in Appendix A given online.

We already showed the RGP for the Alarm example in Example 5 (in that case, there were irrelevant rules about *mary*, but no inactive rules because there was no negative evidence). To illustrate our approach for cyclic programs, we use the well-known Smokers example (Domingos *et al.* 2008).

Example 6 (*ProbLog program for Smokers*)
The ProbLog program for the Smokers example models two causes for people to smoke: either they spontaneously start because of stress or they are influenced by one of their friends.

```
0.2::stress(P) :- person(P).
0.3::influences(P1,P2) :- friend(P1,P2).
person(p1).       person(p2).       person(p3).
friend(p1,p2).    friend(p1,p3)
friend(p2,p1).    friend(p3,p1).

smokes(X) :- stress(X).
smokes(X) :- smokes(Y), influences(Y,X).
```

With the evidence $\{smokes(p2) = true, smokes(p3) = false\}$ and the query set $\{smokes(p1)\}$, we obtain the following ground program:

```
0.2::stress(p1).       0.2::stress(p2).       0.2::stress(p3).
0.3::influences(p2,p1).0.3::influences(p1,p2).0.3::influences(p1,p3).
% irrelevant probabilistic fact !!  0.3::influences(p3,p1).

smokes(p1) :- stress(p1).
smokes(p1) :- smokes(p2), influences(p2,p1).
% inactive rule !!  smokes(p1) :- smokes(p3), influences(p3,p1).
smokes(p2) :- stress(p2).
smokes(p2) :- smokes(p1), influences(p1,p2).
smokes(p3) :- stress(p3).
smokes(p3) :- smokes(p1), influences(p1,p3).
```

The evidence $smokes(p3) = false$ makes the third rule for `smokes(p1)` inactive. This in turn makes the probabilistic fact for `influences(p3,p1)` irrelevant. Nevertheless, the rules for `smokes(p3)` have to be in the grounding, as the truth value of the head of a rule constrains the truth values of the bodies.

5.2 The Boolean formula for the ground program

We now discuss how to convert the rules in the relevant ground program L_g to an equivalent Boolean formula φ_r. Converting a set of LP rules to an equivalent

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Boolean formula is a purely logical (non-probabilistic) problem. This has been well studied in the LP literature, where several conversions have been proposed, e.g. Janhunen (2004). Note that the conversion is not merely a syntactical rewriting issue; the point is that the rules and the formula are to be interpreted according to a different semantics. Hence, the conversion should compensate for this: the rules under LP semantics (with CWA) should be equivalent to the formula under FOL semantics (without CWA).

For acyclic rules, the conversion is straightforward, we can simply take *Clark's completion* of the rules (Lloyd 1987; Janhunen 2004). We illustrate this on the Alarm example, which is indeed acyclic.

Example 7 (*Formula for the* `alarm` *rules*)
As shown in Example 5, the grounding of the Alarm example contains two rules for `alarm`, namely, `alarm :- burglary` and `alarm :- earthquake`. Clark's completion of these rules is the propositional formula $alarm \leftrightarrow burglary \vee earthquake$, i.e. the alarm goes off if and only if there is burglary or earthquake. Once we have the formula, we often need to rewrite it in the CNF form, which is straightforward for a completion formula. For the completion of `alarm`, the resulting CNF has three clauses: $alarm \vee \neg burglary$, $alarm \vee \neg earthquake$ and $\neg alarm \vee burglary \vee earthquake$. The last clause reflects the CWA.[10]

For cyclic rules, the conversion is more complicated. This holds in particular for rules with *positive loops*, i.e. loops with atoms that depend positively on each other, as in the recursive rule for `smokes/1`. It is well known that in the presence of positive loops, Clark's completion is not correct, i.e. the resulting formula is not equivalent to the rules (Janhunen 2004).

Example 8 (*Simplified Smokers example*)
Let us focus on the Smokers program of Example 6, but restricted to persons p1 and p2.

```
0.2::stress(p1).              0.3::influences(p2,p1).
0.2::stress(p2).              0.3::influences(p1,p2).
smokes(p1) :- stress(p1).
smokes(p1) :- smokes(p2), influences(p2,p1).
smokes(p2) :- stress(p2).
smokes(p2) :- smokes(p1), influences(p1,p2).
```

Clark's completion of the rules for `smokes(p1)` and `smokes(p2)` would result in a formula which has as a model $\{smokes(p1), smokes(p2), \neg stress(p1), \neg stress(p2), influences(p1,p2), influences(p2,p1), \ldots\}$, but this is not a model of the ground ProbLog program: the only model resulting from the total choice $\{\neg stress(p1), \neg stress(p2), influences(p1,p2), influences(p2,p1), \ldots\}$ is the model in which both $smokes(p1)$ and $smokes(p2)$ are false.

[10] The Alarm example models a Bayesian network for the MARG task. For Bayesian networks, the problem of conversion to a weighted CNF formula has been considered before, and several encodings exist (Sang *et al.* 2005; Darwiche 2009). For ProbLog programs modelling Boolean Bayesian networks, such as Alarm, our CNF encoding coincides with that of Sang *et al.* (2005).

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Since Clark's completion is inapplicable with positive loops, a range of more sophisticated conversion algorithms have been developed in the LP literature. Since the problem is of a highly technical nature, we are unable to repeat the full details in this paper. Instead, we briefly discuss the two conversion methods that we use in our work and refer to the corresponding literature for more details.

Both conversion algorithms take a set of rules and construct an equivalent formula. The formulas generated by the two algorithms are typically syntactically different because the algorithms introduce a set of auxiliary atoms in the formula and these sets might differ. For both algorithms, the size of the formula typically increases with the number of positive loops in the rules. Following are the two algorithms.

- The first algorithm is from the ASP literature (Janhunen 2004). It first rewrites the given rules into an equivalent set of rules without positive loops (all resulting loops involve negation). This requires the introduction of auxiliary atoms and rules. Since the resulting rules are free of positive loops, these can be converted by taking Clark's completion. The result can then be written as a CNF. This algorithm is *rule-based*, as opposed to the next algorithm.
- The second algorithm was introduced in the LP literature (Mantadelis and Janssens 2010) and is *proof-based*. It first constructs all proofs of all the atoms of interest, in our case all atoms in $\mathbf{Q} \cup \mathbf{E}$, using the tabled SLD resolution. The proofs are collected in a recursive structure, namely, a set of nested tries (Mantadelis and Janssens 2010), which will have loops if the given rules had loops. The algorithm then operates on this structure in order to 'break' the loops and obtain an equivalent Boolean formula. This formula can then be written as a CNF.

Both rule- and proof-based conversion algorithms return a formula that is 'equivalent' to the rules in L_g in the sense of the following lemma.

Lemma 1
Let L_g be a ground ProbLog program. Let φ_r denote the formula derived from the rules in L_g. Then $SAT(\varphi_r) = MOD(L_g)$.

Recall that $MOD(L_g)$ denotes the set of models of a ProbLog program L_g, as defined in Section 3.2. On the formula side, we use $SAT(\varphi_r)$ to denote the set of models of a formula φ_r.[11]

Example 9 (*Boolean formula for the simplified Smokers example*)
Consider a ground program for the simplified Smokers example given in Example 8. The proof-based conversion algorithm converts the ground rules in this program to

[11] Both conversions for cyclic rules introduce additional or 'auxiliary' atoms into φ_r. We can safely omit these atoms from the models in $SAT(\varphi_r)$ because both conversions are 'faithful', so the truth value of auxiliary atoms is uniquely defined by the truth value of original atoms. This means that the introduction of auxiliary atoms does not create extra models. Hence, w.r.t. the original atoms we have the stated equivalence: $SAT(\varphi_r) = MOD(L_g)$; w.r.t. all atoms, φ_r and L_g are equisatisfiable.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

an equivalent formula (in the sense of Lemma 1) consisting of the conjunction of the following four sub-formulas.

$$smokes(p1) \leftrightarrow aux1 \vee stress(p1)$$
$$smokes(p2) \leftrightarrow aux2 \vee stress(p2)$$
$$aux1 \leftrightarrow smokes(p2) \wedge influences(p2, p1)$$
$$aux2 \leftrightarrow stress(p1) \wedge influences(p1, p2)$$

Here *aux1* and *aux2* are auxiliary atoms that are introduced by the conversion (although they could be avoided in this case). Intuitively, *aux1* says that person $p1$ started smoking because he is influenced by person $p2$, who smokes himself. Note that while the ground program (in Example 8) is cyclic, the loop has been broken by the conversion process; this surfaces in the fact that the last sub-formula uses $stress(p1)$ instead of $smokes(p1)$.

5.3 The weighted Boolean formula

The final step of the conversion constructs the weighted Boolean formula starting from the Boolean formula for the rules φ_r. First, the formula φ is defined as the conjunction of φ_r and the formula φ_e capturing the evidence $\mathbf{E} = \mathbf{e}$. Here φ_e is a conjunction of unit clauses: there is a unit clause a for each true atom and a clause $\neg a$ for each false atom in the evidence. Second, we define the weight function for all literals in the resulting formula. The weight of a *probabilistic literal* is derived from the probabilistic facts in the program: If the RGP contains a probabilistic fact `p::f`, then we assign weight p to f and weight $1 - p$ to $\neg f$. The weight of a *derived literal* (a literal not occurring in a probabilistic fact) is always 1. The *weight of a world* ω, denoted as $w(\omega)$, is defined to be the product of the weight of all literals in ω.

Example 10 (*Weighted formula for Alarm*)
We have seen the formula for the Alarm program in Example 7. If we have evidence that $calls(john)$ is true, we add a positive unit clause $calls(john)$ to this formula (after doing this, we can potentially apply unit propagation to simplify the formula). Then we define the weight function. The formula contains three probabilistic atoms `burglary`, `earthquake` and `hears_alarm(john)`. The other atoms in the formula, `alarm` and `calls(john)`, are the derived atoms. Hence, the weight function is as follows.

$burglary \mapsto 0.1$	$\neg burglary \mapsto 0.9$
$earthquake \mapsto 0.2$	$\neg earthquake \mapsto 0.8$
$hears_alarm(john) \mapsto 0.7$	$\neg hears_alarm(john) \mapsto 0.3$
$alarm \mapsto 1$	$\neg alarm \mapsto 1$
$calls(john) \mapsto 1$	$\neg calls(john) \mapsto 1$

We have now seen how to construct the entire weighted formula from the RGP. The following theorem states that this weighted formula is equivalent – in a particular sense – to the RGP. We will make use of this result when performing inference on the weighted formula.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Theorem 2
Let L_g be the RGP for some ProbLog program with respect to $\mathbf{Q}$ and $\mathbf{E} = \mathbf{e}$. Let $MOD_{\mathbf{E}=\mathbf{e}}(L_g)$ be those models in $MOD(L_g)$ that are consistent with the evidence $\mathbf{E} = \mathbf{e}$. Let φ denote the formula and $w(.)$ the weight function of the weighted formula derived from L_g. Then:

- **(model equivalence)** $SAT(\varphi) = MOD_{\mathbf{E}=\mathbf{e}}(L_g)$,
- **(weight equivalence)** $\forall \omega \in SAT(\varphi)$: $w(\omega) = P_{L_g}(\omega)$, i.e. the weight of ω according to $w(.)$ is equal to the probability of ω according to L_g.

Note the relationship with above-mentioned Lemma 1: Lemma 1 applies to the formula φ_r *prior* to asserting the evidence, whereas Theorem 2 applies to the formula φ *after* asserting evidence.

Example 11 (*Equivalence of weighted formula and ground program*)
The ground Alarm program of Example 5 has three probabilistic facts and hence $2^3 = 8$ total choices and corresponding possible worlds. Three of these possible worlds are consistent with the evidence $calls(john) = true$, namely, the worlds resulting from choices in which $hears_alarm(john)$ is always true and at least one of $\{burglary, earthquake\}$ is true. The reader can verify that the Boolean formula constructed in Example 10 has exactly the same three models, and that weight equivalence holds for each of these models.

There is also a link between the weighted formula and MLNs. Readers unfamiliar with MLNs can consult Appendix B given online. The weighted formula that we construct can be regarded as a ground MLN. The MLN contains the Boolean formula as a 'hard' formula (with infinite weight). The MLN also has two weighted unit clauses per probabilistic atom: for a probabilistic atom a and weight function $\{a \mapsto p, \neg a \mapsto 1-p\}$, the MLN contains a unit clause a with weight $\ln(p)$ and a unit clause $\neg a$ with weight $\ln(1-p)$.[12]

Example 12 (*MLN for the Alarm example*)
The Boolean formula φ for our 'Alarm' running example was shown in Example 5. The corresponding MLN contains this formula as a hard formula. The MLN also contains the following six weighted unit clauses.

$\ln(0.1)$ *burglary* $\qquad \ln(0.9)$ $\neg$*burglary*
$\ln(0.2)$ *earthquake* $\qquad \ln(0.8)$ $\neg$*earthquake*
$\ln(0.7)$ $hears_alarm(john)$ $\qquad \ln(0.3)$ $\neg hears_alarm(john)$

We have the following equivalence result.

Theorem 3
Let L_g be the RGP for some ProbLog program with respect to $\mathbf{Q}$ and $\mathbf{E} = \mathbf{e}$. Let $\mathscr{M}$ be the corresponding ground MLN. The distribution $P(\mathbf{Q})$ according to $\mathscr{M}$ is the same as the distribution $P(\mathbf{Q} \mid \mathbf{E} = \mathbf{e})$ according to L_g.

Note that for the MLN we consider the distribution $P(\mathbf{Q})$ (not conditioned on the evidence). This is because the evidence is already hard-coded in the MLN.

[12] The values of the logarithms (and hence the weights) are negative, but any MLN with negative weights can be rewritten into an equivalent MLN with only positive weights (Domingos *et al.* 2008).

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

6 Inference on the weighted formula

To solve the given inference task for the probabilistic logic program L, query $\mathbf{Q}$ and evidence $\mathbf{E} = \mathbf{e}$, we have converted the program to a weighted Boolean formula. A key advantage is that the inference task (be it MARG, MPE or EVID) can now be reformulated in terms of well-known tasks such as WMC or weighted MAX-SAT on the weighted formula. This implies that we can use any of the existing state-of-the-art algorithms for solving these tasks. In other words, by the conversion of ProbLog to weighted formula, we get the inference algorithms for free.

6.1 Task 1: Computing the probability of evidence (EVID)

Computing the probability of evidence reduces to WMC, a well-studied task in the SAT community. Model counting for a propositional formula is the task of computing the number of models of the formula. WMC is the generalization where every model has a weight and the task is to compute the sum of weights of all models. The fact that computing the probability of evidence $P(\mathbf{E} = \mathbf{e})$ reduces to WMC on our weighted formula can be seen as follows:

$$P(\mathbf{E} = \mathbf{e}) = \sum_{\omega \in MOD_{\mathbf{E}=\mathbf{e}}(L)} P_L(\omega) = \sum_{\omega \in SAT(\varphi)} w(\omega)$$

The first equality holds because $P(\mathbf{E} = \mathbf{e})$ by definition equals the total probability of all worlds consistent with the evidence. The second equality follows from Theorem 2: model equivalence implies that the sets over which the sums range are equal, weight equivalence implies that the summed terms are equal. Computing $\sum_{\omega \in SAT(\varphi)} w(\omega)$ is exactly what WMC on the weighted formula φ does. It is well known that inference with the Bayesian networks can be solved using WMC (Sang *et al.* 2005). In Fierens *et al.* (2011), we were the first to point out that this also holds for inference with probabilistic logic programs. As we will see in the experiments, this approach improves upon state-of-the-art methods in PLP.

The above leaves open as to *how* we solve the WMC problem. There exist many approaches to WMC, both exact (Darwiche 2004) and approximate (Gomes *et al.* 2007). An approach that is particularly useful in our context is that of *knowledge compilation*, ‘compiling’ the weighted formula into a more ‘efficient’ form. While knowledge compilation has been studied for many different tasks (Darwiche and Marquis 2002), we need a form that allows for efficient WMC. Concretely, we compile the weighted formula into a so-called *arithmetic circuit* (Darwiche 2009), which is closely linked to the concept of deterministic, decomposable negation normal form (d-DNNF) (Darwiche 2004).

6.1.1 Compilation into an arithmetic circuit via d-DNNF

We now introduce the necessary background on knowledge compilation and illustrate the approach with an example.

Knowledge compilation is concerned with compiling a logical formula, for which a certain family of inference tasks is hard to compute, into a representation where

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

the same tasks are tractable (so the complexity of the problem is shifted to the compilation phase). In this case, the hard task is to compute weighted model counts (which is #P-complete in general). After compiling a logical formula into a d-DNNF circuit representation (Darwiche 2004) and converting the d-DNNF into an arithmetic circuit, the weighted model count of the formula can efficiently be computed, conditioned on any set of evidence. This allows us to compile a single d-DNNF circuit and evaluate all marginals efficiently using this circuit.

A negation normal form (NNF) formula is a rooted directed acyclic graph in which each leaf node is labelled with a literal, and each internal node is labelled with a conjunction or disjunction. A decomposable negation normal form (DNNF) is a NNF satisfying *decomposability*: for every conjunction node, it should hold that no two children of the node share any atom with each other. A d-DNNF is a DNNF satisfying *determinism*: for every disjunction node, all children should represent formulas that are logically inconsistent with each other. For WMC, we need a d-DNNF that also satisfies *smoothness*: for every disjunction node, all children should use exactly the same set of atoms. The reason for this can be found in Appendix C given online. Compiling a Boolean formula to a (smooth) d-DNNF is a well-studied problem, and several compilers are available (Darwiche 2004; Muise *et al.* 2012). These circuits are the most compact circuit language we know of today that supports tractable WMC (Darwiche and Marquis 2002).

A d-DNNF is a purely logical construct. It is constructed by compiling the formula, irrespective of the associated weighting function. Hence, a d-DNNF allows for model counting, but not for WMC. In order to do WMC, we need to convert the d-DNNF into an arithmetic circuit by taking into account the weighting function of our weighted formula. This conversion is done in two steps (Darwiche 2009): (1) replace all conjunctions in the internal nodes by multiplications, and all disjunctions by summations, and (2) replace every leaf node involving a literal l by a subtree consisting of a multiplication node having two children, namely, a leaf node with an *indicator variable* for the literal l and a leaf node with the weight of l according the weighted formula. We now illustrate this for the Alarm example.

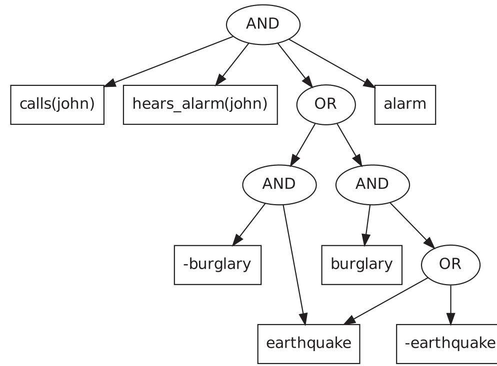
Example 13 (*d-DNNF and arithmetic circuit for the Alarm example*)
We continue the Alarm example (Example 10). The formula for this example, under the evidence $calls(john) = true$, is the conjunction of the following three sub-formulas:

$$
\begin{aligned}
&alarm \leftrightarrow burglary \vee earthquake\\
&calls(john) \leftrightarrow alarm, hears_alarm(john)\\
&calls(john)
\end{aligned}
$$

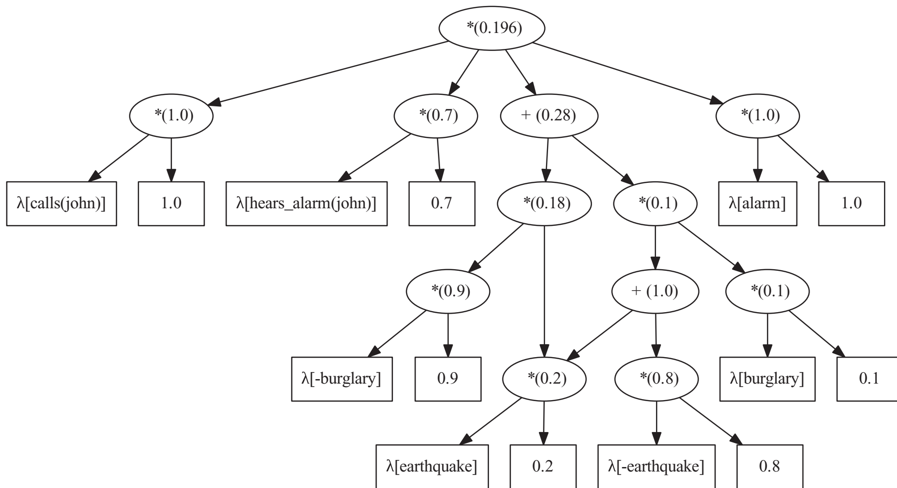
A corresponding d-DNNF is shown in Figure 1(a). Note that the AND-nodes in the d-DNNF (similar to the root note) indeed satisfy the property of decomposability; while the OR-nodes satisfy determinism. The function of the OR-node on the lower right is to make the d-DNNF smooth.

The arithmetic circuit corresponding to this d-DNNF is shown in Figure 1(b). The values in parentheses in the internal nodes will be used later and can be ignored

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press



(a) d-DNNF



(b) Arithmetic circuit

Fig. 1. The d-DNNF for the Alarm example and the corresponding arithmetic circuit.

for now. The λ-variables in the leaves are the indicator variables for the literals. The indicator variable for the literal l is multiplied with a number, which is the weight of l according to our weighting function.

Now that we have an arithmetic circuit for our weighted formula, we are ready to perform WMC and compute the weighted model count $\sum_{\omega \in SAT(\varphi)} w(\omega)$. This count is found by simply *evaluating* the arithmetic circuit: We instantiate all indicator variables to the value 1 and then bottom-up evaluate all nodes until we arrive at the root node. The value found at the root is the desired weighted model count and also equals the probability of the evidence $P(\mathbf{E} = \mathbf{e})$.

Example 14 (*Evaluating the arithmetic circuit for the Alarm example*)

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

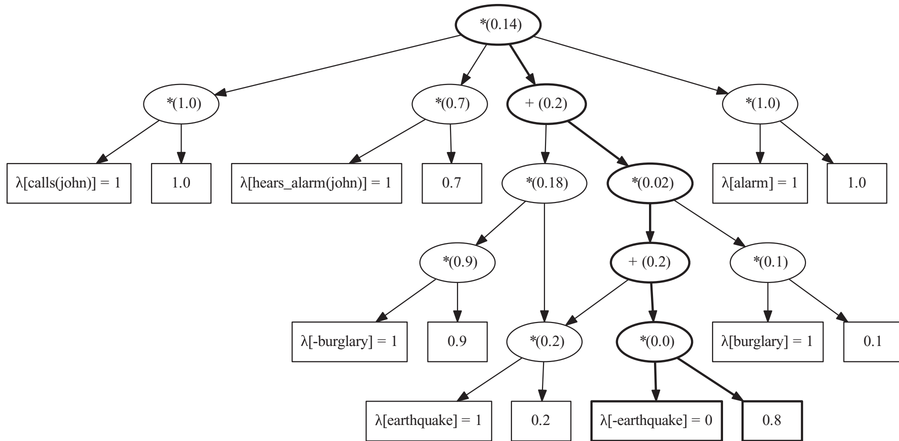


Fig. 2. Evaluating an arithmetic circuit with additional evidence (nodes that get different values than in Figure 1(b) are in boldface).

We use the arithmetic circuit for the Alarm program given in Example 13. Recall that this program and circuit were obtained using $calls(john) = true$ as the evidence, so we can use this circuit to calculate the probability of evidence $P(calls(john) = true)$. This is done by instantiating all indicator variables λ to 1, and then evaluting the circuit. Figure 1(b) illustrates this: the obtained values in each node are given in parentheses. The value for the root is 0.196. This is the probability of evidence.

The above does not explain why we really need indicator variables. The indicator variables allow us to add further evidence, on top of $\mathbf{E} = \mathbf{e}$, which is useful for MARG inference, as we will see later. For instance, we can compute $P(\mathbf{E} = \mathbf{e} \wedge X = true)$ for some additional atom X in the arithmetic circuit by setting the indicator variable $\lambda[X]$ to 1 and $\lambda[-X]$ to 0 when evaluating the circuit.[13]

Example 15 (*Evaluating the arithmetic circuit in case of additional evidence*)
Assume we want to compute $P(calls(john) = true \wedge earthquake = true)$ using the same arithmetic circuit seen before, namely, the circuit for $calls(john) = true$. Since we additionally have $earthquake = true$, we set $\lambda[earthquake]$ to 1, $\lambda[-earthquake]$ to 0, and all other indicator variables to 1 as before. The evaluation is illustrated in Figure 2, yielding the result 0.14. Hence, $P(calls(john) = true \wedge earthquake = true) = 0.14$.

In the same way, the probability of any set of evidence can be computed, provided that this set extends the initial set $\mathbf{E} = \mathbf{e}$ (and that the additional atoms also appear in the compiled circuit). This also means that Step 3 of our conversion algorithm (Section 5.3), where we add the evidence φ_e to the weighted Boolean formula, is not strictly needed: We can achieve the same result by using only the formula φ_r (capturing the rules of the program) and setting the indicator variables in the circuit

[13] In a purely logical context, setting indicator variables to 0 corresponds to *conditioning* the d-DNNF circuit.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

according to the evidence $\mathbf{E} = \mathbf{e}$. However, asserting the evidence φ_e early makes the compilation phase more efficient (it allows for more unit propagation etc).

In SRL, the work of Chavira *et al.* (2006) is closest to the approach given in this section. They perform inference in relational Bayesian networks by encoding them into a weighted Boolean formula and compiling this formula into an arithmetic circuit. The main difference is that relational Bayesian networks are not a programming language and assume acyclicity. That assumption greatly simplifies the step of converting to a weighted Boolean formula (cf. Section 5).

In summary, to compute the probability of evidence we (1) compile the formula to a d-DNNF, (2) convert the d-DNNF into an arithmetic circuit and (3) evaluate the arithmetic circuit.

6.1.2 Compilation into an arithmetic circuit via BDD

In the PLP community, the state of the art (De Raedt *et al.* 2007) is to compile the program into another form, namely, a reduced ordered BDD (Bryant 1986). This approach is a special case of our above WMC approach (although it is usually not formulated similar to that; in fact, in Fierens *et al.* (2011) we were the first to point out the connection of the PLP–BDD approach to WMC).

A BDD is a special kind of d-DNNF, namely, one that satisfies the additional properties of *ordering* and *decision*, see Darwiche (2004). In our approach, we can alternatively replace a d-DNNF compiler with a BDD compiler. Computing the probability of evidence can then be done by either operating directly on the BDD, or by converting the BDD to an arithmetic circuit and evaluating the circuit (the first approach is merely a reformulation of the second). So, while compilation into both BDD and d-DNNF are possible, there is theoretical and empirical evidence in the model counting literature that d-DNNFs outperform BDDs (Darwiche 2004). Our experimental results confirm the superiority of d-DNNFs (Section 9).

We have now seen two ways of computing the probability of evidence: via d-DNNFs, or via BDDs. We will now see how this approach of computing the probability of evidence can be used as a building block for the MARG inference task (as is standard in the probabilistic literature).

6.2 Task 2: Computing marginal probabilities (MARG)

In MARG, we are given a set of query atoms $\mathbf{Q}$ and for each $Q \in \mathbf{Q}$ we need to compute $P(Q \mid \mathbf{E} = \mathbf{e})$. By definition, $P(Q \mid \mathbf{E} = \mathbf{e}) = \frac{P(Q \wedge \mathbf{E}=\mathbf{e})}{P(\mathbf{E}=\mathbf{e})}$. Hence, if we have N atoms in the query set $\mathbf{Q}$, solving MARG reduces to computing the probability of the evidence, and computing N probabilities of the form $P(Q \wedge \mathbf{E} = \mathbf{e})$, i.e. the probability of the conjunction of the evidence with a single atom. In the previous section, we have already seen how we can compute such probabilities from the compiled arithmetic circuit by appropriately instantiating the indicator variables λ and evaluating the circuit. The simplest approach is to apply this one for each query atom $Q \in \mathbf{Q}$ separately. However, we can solve this even more efficiently. Concretely, all required probabilities can be found *in parallel*. To be precise, all

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

probabilities of the form $P(X \wedge \mathbf{E} = \mathbf{e})$, with X being any atom in the circuit, can be computed simultaneously by traversing the circuit twice (bottom-up and top-down). The required traversal algorithm can be found in the literature, see Algorithms 34 (simple version) and 35 (optimized version) in Darwiche (2009). From this we obtain all probabilities of the form $P(X \wedge \mathbf{E} = \mathbf{e})$. We then retain those that involve an atom from the query set ($X \in \mathbf{Q}$) and compute the required conditional probabilities $P(Q \mid \mathbf{E} = \mathbf{e})$ as $\frac{P(Q \wedge \mathbf{E}=\mathbf{e})}{P(\mathbf{E}=\mathbf{e})}$. As in the previous section, this entire approach can be performed using an arithmetic circuit derived from a compiled d-DNNF or a BDD.

The knowledge compilation approach is typically used for exact inference. When dealing with large domains, we often need to resort to computing *approximate* marginals. Approximate inference is often achieved by means of sampling techniques, such as Markov Chain Monte Carlo (MCMC). Standard MCMC approaches similar to Gibbs sampling cannot deal with weighted formulas because the formula itself is deterministic. Instead, we use the *MC-SAT* algorithm, developed specifically to deal with determinism (Poon and Domingos 2006). MC-SAT is an MCMC algorithm that in every step of the Markov chain calls a SAT solver to construct a new sample. MC-SAT takes an MLN as input. Theorem 3 ensures that if we apply MC-SAT on the appropriate MLN, we indeed obtain samples from the distribution $P(\mathbf{Q} \mid \mathbf{E} = \mathbf{e})$.

To summarize, we currently have three methods for the MARG task: exact inference by compilation of (1) d-DNNFs or (2) BDDs or (3) approximate inference with MC-SAT.

6.3 Task 3: Finding the most likely explanation (MPE)

MPE is a task of finding the most similar interpretation (joint state) of all unobserved atoms, given the evidence, i.e. finding $argmax_{\mathbf{u}} P(\mathbf{U} = \mathbf{u} \mid \mathbf{E} = \mathbf{e})$, with $\mathbf{U}$ all unobserved atoms (i.e. all atoms in the ground program that are not in $\mathbf{E}$). MPE inference on weighted formulas has been studied before. We consider two approaches.

The first approach is to perform MPE by means of *knowledge compilation*. The compilation step (to compile an arithmetic circuit via a d-DNNF or a BDD) is the same as before, only the traversal step differs.[14] Again, the traversal algorithm can be found in the literature, see Algorithm 36 in Darwiche (2009). This yields the exact MPE solution.[15]

The second approach is to perform MPE using techniques from the SAT solving community. Concretely, it is known that MPE reduces to partially weighted MAX-SAT (Park 2002). A popular approximate approach for solving this task is the

[14] For the MPE task, it is sufficient to compile into a DNNF circuit, which is not necessarily deterministic. DNNF circuits are potentially more succinct than d-DNNF circuits, but unfortunately there exist no compilers specifically for DNNF.

[15] This approach yields the truth value of all ground atoms that occur in the RGP for a given evidence. All probabilistic atoms that do not occur in the RGP are irrelevant w.r.t. the evidence (i.e. they are probabilistically independent of the evidence). Hence, for each of these atoms, we can simply independently chose the truth value with maximum probability according to the associated probabilistic fact. The truth value of all *derived atoms* that do not occur in the RGP is then found by computing the well-founded model of the MPE total choice.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

stochastic local search (Park 2002). An example algorithm is *MaxWalkSAT*, which is also the standard MPE algorithm for MLNs (Domingos *et al.* 2008).

Since our current ProbLog implementation focusses on MARG inference rather than MPE, we do not discuss these approaches in detail and will not consider these further in this paper.

7 Learning probabilistic logic programs from partial interpretations

We now present an algorithm for learning the parameters (the probabilities of probabilistic facts) of a ProbLog program from data. We use the LFI setting.

7.1 The learning setting

Learning from (possibly partial) interpretations is a common setting in SRL, which has not been studied so far in its full generality for probabilistic programming languages (but see also Gutmann *et al.* 2011).

In the terminology used for inference in Section 4, partial interpretations correspond to evidence, and hence in this section we shall often use the term evidence instead of partial interpretation. Let $\mathbf{At}$ be the Herbrand base, i.e. the set of all ground (probabilistic and derived) atoms in a given ProbLog program. In the fully observable case, we learn from a set of complete interpretations that is, the observed truth-values $\mathbf{e}$ of all the atoms in the Herbrand base $\mathbf{At}$ are given and the evidence variables $\mathbf{E}$ coincide with $\mathbf{At}$. On the other hand, in the partially observable case, we learn from a set of partial interpretations that we only observe the truth-values $\mathbf{e}$ of a set $\mathbf{E} \subset \mathbf{At}$ of observed atoms. We now develop an algorithm, called LFI-ProbLog, that learns from (possibly partial) interpretations of a ProbLog program. In a generative setting, one is typically interested in the maximum likelihood parameters given in the training data. This can be formalized as follows.

Given:

- a ProbLog program $T_\mathbf{p}$ containing a set of rules R and a set of probabilistic facts $F = \{p_i :: f_i\}$ with *unknown* parameters $\mathbf{p} = \langle p_1, \ldots, p_N \rangle$
- a set of (possibly partial) interpretations $D = \{\mathbf{E_1} = \mathbf{e_1}, \ldots, \mathbf{E_M} = \mathbf{e_M}\}$ (the training examples)

Find: the maximum likelihood probabilities $\widehat{\mathbf{p}} = \langle \widehat{p_1}, \ldots, \widehat{p_N} \rangle$, that is,

$$\widehat{\mathbf{p}} = \arg\max_{\mathbf{p}} P_{T_\mathbf{p}}(D) = \arg\max_{\mathbf{p}} \prod_{m=1}^{M} P_{T_\mathbf{p}}(\mathbf{E_m} = \mathbf{e_m})$$

where $P_{T_\mathbf{p}}(\mathbf{E_m} = \mathbf{e_m})$ is the probability of evidence $\mathbf{E_m} = \mathbf{e_m}$ in the ProbLog program $T_\mathbf{p}$ with parameters $\mathbf{p}$.

Example 16 illustrates the LFI setting using the Alarm program from Example 1.

Example 16 (*Learning from interpretations*)

```
P1::burglary.                      person(mary).        alarm:- burglary.
P2::earthquake.                    person(john).        alarm:- earthquake.
P3::hears_alarm(X):- person(X).    calls(X) :- alarm, hears_alarm(X).
```

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

A ProbLog program is given in which the probabilities `P1`, `P2` and `P3` are unknown and should be learned from partial interpretations, which contain the truth value for some of the atoms: $\{alarm = true\}, \{earthquake = true, calls(mary) = true\}, \{calls(john) = true\}$. The goal is to find the probabilities `P1`, `P2` and `P3` such that the combined probability of the partial interpretations is maximal.

One has to consider two cases when computing the maximum likelihood parameters $\widehat{\mathbf{p}}$. In the fully observable case where the truth values for each of the atoms in the Herbrand base is known, one can obtain $\widehat{\mathbf{p}}$ by counting. In the more complex case of partial interpretations, one has to use an approach such as expectation-maximization to deal with the partial observability.

7.2 Full observability

In the fully observable case, the maximum likelihood estimate $\widehat{p_n}$ for a probabilistic fact $p_n :: f_n$ can be calculated directly from the interpretations. When $p_n :: f_n$ is intensional, it represents multiple ground instances, that is, probabilistic facts: $p_n :: f_{n,1}, \ldots, p_n :: f_{n,K_n^m}$ where K_n^m is the number of ground instances represented by the fact $p_n :: f_n$ in interpretation $\mathbf{E_m} = \mathbf{e_m}$. When $p_n :: f_n$ is ground and extensional, K_n^m is equal to 1 and the fact represents itself only. The maximal likelihood estimates can be calculated using the following formula:

$$\widehat{p_n} = \frac{1}{Z_n} \sum_{m=1}^{M} \sum_{k=1}^{K_n^m} \delta_{n,k}^m \text{ where } \delta_{n,k}^m = \begin{cases} 1 & \text{if } f_{n,k} = true \in \mathbf{E_m} = \mathbf{e_m} \\ 0 & \text{if } f_{n,k} = false \in \mathbf{E_m} = \mathbf{e_m} \end{cases} \quad (1)$$

The sum is normalized by $Z_n = \sum_{m=1}^{M} K_n^m$, the total number of probabilistic facts represented by f_n in all training examples. When Z_n is 0 in the data, $\widehat{p_n}$ is not calculated (there is no data to estimate it from).

7.3 Partial observability

In many applications, the training examples are observed only partially. In the Alarm example, we may receive a phone call but we may not know whether an earthquake has occurred. In the partially-observable case – similar to the Bayesian networks – it is impossible to compute the maximum likelihood estimates in closed form. Instead, we use expectation-maximization, see Algorithm 1. In this algorithm, the parameters p_n^0 are initialized randomly. During each iteration i, the ProbLog program $T_{\mathbf{p}^i}$ with parameters $\mathbf{p}^i$ is used to estimate the probability of the unobserved atoms being true in each interpretation, $P_{T_{\mathbf{p}^i}}(f_{n,k}|\mathbf{E_m} = \mathbf{e_m})$ (the expectation step). These expectations are then used as to update the parameters of the program using the following equation (the maximization step):

$$p_n^{i+1} = \frac{1}{Z_n} \sum_{m=1}^{M} \sum_{k=1}^{K_n^m} P_{T_{\mathbf{p}^i}}(f_{n,k}|\mathbf{E_m} = \mathbf{e_m}) \quad (2)$$

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Algorithm 1 uses the inference mechanism described in Section 6.2 for computing marginals in the expectation step. We can make two optimizations. Firstly, for the facts $f_{n,k}$ that are not contained in the dependency set of a partial interpretation $\mathbf{E_m} = \mathbf{e_m}$, the probability $P_{T_{\mathbf{p}^i}}(f_{n,k}|\mathbf{E_m} = \mathbf{e_m})$ is equal to p_n^i. These facts slow down the updating process and should therefore not be included in the sum. This can be realized by compiling the d-DNNF for the query $P_{T_{\mathbf{p}^i}}(\mathbf{E_m} = \mathbf{e_m})$ and to use the resulting d-DNNFs to compute the marginal probabilities $P_{T_{\mathbf{p}^i}}(f_{n,k}|\mathbf{E_m} = \mathbf{e_m})$ only for those facts $f_{n,k}$ that are included in the d-DNNF. For example, when we compile the d-DNNF for the third partial interpretation of Example 16, we obtain the ground program from Example 5 and the d-DNNF from Figure 1(a). This d-DNNF does not contain `calls(mary)`, so this atom will not be used to update the probabilities for the third partial interpretation. When no groundings for a learnable fact are present in any of the d-DNNFs, a zero probability is learned as no information is given. Secondly, one can observe that changing the parameters of a ProbLog program does not change the structure of compiled d-DNNFs. This means that the d-DNNFs that have been compiled in the first iteration can be reused in all further iterations. The algorithm keeps on updating the parameters until the log likelihood of the interpretations is maximal. Each iteration of the algorithm is guaranteed to improve the likelihood of the data.

Algorithm 1 The main loop of LFI-ProbLog. The ProbLog program is compiled into a d-DNNF for each partial interpretation $\mathbf{E_m} = \mathbf{e_m}$. After the compilation step, the algorithm follows an EM update scheme, first using the current model to complete the data and then estimating the new model parameters from the resulting counts until convergence.

1: **function** LFI-PROBLOG($T = \{p_1 :: f_1, \ldots, p_N :: f_N\} \cup R, D = \{\mathbf{E_1} = \mathbf{e_1}, \ldots, \mathbf{E_M} = \mathbf{e_M}\}$)
2:     **for** $1 \leqslant n \leqslant N$ **do**
3:         $p_n^0 \leftarrow rand(0,1)$ ▷ The fact probabilities are initialized with a random probability
4:     **for** $1 \leqslant m \leqslant M$ **do** ▷ Loop over training examples
5:         d-DNNF$_m \leftarrow$ COMPILE($P_{T_0}(\mathbf{E_m} = \mathbf{e_m})$)
6:     $i \leftarrow 0$
7:     **while** not converged **do** ▷ EM algorithm
8:         $i \leftarrow i + 1$
9:         **for** $1 \leqslant m \leqslant M$ **do**
10:             **for** $1 \leqslant n \leqslant N$ **do**
11:                 **for** $1 \leqslant k \leqslant K_n^m$ **do**
12:                     compute $P_{T_{i-1}}(f_{n,k}|\mathbf{E_m})$ using d-DNNF$_m$ ▷ E Step
13:         **for** $1 \leqslant n \leqslant N$ **do** ▷ Loop over probabilistic facts
14:             $p_n^i \leftarrow \frac{1}{Z_n} \sum_{m=1}^{M} \sum_{k=1}^{K_n^m} P_{T_{i-1}}(f_{n,k}|\mathbf{E_m})$ ▷ M Step (cf. Eq. 2)
15:     **return** $\{p_n^i :: f_n \mid f_n \in F\} \cup R$

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

The learning algorithm uses a black box for the MARG inference task (line 12). In principle, any inference algorithm will work, including approximate ones. However, by choosing knowledge compilation for inference, we need to compile a circuit only once for each training example. This is a hard task. Once we have a circuit, computing expectations becomes easy, and we can reuse the circuit many times for all i, k and n in lines 6, 10 and 11 of Algorithm 1. Furthermore, all marginal probabilities $P_{T_{\mathbf{p}}}(f_{n,k}|\mathbf{E_m})$ for the same evidence set $\mathbf{E_m}$ and parameterization $\mathbf{p}$ can be computed at once in only two passes over the d-DNNF circuit.

7.4 Discussion

The above learning algorithm is based on the LFI algorithm that we developed in an earlier work (Gutmann *et al.* 2011), see the discussion in Section 1. Our new algorithm has some advantages over the earlier version. First, the new algorithm can deal with cyclic programs (the old algorithm uses Clark's completion, which applies only to acyclic programs, cf. Section 5.2). Second, the new algorithm scales better as it employs a more efficient approach for inference in the expectation step, namely, compilation of d-DNNFs instead of BDDs (see the experiments in Section 9.3.3). Furthermore, the new description of the algorithm more clearly separates the learning from the inference steps.

The complexity of parameter learning (and of MARG and MPE inference) is the worst case exponential in the *treewidth* (Robertson and Seymour 1986) of the weighted Boolean formula when using knowledge compilation of d-DNNF (Darwiche 2001). This theoretical complexity bound is in line with the complexity of classical algorithms for inference and learning in probabilistic graphical models. For example, hidden Markov models have a constant treewidth in terms of the number of time steps considered. Learning the parameters of these models is linear in the number of time steps, both when using LFI-ProbLog with d-DNNF compilation and, for example, expectation-maximization with the classical junction tree algorithm. These bounds assume that both algorithms succeed at finding the optimal tree decomposition of the model, which itself is a hard task in theory. In practice, however, there exist heuristics that can find good tree decompositions of many different kinds of models.

8 Implementation of the system ProbLog2

The first ProbLog system (Kimmig *et al.* 2010) focused on the inference task of computing the success probability of a single atom (Section 4) and on learning from entailment (Gutmann *et al.* 2008b). ProbLog2, the new ProbLog system described in this paper, focusses on different tasks, namely, computing marginal probabilities and the probability of evidence, as well as learning from interpretations. This new setting is closer in spirit to the work on graphical models and SRL (such as Markov Logic). As a consequence of this new emphasis, the design of ProbLog2 is quite different from that of the first ProbLog. The implementation of the first ProbLog was tightly integrated in the YAP Prolog (Kimmig *et al.* 2010). In contrast, ProbLog2 consists of

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

a number of relatively loosely coupled components, and involves almost no Prolog code. This new design is closer in spirit to some ASP systems than to Prolog.

We now briefly discuss the different components of the implementation. Most of these components are existing state-of-the-art programs rather than being tailor-made for ProbLog.

- The *grounding* component computes the RGP from the given ProbLog program (and query and evidence). This is the only component that is written in (YAP) Prolog. It is essentially a meta-interpreter that collects proofs to construct the dependency set (Section 5.1).
- The *conversion* component converts the rules in the RGP to a Boolean (CNF) formula. The user can choose between the proof-based and the rule-based conversions (Section 5.2). For the proof-based conversion (Mantadelis and Janssens 2010), we use our own implementation. For the rule-based conversion, we use the code of Janhunen (2004), as used in the ASP community.
- The *exact inference* component is based on knowledge compilation and consists of two parts: a compiler and an evaluation algorithm. For compilation of d-DNNF, the user can choose between the 'c2d' compiler by Darwiche (2004) or the more recent 'DSHARP' compiler (Muise *et al.* 2012).[16] For compilation of a BDD, we use CUDD (see `http://vlsi.colorado.edu/~fabio/CUDD/`). For constructing and evaluating the corresponding arithmetic circuit we use our own code.
- The *approximate inference* component converts the weighted formula to an MLN and then uses the MC-SAT (Poon and Domingos 2006) code from the Alchemy package to perform sampling.
- The *learning component*, LFI-ProbLog, builds heavily on the inference component, as explained before (Section 7). It is essentially an EM loop around the inference component.

As mentioned, the above components are relatively loosely coupled. They are bundled into a pipeline by means of the Python code. A major advantage of our design is that it allows to build an entire ProbLog system by (mostly) using the existing state-of-the-art programs for different components such as Janhunen's conversion program and various d-DNNF and BDD compilers. Moreover, research on conversion of logic programs, knowledge compilation, WMC, etc. continues with new tools being released. Whenever a new tool becomes available for a particular component, we can benefit from this and integrate it into our system. Such a design of course also has drawbacks. The two main drawbacks are that there is a certain latency between the components because of I/O issues, and the system is complex to install and configure because of different components written in different programming languages.

ProbLog2 is available on `http://dtai.cs.kuleuven.be/problog/`.[17]

[16] All experiments in this paper use c2d.

[17] In addition to the MARG, MPE and learning tasks, the ProbLog2 system supports MAP and decision-theoretic inference (Van den Broeck *et al.* 2010), which are not discussed here.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

9 Experiments

The goal of our experiments is to establish the feasibility of our approach and to analyze the influence of different parameters. We focus on MARG inference and learning. Concretely, we aim to answer the following six questions:

- **Q1** Is working with the relevant rather than the complete ground program more efficient?
- **Q2** Which of the two considered algorithms for converting the ground program to a Boolean formula (rule-based or proof-based conversion) performs best?
- **Q3** Which of the two considered approaches for knowledge compilation (using d-DNNFs or BDDs) performs best?
- **Q4** When computing success probabilities (the 'classical' ProbLog setting), does ProbLog2 outperform the previous ProbLog implementation?
- **Q5** When learning from the data generated from a known ProbLog program, can we recover the parameters of the original program, given a reasonable amount of data?
- **Q6** When learning from real-world data, can we obtain results comparable to the ones obtained with a state-of-the-art system (namely, Alchemy)?

Note that in **Q6** we compare our system with Alchemy, which is the standard system for Markov Logic (see `http://alchemy.cs.washington.edu/`).

9.1 Programs and datasets

We perform experiments on the following three types of applications:

Social networks. We use the standard *Smokers* application (Domingos *et al.* 2008). The ProbLog program contains the following intensional probabilistic facts and rules.

```
0.2::stress(P) :- person(P).
0.3::influences(P1,P2) :- friend(P1,P2).
0.1::cancer_spont(P) :- person(P).
0.3::cancer_smoke(P) :- person(P).

smokes(X) :- stress(X).
smokes(X) :- smokes(Y), influences(Y,X).
cancer(P) :- cancer_spont(P).
cancer(P) :- smokes(P), cancer_smoke(P).
```

In addition, the program contains ground (non-probabilistic) facts for the predicates *person*/1 and *friend*/2. The number of such facts is varied; see the next section.

Collective classification. We use the relational WebKB dataset.[18] In WebKB, university web pages need to be tagged with classes (such as course page, student page, etc.). This is modelled with a predicate *has_class*(*Page*, *Class*). Following ate the

[18] See `http://www.cs.cmu.edu/~webkb/`.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

rules in the ProbLog program: They specify how the class C of a page P depends on the textual content of P (the words W on the page), and on the classes of pages that link to P.

```
has_class(P,C) :- word_class(P,W,C).
has_class(P,C) :- has_class(P2,C2), link_class(P,P2,C,C2).
```

For the predicate *word_class*/3, there is a different intensional probabilistic fact for every (word,class)-pair in the dataset. Each such intensional probabilistic fact looks as follows.

```
prob::word_class(P,word1,class1) :- has_word(P,word1).
```

The reason why we need a different intensional probabilistic fact for each (word,class) pair is that for every such pair the involved probability (`prob`) can be different. Similarly, for the predicate *link_class*/4, there is one intensional probabilistic fact for every pair of classes in the dataset.

```
prob::link_class(P,P2,class1,class2) :- links_to(P,P2).
```

The predicates that occur in the 'bodies' of these intensional probabilistic facts (*has_word*/2 and *links_to*/2) are defined in the dataset. The probabilities of the probabilistic facts were learned from data using LFI-ProbLog.

Probabilistic grids. For comparing ProbLog2 with the previous ProbLog implementation, we use the classical ProbLog application of probabilistic graphs (De Raedt *et al.* 2007). The program represents a graph in which edges are labelled with a probability. Here we use a 16×16 *grid* as the graph. This consists of nodes $n_{x,y}$, with $x, y \in \{1, \ldots, 16\}$, lined out on a square grid with horizontal, vertical and diagonal directed edges between adjacent nodes. Concretely, following are the edges.

$$\begin{aligned} n_{x,y} &\rightarrow n_{x+1,y} & \forall x \in \{1,\ldots,15\}, y \in \{1,\ldots,16\}\ (\textit{horizontal}) \\ n_{x,y} &\rightarrow n_{x,y+1} & \forall x \in \{1,\ldots,16\}, y \in \{1,\ldots,15\}\ (\textit{vertical}) \\ n_{x,y} &\rightarrow n_{x+1,y+1} & \forall x \in \{1,\ldots,15\}, y \in \{1,\ldots,15\}\ (\textit{diagonal}) \end{aligned}$$

Every edge has probability 0.5. Such a probabilistic graph is modelled in ProbLog by a set of probabilistic *edge*/2 facts. For instance, the horizontal edge from $n_{1,1}$ to $n_{2,1}$ is represented as the probabilistic fact `0.5::edge(n_1_1,n_2_1)`. The goal is to find the probability of there being a path between certain nodes in the graph, where the path is defined in the usual Prolog way.

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

9.2 Experimental setup

We now describe how we use these three programs (Smokers, WebKB and grids) in our experimental setup.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

9.2.1 Inference setup

MARG inference. We test MARG influence on Smokers and WebKB. The main parameter influencing the complexity of our inference experiments is the 'domain size', i.e. the number of constants considered. For Smokers, the domain size is the number of people; for WebKB, it is the number of webpages (we take subsets of all pages that occur in the dataset). In our experiments, we vary the domain size and see how different measures, such as runtime, evolve. For each considered domain size we generate multiple different instances of the MARG task (10 for Smokers, eight for WebKB) as described below. We report median results over these different instances (we use median because it is more stable than arithmetic average).

Given a particular domain size, one instance of the MARG task is generated as follows. (1) For both Smokers and WebKB, the program involves ground (non-probabilistic) facts for certain 'background' predicates. We first generate interpretations for these predicates. For Smokers, the background predicate is *friend*/2, which determines the actual social network. We use a generator of synthetic power law random graphs (since such graphs are known to resemble real social networks) and convert the obtained graph to *friend*/2 facts. For WebKB, the background predicates are *has_word*/2 and *links_to*/2, for which interpretations can be found in the dataset. (2) Given the domains and background facts, we select the set of query and evidence atoms, **Q** and **E**. For Smokers, we use 50% of all *smokes*/1 and *cancer*/1 atoms as evidence and the other *smokes*/1 and *cancer*/1 atoms as queries. All atoms for the other predicates are neither query nor evidence. For WebKB, we have a similar setup: we use 50% of all *has_class*/2 atoms as evidence and the other *has_class*/2 atoms as query. (3) The previous step generates the sets **Q** and **E**, but not yet the values for the evidence atoms, i.e. the vector of truth values **e**. To do so, we generate a 'sample' of the ProbLog program. This is done by independently sampling each ground probabilistic fact, and then computing the well-founded model of the resulting logic program (as dictated by the ProbLog semantics). The result is a complete interpretation of all predicates in the program. From this interpretation, we extract the truth values of all atoms in the evidence set **E**, and we use these truth values to construct vector **e**. (We similarly store the values of all atoms in the query set **Q** because we need them later as 'query ground truth'; see Section 9.3.2).

Special case: Success probability. The above is for MARG inference in the presence of multiple queries and evidence. In addition, we also perform an experiment in the classical *success probability* setting, where the goal is to compute the probability of a single query without evidence (Kimmig *et al.* 2010). For this experiment we use the probabilistic grid program. Per experiment, we ask a single query of the form `path(n_i_i,n_16_16)`, where i is being varied from 1 to 15. In other words, we are asking for the probabibity of there being a path from a node $n_{i,i}$ on the diagonal of the grid to $n_{16,16}$, the lower right corner of the grid. The smaller the value of i, the longer these paths become (and the more possible paths there are), and hence the harder the computation. We measure the effect of the value of i on the runtime of query in both ProbLog2 and the first ProbLog implementation

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

(`http://dtai.cs.kuleuven.be/problog/`). For each value of *i*, we repeat the experiment for 10 times and average the measured runtimes.

9.2.2 Learning setup

In the learning experiments, we estimate the probabilities of all probabilistic facts from data.

For Smokers, we again vary the domain size. For each size we generate 170 experiments. We sample 40, 50,. . . , 200 interpretations, from which we retain 10%, 40%, 70% and 100% of atoms together with their truth value in partial interpretations. From these interpretations, we learn the probabilities for all intensional probabilistic facts in the program (predicates `stress/1`, `influences/2`, `cancer_spont/1` and `cancer_smoke/1`).

For WebKB, the dataset consists of four disjoint sets of webpages, one per university. Per university, we use only 20 words that contain the most information (as measured by information gain with respect to class labels). We perform four-fold cross validation using both Alchemy system (with a standard MLN for this application) and LFI-ProbLog.

The ProbLog program that we use for learning is slightly different from the one we use for inference. In addition to the rules given earlier (Section 9.1), we include in the program two more causes for a page to have a certain class.

```
has_class(P,C) :- fixed_prior(P,C).
has_class(P,C) :- learnable_prior(P,C).
```

The predicate *learnable_prior*/2 accounts for the pages that are tagged with a class that cannot be explained through words and links. There is one such probabilistic fact for each class,

```
prob::learnable_prior(P,class1) :- page(P).
```

The predicate *fixed_prior*/2 makes sure that every page can be tagged with every class,

```
0.001::fixed_prior(P,C) :- page(P), class(C).
```

Finally, for computational reasons, we modify the rule that spreads influence across links (*link_class*/4) such that pages can only influence their direct neighbours.

We learn all `prob`-parameters in the program (not the probability of *fixed_prior*/2). The learned program is too big to perform exact inference. Hence, when evaluating the learned program (which requires running inference), we use an approximation, namely, we remove all probabilistic facts with a learned probability below 0.05.

9.3 Experimental results

We now discuss our results in terms of the six questions raised earlier.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

9.3.1 Q1 – Influence of the grounding algorithm

Question **Q1**: Is working with the relevant rather than the complete ground program more efficient? To answer this question, we measured the time needed for grounding, the size of the resulting ground program and the size of the Boolean formula derived from this program.

The grounding step. The idea behind the RGP is to reduce grounding by pruning clauses that are irrelevant or inactive w.r.t. the queries and evidence. Our setup is such that all clauses are relevant. Hence, the only reduction comes from pruning inactive clauses (that have a false evidence literal in the body). The effect of this pruning is small: on average, the size of the ground program is reduced by 17% (results not shown).

Implications on the conversion to a Boolean formula. The proof-based conversion becomes intractable (i.e. takes prohibitively long) for large domain sizes, but the size where this happens is significantly larger when working on the RGP instead of complete grounding (see Figures 3(a) and 4(a)). Also, the size of the Boolean formula is reduced significantly by using the RGP (up to a 90% reduction in the number of clauses in CNF, Figures 3(b) and 4(b)). The reason why a 17% reduction of the program can yield a 90% reduction in the corresponding formula is that loops in the program cause a 'blow-up' of the formula. Removing only a few rules in the ground program can already break loops and make the formula significantly smaller. Note that the proof-based conversion suffers from this blow-up more than the rule-based conversion.

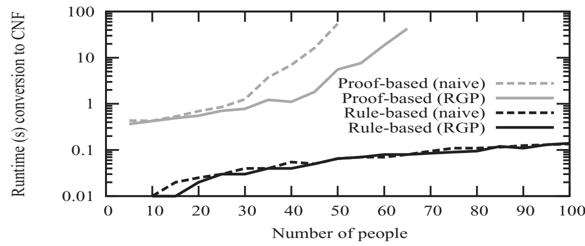
Computing the grounding is always very fast, both for RGP and complete grounding (milliseconds on Smokers; around 1 s for WebKB). Hence, as an answer to question **Q1**, we conclude that using RGP instead of complete grounding is beneficial and comes at almost no computational cost. Hence, from now on we always use RGP.

9.3.2 Q2 – Influence of the conversion algorithm

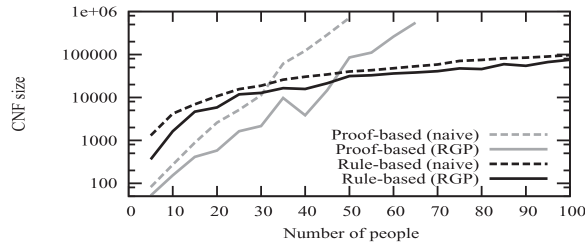
Question **Q2**: Which of the two considered algorithms for converting the ground program to a Boolean formula performs best? Recall that we have seen a rule-based and a proof-based conversion (Section 5.2). To answer this question, we measure the time of the conversion process, the size of the resulting formula and how efficient this formula is for inference.

The conversion step. The proof-based algorithm, by its nature, does more effort to convert the program into a compact formula. This has implication on the scalability of the algorithm: on small domains the algorithm is fast, but on larger domains it becomes intractable (Figures 3(a) and 4(a)). In contrast, the rule-based algorithm is able to deal with all considered domain sizes and is always fast (runtime at most 0.5 s). A similar trend holds in terms of the size of the formula. For small domains, the proof-based algorithm generates smaller formulas than the rule-based algorithm, but for larger domains the opposite holds (Figures 3(b) and 4(b)).

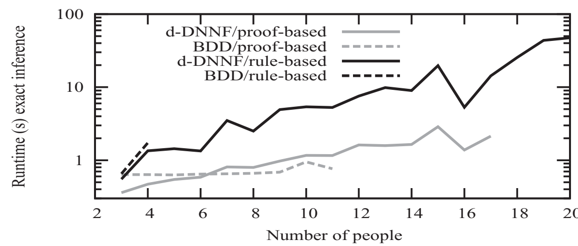
https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press



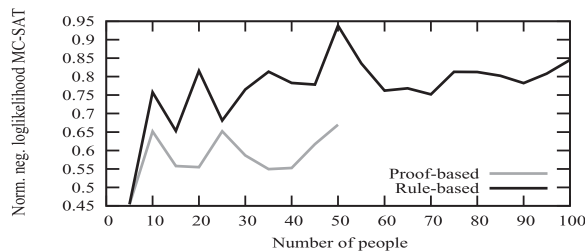
(a) Runtime of conversion to a weighted Boolean formula.



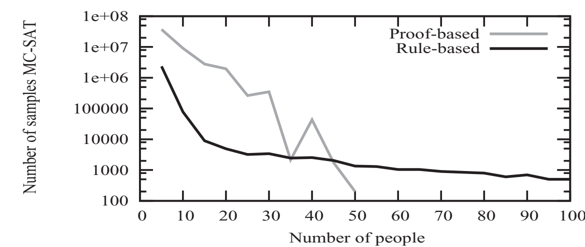
(b) Size of the Boolean formula (number of clauses in the CNF).



(c) Runtime of exact inference (compilation+traversal).



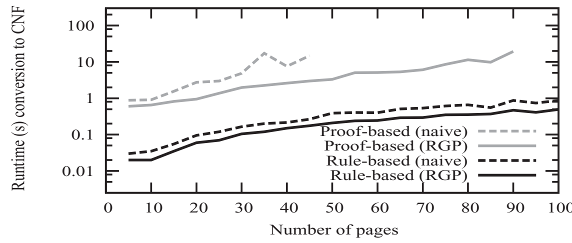
(d) Normalized negative log-likelihood of MC-SAT (lower is better).



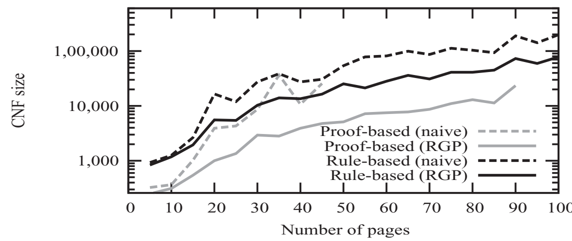
(e) Number of samples drawn by MC-SAT.

Fig. 3. Results for Smokers as a function of domain size. (When the curve for an algorithm ends at a particular domain size, this means that the algorithm is intractable beyond that size).

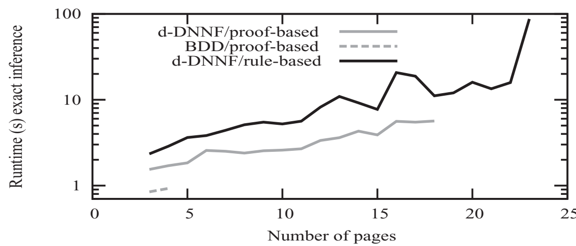
https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press



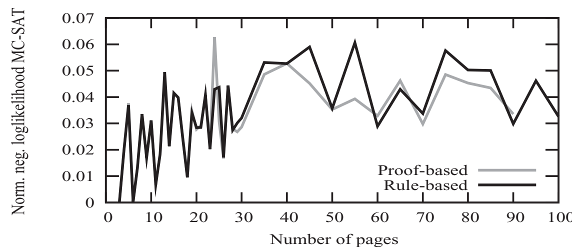
(a) Runtime of conversion to a weighted Boolean formula.



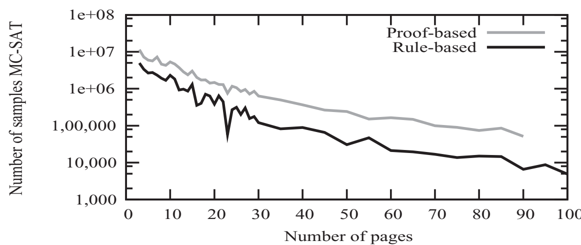
(b) Size of the Boolean formula (number of clauses in the CNF).



(c) Runtime of exact inference (compilation+traversal).



(d) Normalized negative log-likelihood of MC-SAT (lower is better).



(e) Number of samples drawn by MC-SAT.

Fig. 4. Results for *WebKB* as a function of domain size. (When the curve for an algorithm ends at a particular domain size, it means the algorithm is intractable beyond that size).

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Implications on inference. The ultimate question is how efficient are the formulas of different conversions for subsequent inference. We discuss this for exact inference in the next section; here we focus on approximate inference. We use the MC-SAT inference algorithm (Section 6.2) as a tool to evaluate how efficient are the different formulas for inference by running MC-SAT on the two types of formulas and measuring the quality of the estimated marginals. Evaluating the quality of approximate marginals is non-trivial when computing true marginals is intractable. We use the same solution as the original MC-SAT paper: we let MC-SAT run for a fixed time (10 min) and measure the quality of the estimated marginals as the likelihood of the 'query ground truth' according to these estimates; see Poon and Domingos (2006).

On domain sizes where the proof-based algorithm is still tractable, inference results are better with the proof-based formula than with the rule-based formula (see Figure 3(d), and to a smaller extent Figure 4(d)). This is because the proof-based formulas are more compact, and hence more samples can be drawn in the given time (Figures 3(e) and 4(e)).

As an answer to question **Q2**, we conclude that for smaller domains the proof-based algorithm is preferable because of smaller formulas. On larger domains, the rule-based algorithm should be used.

9.3.3 Q3 – Influence of the inference algorithm

For exact inference, our approach consists of knowledge compilation with either d-DNNFs or BDDs. Question **Q3**: Which of the two considered approaches, d-DNNFs or BDDs, performs best? To answer this question, we increase the domain size up to the point where inference (doing the compilation to d-DNNF or BDD) becomes intractable. It is useful to distinguish between compilation of rule-based and proof-based formulas.[19]

Proof-based formulas. On the Smokers domain, BDDs perform relatively well, but they are nevertheless clearly outperformed by d-DNNFs (Figure 3(c)). On WebKB, the difference is even larger: BDDs are only tractable on domains of size 3 or 4, while d-DNNFs reach up to size 18 (Figure 4(c)). When BDDs become intractable, this is mostly due to memory problems.[20]

Rule-based formulas. As seen before, these formulas are less compact than the proof-based formulas (at least for those domain sizes where exact inference is feasible). The results clearly show that d-DNNFs are much better at dealing with these non-compact formulas than BDDs. Concretely, d-DNNFs are still tractable up to reasonable sizes. In contrast, using BDDs on these rule-based formulas is

[19] In the PLP literature, BDDs have almost exclusively been used for proof-based formulas (De Raedt *et al.* 2007; Gutmann *et al.* 2011). Compiling our proof-based formulas to BDDs yields exactly the same BDDs as used by Gutmann *et al.* (2011). In the special case of a single query and no evidence, this also equals the BDDs used in De Raedt *et al.* (2007).

[20] It might be surprising that BDDs, which are the state of the art in PLP, do not perform better. However, one should keep in mind that we are using BDDs for *exact* inference here. BDDs are also used for approximate inference, one simply compiles an *approximate formula* into a BDD (De Raedt *et al.* 2007). The same can be done with d-DNNFs, and we again expect improvement over BDDs.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

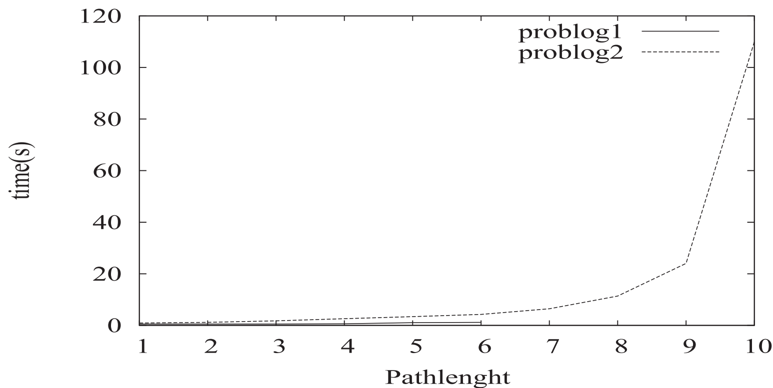


Fig. 5. Runtime of ProbLog1 and ProbLog2 on the probabilistic grid query, as a function of the distance $16 - i$ between the start and end nodes. (When the curve for an algorithm ends at a particular point, this means that the algorithm is intractable beyond that point).

nearly impossible: on Smokers, BDDs only solve size 3 and 4, on WebKB BDDs even do not solve any of the inference tasks on rule-based formulas.

As an answer to question **Q3**, we conclude that the use of d-DNNFs pushes the limit of exact MARG inference significantly further as compared to BDDs, which were the standard in PLP.

9.3.4 Q4 – Computing success probabilities with ProbLog2

Question **Q4**: When using the 'classical' ProbLog setting of computing success probabilities, does ProbLog2 (our new ProbLog implementation) outperform ProbLog1 (the previous ProbLog implementation)?

For ProbLog1 we use the default parameter settings and table the *path/2* predicate. For ProbLog2 we use the proof-based conversion and compile to d-DNNF. These settings are motivated by our previous conclusions, which show that the proof-based conversion works well on programs that are small enough to allow for exact inference (as we do here) and that d-DNNFs are superior to BDDs.

As explained before (Section 9.2.1), we ask the query `path(n_i_i,n_16_16)`, where we vary i from 15 to 1. The smaller the i, the larger the 'distance' $16 - i$ between the start and end nodes, and hence the harder the problem. Figure 5 shows the measured runtimes for ProbLog1 and ProbLog2.

The results show that ProbLog2 scales better than ProbLog1. ProbLog1 is tractable up to distance 6. From distance 7 onwards, it becomes intractable, i.e. it incurs a time-out. We have put the time limit to 300 s and repeated every experiment 10 times. For distance 7, all 10 repetitions timed-out, while for distance 6 the average runtime was only 1.2 s.[21] This shows that the runtime of ProbLog1 explodes beyond distance 6. In contrast, ProbLog2 is tractable up to distance 10. For distance 10, all 10 repetitions finish in time, taking on average 110 s. For distance 11, only five out

[21] To verify that the measurement for distance 7 is not a glitch, we also tried distance 8 and further, but ProbLog1 consistently timed-out for all of these.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

of 10 repetitions finish in time. From distance 12 onwards, all 10 repetitions time out.

9.3.5 Q5 – Ability to learn the original probabilities

Question **Q5**: When learning from the data generated from a known program, can we recover the parameters of the original program, given a reasonable amount of data? We answer this question by generating data from the given Smokers program (Section 9.1), applying our learning algorithm to this data and measuring the difference between the learned probabilities and those in the original program. We measure this difference in two ways. First, we use the mean absolute error (MAE) between both sets of probabilities. Second, we use the Kullback–Leibler(K-L)-divergence, a measure of similarity between a 'true' probability distribution (the one of the original program) and an 'approximating' distribution (the one of the learned program). ProbLog allows for an efficient calculation of the K-L-divergence because of the independence of probabilistic facts; see Appendix D given online.

Both MAE (Figure 6) and K-L-divergence (Figure 7) show that LFI-ProbLog can learn the original probabilities: both MAE and K-L-divergence approach zero when more examples are given. The 100% knowledge line shows the optimal way of calculating the probabilities, given the interpretations. The remaining cases, 10%, 40% and 70% show that the quality of the approximations, as expected, drops when more atoms become unobserved. However, the approximations remain of good quality. Hence, we can conclude that LFI-ProbLog is capable of recovering the original probabilities and is robust against missing values. When we compare the figures for the different domain sizes, we see that the results are independent of the number of persons in the domain.

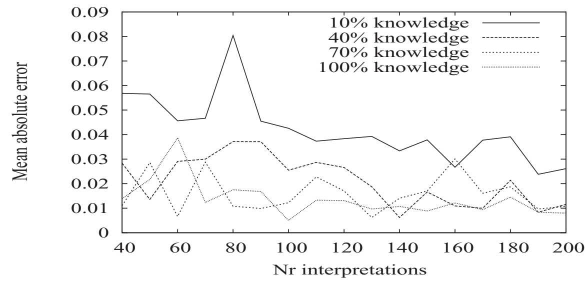
9.3.6 Q6 – Learning of real-world data

Question **Q6**: When learning from the real-world data, can we obtain results comparable with the ones obtained with the state-of-the-art system? To answer this question, we compare LFI-ProbLog with the Alchemy system for Markov Logic, running four-fold cross validation on the WebKB dataset. Table 3 shows the negative log-likelihood obtained with LFI-ProbLog and Alchemy on the test sets of four folds. In the case of Alchemy, we report two results: 'Alchemy' stands for using the system with its default parameters, 'Alchemy*' stands for Alchemy with a modified setting that puts a very strong prior on the weights (prior around zero, with standard deviation 0.1 instead of the default 100).[22]

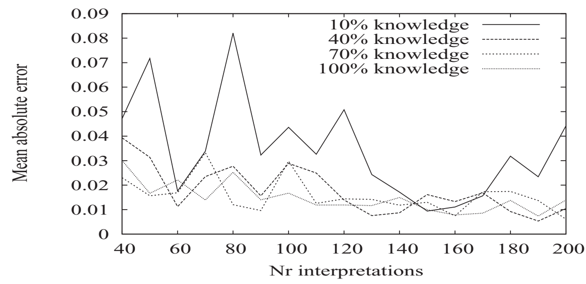
LFI-ProbLog outperforms Alchemy with its default settings on three of the four folds. However, Alchemy with the strong prior (Alchemy*) outperforms LFI-ProbLog on all four folds. We conclude that LFI-ProbLog is competitive with Alchemy, but parameter tuning can have a large impact. These results illustrate the importance of

[22] This modified setting was recommended to us by Alchemy developers (personal communication with Daniel Lowd).

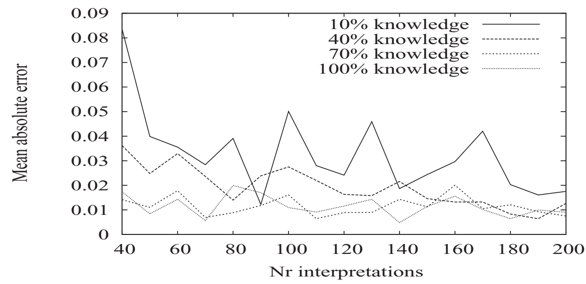
https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press



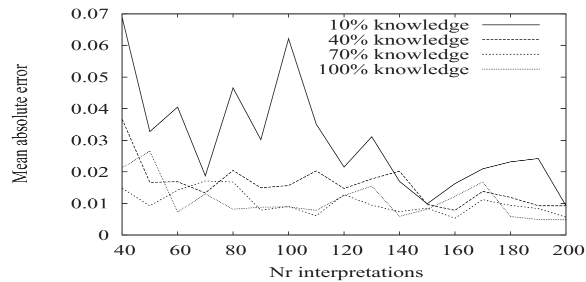
(a) 4 persons



(b) 5 persons



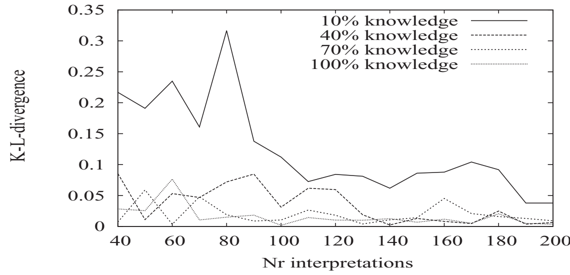
(c) 6 persons



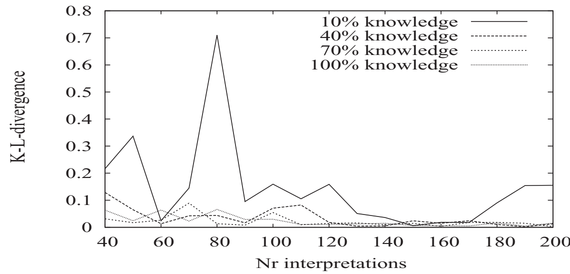
(d) 7 persons

Fig. 6. Mean absolute error (MAE, lower is better) when learning from Smokers data with 10%, 40%, 70% and 100% knowledge of the possible world.

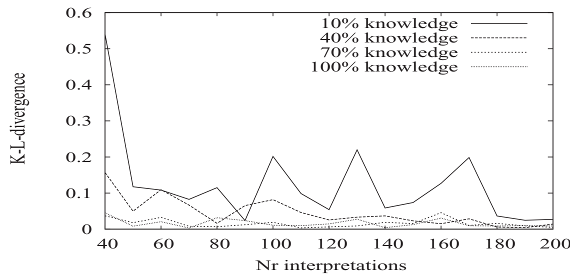
https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press



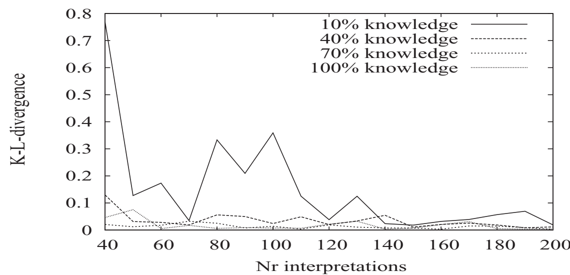
(a) 4 persons



(b) 5 persons



(c) 6 persons



(d) 7 persons

Fig. 7. K-L-divergence (lower is better) when learning from Smokers data with 10%, 40%, 70% and 100% knowldege of the possible world.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Table 3. *Negative log-likelihood (lower is better) on the WebKB learning experiment*

Test set	LFI-Problog	Alchemy*	Alchemy
Cornell	1309.72	613.37	1603.31
Texas	1210.51	640.56	1075.75
Washington	646.39	622.55	1420.87
Wisconsin	1033.90	783.51	3479.04

setting a suitable prior when learning. This is a topic that we have not yet explored in detail for LFI-ProbLog, but we plan to study in the future research.

10 Conclusions

The contributions of this paper are threefold.

First, we have introduced a two-step procedure for MPE and MARG inference in general probabilistic logic programs. In the first step it generates the weighted Boolean formula that captures all relevant information about a specific query, evidence and probabilistic logic program. This step relies on well-known conversion techniques from LP. The second step then invokes the well-known solvers (for instance, WMC and weighted MAX-SAT) on the generated weighted formula.

The resulting inference procedure is akin to that employed by Darwiche (2009) and others (Park 2002; Sang *et al.* 2005) for probabilistic graphical models (where many inference problems are also cast in terms of weighted Boolean formulas) but adapted to the much more expressive class of probabilistic logic programs. Our conversion-based approach is advantageous because it allows us to employ a wide range of well-known and optimized solvers on the weighted formula, essentially giving us 'inference algorithms for free'. Furthermore, the approach also improves upon the state of the art in PLP, where one has typically focussed on inference with a single query atom and no evidence (cf. Section 4), often by using BDDs. By using d-DNNFs instead of BDDs, we obtained speed-ups that push the limit of exact MARG inference significantly further.

Second, we have developed an EM approach to learning probabilistic logic programs from interpretations. This approach employs our novel inference procedures in the expectation step. The LFI setting is akin to that used in the graphical model and SRL communities.

Third, the two approaches have been incorporated in a novel implementation of the PLP language ProbLog, which, unlike its previous implementation in YAP-Prolog (Kimmig *et al.* 2010), is closer to that of ASP systems than to Prolog systems.

Overall, our approach provides new insights into the relationships between PLP, graphical models and SRL. As an immediate outcome, we pointed out a conversion of probabilistic logic programs to ground Markov Logic, which allowed us to apply MC-SAT to PLP inference. This contributes to further bridging the gap between PLP and the field of SRL.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Acknowledgements

We thank the reviewers for their useful suggestions. We thank Maurice Bruynooghe, Jesse Davis, Kristian Kersting, Angelika Kimmig and Theofrastos Mantadelis for useful discussions. Daan Fierens, Guy Van den Broeck and Bernd Gutmann are supported by the Research Foundation-Flanders (FWO-Vlaanderen). Joris Renkens is supported by PF-10/010 NATAR. This research is supported by the European Commission under contract number FP7-248258-First-MM.

References

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers 35,* 8 (August), 677–691.

Chavira, M., Darwiche, A. and Jaeger, M. 2006. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning 42,* 1, 4–20.

Darwiche, A. 2001. On the tractability of counting theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics 11,* 1–2, 11–34.

Darwiche, A. 2004. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of 16th European Conference on Artificial Intelligence*. 328–332.

Darwiche, A. 2009. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, Cambridge, UK, Chap. 12.

Darwiche, A. and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research 17,* 1 (September), 229–264.

Denecker, M., Bruynooghe, M. and Marek, V. W. 2001. Logic programming revisited: Logic programs as inductive definitions. *ACM Transactions on Computational Logic 2,* 4, 623–654.

De Raedt, L. 2008. *Logical and Relational Learning*. Cognitive Technologies. Springer, New York, NY.

De Raedt, L., Frasconi, P., Kersting, K. and Muggleton, S., Eds. 2008. *Probabilistic Inductive Logic Programming – Theory and Applications*. LNCS, vol. 4911. Springer, New York, NY.

De Raedt, L., Kimmig, A. and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of 20th International Joint Conference on Artificial Intelligence*. AAAI Press, Menlo Park, CA, 2468–2473.

Domingos, P., Kok, S., Lowd, D., Poon, H., Richardson, M. and Singla, P. 2008. *Probabilistic Inductive Logic Programming – Theory and Applications*, Chapter "Markov Logic," Lecture Notes in Computer Science. Springer, New York, NY.

Fierens, D., Van den Broeck, G., Bruynooghe, M. and De Raedt, L. 2012. Constraints for probabilistic logic programming. In *Proceedings of the NIPS 2012 Workshop on Probabilistic Programming: Foundations and Applications*.

Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B. and De Raedt, L. 2011. Inference in probabilistic logic programs using weighted CNFs. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI)*, AUAI Press, Corvallis, Oregon, USA, 211–220.

Getoor, L. and Taskar, B. 2007. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. MIT Press, Cambridge, MA.

Gomes, C. P., Hoffmann, J., Sabharwal, A. and Selman, B. 2007. From sampling to model counting. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. 2293–2299.

Grädel, E. 1992. On transitive closure logic. In *Proceedings of the 5th Workshop on Computer*

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Science Logic. Lecture Notes in Computer Science, vol. 626. Springer, New York, NY, 149–163.

Gutmann, B., Kimmig, A., Kersting, K. and Raedt, L. 2008a. Parameter learning in probabilistic databases: A least squares approach. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD '08)* – Part I. Springer-Verlag, Berlin, Germany, 473–488.

Gutmann, B., Kimmig, A., Kersting, K. and Raedt, L. D. 2008b. Parameter learning in probabilistic databases: A least squares approach. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, W. Daelemans, B. Goethals and K. Morik, Eds. Lecture Notes in Computer Science, vol. 5211. Springer, Berlin, Germany, 473–488.

Gutmann, B., Thon, I. and De Raedt, L. 2010. *Learning the Parameters of Probabilistic Logic Programs from Interpretations*. Technical Report CW 584, KU Leuven.

Gutmann, B., Thon, I. and De Raedt, L. 2011. Learning the parameters of probabilistic logic programs from interpretations. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, Part 1. Springer-Verlag, Berlin, Germany, 581–596.

Ishihata, M., Kameya, Y., Sato, T. and Minato, S. 2008. Propositionalizing the EM algorithm by BDDs. In *Late Breaking Papers of the 18th International Conference on Inductive Logic Programming*.

Janhunen, T. 2004. Representing normal programs with clauses. In *Proceedings of the 16th European Conference on Artificial Intelligence*. IOS Press, Amsterdam, Netherlands, 358–362.

Kersting, K., De Raedt, L. and Kramer, S. 2000. Interpreting Bayesian logic programs. In *Proceedings of the AAAI-2000 workshop on learning statistical models from relational data*, AAAI Press, 29–35.

Kimmig, A., Demoen, B., De Raedt, L., Costa, V. S. and Rocha, R. 2010. On the implementation of the probabilistic logic programming language problog. In *Theory and Practice of Logic Programming Systems, 24th International Conference on Logic Programming (ICLP 2008)*, Special Issue, vol. 11, pp.235–262, arXiv: CoRR abs/1006.4442.

Lin, F. and Zhao, Y. 2002. Assat: Computing answer sets of a logic program by sat solvers. In *Artificial Intelligence*. S. Benferhat and E. Giunchiglia, Eds. Elsevier, 112–117.

Lloyd, J. 1987. *Foundations of Logic Programming*, 2nd edn. Springer-Verlag, Berlin, Germany.

Mantadelis, T. and Janssens, G. 2010. Dedicated tabling for a probabilistic setting. In *Technical Communications of 26th International Conference on Logic Programming*, M. Hermenegildo and T. Schaub, Eds. Dagstuhl Publishing, Saarbrücken/Wadern, Germany, 124–133.

Meert, W., Struyf, J. and Blockeel, H. 2009. CP-logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In *Proceedings of the 19th International Conference on Inductive Logic Programming*, L. De Raedt, Ed. Elsevier, 96–109.

Muise, C., McIlraith, S. A., Beck, J. C. and Hsu, E. 2012. DSHARP: Fast d-DNNF compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*, L. Kosseim and D. Inkpen Eds. Springer.

Park, J. D. 2002. Using weighted MAX-SAT engines to solve MPE. In *Proceedings of the 18th National Conference on Artificial Intelligence*, AAAI Press, 682–687.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press

Poole, D. 2008. *Probabilistic Inductive Logic Programming – Theory and Applications*, Chapter: "The Independent Choice Logic and Beyond." Lecture Notes in Computer Science. Springer, Berlin, Germany.

Poon, H. and Domingos, P. 2006. Sound and efficient inference with probabilistic and deterministic dependencies. In *Proceedings of the 21st National Conference on Artificial Intelligence*, AAAI Press.

Riguzzi, F. and Swift, T. 2013. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming*, *13*, 2, 279–302.

Robertson, N. and Seymour, P. 1986. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms 7,* 3, 309–322.

Sang, T., Beame, P., and Kautz, H. 2005. Solving Bayesian networks by weighted model counting. In *Proceedings of the 20th National Conference on Artificial Intelligence*, AAAI Press, 475–482.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP95)*. MIT Press, Cambridge, MA, 715–729.

Sato, T. and Kameya, Y. 2008. *Probabilistic Inductive Logic Programming – Theory and Applications*, Chapter: "New Advances in Logic-Based Probabilistic Modeling by PRISM." Lecture Notes in Computer Science. Springer, Berlin, Germany.

Van den Broeck, G., Thon, I., van Otterlo, M. and De Raedt, L. 2010. DTProbLog: A decision-theoretic probabilistic Prolog. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI Press, 1217–1222.

Van Gelder, A., Ross, K. A. and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of ACM 38,* 3, 620–650.

Vennekens, J., Denecker, M. and Bruynooghe, M. 2009. Cp-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming 9,* 3, 245–308. CoRR abs/0904.1672.

https://doi.org/10.1017/S1471068414000076 Published online by Cambridge University Press