TLP **17** (5–6): 942–960, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000400 First published online 30 August 2017

Computing LP$^{\text{MLN}}$ *using ASP and MLN solvers**

JOOHYUNG LEE, SAMIDH TALSANIA and YI WANG
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA
(*e-mail:* joolee@asu.edu, stalsani@asu.edu, ywang485@asu.edu)

submitted 2 May 2017; revised 20 June 2017; accepted 4 July 2017

Abstract

LP$^{\text{MLN}}$ is a recent addition to probabilistic logic programming languages. Its main idea is to overcome the rigid nature of the stable model semantics by assigning a weight to each rule in a way similar to Markov Logic is defined. We present two implementations of LP$^{\text{MLN}}$, LPMLN2ASP and LPMLN2MLN. System LPMLN2ASP translates LP$^{\text{MLN}}$ programs into the input language of answer set solver CLINGO, and using weak constraints and stable model enumeration, it can compute most probable stable models as well as exact conditional and marginal probabilities. System LPMLN2MLN translates LP$^{\text{MLN}}$ programs into the input language of Markov Logic solvers, such as ALCHEMY, TUFFY, and ROCKIT, and allows for performing approximate probabilistic inference on LP$^{\text{MLN}}$ programs. We also demonstrate the usefulness of the LP$^{\text{MLN}}$ systems for computing other languages, such as ProbLog and Pearl's Causal Models, that are shown to be translatable into LP$^{\text{MLN}}$.

KEYWORDS: Answer Set Programming, Markov Logic, LPMLN

1 Introduction

LP$^{\text{MLN}}$ is a simple extension of answer set programs with the concept of weighted rules, whose weight scheme is adopted from that of Markov Logic (Richardson and Domingos 2006). Like Markov Logic, not all LP$^{\text{MLN}}$ rules have to be true, but, roughly speaking, the more rules that are true, the larger the weight that is assigned to the corresponding stable model.

It is shown that both answer set programs and Markov Logic can be easily embedded in LP$^{\text{MLN}}$ [Theorems 1, 2 of the paper by Lee and Wang (2016)]. The other direction is more interesting from a computational point of view because it provides ways to compute LP$^{\text{MLN}}$ using existing implementations of ASP (Answer Set Programming) and MLN (Markov Logic Network) solvers. This paper further develops the translations in this direction, presents two implementations of LP$^{\text{MLN}}$

* This work was partially supported by the National Science Foundation under Grants IIS-1319794 and IIS-1526301.

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

based on them along with a few new theorems justifying the implementations, and illustrates the usefulness of the systems.

It is shown by Balai and Gelfond (2016) that LP^{MLN} programs can be turned into P-log programs. We present a similar but simpler translation that turns LP^{MLN} programs into answer set programs. However, for a non-ground LP^{MLN} program, the translation yields an answer set program that is unsafe, so a direct implementation of this idea has a drawback. Instead, we develop an implementation based on the translation by Lee and Yang (2017) that turns (ground) LP^{MLN} programs into answer set programs containing weak constraints so that the most probable stable models of an LP^{MLN} program coincide with the optimal stable models of the translated ASP program. Going further, we show how to map the penalty of each stable model of the translated ASP program to the probability of each stable model of a (non-ground) LP^{MLN} program, and use this result to compute probabilistic queries for an LP^{MLN} program using an ASP solver CLINGO. The input language of LPMLN2ASP is familiar to the users of CLINGO because its syntax is a simple extension of CLINGO rules prepended by weights, thereby allowing many advanced constructs of the CLINGO language, such as aggregates and conditional literals, in the context of LP^{MLN}.

A different method to compute an LP^{MLN} program is by converting it into a Markov Logic Network (Lee and Wang 2016, Theorem 3), similar to the reduction of answer set programs to propositional logic, and then invoking MLN solvers, such as ALCHEMY, TUFFY, and ROCKIT. While it is possible to turn any LP^{MLN} program into an equivalent MLN by adding all loop formulas, in practice, the straightforward implementation does not yield an effective computation. Thus, we limit attention to the "tight" fragment of LP^{MLN} programs that can be easily converted into MLNs using the process of completion. Even so, the straightforward encoding of completion formulas in Markov Logic may lead to a blow-up in CNF (Conjunctive Normal Form) conversion performed by ALCHEMY because the conversion is naively implemented in ALCHEMY. Furthermore, the input languages of TUFFY and ROCKIT do not even allow nested formulas, which are needed to encode completion formulas. So, LPMLN2MLN implements some equivalent transformation using auxiliary atoms to avoid the blow-up in CNF conversion and takes care of differences in the input language of different MLN solvers. The input language of LPMLN2MLN resembles that of ALCHEMY and is converted into one of the input languages of ALCHEMY, TUFFY, and ROCKIT depending on the mode selected. The system utilizes approximate probabilistic inference methods or exact optimization methods supported by the MLN solvers.

The implementations are not only interesting for computing LP^{MLN}. System LPMLN2ASP can be used to derive the most probable stable models even when the standard answer set program is inconsistent. This feature could be useful in debugging an inconsistent answer set program or deriving some meaningful conclusions from an inconsistent knowledge base. Also, both implementations can be used to compute other formalisms, such as Markov Logic, ProbLog (De Raedt *et al.* 2007), Pearl's Probabilistic Causal Models (Pearl 2000), and P-log (Baral *et al.* 2009), which are shown to be translatable into LP^{MLN} (Lee *et al.* 2015; Lee and Wang 2016; Lee and Yang 2017).

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

The systems are publicly available at `http://reasoning.eas.asu.edu/lpmln/` along with the user manual and examples.

The paper is organized as follows. Section 2 reviews the language LP$^{\text{MLN}}$, which is based on the concept of reward, and presents a reformulation of LP$^{\text{MLN}}$ based on the concept of penalty. Section 3 shows two translations of LP$^{\text{MLN}}$ programs into answer set programs, one based on the reward-based way and another based on the penalty-based way, and presents system LPMLN2ASP that implements the penalty-based way. Section 4 shows a translation of tight LP$^{\text{MLN}}$ programs into Markov Logic Networks and presents system LPMLN2MLN that implements the translation. Section 5 gives a comparison and running statistics of these implementations. Section 6 shows how to use these systems to compute other probabilistic logic languages that are shown to be translatable into LP$^{\text{MLN}}$. The proofs of the theorems and more experiments can be found in the online appendix accompanying the paper at the TPLP archive (Lee *et al.* 2017).

2 Language LP$^{\text{MLN}}$

2.1 *Review:* LP$^{\text{MLN}}$

We review the definition of LP$^{\text{MLN}}$ from the paper by Lee and Wang (2016). An LP$^{\text{MLN}}$ program is a finite set of weighted rules $w : R$, where R is a rule (as allowed in the input language of ASP solver CLINGO), and w is a real number (in which case, the weighted rule is called *soft*) or α for denoting the infinite weight (in which case, the weighted rule is called *hard*). An LP$^{\text{MLN}}$ program is called *ground* if its rules contain no variables. We assume a finite Herbrand Universe so that the ground program is finite. Each ground instance of a non-ground rule receives the same weight as the original non-ground formula.

For any ground LP$^{\text{MLN}}$ program Π and any interpretation I, $\overline{\Pi}$ denotes the usual (unweighted) ASP program obtained from Π by dropping the weights, and Π_I denotes the set of $w : R$ in Π such that $I \models R$, and SM[Π] denotes the set $\{I \mid I \text{ is a stable model of } \overline{\Pi_I}\}$. The *unnormalized weight* of an interpretation I under Π is defined as

$$W_\Pi(I) = \begin{cases} \exp\left(\sum_{w:R \in \Pi_I} w\right) & \text{if } I \in \text{SM}[\Pi]; \\ 0 & \text{otherwise.} \end{cases}$$

The *normalized weight* (a.k.a. *probability*) of an interpretation I under Π is defined as

$$P_\Pi(I) = \lim_{\alpha \to \infty} \frac{W_\Pi(I)}{\sum_{J \in \text{SM}[\Pi]} W_\Pi(J)}.$$

Interpretation I is called a *(probabilistic) stable model* of Π if $P_\Pi(I) \neq 0$. The most probable stable models of Π are the stable models with the highest probability.

2.2 *Reformulating* LP$^{\text{MLN}}$ *based on the concept of penalty*

In the definition of the LP$^{\text{MLN}}$ semantics by Lee and Wang (2016), the weight assigned to each stable model can be regarded as "rewards": the more rules that

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

are true in deriving the stable model, the larger the weight that is assigned to it. In this section, we reformulate the LP$^{\text{MLN}}$ semantics in a "penalty" based way. More precisely, the penalty-based weight of an interpretation I is defined as the exponentiated negative sum of the weights of the rules that are not satisfied by I (when I is a stable model of $\overline{\Pi_I}$). Let

$$W_\Pi^{\text{pnt}}(I) = \begin{cases} \exp\left(- \sum_{w:R \in \Pi \text{ and } I \not\models R} w \right) & \text{if } I \in \text{SM}[\Pi]; \\ 0 & \text{otherwise} \end{cases}$$

and

$$P_\Pi^{\text{pnt}}(I) = \lim_{\alpha \to \infty} \frac{W_\Pi^{\text{pnt}}(I)}{\sum_{J \in \text{SM}[\Pi]} W_\Pi^{\text{pnt}}(J)}.$$

The following theorem tells us that the LP$^{\text{MLN}}$ semantics can be reformulated using the concept of a penalty-based weight.

Theorem 1
For any LP$^{\text{MLN}}$ program Π and any interpretation I,

$$W_\Pi(I) \propto W_\Pi^{\text{pnt}}(I) \quad \text{and} \quad P_\Pi(I) = P_\Pi^{\text{pnt}}(I).$$

Although the penalty-based reformulation appears to be more complicated, it has a few desirable features. One of them is that adding a trivial rule does not affect the weight of an interpretation, which is not the case with the original definition. More importantly, this reformulation leads to a better translation of LP$^{\text{MLN}}$ programs into answer set programs as we discuss in Section 3.3.

3 Turning LP$^{\text{MLN}}$ into ASP with weak constraints

3.1 Review: Weak constraints

A *weak constraint* (Buccafurri *et al.* 2000; Calimeri *et al.* 2012) has the form

$$:\sim F \quad [\mathit{Weight} \ @ \ \mathit{Level}]$$

where F is a conjunction of literals, *Weight* is a real number, and *Level* is a non-negative integer.

Let Π be a program $\Pi_1 \cup \Pi_2$, where Π_1 is an answer set program that does not contain weak constraints, and Π_2 is a set of ground weak constraints. We call I a stable model of Π if it is a stable model of Π_1. For every stable model I of Π and any non-negative integer l, the *penalty* of I at level l, denoted by $\mathit{Penalty}_\Pi(I,l)$, is defined as

$$\sum_{:\sim F[w@l] \in \Pi_2, \ I \models F} w.$$

For any two stable models I and I' of Π, we say I is *dominated* by I' if

- there is some non-negative integer l such that $\mathit{Penalty}_\Pi(I',l) < \mathit{Penalty}_\Pi(I,l)$ and
- for all integers $k > l$, $\mathit{Penalty}_\Pi(I',k) = \mathit{Penalty}_\Pi(I,k)$.

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

A stable model of Π is called *optimal* if it is not dominated by another stable model of Π.

The input language of CLINGO allows non-ground weak constraints that contain tuples of terms. We refer the reader to the CLINGO manual for more details.

3.2 *Turning* LP$^{\text{MLN}}$ *into ASP: Reward way*

In the paper by Balai and Gelfond (2016), it is shown that LP$^{\text{MLN}}$ programs can be turned into P-log. In this section, we show that using a similar translation, it is even possible to turn LP$^{\text{MLN}}$ programs into answer set programs.

We turn each (possibly non-ground) rule

$$w_i: \quad Head_i(\mathbf{x}) \leftarrow Body_i(\mathbf{x})$$

in an LP$^{\text{MLN}}$ program Π, where i is the index of the rule and $\mathbf{x}$ is the list of global variables in the rule, into ASP rules

$$\begin{aligned} \texttt{sat}(i, w_i, \mathbf{x}) &\leftarrow Head_i(\mathbf{x}) \\ \texttt{sat}(i, w_i, \mathbf{x}) &\leftarrow \texttt{not } Body_i(\mathbf{x}) \\ Head_i(\mathbf{x}) &\leftarrow Body_i(\mathbf{x}), \texttt{ not not sat}(i, w_i, \mathbf{x}) \\ &:\sim \texttt{sat}(i, w_i, \mathbf{x}). \quad [-w_i'@l, i, \mathbf{x}] \end{aligned} \tag{3.1}$$

where (i) $w_i' = 1$ and $l = 1$ if w_i is α; and (ii) $w_i' = w_i$ and $l = 0$ otherwise[1].

Intuitively, a ground `sat` atom is true if the corresponding ground rule obtained from the original program is true. For each true `sat` atom, a weak constraint imposes on the stable model the opposite of the weight as a penalty, which can be viewed as imposing the weight as a reward.

By lpmln2asp$^{\text{rwd}}(\Pi)$, we denote the resulting ASP program containing weak constraints. The following theorem states the correctness of the translation. Let $Gr(\Pi)$ be the ground program obtained from Π by replacing global variables with the Herbrand Universe.

Theorem 2
For any LP$^{\text{MLN}}$ program Π, there is a 1-1 correspondence ϕ between SM[Π] and the set of stable models of lpmln2asp$^{\text{rwd}}(\Pi)$, where

$$\phi(I) = I \cup \{\texttt{sat}(i, w_i, \mathbf{c}) \mid w_i : Head_i(\mathbf{c}) \leftarrow Body_i(\mathbf{c}) \text{ in } Gr(\Pi), I \models Body_i(\mathbf{c}) \rightarrow Head_i(\mathbf{c})\}.$$

Furthermore,

$$W_\Pi(I) = exp\left(\sum_{\texttt{sat}(i,w_i,\mathbf{c}) \in \phi(I)} w_i\right). \tag{3.2}$$

Also, ϕ is a 1-1 correspondence between the most probable stable models of Π and the optimal stable models of lpmln2asp$^{\text{rwd}}(\Pi)$.

While the translation is simple and modular, there are a few problems with using this translation to compute LP$^{\text{MLN}}$ using ASP solvers. First, the translation does

[1] CLINGO restricts the weights in weak constraints to be integers only. To implement the translation using CLINGO, we need to turn w_i' into an integer by multiplying some factor.

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

not necessarily yield a program that is acceptable in CLINGO and requires a further translation. In particular, the first and the second rules of (3.1) may not be in the syntax of CLINGO. (The third rule contains double negations, which are allowed in CLINGO from version 4.) Second, more importantly, when we translate non-ground LP$^{\text{MLN}}$ rules into the input language of ASP solvers, the first and the second rules of (3.1) may be unsafe, so CLINGO cannot ground the program. An alternative translation in the next section avoids these problems by basing on the penalty-based concept of weights.

3.3 Turning LP$^{\text{MLN}}$ *into ASP: Penalty way*

Based on the reformulation of LP$^{\text{MLN}}$ in Section 2.2, we introduce another translation that turns LP$^{\text{MLN}}$ programs into ASP programs. The translation ensures that a safe LP$^{\text{MLN}}$ program is always turned into a safe ASP program, and the resulting program is readily acceptable as an input to CLINGO[2].

We define the translation $\mathsf{lpmln2asp}^{\mathsf{pnt}}(\Pi)$ by translating each (possibly non-ground) rule

$$w_i : \quad Head_i(\mathbf{x}) \leftarrow Body_i(\mathbf{x})$$

in an LP$^{\text{MLN}}$ program Π, where i is the index of the rule and $\mathbf{x}$ is the list of global variables in the rule, into ASP rules

$$\begin{aligned} \texttt{unsat}(i, w_i, \mathbf{x}) &\leftarrow Body_i(\mathbf{x}),\ \texttt{not}\ Head_i(\mathbf{x}) \\ Head_i(\mathbf{x}) &\leftarrow Body_i(\mathbf{x}),\ \texttt{not unsat}(i, w_i, \mathbf{x}) \\ &:\sim \texttt{unsat}(i, w_i, \mathbf{x}). \quad [w_i'@l, i, \mathbf{x}] \end{aligned} \tag{3.3}$$

where (i) $w_i' = 1$ and $l = 1$ if w_i is α; and (ii) $w_i' = w_i$ and $l = 0$ otherwise[3].

Intuitively, the first rule of (3.3) makes atom $\texttt{unsat}(i, w_i, \mathbf{x})$ true when the ith rule in the original program is not satisfied. In that case, the second rule is not effective, and w_i is imposed on the penalty of the stable model. On the other hand, if the ith rule is satisfied, atom $\texttt{unsat}(i, w_i, \mathbf{x})$ is false, the rule $Head_i \leftarrow Body_i$ is effective, and the penalty is not imposed.

The following theorem is an extension of Corollary 2 by Lee and Yang (2017) to allow non-ground programs and to consider the correspondence between all stable models, not only the most probable ones.

Theorem 3

For any LP$^{\text{MLN}}$ program Π, there is a 1-1 correspondence ϕ between SM[Π] and the set of stable models of $\mathsf{lpmln2asp}^{\mathsf{pnt}}(\Pi)$, where

$$\begin{aligned} \phi(I) = I \cup \{\texttt{unsat}(i, w_i, \mathbf{c}) \mid\ & w_i : Head_i(\mathbf{c}) \leftarrow Body_i(\mathbf{c}) \text{ in } Gr(\Pi), \\ & I \not\models Body_i(\mathbf{c}) \rightarrow Head_i(\mathbf{c})\}. \end{aligned}$$

[2] An LP$^{\text{MLN}}$ program Π is *safe* if its unweighted program $\overline{\Pi}$ is safe as defined by Calimeri *et al.* (2012).
[3] In the case $Head_i$ is a disjunction $l_1; \ldots, l_n$, expression $\texttt{not}\ Head_i$ stands for $\texttt{not}\ l_1, \ldots, \texttt{not}\ l_n$.

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

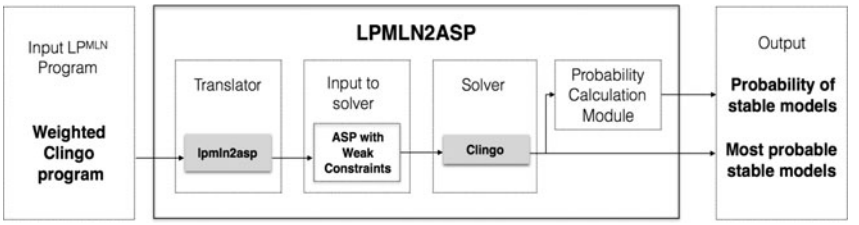


Fig. 1. Architecture of system LPMLN2ASP.

Furthermore,

$$W_{\Pi}^{\text{pnt}}(I) = exp\Big(- \sum_{\texttt{unsat}(i,w_i,\mathbf{c})\in\phi(I)} w_i \Big). \tag{3.4}$$

Also, ϕ is a 1-1 correspondence between the most probable stable models of Π and the optimal stable models of $\mathsf{lpmln2asp}^{\text{pnt}}(\Pi)$.

Theorem 3, in conjunction with Theorem 1, provides a way to compute the probability of a stable model of an LP^{MLN} program by examining the `unsat` atoms satisfied by the corresponding stable model of the translated ASP program.

3.4 *System* LPMLN2ASP

System LPMLN2ASP is an implementation of LP^{MLN} based on the result in Section 3.3 using CLINGO v4.5. It can be used for computing the probabilities of stable models, marginal/conditional probability of a query, as well as the most probable stable models. The architecture of the system is shown in Figure 1.

In the input language of LPMLN2ASP, a soft rule is written in the form

$$w_i \quad Head_i \leftarrow Body_i \tag{3.5}$$

where w_i is a real number in decimal notation, and $Head_i \leftarrow Body_i$ is a CLINGO rule. A hard rule is written without weights and is identical to a CLINGO rule. For instance, the "Bird" example from the paper by Lee and Wang (2016) can be represented in the input language of LPMLN2ASP as follows. The first three rules represent definite knowledge, while the last two rules represent uncertain knowledge with different confidence levels.

```
% bird.lpmln
bird(X) :- residentbird(X).
bird(X) :- migratorybird(X).
:- residentbird(X), migratorybird(X).
2 residentbird(jo).
1 migratorybird(jo).
```

The basic command line syntax of executing LPMLN2ASP is

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

```
lpmln2asp -i <input file> [-r <output file>] [-e <evidence file>]
     [-q <query predicates>] [-hr] [-all] [-clingo "<clingo options>"]
```

which follows the ALCHEMY command line syntax.

The mode of computation is determined by the options provided to LPMLN2ASP. By default, the system finds a most probable stable model of $\mathsf{lpmln2asp}^{\text{pnt}}(\Pi)$ (MAP estimate) by leveraging CLINGO's built-in optimization method for weak constraints.

For computing marginal probability, LPMLN2ASP utilizes CLINGO's interface with Python. When CLINGO enumerates each stable model of $\mathsf{lpmln2asp}^{\text{pnt}}(\Pi)$, the computation is interrupted by the *probability computation module*, a Python program which records the stable model as well as its penalty specified in the `unsat` atoms true in the stable model. Once all the stable models are generated, the control returns to the module, which sums up the recorded penalties to compute the normalization constant as well as the probability of each stable model. The probabilities of query atoms (specified by the option `-q`) are also calculated by adding the probabilities of the stable models that contain the query atoms. For instance, the probability of a query atom `residentbird(jo)` is $\Sigma_{I \models \texttt{residentbird(jo)}} P(I)$. The option `-all` instructs the system to display all stable models and their probabilities.

For conditional probability, the evidence file `<evidence file>` is specified by the option `-e`. The file may contain any CLINGO rules, but usually they are constraints, i.e., rules with the empty head. The main difference from the marginal probability computation is that CLINGO computes $\mathsf{lpmln2asp}^{\text{pnt}}(\Pi) \cup$ `<evidence file>` instead of $\mathsf{lpmln2asp}^{\text{pnt}}(\Pi)$.

Below, we illustrate how to use the system for various inferences.

MAP (Maximum A Posteriori) inference: The command line to use is

```
lpmln2asp -i <input file>
```

By default, LPMLN2ASP computes MAP inference. For example, `lpmln2asp -i bird.lpmln` returns

```
residentbird(jo) bird(jo) unsat(5,"1.000000")
Optimization: 1000
OPTIMUM FOUND
```

Marginal probability of all stable models: The command line to use is

```
lpmln2asp -i <input file> -all
```

For example, `lpmln2asp -i bird.lpmln -all` outputs

```
Answer: 1
residentbird(jo) bird(jo)
unsat(5,"1.000000")
Optimization: 1000
Answer: 2
unsat(4,"2.000000") unsat(5,"1.000000")
Optimization: 3000
Answer: 3
unsat(4,"2.000000") bird(jo)
```

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

```
migratorybird(jo)
Optimization: 2000

Probability of Answer 1 : 0.665240955775
Probability of Answer 2 : 0.0900305731704
Probability of Answer 3 : 0.244728471055
```

Marginal probability of query atoms: The command line to use is

```
lpmln2asp -i <input file> -q <query predicates>
```

This mode calculates the marginal probability of the atoms whose predicates are specified by `-q` option. For example, `lpmln2asp -i birds.lp -q residentbird` outputs

```
residentbird(jo) 0.665240955775
```

Conditional probability of query given evidence: The command line to use is

```
lpmln2asp -i <input file> -q <query predicates> -e <evidence file>
```

This mode computes the conditional probability of a query given the evidence specified in the `<evidence file>`. For example,

```
lpmln2asp -i birds.lp -q residentbird -e evid.db
```

where `evid.db` contains

```
:- not bird(jo).
```

outputs the conditional probability $P(residentbird(X) \mid bird(jo))$:

```
residentbird(jo) 0.73105857863
```

Debugging ASP programs: The command line to use is

```
lpmln2asp -i <input file> -hr -all
```

By default, LPMLN2ASP does not translate hard rules and pass them to CLINGO as is. The option `-hr` instructs the system to translate hard rules as well. According to Proposition 2 by Lee and Wang (2016), as long as the LP^{MLN} program has a probabilistic stable model that satisfies all hard rules, the simpler translation that does not translate hard rules gives the same result as the full translation and is more computationally efficient. Since in many cases hard rules represent definite knowledge that should not be violated, this is desirable.

On the other hand, translating hard rules could be relevant in some other cases, such as debugging an answer set program by finding which rules cause inconsistency. For example, consider a CLINGO input program `bird.lp`, that is similar to `bird.lpmln` but drops the weights in the last two rules. CLINGO finds no stable models for this program. However, if we invoke LPMLN2ASP on the same program as

```
lpmln2asp -i bird.lp -hr
```

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

the output of LPMLN2ASP shows three probabilistic stable models, each of which shows a way to resolve the inconsistency by ignoring the minimal number of the rules. For instance, one of them is {`bird(jo)`, `residentbird(jo)`}, which disregards the last rule. The other two are similar.

Note that the probability computation involves enumerating all stable models so that it can be much more computationally expensive than the default MAP inference. On the other hand, the computation is exact, so compared to an approximate inference, the "gold standard" result is easy to understand. Also, the conditional probability is more effectively computed than the marginal probability because CLINGO effectively prunes many answer sets that do not satisfy the constraints specified in the evidence file.

3.5 *Computing MLN with* LPMLN2ASP

A typical example in the MLN literature is a social network domain that describes how smokers influence other people, which can be represented in LP$^{\text{MLN}}$ as follows. We assume three people *alice*, *bob*, and *carol*, and assume that *alice* is a smoker, *alice* influences *bob*, *bob* influences *carol*, and nothing else is known.

$$\begin{array}{l} w: \quad smoke(x) \wedge influence(x,y) \rightarrow smoke(y) \\ \alpha: smoke(alice) \qquad \alpha: influence(alice,bob) \qquad \alpha: influence(bob,carol). \end{array} \tag{3.6}$$

(w is a positive number.) One may expect *bob* is less likely a smoker than *alice*, and *carol* is less likely a smoker than *bob*.

Indeed, the program above defines the following distribution (we omit the *influence* relation, which has a fixed interpretation.)

Possible world	Weight
$\{smoke(alice), \neg smoke(bob), \neg smoke(carol)\}$	$k \cdot e^{8w}$
$\{smoke(alice), smoke(bob), \neg smoke(carol)\}$	$k \cdot e^{8w}$
$\{smoke(bob), \neg smoke(alice), smoke(carol)\}$	0
$\{smoke(alice), smoke(bob), smoke(carol)\}$	$k \cdot e^{9w}$

where $k = e^{3\alpha}$. The normalization constant is the sum of all the weights: $k \cdot e^{9w} + 2k \cdot e^{8w}$. This means $P(smoke(alice)) = 1$ and

$$P(smoke(bob)) = \lim_{\alpha \to \infty} \frac{k \cdot e^{8w} + k \cdot e^{9w}}{k \cdot e^{9w} + 2k \cdot e^{8w}} > P(smoke(carol)) = \lim_{\alpha \to \infty} \frac{k \cdot e^{9w}}{k \cdot e^{9w} + 2k \cdot e^{8w}}.$$

The result can be verified by LPMLN2ASP. For $w = 1$, the input program `smoke.lpmln` is

```
1 smoke(Y) :- smoke(X), influence(X, Y).
smoke(alice).    influence(alice, bob).    influence(bob, carol).
```

Executing `lpmln2asp -i smoke.lpmln -q smoke` outputs

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

```
smoke(alice) 1.00000000000000
smoke(bob) 0.788058442382915
smoke(carol) 0.576116884765829
```

as expected.

On the other hand, if (3.6) is understood under the MLN semantics (assuming `influence` relation is fixed as before), similar to above, one can compute

$$P(smoke(bob)) = \frac{e^{8w} + e^{9w}}{3e^{8w} + e^{9w}} = P(smoke(carol)).$$

In other words, the degraded probability along the transitive relation does not hold under the MLN semantics. This is related to the fact that Markov logic cannot express the concept of transitive closure correctly as it inherits the standard first-order logic semantics.

According to Theorem 2 in the paper by Lee and Wang (2016), MLN can be easily embedded in LP^{MLN} by adding a choice rule for each atom with an arbitrary weight, similar to the way propositional logic can be embedded in ASP using choice rules. Consequently, it is possible to use system LPMLN2ASP to compute MLN, which is essentially using an ASP solver to compute MLN.

Let `smoke.mln` be the resulting program. Executing `lpmln2asp -i smoke.mln -q smoke` outputs

```
smoke(alice) 1.0
smoke(bob)   0.650244590946
smoke(carol) 0.650244590946
```

which agrees with the computation above.

4 Turning tight LP^{MLN} into MLN

4.1 Translation

In the implementation of LP^{MLN} using MLN solvers, we limit attention to non-disjunctive logic programs that are tight. Extending Theorem 3 by Lee and Wang (2016) to non-ground programs, LP^{MLN} programs (possibly containing variables) can be turned into MLN instances by first rewriting each rule into Clark normal form

$$w: \quad p(\mathbf{x}) \leftarrow Body$$

using equality in the body, where $\mathbf{x}$ is a list of new distinct variables unique to each predicate p, and then adding the completion formulas

$$\alpha: \quad p(\mathbf{x}) \rightarrow \bigvee_{w:\ p(\mathbf{x}) \leftarrow Body \ \in \ \Pi} Body \tag{4.1}$$

for each atom $p(\mathbf{x})$. In fact, since the built-in algorithm in ALCHEMY for clausifying the completion formulas may yield an exponential blow-up, LPMLN2MLN implements an equivalent rewriting known as Tseytin transformation[4], which introduces an

[4] https://en.wikipedia.org/wiki/Tseytin_transformation

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

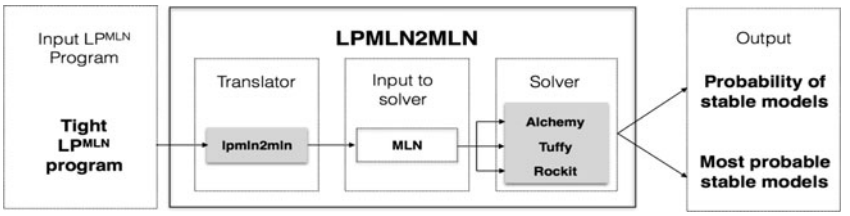


Fig. 2. Architecture of system LPMLN2MLN.

auxiliary predicate for each disjunctive term in (4.1). The resulting MLN instance (possibly containing variables) is fed into ALCHEMY, which grounds the MLN and performs probabilistic inference on the ground network[5].

For any MLN $\mathbb{L}$, let $\mathbb{L}^{hard}$ and $\mathbb{L}^{soft}$ denote the set of hard formulas and soft formulas in $\mathbb{L}$, respectively. For any set $\mathbb{L}$ of weighted formulas, let $\overline{\mathbb{L}}$ be the formulas obtained from $\mathbb{L}$ by dropping the weights. The following proposition justifies the equivalent rewriting using auxiliary atoms.

Proposition 1
For any MLN $\mathbb{L}$ of signature σ, let $F(\mathbf{x})$ be a subformula of some formula in $\mathbb{L}$ where $\mathbf{x}$ is the list of all free variables of $F(\mathbf{x})$, and let $\mathbb{L}^F_{Aux}$ be the MLN program obtained from $\mathbb{L}$ by replacing $F(\mathbf{x})$ with a new predicate $Aux(\mathbf{x})$ and adding the formula

$$\alpha \quad : \quad Aux(\mathbf{x}) \leftrightarrow F(\mathbf{x}).$$

For any interpretation I of $\mathbb{L}$, let I_{Aux} be the extension of I of signature $\sigma \cup \{Aux\}$ defined by $I_{Aux}(Aux(\mathbf{c})) = (F(\mathbf{c}))^I$ for every list $\mathbf{c}$ of elements in the Herbrand Universe. When $\overline{\mathbb{L}^{hard}}$ has at least one model, we have

$$P_{\mathbb{L}}(I) = P_{\mathbb{L}^F_{Aux}}(I_{Aux}).$$

4.2 System LPMLN2MLN

System LPMLN2ASP is an implementation of LP$^{\text{MLN}}$ based on the result in Section 4.1 using ALCHEMY (v2.0), TUFFY (v0.3), and ROCKIT (v0.5). The architecture of the system is shown in Figure 2.

The basic command line syntax of executing LPMLN2MLN is

```
lpmln2mln -i <input file> -r <output file> -q <query predicates>
     [-e <evidence file>]
     [-tuffy| -rockit| -alchemy] [-mln "<options for mln solvers>"]
```

which is similar to the command of executing LPMLN2ASP.

The syntax of the input language of LPMLN2MLN follows that of ALCHEMY, except that it uses a rule form. For example, consider again Example 1 in the paper by Lee and Wang (2016). In the input language of LPMLN2MLN, it is encoded as

[5] Without introducing auxiliary atoms in the process of clausification, most examples cannot be run on ALCHEMY because the CNF conversion method implemented in ALCHEMY is naive.

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

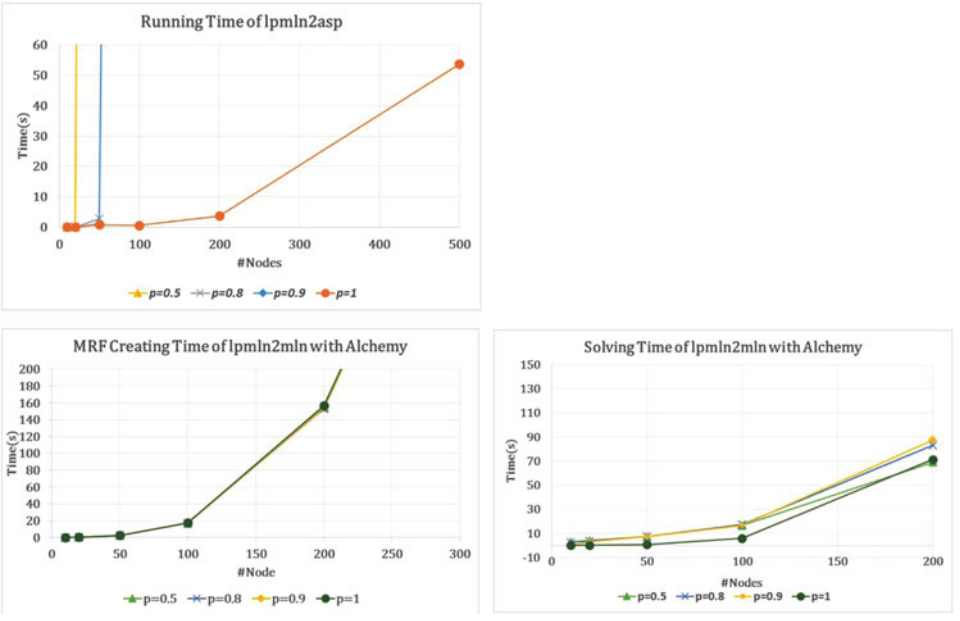


Fig. 3. Running statistics on finding relaxed clique.

```
entity={Jo}                 Bird(x) <= ResidentBird(x).
                            Bird(x) <= MigratoryBird(x).
Bird(entity)                <= ResidentBird(x) ^ MigratoryBird(x).
MigratoryBird(entity)
ResidentBird(entity)        2 ResidentBird(Jo)
                            1 MigratoryBird(Jo)
```

Executing

```
lpmln2mln -i bird.lpmln -r out -q Bird,ResidentBird, MigratoryBird
```

gives

```
Bird(Jo) 0.90296
ResidentBird(Jo) 0.667983
MigratoryBird(Jo) 0.235026
```

(When no MLN solver is specified in the command line, ALCHEMY is called by default.)

5 Comparison between two LPMLN implementations

Both LPMLN2ASP and LPMLN2MLN can compute conditional/marginal probability, as well as finding the most probable stable models (MAP estimates). The implementations use ASP and MLN solvers as blackboxes, so their performance depends on the underlying solvers. Although ASP solvers do not have a built-in concept of probabilistic reasoning, it is interesting to note how the optimal answer set finding is related to MAP estimates in probabilistic reasoning. Grounding in ASP solvers

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

is much more efficient than that in MLN solvers for the examples that we tested, but they have different characters. While grounding methods implemented in MLN solvers are not highly optimized, they do not ground the whole network; rather an essential part of a Markov network can be constructed from Markov blankets relevant to the query. Unlike LPMLN2ASP, system LPMLN2MLN utilizes approximate sampling-based inference methods in underlying MLN solvers. Consequently, its solving is more scalable but gives less accurate results. Its input program is restricted to tight LP^{MLN} programs and does not support advanced ASP constructs, such as aggregates. When the domain is small, our experience is that it is much more convenient to work with LPMLN2ASP because it supports many useful ASP constructs and its exact computation yields outputs that are easier to understand. Once we make sure the program is correct and we do not need advanced ASP constructs nor recursive definitions, we may use LPMLN2MLN for more scalable inference.

We report the running time statistics for both LPMLN2ASP and LPMLN2MLN on the example of finding a maximal "relaxed clique" in a graph, where the goal is to select as many nodes as possible while a penalty is assigned to each pair of disconnected nodes. The penalty assigned to disconnected nodes and the reward given to each node included in the subgraph define how much "relaxed" the clique is.

The LPMLN2ASP encoding of the relaxed clique example is

```
{in(X)} :- node(X).
disconnected(X, Y) :- in(X), in(Y), not edge(X, Y).
5  :- not in(X), node(X).
5  :- disconnected(X, Y).
```

The LPMLN2MLN encoding of the relaxed clique example is

```
{In(x)} <= Node(x).
Disconnected(x, y) <= In(x) ^ In(y) ^ !Edge(x, y).
5  <= !In(x) ^ Node(x)
5  <= Disconnected(x, y)
```

We use a Python script to generate random graphs with each edge generated with a fixed probability p. We experiment with $p = 0.5, 0.8, 0.9, 1$ and different numbers of nodes. For each problem instance, we perform MAP inference to find a maximal relaxed clique with both LPMLN2ASP and LPMLN2MLN. The timeout is 20 minutes. The experiments are performed on a machine powered by 4 Intel(R) Core(TM) i5-2400 CPU with OS Ubuntu 14.04.5 LTS and 8 GB memory.

Figure 3 shows running statistics of utilizing different underlying solvers. For LPMLN2ASP, grounding finishes almost instantly for all problem instances that we tested. We plot how solving times vary according to the number of nodes for different edge generation probabilities (top left graph). Roughly, solving time increases as the number of nodes increases. However, there is no clear correlation between solving time and the edge probability (i.e., the density of the graph). For $p = 0.5$, the LPMLN2ASP system first times out when $\#Nodes = 50$, while for both $p = 0.8$ and $p = 0.9$, it first times out when $\#Node = 100$. On the other hand, when $\#Node = 20$, solving time roughly increases as the edge probability increases except for $p = 0.5$. The running time is sensitive to particular problem instances, due to the exact

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

optimization algorithm CDNL-OPT (Gebser *et al.* 2011) used by CLINGO, which only terminates when a true optimal solution is found. The non-deterministic nature of CDNL-OPT also brings randomness on the path through which an optimal solution is found, which makes the running time differ even among similar-sized problem instances, while in general, as the size of the graph increases, the search space gets larger, thus the solving time increases.

For LPMLN2MLN with ALCHEMY (bottom left and bottom right), grounding (creating Markov random fields) becomes the bottleneck. It increases much faster than solving time, and times out first when $\#Nodes = 500$. Again, the running time increases as the number of nodes increases. On the other hand, unlike LPMLN2ASP, ALCHEMY uses MaxWalkSAT for MAP inference, which allows a suboptimal solution to be returned. The approximate nature of the method allows relatively consistent running times for different problem instances, as long as parameters such as the maximum number of iterations/tries are fixed among all experiments. The running times are not also much affected by the edge probability.

In general, LPMLN2MLN can be more scalable via parameter setting, while LPMLN2ASP grants better solution quality. LPMLN2MLN with TUFFY shows a similar behavior as LPMLN2MLN with ALCHEMY.

6 Using LP$^{\mathrm{MLN}}$ implementations to compute other languages

6.1 Computing ProbLog

ProbLog (De Raedt *et al.* 2007) can be viewed as a special case of the LP$^{\mathrm{MLN}}$ language (Lee and Wang 2016), in which soft rules are atomic facts only. The precise relation between the semantics of the two languages is stated by Lee and Wang (2016). System PROBLOG2 implements a native inference and learning algorithm which converts probabilistic inference problems into weighted model counting problems and then solves with knowledge compilation methods (Fierens *et al.* 2013). We compare the performance of LPMLN2ASP with that of PROBLOG2 on ProbLog input programs. We encode the problem of reachability in a probabilistic graph in both languages, and perform MAP inference ("given that there is a path between two nodes, what is the most likely graph?") as well as marginal probability computation ("given two particular nodes, what is the probability that there exists a path between them?"). We use a Python script to generate edges with probabilities randomly assigned. For the probabilistic facts $p :: \mathtt{edge}(n_1, n_2)$ $(0 < p < 1)$ in PROBLOG2, we write $ln(p/(1-p)) : \mathtt{edge}(n_1, n_2)$ for LPMLN2ASP, which makes the probability of the edge being true to be p and being false to be $1 - p$.

The path relation is defined in the input language of LPMLN2ASP as

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

and in the input language of PROBLOG2 as

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

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

Parameter	MAP	
	LPMLN2CL (CLINGO 4.5)	PROBLOG2
$\#edges = 9$	0.013s	0.192s
$\#edges = 25$	0.021s	Timeout
$\#edges = 81$	0.308s	Timeout
$\#edges = 100$	0.756s	Timeout
$\#edges = 225$	6.121s	Timeout
$\#edges = 400$	29.706s	Timeout

Parameter	Marginal		
	LPMLN2ASP (CLINGO 4.5)	PROBLOG2 Default Setting	PROBLOG2 Sample Based $\#_{Sample} = 1000$
$\#edges = 9$	0.520s	0.137s	1.878s
$\#edges = 10$	0.676s	1.468s	1.656s
$\#edges = 11$	1.396s	1.480s	1.684s
$\#edges = 12$	2.524s	1.781s	1.672s
$\#edges = 13$	4.995s	Timeout	1.692s
$\#edges = 14$	9.744s	Timeout	1.796s
$\#edges = 15$	19.568s	Timeout	1.732s
$\#edges = 16$	38.476s	Timeout	1.748s
$\#edges = 18$	164.192s	Timeout	16.676s
$\#edges = 19$	306.000s	Timeout	4.564s
$\#edges = 20$	637.584s	Timeout	4.020s
$\#edges = 21$	Timeout	Timeout	4.344s

Fig. 4. Running statistics on reachability in a probabilistic graph.

Figure 4 shows the running time of each experiment. LPMLN2ASP outperforms PROBLOG2 with the default setting (exact inference) in both MAP inference and marginal probability computation. However, both systems' marginal probability computations are not scalable because they enumerate all models. Using a sampling-based inference instead, PROBLOG2 is able to handle marginal probability computation more effectively (the MAP inference in PROBLOG2 is exact inference only). In general, compared to running on tight programs, PROBLOG2 is slow for non-tight programs such as the program we use here. A possible reason is that it has to convert the input program, combined with the query, into weighted Boolean formulas, which is expensive for non-tight programs.

6.2 Reasoning about probabilistic causal model

Lee *et al.* (2015) show how to represent Pearl's probabilistic causal model (Pearl 2000) by LP$^{\text{MLN}}$. Due to the acyclicity assumption on the causality, the LP$^{\text{MLN}}$ representation is tight, so we can use either implementation of LP$^{\text{MLN}}$ to compute probabilistic queries on Pearl's Causal Model (PCM). (Related to this, Appendix A (Lee *et al.* 2017) shows how Bayesian networks can be represented in LP$^{\text{MLN}}$.)

As an example, consider a probabilistic version of the firing squad example, shown in Figure 5. The court orders the execution (U) with probability p and Rifleman A is nervous (W) with probability q. The nervousness of Rifleman A causes him shooting at the prisoner (A). The court order causes the Captain to signal (C), which again causes Rifleman A and Rifleman B to shoot at the prisoner. Either of Rifleman A and Rifleman B shooting causes the prisoner's death (D). We illustrate how we use LP$^{\text{MLN}}$ systems to compute the counterfactual query "Given that the prisoner is dead, what is the probability that the prisoner would be alive if Rifleman A had not shot?" According to Pearl (2000), the answer is $\frac{(1-p)q}{1-(1-p)(1-q)}$.

Theorem 4 from the paper by Lee *et al.* (2015) states that the counterfactual reasoning in PCM can be reduced to LP$^{\text{MLN}}$ computation. The translation of PCM into LP$^{\text{MLN}}$ in Section 4.4 by Lee *et al.* (2015) can be represented in the input

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

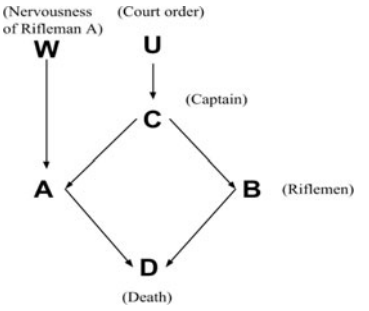


Fig. 5. Firing squad example.

language of LPMLN2ASP as follows, where `as`, `bs`, `cs`, `ds` are nodes in the twin network, `a1` means that `a` is true; `a0` means that `a` is false; other atoms are defined similarly. Let $p = 0.7$ and $q = 0.2$.

```
@log(0.7/0.3)  u.
@log(0.2/0.8)  w.

c :- u.
a :- c.

cs :- u, not do(c1), not do(c0).
as :- cs, not do(a1), not do(a0).
as :- w, not do(a1), not do(a0).
bs :- cs, not do(b1), not do(b0).
ds :- as, not do(d1), not do(d0).
ds :- bs, not do(d1), not do(d0).
```

```
a :- w.
b :- c.
d :- a.
d :- b.

cs :- do(c1).
as :- do(a1).
bs :- do(b1).
ds :- do(d1).
```

To represent the counterfactual query, the evidence file contains

```
do(a0).
:- not d.
```

Note the different ways that intervention (`do(a0)`) and observation (`d`) are encoded.

With the command `lpmln2asp -i pcm.lp -r out -e evid.db -q ds`, we obtain `ds 0.921047297896`, which means there is a 8% chance that the prisoner would be alive.

7 Conclusion

We presented two implementations of LP$^{\mathrm{MLN}}$ using ASP and MLN solvers. This is based on extending the translations that turn LP$^{\mathrm{MLN}}$ into answer set programs and Markov logic to allow non-ground weighted rules. Although the input language of CLINGO does not have a built-in concept of probabilistic reasoning, its optimal answer set finding algorithm is shown to be effective in finding MAP estimates (most probable stable models). It is also interesting that the efficient stable model enumeration leads to competitive exact probability computation.

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

The implementations also serve for other probabilistic logic languages that are shown to be embeddable in LP^{MLN}, such as ProbLog, Pearl's causal models, Bayesian networks, and P-log.

PrASP (Nickles 2016) is another system whose input language extends answer set programs with weights, although the semantics is different from that of LP^{MLN}. While LP^{MLN} systems turn an input program into another input program that can be computed by existing systems, PrASP implements several native inference algorithms, including model counting, simulated annealing, flip-sampling, iterative refinement, etc. The formal relationships between the language of PrASP and other languages have not been established.

The LP^{MLN} implementations suggest how to combine the solving techniques from the two solvers. While CLINGO is efficient for grounding, MLN solvers consider subnetworks derived from the Markov blanket of query atoms and evidence. While CLINGO does exact inference only, MLN solvers can perform sampling-based approximate inference. Future work includes building a native algorithm for LP^{MLN} borrowing the techniques from the related systems.

Acknowledgements

We are grateful to Brandon Gardell, Roland Kaminski, Zhun Yang, and the anonymous referees for their useful comments and pointers.

Supplementary materials

For supplementary material for this article, please visit https://doi.org/10.1017/S1471068417000400

References

Balai, E. and Gelfond, M. 2016. On the relationship between P-log and LP^{MLN}. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 915–921.

Baral, C., Gelfond, M. and Rushton, J. N. 2009. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming 9,* 1, 57–144.

Buccafurri, F., Leone, N., and Rullo, P. 2000. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering 12,* 5, 845–860.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F. and Schaub, T. 2012. ASP-Core-2: Input language format. *ASP Standardization Working Group, Technical Report*.

De Raedt, L., Kimmig, A. and Toivonen, H. 2007. ProbLog: A probabilistic prolog and its application in link discovery. In *Proc. of IJCAI*, vol. 7. 2462–2467.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G. and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming 15,* 3, 358–401.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2011. Multi-criteria optimization in answer set programming. In *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 11. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 1–10.

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press

Lee, J., Meng, Y. and Wang, Y. 2015. Markov logic style weighted rules under the stable model semantics. In *Technical Communications of the 31st International Conference on Logic Programming*.

Lee, J., Talsania, S. and Wang, Y. 2017. Online appendix for the paper "Computing LP^{MLN} using ASP and MLN solvers". TPLP Archive. https://doi.org/10.1017/S1471068417000400

Lee, J. and Wang, Y. 2016. Weighted rules under the stable model semantics. In *Proc. of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 145–154.

Lee, J. and Yang, Z. 2017. LPMLN, weak constraints, and P-log. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*, 1170–1177.

Nickles, M. 2016. A tool for probabilistic reasoning based on logic programming and first-order theories under stable model semantics. In *Proc. of European Conference on Logics in Artificial Intelligence (JELIA)*, 369–384.

Pearl, J. 2000. *Causality: Models, Reasoning and Inference*. Vol. 29. Cambridge University Press.

Richardson, M. and Domingos, P. 2006. Markov logic networks. *Machine Learning 62,* 1–2, 107–136.

https://doi.org/10.1017/S1471068417000400 Published online by Cambridge University Press