Computing a Probabilistic Extension of Answer Set Program Language Using ASP and Markov Logic Solvers

by

Samidh Talsania

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2017 by the
Graduate Supervisory Committee:

Joohyung Lee, Chair
Chitta Baral
Yezhou Yang

ARIZONA STATE UNIVERSITY

August 2017

ABSTRACT

$\mathrm{LP}^{\mathrm{MLN}}$ is a recent probabilistic logic programming language which combines both Answer Set Programming (ASP) and Markov Logic. It is a proper extension of Answer Set programs which allows for reasoning about uncertainty using weighted rules under the stable model semantics with a weight scheme that is adopted from Markov Logic. $\mathrm{LP}^{\mathrm{MLN}}$ has been shown to be related to several formalisms from the knowledge representation (KR) side such as ASP and P-Log, and the statistical relational learning (SRL) side such as Markov Logic Networks (MLN), Problog and Pearl's causal models (PCM). Formalisms like ASP, P-Log, Problog, MLN, PCM have all been shown to embeddable in $\mathrm{LP}^{\mathrm{MLN}}$ which demonstrates the expressivity of the language. Interestingly, $\mathrm{LP}^{\mathrm{MLN}}$ has also been shown to reducible to ASP and MLN which is not only theoretically interesting, but also practically important from a computational point of view in that the reductions yield ways to compute $\mathrm{LP}^{\mathrm{MLN}}$ programs utilizing ASP and MLN solvers. Additionally, the reductions also allow the users to compute other formalisms which can be reduced to $\mathrm{LP}^{\mathrm{MLN}}$.

This thesis realizes two implementations of $\mathrm{LP}^{\mathrm{MLN}}$ based on the reductions from $\mathrm{LP}^{\mathrm{MLN}}$ to ASP and $\mathrm{LP}^{\mathrm{MLN}}$ to MLN. This thesis first presents an implementation of $\mathrm{LP}^{\mathrm{MLN}}$ called LPMLN2ASP that uses standard ASP solvers for computing MAP inference using *weak constraints*, and marginal and conditional probabilities using stable models enumeration. Next, in this thesis, another implementation of $\mathrm{LP}^{\mathrm{MLN}}$ called LPMLN2MLN is presented that uses MLN solvers which apply *completion* to compute the *tight* fragment of $\mathrm{LP}^{\mathrm{MLN}}$ programs for MAP inference, marginal and conditional probabilities. The computation using ASP solvers yields *exact* inference as opposed to approximate inference using MLN solvers. Using these implementations, the usefulness of $\mathrm{LP}^{\mathrm{MLN}}$ for computing other formalisms is demonstrated by reducing them to $\mathrm{LP}^{\mathrm{MLN}}$. The thesis also shows how the implementations are better than the native solvers of some of these formalisms on certain domains. The implementations makes use of the current state of the art solving technologies in ASP and

MLN, and therefore they benefit from any theoretical and practical advances in these technologies, thereby also benefiting the computation of other formalisms that can be reduced to $\mathrm{LP}^{\mathrm{MLN}}$. Furthermore, the implementation also allows for certain SRL formalisms to be computed by ASP solvers, and certain KR formalisms to be computed by MLN solvers.

AKCNOWLEDGMENTS

Working on my Masters thesis has been a fun and exciting journey which would not have been possible without support from a lot of people. First of all, I would like to thank my advisor Dr. Joohyung Lee for his constant support and guidance throughout the course of this thesis. His enthusiasm has guided me from my first lecture introducing ASP right into this Masters thesis program. Dr. Lee worked closely with me throughout my Masters thesis work and I have learned a lot from him. I consider myself fortunate to have a dedicated and a patient advisor.

This thesis wouldn't have been possible without help from Zhun Young and Yi Wang to whom I have asked countless questions and who were patient enough to explain it to me until I understood them. I am also grateful to Nikhil, Manjula, and Brandon for providing critical feedback to my work. I would also like to thank my friends Abhinav, Umang, Manan, Rakhi, Anurag, Krishna, Arjav, Jiten, Shalmali, Kaustav, Shubham and others for always being supportive during my research and providing the necessary distractions. Thank you all for making my time at ASU memorable.

I would like to thank my family who supported me in pursuing my dreams of doing a Masters in computer science and encouraged me to undertake this research. Thank you, mummy, papa and my dear brother Jainam for always believing in me even when I didn't. Finally, I would like to thank Pooja for always being loving, supporting and encouraging during this journey. Thank you all.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Knowledge Representation and Reasoning (KRR) is a field of Artificial Intelligence that is dedicated to studying about representing information about the world in a way that can be utilized by computers for automated reasoning. Answer Set Programming (ASP) (Gelfond and Lifschitz, 1988), a formalism for KRR, is a declarative programming paradigm which is based on the stable model semantics, also called the answer set semantics.

Answer set programming has its roots in nonmonotonic reasoning, deductive databases and logic programming with negation as failure. Answer set programming is a primary candidate tool for knowledge representation because of the emergence of highly efficient solvers and has become a major driving force for KRR. ASP has been successfully applied in a large number of applications because of its expressivity which allows simple ways to encode concepts like defaults and aggregates. ASP is particularly suited for solving difficult combinatorial search problems like plan generation, product configuration, diagnosis and graph theoretical problems.

However, the language of ASP is still deterministic and it is difficult to express uncertain or probabilistic knowledge about a domain. To overcome this limitation of ASP, a few probabilistic extensions to ASP have been proposed such as *weak constraints* (Buccafurri *et al.*, 2000) and P-Log (Baral *et al.*, 2009a). While weak constraints enable an ASP program to find an optimal stable model, it still does not have the notion of probability.

LP^{MLN} (Lee and Wang, 2016a) is a recent probabilistic logic language that combines ASP and Markov Logic to overcome this limitation of ASP by introducing the notion of probability into the stable model semantics. Markov Logic (Richardson and Domingos, 2006) combines first order logic with Markov Networks. Markov logic does not make

the assumption of independently and identically distributed data made by many statistical learners and leverages first-order logic to model complex dependencies. A Markov Logic Network (MLN) is a representation that is used to encode domains with Markov Logic semantics. MLN have been successfully used for tasks such as collective classification, logistic regression, social network analysis, entity resolution, information extraction, etc. Approximate methods like MC-SAT, Markov Chain Monte Carlo are used to perform inference over a relevant minimal subset of the generated markov network which is required for answering the query. Usage of sampling techniques for inference allow MLN inference to scale well. MLN is however based on classical model semantics and therefore concepts like inductive definitions, defaults, aggregates cannot be directly encoded using MLN representation.

$\mathrm{LP}^{\mathrm{MLN}}$ introduces the notion of weighted rules under the stable model semantics where the weight scheme is adopted from Markov Logic. This provides a versatile language to overcome the deterministic nature of the stable model semantics. The logical component of $\mathrm{LP}^{\mathrm{MLN}}$ is stable models instead of classical models adopted in Markov Logic. $\mathrm{LP}^{\mathrm{MLN}}$ extends ASP programs probabilistically in a way similar to how Markov Logic extends SAT. Knowledge Reasoning (KR) formalisms such as P-Log and ASP can be embedded into $\mathrm{LP}^{\mathrm{MLN}}$ (Lee and Wang, 2016b; Lee and Yang, 2017b). On a similar note, various Statistical Relational Learning (SRL) formalisms like MLN, Problog and Pearl's Causal Models can be embedded into $\mathrm{LP}^{\mathrm{MLN}}$ as well (Lee and Wang, 2016b; Lee *et al.*, 2015). This proves $\mathrm{LP}^{\mathrm{MLN}}$ to be a viable middle ground language that links the KR formalisms and SRL formalisms. Moreover, $\mathrm{LP}^{\mathrm{MLN}}$ itself has been shown to be translatable to languages like ASP and MLN. This yields methods to compute $\mathrm{LP}^{\mathrm{MLN}}$ programs using ASP and MLN solvers using translation from $\mathrm{LP}^{\mathrm{MLN}}$ to ASP and $\mathrm{LP}^{\mathrm{MLN}}$ to MLN respectively.

$\mathrm{LP}^{\mathrm{MLN}}$ can be translated to an ASP program using weak constraints. While weak constraints in ASP impose a preference over the stable models of a program, adding weak

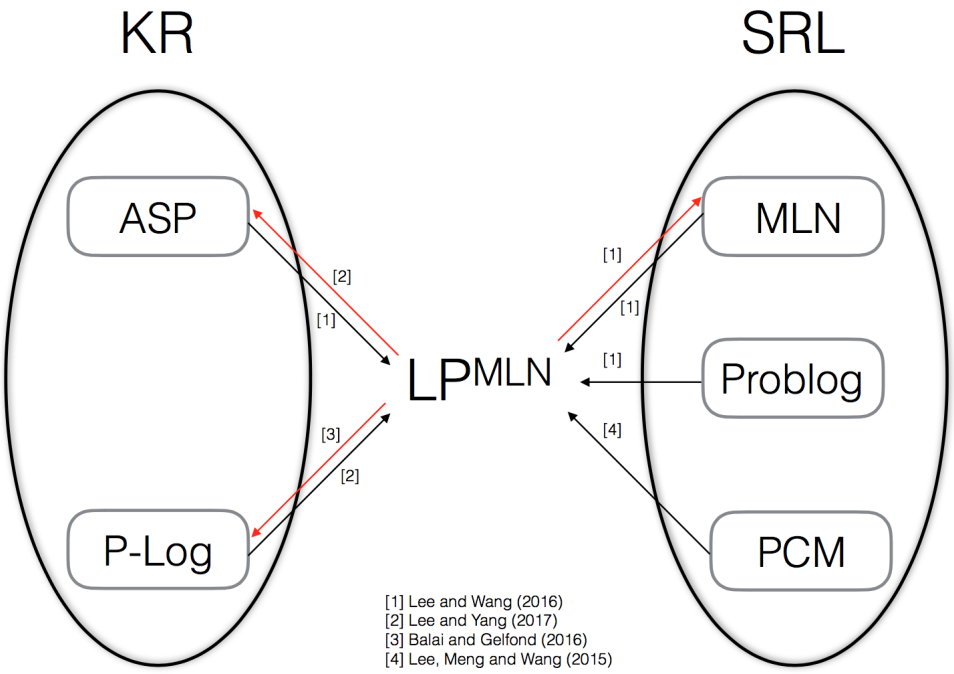

Figure 1.1: Relationship of $\mathrm{LP}^{\mathrm{MLN}}$ with Other Formalisms

constraints to a program does not add or remove stable models of the original program. The translation shown in Theorem 1 (Lee and Yang, 2017b) states how an $\mathrm{LP}^{\mathrm{MLN}}$ program can be translated to an ASP program with weak constraints such that the stable models with the highest probability precisely correspond to the optimal stable models of the program with weak constraints. The thesis uses this result not only to compute the $\mathrm{LP}^{\mathrm{MLN}}$ program and perform *Maximum A Posteriori* (MAP) inference but also to compute the marginal and conditional probability of atoms using standard ASP solvers by enumerating over all the stable models of a program. This gives gold standard results for any program since the probability is computed using *exact* methods. Since this probability calculation method is based on stable models enumeration, it also provides a way to find the probability of each stable model of the program.

The relationship between $\mathrm{LP}^{\mathrm{MLN}}$ and MLN is analogous to the relationship between ASP and SAT and consequently many results about the relationship between ASP and SAT carry over between $\mathrm{LP}^{\mathrm{MLN}}$ and MLN. One such result is *completion* (Clark, 1978) which allows the computation of *tight* $\mathrm{LP}^{\mathrm{MLN}}$ programs using Markov Logic Solvers similar to the way *tight* ASP programs can be computed by SAT solvers. While in theory loop formulas can be used to translate any $\mathrm{LP}^{\mathrm{MLN}}$ program to an equivalent MLN program, in practice, this method does not yield an effective computation. Thus, we limit our attention to the *tight* fragment of $\mathrm{LP}^{\mathrm{MLN}}$ programs.

The thesis explores whether these theoretical translations can result in practical implementation of the $\mathrm{LP}^{\mathrm{MLN}}$ system. The thesis first presents an implementation of $\mathrm{LP}^{\mathrm{MLN}}$ called LPMLN2ASP which uses weak constraints to translate an $\mathrm{LP}^{\mathrm{MLN}}$ program to an ASP program which uses CLINGO as the solver.

- The alternative translation as stated in (Lee and Yang, 2017b) is realized in this implementation which can compute the most probable stable models of the $\mathrm{LP}^{\mathrm{MLN}}$ program which corresponds to the optimal stable models of the program with weak constraints.

- We add a *probability computation module* to CLINGO to calculate the marginal and conditional probability of an atom by examining the *unsat* atoms which are introduced by the translation in each stable model. In doing so, we reformulate the $\mathrm{LP}^{\mathrm{MLN}}$ semantics in a "penalty" based way and introduce a translation which guarantees that a safe $\mathrm{LP}^{\mathrm{MLN}}$ program is always converted to a safe ASP program.

- We show that other formalisms which are proven to be translatable in $\mathrm{LP}^{\mathrm{MLN}}$ like P-log, ProbLog, Markov Logic Networks can be computed using ASP solvers.

- We also use this translation to show a method to debug inconsistent ASP programs and find out which rules cause inconsistency.

Further, this thesis develops another implementation of $\mathrm{LP}^{\mathrm{MLN}}$ called LPMLN2MLN which applies *completion* and a transformation similar to *Tseitin's transformation* (Tseitin, 1968), and uses MLN solvers like ALCHEMY, TUFFY and ROCKIT for computing the $\mathrm{LP}^{\mathrm{MLN}}$ program. LPMLN2MLN translates *tight* $\mathrm{LP}^{\mathrm{MLN}}$ programs to its equivalent MLN encoding.

- The direct implementation of *completion* method generates formulas that blow up when it is converted to the Conjunctive Normal Form (CNF) which is required for the formulas to be processed by MLN solvers. This blow up is exponential in the size of the formula in the worst case and the MLN solvers do not handle it efficiently. This results in timeouts during the grounding phase of the computation. To overcome this, we introduce *auxiliary* atoms for each disjunctive term in the formula which is generated by the *completion* method. This avoids the exponential blow up that results from the naive implementation of the CNF conversion method used in ALCHEMY.

- Furthermore, the input languages of TUFFY and ROCKIT do not allow nested formulas which are required to encode completion formulas. LPMLN2MLN takes care of these differences in the input language of different MLN solvers.

Additionally, we use these implementations to test on several benchmark problems. We analyze the two implementations based on running time, grounding time and accuracy of the outputs obtained from LPMLN2MLN and LPMLN2ASP. We then discuss the relative strengths and weakness of these two implementations based on the experiments performed.

This thesis is organized as follows: In Chapter 2, we give a review of the background theories on $\mathrm{LP}^{\mathrm{MLN}}$, turning $\mathrm{LP}^{\mathrm{MLN}}$ to ASP and turning $\mathrm{LP}^{\mathrm{MLN}}$ to MLN and set up the terminologies used for translations. Chapter 3 introduces LPMLN2ASP system, its architecture and describes the translations used and its usage. Chapter 4 introduces LPMLN2MLN and describes the completion procedure used in the translation, the Tseytin's transforma-

tion used during the completion procedure, the syntax of the input $\mathrm{LP}^{\mathrm{MLN}}$ language and its usage. Chapter 5 describes the examples of formalisms that can be executed using these implementations and report the results of the experiments done on certain benchmark problems.

Chapter 2

BACKGROUND

2.1 Review: Stable model semantics

In this section we introduce the concepts needed in this thesis. We assume a finite first-order signature σ that contains no function constants of positive arity. There are finitely many Herbrand interpretations of σ each of which are finite as well. A *rule* R over signature σ is of the form

$$A_1; ...; A_k \leftarrow A_{k+1}, ..., A_l, \mathbf{not}\ A_{l+1}, ..., \mathbf{not}\ A_m, \mathbf{not\ not}\ A_{m+1}, ..., \mathbf{not\ not}\ A_n \quad (2.1)$$

Each A_i is an *atom* of σ possibly containing variables. An atom is a predicate constant followed by terms. In this definition, **not** stands for *default* negation, comma for conjunction and semi-colon for disjunction. $A_1; ...; A_k$ is called the $head$ of the rule and $A_{k+1}, ..., A_l, \mathbf{not}\ A_{l+1}, ..., \mathbf{not}\ A_m, \mathbf{not\ not}\ A_{m+1}, ..., \mathbf{not\ not}\ A_n$ is called the *body* of the rule. We can write $\{A_1\}^{ch} \leftarrow body$, where A_1 is an atom, to denote the rule $A_1 \leftarrow body, \mathbf{not\ not}\ A_1$. This expression is called a *choice* rule. If the head of the rule $A_1; ...; A_k$ is empty ($\bot$) the rule is called a *constraint*.

A logic programs is a set of rules R. A logic program is called *ground* if it contains no variables. Grounding replaces a program with its equivalent program without variables.

For a ground program Π and an interpretation I, Π^I denotes the *reduct* of Π relative to I. Π^I consists of $A_1; ...; A_k \leftarrow A_{k+1}, ..., A_l$ for all rules in Π such that $I \models \mathbf{not}\ A_{l+1}, ..., \mathbf{not}\ A_m, \mathbf{not\ not}\ A_{m+1}, ..., \mathbf{not\ not}\ A_n$. The Herbrand interpretation I is called the *stable model* of Π if I is a *minimal* Herbrand model of Π^I. Here, minimality is understood in terms of set inclusion. We identify an Herbrand interpretation with the set of atoms that are true in it.

Example 1. Consider the program from (Lee and Wang, 2015).

$$p \leftarrow q$$

$$q \leftarrow p$$

$$p \leftarrow not\ r$$

$$r \leftarrow not\ p$$

The stable models for Π are $\{r\}$ and $\{p, q\}$. The reduct of Π relative to $\{p, q\}$ is $\{p \leftarrow q.\ \ q \leftarrow p.\ \ p\}$ for which $\{p, q\}$ is the minimal model. The reduct relative to $\{r\}$ is $\{p \leftarrow q.\ q \leftarrow p.\ r\}$ for which $\{r\}$ is the minimal model.

2.2 Review: Weak Constraints

Weak constraints are part of the ASP Core 2 language (Calimeri *et al.*, 2012) and are implemented in standard ASP solvers such as CLINGO. A weak constraint is of the form

$$:\sim A_{k+1}, ..., A_l, \mathbf{not}\ A_{l+1}, ..., \mathbf{not}\ A_m, \mathbf{not\ not}\ A_{m+1}, ..., \mathbf{not\ not}\ A_n\ [w@l, t_1, ..., t_o]$$

where each A_i is an atom of signature σ, w is a *real number* denoting the weight of the weak constraint, l is an *interger* denoting the level, and $t_1, ..., t_o$ is a list of terms. Unlike constraints in ASP, weak constraints cannot be used to derive a rule or prune out stable models, rather a weight associated with the weak constraint body is added to the stable model if the body is true.

Semantics of weak constraints (Calimeri *et al.*, 2012)

Weak constraints impose an ordering over all the answer sets of a program, in a way specifying which answer set is "better" than others. A weak constraint rule W is *safe* if every

variable in W occurs in atleast one of the positive literals $A_{k+1}, ..., A_l$ of W. At each priority level l, the aim is to discard models that do not minimize the sum of the weights of the ground weak constraints with level l whose bodies are true. Higher levels are minimized first. Terms determine which ground weak constraints are unique (only unique weight tuples are considered when adding weights).

For any integer l (level) and an interpretation I for a grounded program P, let

$$\begin{aligned} weak(P, I) = &\{(w@l, t_1, ..., t_o)\} \mid \\ &:\sim A_{k+1}, ..., A_l, \mathbf{not}\ A_{l+1}, ..., \mathbf{not}\ A_m, \mathbf{not\ not}\ A_{m+1}, ..., \\ &\qquad \mathbf{not\ not}\ A_n\ [w@l, t_1, ..., t_o] \\ &occurs\ in\ P\ and \\ &A_{k+1}, ..., A_l, \mathbf{not}\ A_{l+1}, ..., \mathbf{not}\ A_m, \mathbf{not\ not}\ A_{m+1}, ..., \\ &\qquad \mathbf{not\ not}\ A_n\ is\ true\ in\ I\}, \end{aligned}$$

then

$$P_l^I = \sum_{w@l,t_1,...,t_m \in weak(P,I), w \text{ is an integer}} w$$

denotes the sum of weights w for an interpretation I for level l. An answer set I of P is *dominated* by an answer set I' of P if there is some integer l such that $P_l^{I'} < P_l^I$ and $P_{l'}^{I'} = P_{l'}^I$ for all integers $l' > l$. An answer set I of P is *optimal* if there is no answer set I' of P such that I is dominated by I'.

Example 2. Consider an ASP program with weak constraints

$$\{p, q\}^{ch} \tag{r1}$$

$$:\sim p\ [10@0] \tag{r2}$$

$$:\sim q\ [5@1] \tag{r3}$$

The *optimal* answer set for the above program is the one with the minimum weight at highest level. Rule (r2) says that add weight 10 at level 0 for all stable models in which p is included. Similarly, Rule (r3) says that add weight at level 1 for all stable models in which q is included. This program results in 4 stable models, $\varnothing, \{p\}, \{q\}, \{p, q\}$. The weights at each level for the stable models is given by

I	$weight@0$	$weight@1$
$\varnothing$	0	0
$\{p\}$	10	0
$\{q\}$	0	5
$\{p, q\}$	10	5

$\varnothing$ has the lowest weight at level 1 and thus it is the *optimal* answer set of the program.

2.3 Review: $\mathrm{LP}^{\mathrm{MLN}}$

An $\mathrm{LP}^{\mathrm{MLN}}$ program is a set of weighted rules $w \ : \ R$, where R is a rule as described in 2.1 and w is a real number denoting the weight of the rule or α denoting infinite weight. A rule is called a *hard* rule if its weight is α and a *soft* rule if the weight is w. We assume the same signature σ as in the stable model semantics. An $\mathrm{LP}^{\mathrm{MLN}}$ program is called *ground* if the rules contain no variables. Grounding replaces a program with its equivalent program without variables. Each of the ground rules receive the same weight as original rules.

$\mathrm{LP}^{\mathrm{MLN}}$ semantics

For any $\mathrm{LP}^{\mathrm{MLN}}$ program Π, we denote the unweighted logic program obtained from Π as $\overline{\Pi}$, i.e., $\overline{\Pi} = \{R \mid w \ : \ R \in \Pi\}$. For any interpretation I of Π, by Π_I we denote a set of rules $w \ : \ R$ in Π such that $I \vDash R$, by Π^{hard} we denote the set of all hard rules in Π and by $SM'[\Pi]$ we denote the set $\{I \mid I \text{ is a stable model of } \overline{\Pi_I} \text{ that satisfy } \overline{\Pi^{hard}}\}$. The *unnormalized weight* of an interpretation I under Π, denoted by $W_\Pi(I)$, is defined as

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

The *normalized weight* a.k.a. *probability* of an interpretation I of Π, denoted by $P_\Pi(I)$, is given by

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

where I is called a *probabilistic stable model* of Π if $P_\Pi(I) \neq 0$.

Example 3. Consider an example from (Lee and Wang, 2016b).

$$\alpha \ : Bird(Jo) \leftarrow ResidentBird(Jo) \tag{r1}$$

$$\alpha \ : Bird(Jo) \leftarrow MigratoryBird(Jo) \tag{r2}$$

$$\alpha \ : \leftarrow ResidentBird(Jo)\ ,\ MigratoryBird(Jo) \tag{r3}$$

$$2 \ : ResidentBird(Jo) \tag{r4}$$

$$1 \ : MigratoryBird(Jo) \tag{r5}$$

The table below shows the *satisfied rules*, *unnormalized weight* and *probability* of each interpretation I of the above program.

I	Π_I	$W_\Pi(I)$	$P_\Pi(I)$
$\varnothing$	$\{r_1, r_2, r_3\}$	$e^{3\alpha}$	$\frac{e^0}{e^2+e^1+e^0}$
$\{R(Jo)\}$	$\{r_1, r_3, r_4\}$	$e^{2\alpha+2}$	0
$\{M(Jo)\}$	$\{r_1, r_3, r_5\}$	$e^{2\alpha+1}$	0
$\{B(Jo)\}$	$\{r_1, r_2, r_3\}$	0	0
$\{R(Jo), B(Jo)\}$	$\{r_1, r_2, r_3, r_4\}$	$e^{3\alpha+2}$	$\frac{e^2}{e^2+e^1+e^0}$
$\{M(Jo), B(Jo)\}$	$\{r_1, r_2, r_3, r_5\}$	$e^{3\alpha+1}$	$\frac{e^1}{e^2+e^1+e^0}$
$\{R(Jo), M(Jo)\}$	$\{r_4, r_5\}$	e^3	0
$\{R(Jo), M(Jo), B(Jo)\}$	$\{r_1, r_2, r_4, r_5\}$	$e^{2\alpha+3}$	0

There are 7 stable models of Π. $\{B(Jo)\}$ is not a stable model of $\{r_1, r_2, r_3\}$ and hence its weight is 0 according to the definition. Therefore there are only 3 probabilistic stable models $\varnothing, \{R(Jo), B(Jo)\}, \{M(Jo), B(Jo)\}$. The model $\{R(Jo), B(Jo)\}$ has the highest weight and it is the most probable stable model of the program.

2.4 Review: LP$^{\text{MLN}}$ to ASP (Lee and Yang, 2017a)

For any LP$^{\text{MLN}}$ program Π, the translation `lpmln2wc(`Π`)` is defined as follows. An LP$^{\text{MLN}}$ rule of the form

$$w_i \ : \ Head_i \leftarrow Body_i$$

is turned into

$$unsat(i) \ \leftarrow \ Body_i, not \ Head_i$$

$$Head_i \leftarrow \ Body_i, not \ unsat(i)$$

$$:\sim unsat(i) \ [w_i@l]$$

where i is the index of the rule, $Head_i$ is the head of the rule, $Body_i$ is the body of the rule and l is the level. $l = 1$ if w_i is α and $l = 0$ otherwise. $unsat$ is a new predicate that is introduced by this translation.

Corollary 2 from (Lee and Yang, 2017a) states that for any LP^{MLN} program Π, there is a $1-1$ correspondence ϕ between the most probable stable models of Π and the optimal stable models of `lpmln2wc(`Π`)`, where $\phi(I) = I \cup \{unsat(i) \mid w_i : R_i \in \Pi, I \not\models R_i\}$. While the most probable stable models of Π and the optimal stable models of the translation coincide, their weights and penalties are not proportional to each other. The former is defined by an exponential function whose exponent is the sum of the weights of the satisfied formulas, while the latter simply adds up the penalties of the unsatisfied formulas. On the other hand, they are monotonically increasing/decreasing as more formulas are unsatisfied.

Example 4. Consider the same example as the previous section from (Lee and Wang, 2016b).

$$\alpha : Bird(Jo) \leftarrow ResidentBird(Jo)$$

$$\alpha : Bird(Jo) \leftarrow MigratoryBird(Jo)$$

$$\alpha : \leftarrow ResidentBird(Jo)\ ,\ MigratoryBird(Jo)$$

$$2 : ResidentBird(Jo)$$

$$1 : MigratoryBird(Jo)$$

Using the translation defined before, the above program can be translated into

$$
\begin{aligned}
unsat(1) &\leftarrow ResidentBird(Jo), not\ Bird(Jo)\\
Bird(Jo) &\leftarrow ResidentBird(Jo), not\ unsat(1)\\
&:\sim unsat(1)\ [1@1]\\
unsat(2) &\leftarrow MigratoryBird(Jo), not\ Bird(Jo)\\
Bird(Jo) &\leftarrow MigratoryBird(Jo), not\ unsat(2)\\
&:\sim unsat(2)\ [1@1]\\
unsat(3) &\leftarrow ResidentBird(Jo), MigratoryBird(Jo)\\
&\leftarrow ResidentBird(Jo), MigratoryBird(Jo), not\ unsat(3)\\
&:\sim unsat(3)\ [1@1]\\
unsat(4) &\leftarrow not\ ResidentBird(Jo)\\
ResidentBird(Jo) &\leftarrow not\ unsat(4)\\
&:\sim unsat(4)\ [2@0]\\
unsat(5) &\leftarrow not\ MigratoryBird(Jo)\\
MigratoryBird(Jo) &\leftarrow not\ unsat(5)\\
&:\sim unsat(5)\ [1@0]
\end{aligned}
$$

The optimal stable model of the program is $\{ResidentBird(Jo), Bird(Jo), unsat(5)\}$. This model has the lowest weight, 0 at $l = 1$ and 1 at $l = 0$ which corresponds to the most probable stable model in Example 3.

Since hard rules encode definite knowledge, one may not want any hard rules to be violated. The above translation can then be turned into a simple translation by turning all hard rules into the usual ASP rules instead [1] . The translation then becomes

$$\begin{aligned}
\alpha\ &: Bird(Jo) \leftarrow ResidentBird(Jo) \\
\alpha\ &: Bird(Jo) \leftarrow MigratoryBird(Jo) \\
\alpha\ &: \leftarrow ResidentBird(Jo)\ ,\ MigratoryBird(Jo) \\
unsat(3) &\leftarrow not\ ResidentBird(Jo) \\
ResidentBird(Jo) &\leftarrow not\ unsat(3) \\
&:\sim unsat(3)\ [2@0] \\
unsat(4) &\leftarrow not\ MigratoryBird(Jo) \\
MigratoryBird(Jo) &\leftarrow not\ unsat(4) \\
&:\sim unsat(4)\ [1@0]
\end{aligned}$$

2.5 Review: $\mathrm{LP}^{\mathrm{MLN}}$ to MLN (Lee and Wang, 2016b)

The stable models of a tight logic program coincide with the models of the program's completion (Erdem and Lifschitz, 2003). The same method can be extended to $\mathrm{LP}^{\mathrm{MLN}}$ program such that the probability queries involving the stable models can be computed using existing implementations of MLNs like ALCHEMY. The *completion* of an $\mathrm{LP}^{\mathrm{MLN}}$ program Π as $Comp(\Pi)$, is defined as an MLN which is the union of Π and the hard

[1] This translation can only be used when Π has atleast one stable model.

formula

$$\alpha : A \rightarrow \bigvee_{\substack{w:A_1 \vee \ldots \vee A_k \leftarrow Body \in \Pi \\ A \in \{A_1,\ldots,A_k\}}} \left(Body \wedge \bigwedge_{A' \in \{A_1,\ldots,A_k\} \setminus \{A\}} \neg A' \right)$$

for each ground atom A, where A is as defined in 2.1.

The result is an extension of the completion from (Lee and Lifschitz, 2003) by assigning infinite weight α to the completion formulas. For an $\mathrm{LP}^{\mathrm{MLN}}$ program Π and interpretation I, let $\overline{\Pi}$ denote the set of unweighted rules $\overline{\Pi} = \{R \mid w : R \in \Pi\}$, Π^{hard} denote the set of all hard rules of Π, Π_I denote the set of rules satisfied by interpretation I and $SM'[\Pi]$ denote the set $\{I \mid I$ is a stable model of $\overline{\Pi_I}$ that satisfy $\overline{\Pi^{hard}}\}$. Then, by Theorem 3 from (Lee and Wang, 2016b), any tight $\mathrm{LP}^{\mathrm{MLN}}$ program Π where $SM'[\Pi]$ is not empty, Π and $Comp(\Pi)$ have the same probability distribution, where Π follows the $\mathrm{LP}^{\mathrm{MLN}}$ semantics and $Comp(\Pi)$ follows the MLN semantics. The theorem can be generalized to non-tight programs by using loop formulas (Lin and Zhao, 2003).

Example 5. Consider the same example as previous section from (Lee and Wang, 2016b).

$$\alpha \ : Bird(Jo) \leftarrow ResidentBird(Jo) \tag{r1}$$

$$\alpha \ : Bird(Jo) \leftarrow MigratoryBird(Jo) \tag{r2}$$

$$\alpha \ : \leftarrow ResidentBird(Jo) \ , \ MigratoryBird(Jo) \tag{r3}$$

$$2 \ : ResidentBird(Jo) \tag{r4}$$

$$1 \ : MigratoryBird(Jo) \tag{r5}$$

$Comp(\Pi)$ for Π is

$$\alpha \ : Bird(Jo) \rightarrow ResidentBird(Jo) \vee MigratoryBird(Jo)$$

$$\alpha \ : ResidentBird(Jo)$$

$$\alpha \ : MigratoryBird(Jo)$$

Chapter 3

LPMLN2ASP SYSTEM

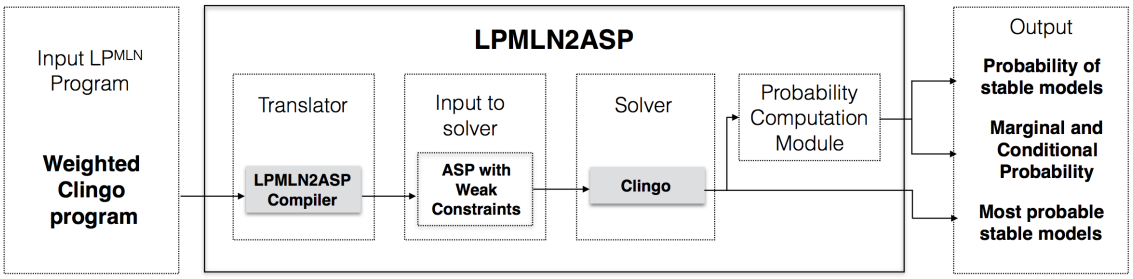

Figure 3.1: LPMLN2ASP System

3.1 Introduction

In this chapter, we extend the Lee-Yang translation to non-ground programs and use this result to to compute $\mathrm{LP}^{\mathrm{MLN}}$ programs using ASP solver CLINGO [1] . This implementation is realized in the form of LPMLN2ASP system as shown in Figure 3.1. We introduce the *probability computation module* as a part of the system to compute the probabilities of atoms in an $\mathrm{LP}^{\mathrm{MLN}}$ program. We then demonstrate how to use LPMLN2ASP for computing the probability of stable models, the most probable stable model, and marginal and conditional probability of *query* atoms for $\mathrm{LP}^{\mathrm{MLN}}$ programs.

3.2 Reformulating $\mathrm{LP}^{\mathrm{MLN}}$ based on the Concept of Penalty

In the definition of the $\mathrm{LP}^{\mathrm{MLN}}$ semantics reviewed in Section 2.3, the weight assigned to each stable model can be regarded as "rewards" i.e. the more rules that are true in

[1] We use CLINGO version 4.5.4 for this implementation

deriving the stable model, the larger the weight that is assigned to the stable model. We reformulate the $\mathrm{LP}^{\mathrm{MLN}}$ semantics in a "penalty" based way. 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

$$W_{\Pi}^{\mathrm{pnt}}(I) = \begin{cases} exp\left(- \sum\limits_{w:R \in \Pi \text{ and } I \not\models R} w \right) & \text{if } I \in SM[\Pi]; \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

and

$$P_{\Pi}^{\mathrm{pnt}}(I) = \lim_{\alpha \to \infty} \frac{W_{\Pi}^{\mathrm{pnt}}(I)}{\sum\limits_{J \in \mathrm{SM}[\Pi]} W_{\Pi}^{\mathrm{pnt}}(J)}. \tag{3.2}$$

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

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

This penalty based reformulation has a desirable property that adding a trivial rule that is satisfied by all interpretations does not affect the weight of an interpretation, which is not the case with the original definition. Another advantage is that this reformulation can be easily extended to the non-ground case as we show in the next section, so that a safe [2] non-ground $\mathrm{LP}^{\mathrm{MLN}}$ program can be translated into a safe non-ground ASP program.

3.3 Extending Lee-Yang translation to non-ground programs

We extend the Lee-Yang translation as described in section 2.4 by extending it to non-ground rules. We define the translation $lpmln2asp(\Pi)$ by translating each (possibly non-ground) rule

$$w_i \ : \ Head_i \leftarrow Body_i$$

[2] An $\mathrm{LP}^{\mathrm{MLN}}$ program Π is *safe* if the unweighted $\mathrm{LP}^{\mathrm{MLN}}$ program $\overline{\Pi}$ is safe as defined in (Calimeri *et al.*, 2013)

in an LP^{MLN} program Π, where i ranges from 1 to n and n being the total number of rules in Π, into

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

where

- $w_i' = 1$ if $w_i = \alpha$ and $w_i' = \lfloor w_i \times 10^m \rfloor$ otherwise, where m is a user-specified *multiplying factor* whose default value is 3
- $w_i^s =$ "a" if $w_i = \alpha$ and $w_i^s =$ "w_i" otherwise
- $l = 1$ if $w_i = \alpha$ and $l = 0$ otherwise
- $\mathbf{x}$ is a list of *global* [3]variables in the rule.

The distinction between hard and soft rules can be simulated by the different levels l of weak constraint. In the case when $Head_i$ is a disjunction of atoms $A_1; ...; A_k$, the expression $\texttt{not}\ Head_i$ stands for $\texttt{not}\ A_1, ..., \texttt{not}\ A_k$.

The list $\mathbf{x}$ of global variables is appended as arguments to the $unsat$ atom to ensure that a unique $unsat$ atom is generated for all groundings of a rule. Adding the index of the rule i in the list of terms of weak constraint further ensures that unique $unsat$ atoms are generated for two different grounded $lpmln2asp(\Pi)$ rules with the same list of terms. For any ASP program P, let $grnd(P)$ denote the *ground instantiation* of program P obtained

[3]The definition of a *global* variable is as defined in (Gebser *et al.*, 2015). One simple way to check if a variable is global or not is to ensure that it satisfies the two conditions: (1) the variable is not present in an aggregate in CLINGO i.e. the variable is not present between { and }, and (2) the variable is not present as a tuple of a symbolic or an arithmetic literal in a conditional literal i.e. in a conditional literal $\mathbf{H} : L$ the variable is not present in L.

by grounding the rules in P as defined in (Calimeri *et al.*, 2013). It is easy to see that there is a $1-1$ correspondence between the optimal stable models of $grnd(lpmln2asp(\Pi))$ and the optimal stable models of $lpmln2wc(grnd(\Pi))$.

w_i^s is a enclosed in *quotes* to make w_i a *string* value. Decimal values are not allowed as an argument of a predicate in CLINGO syntax and therefore the workaround is to encode decimals as string values. w_i' has to be an integral weight according to the CLINGO syntax. When w_i is not an integral weight and $w_i \neq \alpha$, it is converted to w_i' by applying the translation rule

$$w_i' = \begin{cases} 1 & if \quad w_i = \alpha \\ \lfloor w_i \times 10^m \rfloor & otherwise. \end{cases} . \tag{3.4}$$

The first rule of (3.3) says that, for an interpretation I, if the i^{th} rule is not satisfied then $unsat(i, w_i^s, \mathbf{x})$ is true for I. Consequently, I also gets a penalty of w_i' due to the third rule of (3.3) at level 0 or 1 depending on whether the rule is soft or hard, respectively. If the i^{th} rule is true in I, then $unsat(i, w_i^s, \mathbf{x})$ is false in I, $Head_i \leftarrow Body_i$ is effective and as a result no penalty is assigned to I.

3.4 MAP inference using LPMLN2ASP

System LPMLN2ASP uses the translation as described in Equation (3.3) in conjunction with Corollary 2 from (Lee and Yang, 2017a) to compute MAP inference on an $\mathrm{LP}^{\mathrm{MLN}}$ program. MAP inference in an $\mathrm{LP}^{\mathrm{MLN}}$ program is reduced to the optimal stable model finding of the program with weak constraints. For MAP inference, CLINGO does not enumerate all the stable models of the program and therefore in practice MAP inference is more scalable than exact probability computation on non trivial domains.

For any integer l (level) and an interpretation I of the program lpmln2asp(Π), let

$$
\begin{aligned}
weak(lpmln2asp(\Pi), I) = \{&(w_i@l, i, \mathbf{c}) \mid \\
&(w_i@l, i, \mathbf{c}) \; is \; obtained \; from \; (w_i@l, i, \mathbf{x}) \; by \\
&replacing \; all \; \mathbf{x} \; with \; the \; elements \; from \; the \; Herbrand \; Universe, \\
&:\sim unsat(i, w_i^s, \mathbf{c}) \; [w_i'@l, i, \mathbf{c}] \; occurs \; in \; lpmln2asp(\Pi) \; and \\
&unsat(i, w_i^s, \mathbf{c}) \; is \; true \; in \; I\},
\end{aligned}
$$

then the penalty of I at l, defined by pnt_l^I is

$$
pnt_l^I = \sum_{(w_i@l, i, \mathbf{c}) \; \in \; weak(lpmln2asp(\Pi), I)} w_i' \tag{3.5}
$$

The optimal stable models of $lpmln2asp(\Pi)$ are the stable models that minimize pnt_l^I according to the weak constraint semantics. The optimal stable models are given by

$$
I \in \underset{J \; : \; J \; \in \; \underset{K \in SM[\Pi]}{argmin_K} \; pnt_1^K}{argmin_J} pnt_0^J \tag{3.6}
$$

The minimization first happens at the highest level $l = 1$ first and then at $l = 0$. In case if the hard rules are not translated the optimal stable models are the models that minimize pnt_0^I. The optimal stable models of $lpmln2asp(\Pi)$ have a $1 - 1$ correspondence with the optimal stable models of $lpmln2wc(\Pi)$.

The translation w.r.t. w_i' as defined in Equation (3.4) does not affect the resulting optimal stable models and therefore the MAP estimates. In Equation (3.5), pnt_l^I is a monotonically increasing function of w_i', w_i' is a linear function of w_i and therefore pnt_l^I is a monotonically increasing function of w_i. Thus, multiplying each w_i with the a positive factor still results in a monotonically increasing function.

Example 6. Consider the non-ground *Bird* example adapted from (Lee and Wang, 2016b)

$$\alpha \ : Bird(X) \leftarrow ResidentBird(X) \tag{r1}$$

$$\alpha \ : Bird(X) \leftarrow MigratoryBird(X) \tag{r2}$$

$$\alpha \ : \leftarrow ResidentBird(X)\ ,\ MigratoryBird(X) \tag{r3}$$

$$2 \ : ResidentBird(Jo) \tag{r4}$$

$$1 \ : MigratoryBird(Jo) \tag{r5}$$

Here X is a variable. Using the translation 3.3 described above, the program is turned into

$$
\begin{aligned}
unsat(1, \text{“a”}, X) &\leftarrow ResidentBird(X), not\ Bird(X).\\
Bird(X) &\leftarrow ResidentBird(X), not\ unsat(1, \text{“a”}, X).\\
&:\sim unsat(1, \text{“a”}, X).\ [1@1, 1, X]\\
unsat(2, \text{“a”}, X) &\leftarrow MigratoryBird(X), not\ Bird(X).\\
Bird(X) &\leftarrow MigratoryBird(X), not\ unsat(2, \text{“a”}, X).\\
&:\sim unsat(2, \text{“a”}, X)\ [1@1, 2, X]\\
unsat(3, \text{“a”}, X) &\leftarrow ResidentBird(X), MigratoryBird(X).\\
&\leftarrow ResidentBird(X), MigratoryBird(X), not\ unsat(3, \text{“a”}, X).\\
&:\sim unsat(3, \text{“a”}, X)\ [1@1, 3, X]\\
unsat(4, \text{“2”}) &\leftarrow not\ ResidentBird(Jo).\\
ResidentBird(Jo) &\leftarrow not\ unsat(4, \text{“2”}).\\
&:\sim unsat(4, \text{“2”})\ [2000@0, 4]\\
unsat(5, \text{“1”}) &\leftarrow not\ MigratoryBird(Jo).\\
MigratoryBird(Jo) &\leftarrow not\ unsat(5, \text{“1”}).\\
&:\sim unsat(5, \text{“1”})\ [1000@0, 5]
\end{aligned}
$$

The optimal stable model of the program is $I = \{ResidentBird(Jo), Bird(Jo), unsat(5, \text{“1”})\}$. This model has the lowest penalty $pnt_0^{I'} = 1000$ at $l = 0$ and $pnt_1^{I'} = 0$ at $l = 1$ amongst all the interpretations I' of Π. The optimal stable model I is also the most probable stable model of the program which is in accordance with the answer in Example 3.

3.5 Probability computation using LPMLN2ASP

We introduce the *probability computation module* which is used by LPMLN2ASP for computing the marginal and conditional probability of queried atoms. Probability computation involves enumerating all the stable models of an $\mathrm{LP}^{\mathrm{MLN}}$ program. This is computationally expensive for all but the most trivial domains. However, the computation is exact. This gives the "gold" standard result which is easy to understand. Conditional probability of an atom is calculated in presence of certain evidence. Conditional probability calculation is more effective than marginal probability computation since adding evidence prunes out all the stable models where the evidence is not satisfied thereby resulting in fewer stable models to enumerate. The stable model enumeration also facilitates probability calculation of each stable model. The probability computation module works by examining the $unsat$ atoms present in the stable model. The output from CLINGO is given as input to this module.

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

Theorem 2. *For any* $\mathrm{LP}^{\mathrm{MLN}}$ *program* Π*, there is a 1-1 correspondence* ϕ *between* $SM[\Pi]$ [4] *and the set of stable models of* $\mathrm{lpmln2asp}(\Pi)$*, where* $\phi(I) = I \cup \{\mathtt{unsat}(i, w_i^s, \mathbf{c}) \mid w_i : Head_i \leftarrow Body_i \in \Pi, I \models Body_i, I \not\models Head_i\}$*. Furthermore,*

$$W_{\Pi}^{\mathrm{pnt}}(I) = exp\left(- \sum_{\mathtt{unsat}(i,w_i^s,\mathbf{c})\in\phi(I)} w_i \right). \tag{3.7}$$

w_i is obtained from w_i^s by a $1-1$ string to real number conversion. If a rule is a hard rule i.e. $w_i^s =$ "a" than a penalty of $w_i = \alpha$ is added to the stable model and if it is a soft rule i.e. $w_i^s =$ "w_i" than a penalty of w_i is added. Theorem 2, in conjunction with Theorem 1,

[4] $SM[\Pi]$ is as defined in Section 2.3

provides a way to compute the probability of a stable model of an LP^{MLN} program by examining the $unsat(i, w_i^s, \mathbf{c})$ atoms satisfied by the corresponding stable model of the translated ASP program. The penalty of a stable model is the exponentiated negative sum of w_i obtained from w_i^s specified by the corresponding $unsat(i, w_i^s, \mathbf{c})$ atoms present in the stable model.

For probability computation the translation used is the same as defined in Equation (3.3). The penalty of an interpretation I such that $I \in SM[\Pi]$ is given by $W_\Pi^{\text{pnt}}(I)$ as defined in Equation (3.7) and 0 if $I \notin SM[\Pi]$. The probability of an interpretation I is given by $P_\Pi^{\text{pnt}}(I)$ as defined in Equation (3.2). The probability of a ground query atom $q(\mathbf{c})$, denoted as $Pr_\Pi(q(\mathbf{c}))$, is given by

$$Pr_\Pi(q(\mathbf{c})) = \sum_{q(\mathbf{c}) \in I} P_\Pi^{\text{pnt}}(I). \tag{3.8}$$

In the presence of evidence e, $Pr_\Pi(q(\mathbf{c}))$ is the conditional probability of $q(\mathbf{c})$ given evidence e. It is given by

$$Pr_\Pi(q(\mathbf{c}) \mid e) = \sum_{\substack{q(\mathbf{c}) \in I \\ I \models e}} P_\Pi^{\text{pnt}}(I). \tag{3.9}$$

Without evidence, $Pr_\Pi(q(\mathbf{c}))$ is the marginal probability of $q(\mathbf{c})$. In the calculations of marginal and conditional probability, and probability of all the stable models of programs, weak constraints from Equation (3.3) are not used in the computation and therefore ignored by the module.

Example 7. Consider the Bird encoding from Example 6.

$$\alpha \ : Bird(X) \leftarrow ResidentBird(X) \tag{r1}$$

$$\alpha \ : Bird(X) \leftarrow MigratoryBird(X) \tag{r2}$$

$$\alpha \ : \leftarrow ResidentBird(X) \ , \ MigratoryBird(X) \tag{r3}$$

$$2 \ : ResidentBird(Jo) \tag{r4}$$

$$1 \ : MigratoryBird(Jo) \tag{r5}$$

the above program is translated into (weak constraints are removed for brevity)

$$
\begin{aligned}
unsat(1, \text{“a”}, X) &\leftarrow ResidentBird(X), not\ Bird(X) \\
Bird(X) &\leftarrow ResidentBird(X), not\ unsat(1, \text{“a”}, X) \\
unsat(2, \text{“a”}, X) &\leftarrow MigratoryBird(X), not\ Bird(X) \\
Bird(X) &\leftarrow MigratoryBird(X), not\ unsat(2, \text{“a”}, X) \\
unsat(3, \text{“a”}, X) &\leftarrow ResidentBird(X), MigratoryBird(X) \\
&\leftarrow ResidentBird(X), MigratoryBird(X), not\ unsat(3, \text{“a”}, X) \\
unsat(4, \text{“2”}) &\leftarrow not\ ResidentBird(Jo) \\
ResidentBird(Jo) &\leftarrow not\ unsat(4, \text{“2”}) \\
unsat(5, \text{“1”}) &\leftarrow not\ MigratoryBird(Jo) \\
MigratoryBird(Jo) &\leftarrow not\ unsat(5, \text{“1”})
\end{aligned}
$$

The following table illustrates $W^{\text{pnt}}_{\Pi}(I)$ and $P^{\text{pnt}}_{\Pi}(I)$ for each stable model $SM[\Pi]$ for the above example

	I	$W^{\text{pnt}}_{\Pi}(I)$	$P^{\text{pnt}}_{\Pi}(I)$
I_1	$\{RB(Jo), MB(Jo), un(1, \text{“a”}, Jo), un(2, \text{“a”}, Jo), un(3, \text{“a”}, Jo)\}$	$e^{-3\alpha}$	0
I_2	$\{un(4, \text{“2”}), MB(Jo), un(2, \text{“a”}, Jo)\}$	$e^{-\alpha-2}$	0
I_3	$\{RB(Jo), un(5, \text{“1”}), un(1, \text{“a”}, Jo)\}$	$e^{-\alpha-1}$	0
I_4	$\{un(4, \text{“2”}), un(5, \text{“1”})\}$	e^{-3}	$\frac{e^{-3}}{e^{-3}+e^{-2}+e^{-1}}$
I_5	$\{RB(Jo), MB(Jo), B(Jo), un(3, \text{“a”}, Jo)\}$	$e^{-\alpha}$	0
I_6	$\{un(4, \text{“2”}), MB(Jo), B(Jo)\}$	e^{-2}	$\frac{e^{-2}}{e^{-3}+e^{-2}+e^{-1}}$
I_7	$\{RB(Jo), un(5, \text{“1”}), B(Jo)\}$	e^{-1}	$\frac{e^{-1}}{e^{-3}+e^{-2}+e^{-1}}$

$$RB = ResidentBird, MB = Migratorybird, B = Bird, un = unsat$$

Table 3.1: Stable Models of Π from Example 6

To calculate the probability of $Bird$, we add the probability of all $P_{\Pi}^{\text{pnt}}(I)$ such that $Bird(X)$ is true in I. In this case, since there is only one grounding for $Bird$, we check for all I where $Bird(Jo)$ is true. From the above table, rows 5,6 and 7 correspond to the stable models where $Bird(Jo)$ is satisfied. Therefore, $Pr_{\Pi}(Bird(Jo)) = P_{\Pi}^{\text{pnt}}(I_5) + P_{\Pi}^{\text{pnt}}(I_6) + P_{\Pi}^{\text{pnt}}(I_7)$ where I_i corresponds to the i-th interpretation in Table 3.5.

3.6 LPMLN2ASP system architecture

Figure 3.1 shows the architecture of the LPMLN2ASP system. LPMLN2ASP is an encompassing system comprising of the LPMLN2ASP compiler, CLINGO and the probability computation module. It provides an interface similar to popular MLN tools like ALCHEMY, TUFFY, ROCKIT. The input to the system is an LP^{MLN} program. The syntax of the input program is detailed in section 3.7. The input is a set of weighted CLINGO rules given to the LPMLN2ASP compiler called LPMLN2CL. The compiler outputs an ASP encoding with weak constraints according to the translation 3.3 which is given as input to the solver. The solver used in LPMLN2ASP is CLINGO 4.

There are three modes of computation in LPMLN2ASP : MAP estimates, marginal probability and conditional probability. The mode of computation is determined by the arguments provided to LPMLN2ASP . For MAP inference, the output from the compiler is given to CLINGO and the output is the most optimal stable model and the penalty assigned to the model.

For marginal and conditional probability, the output from CLINGO is given to the *probability computation module*. The implementation of the module is based on the equations (3.2), (2) and (3.8). The module is a PYTHON program that integrates with CLINGO and

post-processes the CLINGO output to perform probabilistic inference. There is no difference between the marginal and conditional probability computation from the context of the module. The difference between the two is the addition of evidence file required for conditional probability. The evidence file is a set of ASP facts and constraints. When an evidence file e is provided, the input to CLINGO is $lpmln2asp(\Pi) \cup e$. Marginal and conditional probability computation also requires a $query$ predicate constant as input. The query predicate is any predicate that is present in $lpmln2asp(\Pi)$ or e. Note that now since e is also a part of the input program in conditional probability, all interpretations I that satisfy $lpmln2asp(\Pi)$ but do not satisfy e are discarded[5]. This makes computation of conditional probability faster than marginal probability since there are lesser stable models to enumerate through.

Solver CLINGO has a feature that integrates PYTHON code along with the CLINGO encoding by utilizing its suite of APIs. The probability computation module makes use of these APIs for computations. When CLINGO finds a stable model for the $lpmln2asp(\Pi)$, the stable model computation is interrupted by the module which processes the stable model generated. The module calculates the penalty of the stable model by examining the $unsat(i, w_i^s, \mathbf{c})$ atoms and stores the penalty and model for later use. The module also keeps a track of which models have the queried predicates. Once all the stable models have been generated by CLINGO, the control again returns to the module. At this point, the module adds up the stored penalties to compute the normalization factor and finds the probabilities of each stable models. Probabilities of queried predicates is calculated by adding the probabilities of stable models where the atoms of the predicate are satisfied. For each

[5]The exception to this is adding interceptions in evidence files in formalisms like Pearl's Causal Models(Pearl, 2000). Adding interceptions in evidence increases the number of stable models compared to the original program Π without any evidence.

queried predicate, a single pass over the stored models and their values is made to compute the probabilities of all grounded atoms of the queried predicate.

3.7 LPMLN2ASP input language syntax

The input language of LPMLN2ASP consists of *rules* of the form,

$$w_i \; Head_i \leftarrow Body_i. \tag{3.10}$$

where w_i is the weight of the i^{th} rule and $Head_i \leftarrow Body_i$ is a safe CLINGO rule. A hard rule is written without weights and is identical to a CLINGO rule. The CLINGO syntax is described in (Calimeri *et al.*, 2013). Every valid rule for clingo is also a valid rule for LPMLN2ASP . Weight w_i can be a positive or a negative decimal value or defined by a function expression as described later. For hard rules w_i is dropped. Every CLINGO program can be converted to an LPMLN2ASP program by appending w to the CLINGO rule.

Example 8. Encoding of Example 6 in the input language of LPMLN2ASP

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

Functions in LPMLN2ASP

System LPMLN2ASP allows for using functions log [6] and exp in the input language. The syntax to use such functions is,

$$@function\_name(expression)$$

[6]Natural logarithm

The functions allowed are log which evaluates natural logarithm of an expression and exp which evaluates exponential of an expression. The expression may be any non-trivial arithmetic expression consisting of $\{+, -, *, /, exp, log\}$ and decimal values. The system throws the error *resulting value infinity or NAN* if an expression or subexpression evaluates to infinity or a malformed expression is provided in input. Internally, the function is evaluated and the resuting value is used as w in $w : R$.

Some examples of using functions in LPMLN2ASP

```
@exp(2) bird(X) :- migratorybird(X).
@exp(2/exp(2)) :- residentbird(X) , migratorybird(X).

@log(10/1.0) residentbird(bob).
@log((0.75*10)+1-(2*(3/6))) migratorybird(bob).
```

3.8 LPMLN2ASP usage

The basic usage of LPMLN2ASP is

```
lpmln2asp -i <input_file> [-e <evid_file>] [-r <output_file>]
[-q <query>] [-hr] [-mf <multiplying_factor>] [-d] [-all]
[-clingo <clingo_options>]
```

Command line options for LPMLN2ASP ,

```
-h, --help                    show this help message and exit
-i <input_file>               input file. [REQUIRED]
-e <evidence_file>            evidence file
-r <output_file>              output file. Default is STDOUT
-q <query>                    List of comma separated query
                              predicates.
-clingo <clingo_options>      clingo options passed as it is to
```

```
                                the solver. Pass all clingo options
                                enclosed in 'single quotes'
-hr                             [FALSE] Translate hard rules
-all                            Display probability of all stable
                                models.
-mf <multiplying_factor>        [1000] Integer value of multiplying
                                factor
-d                              [FALSE] Debug. Print all debug info
```

The implementation bypasses all python/lua code enclosed in `#script(X) ... #end.` tags where `X` is either `python` or `lua`. Do note that this implementation does not check for the correctness of syntax of input and in case of syntactically incorrect input, the implementation would output the translated rules which would be syntactically wrong as well.

The option *-hr* translates all the hard rules as well. While translating hard rules is useful for debugging, it increases the number of rules generated and therefore increases the grounding size and the number of stable models. This is computationally expensive since the module enumerates all stable models for computations. Since hard rules always need to be satisfied, this option is not much useful except for debugging inconsistent ASP programs as shown later in Section 4.6.

-mf X option is the multiplying factor and the default value is 3. Consider rules

$$0.542\ a(X) \text{ :- } b(X).$$

$$0.148\ a(X) \text{ :- } b(X).$$

$$0.986\ a(X) \text{ :- } b(X).$$

if $m = 0$, following the definition from Equation (3.4), the above 3 rules will generate the weak constraints as

$$:\sim unsat(0, \text{“0.542”}, X).\ [0@0, 0, X]$$
$$:\sim unsat(1, \text{“0.148”}, X).\ [0@0, 1, X]$$
$$:\sim unsat(2, \text{“0.986”}, X).\ [0@0, 2, X]$$

For priority 0, the weights considered by CLINGO is 0 for all rules. This is obviously wrong. If a multiplying factor of $m = 1$ is provided to LPMLN2ASP, the weights generated would be

$$:\sim unsat(0, \text{“0.542”}, X).\ [5@0, 0, X]$$
$$:\sim unsat(1, \text{“0.148”}, X).\ [1@0, 1, X]$$
$$:\sim unsat(2, \text{“0.986”}, X).\ [9@0, 2, X]$$

This allows for more fine grained control over the weight scheme for MAP inference. For probabilistic inference the -mf option is ignored.

Command line usage

This section describes the command line usage of LPMLN2ASP for different modes of computation and example usage [7] of the respective modes on Example 8 and the respective outputs.

[7] The filename is birds.lp for the usage

- MAP inference

```
lpmln2asp -i <input_file>
```

By default, the mode of computation in LPMLN2ASP is MAP inference. Only providing an input file defaults to this mode.

Example 9. `lpmln2asp -i birds.lp`

Output:

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

- Marginal Probability of all models

```
lpmln2asp -i <input_file> -all
```

Providing the `-all` argument invokes the probability computation module in LPMLN2ASP and also serves as verbose mode to list all models and their respective probabilities.

Example 10. `lpmln2asp -i birds.lp -all`

Output:

```
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) 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 a list of query predicates

  `lpmln2asp -i <input_file> -q [query_predicates]`

  This mode calculates the marginal probability of the multiple query predicates which should be comma(,) separated.

  Example 11. `lpmln2asp -i birds.lp -q residentbird`

  Output:

  ```
  residentbird(jo) 0.665240955775
  ```

- Marginal probability of query predicates and probability of all models

  `lpmln2asp -i <input_file> -q [query_predicate] -all`

  This mode is the same as previous mode except it provides a verbose output where the marginal probability of all models is printed along with the probability of query predicates.

  Example 12. `lpmln2asp -i birds.lp -q residentbird -all`

  Output:

  ```
  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) migratorybird(jo)
Optimization: 2000



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


residentbird(jo) 0.665240955775
```

- Conditional probability of query predicates given evidence e

  `lpmln2asp -i <input_file> -q [query_predicate] -e <evidence_file>`

  Since evidence is provided in this mode, conditional probability of query given evidence is computed.

  Example 13. `lpmln2asp -i birds.lp -q residentbird -e evid.db`

  where `evid.db` contains

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

  Output:

  ```
  residentbird(jo) 0.73105857863
  ```

- Conditional probability of query predicates given evidence e and probability of all models

  ```
  lpmln2asp -i <input_file> -q [query_predicate] -e <evidence_file> -all
  ```

  This mode is the same as previous mode except it provides a verbose output where the conditional probability of all models is printed along with the probability of query predicates.

  Example 14. `lpmln2asp -i birds.lp -q residentbird -e evid.db -all`

  where `evid.db` contains

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

  Output:

  ```
  Answer: 1
  residentbird(jo) bird(jo) unsat(5,"1.000000")
  Optimization: 1000
  Answer: 2
  unsat(4,"2.000000") bird(jo) migratorybird(jo)
  Optimization: 2000

  Probability of Answer 1 : 0.73105857863
  Probability of Answer 2 : 0.26894142137

  residentbird(jo) 0.73105857863
  ```

Chapter 4

COMPUTING OTHER FORMALISMS IN LPMLN2ASP

It has been shown that formalisms like MLN, Pearl's Causal Models, Bayes Net, P-Log and ASP can be embedded in $\mathrm{LP}^{\mathrm{MLN}}$ (Lee and Wang, 2016b; Lee and Yang, 2017b; Lee *et al.*, 2015). In this chapter we demonstrate how to use LPMLN2ASP to compute these formalisms. We also show how to use LPMLN2ASP to resolve inconsistencies in an ASP program.

4.1 Computing MLN in LPMLN2ASP

Markov Logic can be embedded into $\mathrm{LP}^{\mathrm{MLN}}$ similar to the way SAT can be embedded into ASP as described in (Lee and Wang, 2016b). For any MLN $\mathbb{L}$, $\mathrm{LP}^{\mathrm{MLN}}$ program $\Pi_{\mathbb{L}}$ is obtained from $\mathbb{L}$ by adding $w : \{A\}^{ch}$ for every ground atom A of σ. The effect of adding the choice rules is to exempt A from minimization under the stable model semantics. Theorem 2 from (Lee and Wang, 2016b) states that any MLN $\mathbb{L}$ and its $\mathrm{LP}^{\mathrm{MLN}}$ representation $\Pi_{\mathbb{L}}$ have the same probability distribution over all interpretations.

Example 15. Social network domain is a typical example in the Markov Logic literature. Consider an example of a domain that describes the relationship between smokers who are friends. We assume three people: Alice,Bob and Carol, and assume that Alice smokes and Alice is friends with Bob. We assume that smoking causes cancer up to a certain degree,

and that friends of each other are more likely to smoke.

$$1.5 \;\; \forall x, y \; \big(Smokes(x) \wedge Friends(x, y) \rightarrow Smokes(y)\big)$$

$$1.1 \;\; \forall x \; \big(Smokes(x) \rightarrow Cancer(x)\big)$$

$$Smokes(Alice)$$

$$Friends(Alice, Bob)$$

There are eight possible worlds. The weight of each world according to the MLN semantics is given by

Possible World	**Weight**
$\{S(B), \neg C(A), \neg C(B)\}$	e^{6}
$\{S(B), C(A), \neg C(B)\}$	$e^{7.1}$
$\{\neg S(B), \neg C(A)\}$	$e^{5.6}$
$\{\neg S(B), C(A)\}$	$e^{6.7}$
$\{S(B), \neg C(A), C(B)\}$	$e^{7.1}$
$\{S(B), C(A), C(B)\}$	$e^{8.2}$
$\{\neg S(B), \neg C(A), C(B)\}$	$e^{5.6}$
$\{\neg S(B), C(A), C(B)\}$	$e^{6.7}$

The probabilities of Cancer(**x**) is given by

$Cancer(Alice)$	$\frac{e^{7.1}+e^{6.7}+e^{8.2}+e^{6.7}}{e^{6}+e^{7.1}+e^{7.1}+e^{8.2}+e^{6.7}+e^{5.6}+e^{5.6}+e^{6.7}} = 0.75$
$Cancer(Bob)$	$\frac{e^{7.1}+e^{8.2}+e^{5.6}+e^{6.7}}{e^{6}+e^{7.1}+e^{7.1}+e^{8.2}+e^{6.7}+e^{5.6}+e^{5.6}+e^{6.7}} = 0.6874$

The same program can be encoded into LPMLN2ASP as [1]

[1]Note that we only add choice rule for `smoke` and `cancer` because they are assumed to be open world while `friends` is assumed to be closed world.

```
1.1 cancer(X) :- smoke(X).
1.5 smoke(Y) :- smoke(X), friends(X, Y).


smoke(alice).
friends(alice, bob).


{smoke(alice)}.
{smoke(bob)}.
{cancer(alice)}.
{cancer(bob)}.
```

Executing

```
lpmln2asp -i input.lp -q cancer
```

outputs

```
cancer(bob) 0.687487252151
cancer(alice) 0.750260105595
```

4.2 Computing P-log in LPMLN2ASP

P-log (Baral *et al.*, 2009b) is a "KR formalism that combines logic and probabilistic arguments in its reasoning". ASP is used as the logical foundation, while causal Bayes Net serve as probabilistic foundation. We use the translation as described in (Lee and Yang, 2017b) to translate a P-log program into its equivalent $\mathrm{LP}^{\mathrm{MLN}}$ program. We refer the readers to (Lee and Yang, 2017b) for details regarding the translation.

Example 16. We describe the Monty Hall problem that is used in the (Baral *et al.*, 2009b). A player is given the opportunity to select one of three closed doors, behind one of which there is a prize. Behind the other two doors are empty rooms. Once the player has made

a selection, Monty is obligated to open one of the remaining closed doors which does not contain the prize, showing that the room behind it is empty. He then asks the player if he would like to switch his selection to the other unopened door, or stay with his original choice. Here is the problem: does it matter if he switches?

The answer is YES. In fact switching doubles the players chance to win. This problem can be encoded in the language of P-log as

```
doors = {1, 2, 3}.
open, selected, prize : doors.
¬ can_open(D) ← selected = D.
¬ can_open(D) ← prize = D.
can_open(D) ← not ¬ can_open(D).
random(prize).
random(selected).
random(open : {X : can_open(X)}).
```

Suppose that we observed that the player has already selected door 1, and Monty opened door 2 revealing that it did not contain the prize. This is expressed as

```
obs(selected = 1).
obs(open = 2).
obs(prize ≠ 2).
```

Let us refer to the above P-log program as Π_{monty}. Because of the observations Π_{monty} has two possible worlds: the first containing $prize = 1$ and the second containing $prize = 3$. According to the P-log semantics, it follows that

$$P_{\Pi_{monty}}(prize = 1) = 1/3$$

$$P_{\Pi_{monty}}(prize = 3) = 2/3$$

where $P_{\Pi_{monty}}(prize = X)$ is the probability of world with $prize$ in X-th door.

The above program can be encoded in the syntax of LPMLN2ASP following the translation from (Lee and Yang, 2017b) as

```
door(d1;d2;d3).
constant(prize;selected;open).
number(2;3).
boolean(t;f).

canopen(D,f) :- selected(D),door(D).
canopen(D,f) :- prize(D),door(D).
canopen(D,t) :- not canopen(D,f),door(D).

:- canopen(D,t) , canopen(D,f).
:- prize(D1) , prize(D2) , D1!=D2.
:- selected(D1) , selected(D2) , D1!=D2.
:- open(D1) , open(D2) , D1!=D2.

prize(d1); prize(d2); prize(d3) :- not intervene(prize).
selected(d1); selected(d2); selected(d3) :- not intervene(selected).
open(d1); open(d2); open(d3) :- not intervene(open).
:- open(D) , not canopen(D,t) , not intervene(open).

posswithdefprob(prize,D) :- not posswithassprob(prize,D) , not
   intervene(prize),door(D).
numdefprob(prize,X) :- X= #count{D:posswithdefprob(prize,D)} , prize(Y)
    , posswithdefprob(prize,Y),number(X).
```

```
posswithdefprob(selected,D) :- not posswithassprob(selected,D) , not
    intervene(selected),door(D).
numdefprob(selected,X) :- X= #count{D:posswithdefprob(selected,D)} ,
    selected(Y) , posswithdefprob(selected,Y),number(X).

posswithdefprob(open,D) :- not posswithassprob(open,D) , canopen(D,t) ,
     not intervene(open),door(D) , door(D).
numdefprob(open,X) :- X= #count{D:posswithdefprob(open,D)} , open(Y) ,
    posswithdefprob(open,Y),number(X).

obs(selected,d1).
:- obs(selected,d1), not selected(d1).
obs(open,d2).
:- obs(open,d2), not open(d2).

unobs(prize,d2).
:- unobs(prize,d2), prize(d2).



-0.6931 :- not numdefprob(C,2),constant(C).
-0.4054 :- not numdefprob(C,3),constant(C).
```

On executing

```
lpmln2asp -i monty_hall.lp -q prize
```

the output is

```
prize(d1) 0.333343817985
prize(d3) 0.666656182015
```

which corresponds to the output of the P-log program.

4.3 Computing Pearl's Causal Model in LPMLN2ASP

Pearl's probabilistic causal models (PCM) (Pearl, 2000) can be represented in $\mathrm{LP}^{\mathrm{MLN}}$ as described in (Lee *et al.*, 2015). Theorem 3 from (Lee *et al.*, 2015) states that the solutions of the probabilistic causal model $\mathbb{M}$ where $\mathbb{M}$ is a representation of a PCM are identical to the stable models of its translation to $\mathrm{LP}^{\mathrm{MLN}}$ and their probability distributions coincide. Theorem 4 from (Lee *et al.*, 2015) states that the counterfactual reasoning in PCM can be reduced to $\mathrm{LP}^{\mathrm{MLN}}$ computation. LPMLN2ASP allows for computing probabilistic queries on PCMs such as counterfactual queries.

For any PCM $\mathbb{M} = \langle\langle U, V, F\rangle, P(U)\rangle$, where

- U is a set of exogenous atoms [2] ,
- V is a set of endogenous atoms,
- F is a finite set of equations $V_i = F_i$, one for each endogenous atoms V_i, and F_i is a propositional formula and,
- $P(U)$ is a probability distribution over U.

Let $P_{\mathbb{M}}$ be the $\mathrm{LP}^{\mathrm{MLN}}$ program obtained from $\mathbb{M}$ by applying the translation as defined in *Definition 6* in (Lee *et al.*, 2015) as follows

- $\alpha : V_i \leftarrow F_i$ for each $V_i \leftarrow F_i$ in $\mathbb{M}$ [3]
- for every U in $\mathbb{M}$ such that $P(U_i = \mathtt{t}) = p$: (i) $ln(\frac{p}{1-p}) : U_i$ if $0 < p < 1$; (ii) $\alpha : U_i$ if $p = 1$; (iii) $\alpha :\leftarrow U_i$ if $p = 0$.

[2] We assume the exogenous atoms are independent of each other.

[3] The syntax of $\mathrm{LP}^{\mathrm{MLN}}$ is extended to weighted propositional formulas in (Lee *et al.*, 2015) and therefore F_i is defined as a propositional formula. However, for the discussion in this example we assume that F_i can be written in the form as shown in Equation (2.1).

To represent counterfactuals in PCM the translation is as follows

- rule

$$\alpha : V_i^* \leftarrow F_i^*,\ not\ Do(V_i = \mathbf{t}),\ not\ Do(V_i = \mathbf{f})$$

  is added for each equation $V_i = F_i$ in $\mathbb{M}$, where V_i^* is a new symbol corresponding to V_i, and F_i^* is a formula obtained from F_i by replacing every occurrence of endogenous atoms W with W^*.

- rule

$$\alpha : V_i^* \leftarrow Do(V_i = \mathbf{t})$$

  for every $V_i \in V$. Informally, starred atoms represent the counterfactual world.

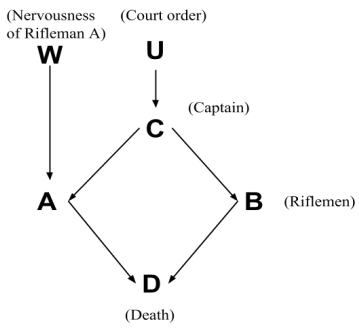

Figure 4.1: Firing Squad Example

Example 17. Consider the probabilistic version of the firing squad example as shown in Figure 4.1 which shows the firing squad scenario expressed as a *causal diagram* [4] . 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). Court orders the execution causes the captain to signal (C), which again causes Rifleman A and Rifleman B

[4]The definition of *causal diagram* is as defined in (Pearl, 2000)

to shoot at the prisoner. Either of Rifleman A and Rifleman B shooting causes the prisoner's death (D).

U denotes "The court orders the execution," C denotes "The captain gives a signal," A denotes "Rifleman A shoots," B denotes Rifleman B shoots, D denotes "The prisoner dies," and W denotes "Rifleman A is nervous." The court has ordered the execution with a probability $p = 0.7$ and rifleman A has a probability $q = 0.2$ of pulling the trigger out of nervousness.

For this example, $P_{\mathbb{M}}$ is

$$ln(\frac{0.7}{1-0.7}) : U$$
$$ln(\frac{0.2}{1-0.2}) : W$$
$$\alpha : C \leftarrow U$$
$$\alpha : A \leftarrow C \vee W^{5}$$
$$\alpha : B \leftarrow C$$
$$\alpha : D \leftarrow A \vee B$$

$$\alpha : C^* \leftarrow U, not\ Do(C = \mathbf{t}), not\ Do(C = \mathbf{f})$$
$$\alpha : A^* \leftarrow (C^* \vee W^*), not\ Do(A = \mathbf{t}), not\ Do(A = \mathbf{f})$$
$$\alpha : B^* \leftarrow C^*, not\ Do(B = \mathbf{t}), not\ Do(B = \mathbf{f})$$
$$\alpha : D^* \leftarrow (A^* \vee B^*), not\ Do(D = \mathbf{t}), not\ Do(D = \mathbf{f})$$

$$\alpha : C^* \leftarrow Do(C = \mathbf{t})$$
$$\alpha : A^* \leftarrow Do(A = \mathbf{t})$$
$$\alpha : B^* \leftarrow Do(B = \mathbf{t})$$
$$\alpha : D^* \leftarrow Do(D = \mathbf{t})$$

This translation is represented in the input language of LPMLN2ASP as follows

```
@log(0.7/0.3)  u.
```

[5] In logic programs $A \leftarrow B \vee C$ is *strongly equivalent* to writing $(A \leftarrow B) \wedge (A \leftarrow C)$.

```
@log(0.2/0.8)  w.

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

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).

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

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.

The different types of inference that can be computed are:

- *Prediction*: If rifleman A did not shoot, what is the probability that the prisoner is alive? We want to calculate the probability $P(\neg D \mid \neg A)$. $\neg A$ is true iff $\neg C$ is true

in which case $\neg B$ is true as well. Since both the rifleman did not fire, $\neg D$ is true. Therefore, $P(D|\neg A) = 0$. To represent prediction, the evidence file contains

```
:- a.
```

On executing

```
lpmln2asp -i pcm.lp -e evid.db -q d
```

the output is empty [6] which means that if rifleman A did not shoot, the prisonser is certainly alive.

- *Abduction*: If the prisoner is alive, what is the probability that the captain did not signal? We want to calculate the probability $P(\neg C \mid \neg D)$. $\neg D$ is true iff $\neg A \wedge \neg B$ is true which is possible only if $\neg C$ is true. Therefore, $P(C \mid \neg D) = 0$. To represent abduction, the evidence file contains

  ```
  :- d.
  ```

  On executing

  ```
  lpmln2asp -i pcm.lp -e evid.db -q c
  ```

  the output is empty which means that if the prisoner is alive then the captain did not order execution.

- *Transduction*: If rifleman A shot, what is the probability that rifleman B shot as well. We want to calcualte the probability $P(B \mid A) = \frac{p}{p+(1-p)q} = 0.92$. To represent transduction, the evidence file contains

  ```
  :- not a.
  ```

  On executing

[6] LPMLN2ASP does not output a query atoms whose probability is 0

```
lpmln2asp -i pcm.lp -e evid.db -q b
```

the output is

```
b 0.921047297896
```

which means there is a 92% chance that rifleman B shot as well.

- *Action*: If the captain gave no signal and rifleman A decides to shoot, what is the probability that the prisoner will die and rifleman B will not shoot. We want to calculate the probability $P(D_A \wedge \neg B_A \mid \neg C)$. Rifleman A decides to shoot regardless of captain's order. If $\neg C$ is true then $\neg B$ is true as well since only rifleman A decides to violate orders. From $P_{\mathbb{M}}$, we can see that $D \leftarrow A \vee B$, therefore if A is true than D is true. So, $P(D_A \mid \neg C) = 1$ and $P(\neg B_A \mid \neg C) = 1$. To represent an action, the evidence file contains

  ```
  :- c.
  do(a1).
  ```

  Here `c` is an *observation* and `do(a0)` is an *intervention* and hence encoded differently. On executing

  ```
  lpmln2asp -i pcm.lp -e evid.db -q ds,bs
  ```

  outputs

  ```
  ds 1.0
  ```

  which means that the prisoner will die and rifleman B will not shoot.

- *Counterfactual*: If the prisoner is dead, what is the probability that the prisoner would be alive if rifleman A had not shot $P(D_{\neg A} \mid D)$? To represent the counterfactual query, the evidence file contains

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

Here `d` is an *observation* and `do(a0)` is an *intervention*. On executing

```
lpmln2asp -i pcm.lp -e evid.db -q ds
```

LPMLN2ASP outputs

```
ds 0.921047297896
```

which means there is a 8% chance that the prisoner would be alive.

4.4 Computing Bayes Net in LPMLN2ASP

Bayes net can be encoded in $\mathrm{LP}^{\mathrm{MLN}}$ in a way similar to (Sang *et al.*, 2005). All random variables are assumed to be Boolean. Each conditional probability table associated with the nodes can be represented by a set of probabilistic facts. For each CPT entry $P(V = \mathbf{t} \mid V_1 = S_1, \ldots, V_n = S_n) = p$ where $S_1, \ldots, S_n \in \{\mathbf{t}, \mathbf{f}\}$, we include a set of weighted facts

- $ln(p/(1-p)) : \ \ PF(V, S_1, \ldots, S_n)$ if $0 < p < 1$;
- $\alpha : \ \ PF(V, S_1, \ldots, S_n)$ if $p = 1$;
- $\alpha : \ \leftarrow \ not\ PF(V, S_1, \ldots, S_n)$ if $p = 0$.

For each node V whose parents are $V_1, \ldots, V_n$, each directed edge can be represented by rules

$$\alpha : \ V \leftarrow V_1^{S_1}, \ldots, V_n^{S_n}, PF(V, S_1, \ldots, S_n) \qquad (S_1, \ldots, S_n \in \{\mathbf{t}, \mathbf{f}\})$$

where $V_i^{S_i}$ is V_i if S_i is $\mathbf{t}$, and $not\ V_i$ otherwise.

Example 18. Consider the example that is widely used in the Bayes net literature described in Figure 4.2. The LPMLN2ASP encoding for the net is

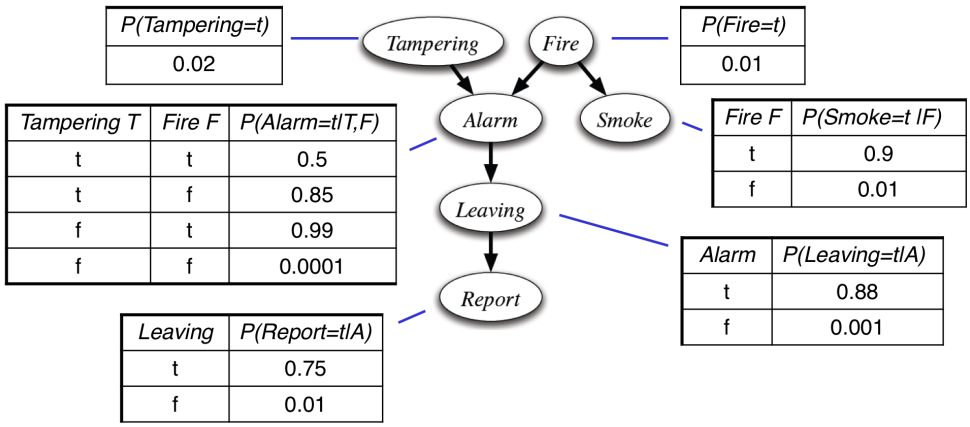

Figure 4.2: Bayes Net Example

```
@log(0.02/0.98) pf(t).

@log(0.01/0.99) pf(f).


@log(0.5/0.5) pf(a,t1f1).

@log(0.85/0.15) pf(a,t1f0).

@log(0.99/0.01) pf(a,t0f1).

@log(0.0001/0.9999) pf(a,t0f0).


@log(0.9/0.1) pf(s,f1).

@log(0.01/0.99) pf(s,f0).


@log(0.88/0.12) pf(l,a1).

@log(0.001/0.999) pf(l,a0).


@log(0.75/0.25) pf(r,l1).

@log(0.01/0.99) pf(r,l0).


tampering :- pf(t).
```

```
fire :- pf(f).


alarm :- tampering, fire, pf(a,t1f1).
alarm :- tampering, not fire, pf(a,t1f0).
alarm :- not tampering, fire, pf(a,t0f1).
alarm :- not tampering, not fire, pf(a,t0f0).


smoke :- fire, pf(s,f1).
smoke :- not fire, pf(s,f0).


leaving :- alarm, pf(l,a1).
leaving :- not alarm, pf (l,a0).


report :- leaving, pf(r,l1).
report :- not leaving, pf(r,l0).
```

The different types of inferences that can be computed are:

- *Diagnostic Inference*: Here we are trying to compute the probability of cause given the effect. To compute $P(fire = \mathbf{t} \mid leaving = \mathbf{t})$, the user can invoke

  ```
  lpmln2asp -i fire-bayes.lpmln -e evid.db -q fire
  ```

  where `evid.db` contains the line

  ```
  :- not leaving.
  ```

  This outputs

  ```
  fire 0.352151116689
  ```

- *Predictive Inference*: Here we are trying to compute the probability of effect given the cause. To compute $P(leaving = \mathbf{t} \mid fire = \mathbf{t})$, the user can invoke

  ```
  lpmln2asp -i fire-bayes.lpmln -e evid.db  -q leaving
  ```

  where `evid.db` contains the line

  ```
  :- not fire.
  ```

  This outputs

  ```
  leaving 0.862603541626
  ```

- *Mixed Inference*: Here we combine *predictive* and *diagnostic* inference in *Mixed Inference*. To compute $P(alarm = \mathbf{t} \mid fire = \mathbf{f}, leaving = \mathbf{t})$, the user can invoke

  ```
  lpmln2asp -i fire-bayes.lpmln -e evid.db  -q alarm
  ```

  where `evid.db` contains two lines

  ```
  :- fire.
  :- not leaving.
  ```

  This outputs

  ```
  alarm 0.938679679707
  ```

- *Intercausal Inference*: Here we compute the probability of a cause given an effect common to multiple causes. To compute $P(tampering = \mathbf{t} \mid fire = \mathbf{t}, alarm = \mathbf{t})$, the user can invoke

  ```
  lpmln2asp -i fire-bayes.lpmln -e evid.db  -q tampering
  ```

  where `evid.db` contains two lines

  ```
  :- not fire.
  :- not alarm.
  ```

This outputs

```
tampering 0.0102021964693
```

- *Explaining away*: Suppose we know that $alarm$ rang. Then we can use *Diagnostic Inference* to calculate $P(tampering = \mathbf{t} \mid alarm = \mathbf{t})$. But what happens if we now know that there was a $fire$ as well? In this case $P(tampering = \mathbf{t} \mid alarm = \mathbf{t})$ will change to $P(tampering = \mathbf{t} \mid fire = \mathbf{t}, alarm = \mathbf{t})$. In this case, knowing that there was a $fire$ explains away $alarm$, and hence affecting the probability of $tampering$. Even though $fire$ and $tampering$ are independent, the knowledge about one changes the probability of other.

  Lets compute $P(tampering = \mathbf{t} \mid alarm = \mathbf{t})$ which states the probability of $tampering$ to be true given $alarm$ is true. The user can invoke

  ```
  lpmln2asp -i fire-bayes.lpmln -e evid.db  -q tampering
  ```

  where `evid.db` contains line

  ```
  :- not alarm.
  ```

  This outputs

  ```
  tampering 0.633397289908
  ```

  If this result is compared with the result of *Intercausal Inference*, we can see that $P(tampering = \mathbf{t} \mid alarm = \mathbf{t}) > P(tampering = \mathbf{t} \mid fire = \mathbf{t}, alarm = \mathbf{t})$. Observing the value of $fire$ explains away the $tampering$ i.e. the probability of $tampering$ decreases.

4.5 Computing ProbLog in LPMLN2ASP

ProbLog (De Raedt *et al.*, 2007) can be viewed as a special case of $\mathrm{LP}^{\mathrm{MLN}}$ language (Lee and Wang, 2016b), in which soft rules are atomic facts only. The precise relation be-

tween the semantics of the two languages is stated in (Lee and Wang, 2016b). 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).

According to the translation defined in (Lee and Wang, 2016b), given a Problog program $\mathbb{P} = \langle PF, \Pi \rangle$, where

- PF is a set of ground probabilistic facts of the form *pr :: a*,
- Π is a set of ground rules of the form $A \leftarrow B_1, ..., B_m, not\ B_{m+1}, ..., not\ B_n$ where $A, B_1, ..., B_n$ are atoms from σ (0 $\leq$ m $\leq$ n), and A is not a probabilistic atom.

the corresponding LP^{MLN} program $\mathbb{P}'$ is obtained from $\mathbb{P}$ as follows

- For each probabilistic fact *pr :: a* in $\mathbb{P}$, LP^{MLN} program $\mathbb{P}'$ contains (i) $ln(pr)\ :\ a$ and $ln(1-pr)\ :\leftarrow a$ if $0 < pr < 1$; [7] (ii) $\alpha\ :\ a$ if $pr = 1$; (iii) $\alpha\ :\leftarrow\ a$ if $pr = 0$.
- For each rule $R \in \Pi$, $\mathbb{P}'$ contains $\alpha\ :\ R$.

We present two different examples taken from the PROBLOG2 website to demonstrate how Problog encodings can be translated to LPMLN2ASP encodings.

4.5.1 Computing ProbLog in LPMLN2ASP *- 1*

Example 19. We encode the problem of probabilistic graphs [8] in Problog. In the probabilistic graph, the existence of some edges between nodes is uncertain. We can use Problog to calculate the probability of path between two nodes. The encoding in Problog syntax is

```
0.6::edge(1,2).
0.1::edge(1,3).
```

[7] This can be shortened as $ln(\frac{pr}{1-pr})\ a$

[8] This example is taken from the Problog website https://dtai.cs.kuleuven.be/problog/tutorial/basic/04_pgraph.html

```
0.4::edge(2,5).

0.3::edge(3,4).

0.8::edge(4,5).


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

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


query(path(1,5)).
```

$0.6 :: edge(1,2)$ states that there is a 60% probability of there being an edge from node 1 to 2. We are querying for the path between nodes 1 and 5. The probability of path from node 1 to 5 according to Problog semantics is 0.25824. The encoding in the syntax of LPMLN2ASP according to the translation above is

```
@log(0.6/0.4) edge(1,2).

@log(0.1/0.9) edge(1,3).

@log(0.4/0.6) edge(2,5).

@log(0.3/0.7) edge(3,4).

@log(0.8/0.2) edge(4,5).


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

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

On executing

```
lpmln2asp -i problog2.lp -q path
```

outputs

```
path(1, 2) 0.600008374025

path(1, 3) 0.100002211982

path(4, 5) 0.800000902219
```

```
path(1, 4) 0.0300027187484
path(1, 5) 0.258254093504
path(2, 5) 0.400015626048
path(3, 4) 0.300020551084
path(3, 5) 0.240016711551
```

The probability of path from node 1 to 5 according to $\mathrm{LP}^{\mathrm{MLN}}$ semantics corresponds the the output from Problog encoding.

4.5.2 Computing ProbLog in LPMLN2ASP - *2*

Example 20. The following example [9] expresses a chain of events that happens when a person throws a rock. Two people, Suzy and Billy, may each decide to throw a rock at a bottle. Suzy throws with a probability 0.5 and if she does, her rock breaks the bottle with probability 0.8. Billy always throws and his rock hits with probability 0.6. The encoding in Problog syntax is

```
0.5::throws(suzy).
throws(billy).

0.8::broken; 0.2::miss :- throws(suzy).
0.6::broken; 0.4::miss :- throws(billy).

query(broken).
```

Rule 3 in Problog encoding is called *annotated disjunction* (Vennekens *et al.*, 2004). Such a disjunction cannot be written in the input language of LPMLN2ASP. We use the translation as defined in (Gutmann, 2011) to translate the annotated disjunction rules into Problog

[9] This example is taken from the Problog website
https://dtai.cs.kuleuven.be/problog/tutorial/various/16_cplogic.html

encoding without annotated disjunctions. We translate every annotated disjunction rule of the form

$$p_1 :: h_1; \ldots p_n :: h_n \ :- \ b_1, \ldots b_m$$

where $h_1, \ldots, h_n$ are atoms, the body $b_1, \ldots, b_m$ is a possibly empty conjunction of atoms and p_i are probabilities such that $\sum_{i=1}^{n} p_i \leq 1$ into a set of probabilistic facts

$$F = \{p_1' :: msw(1, V_1, \ldots, V_k), \ldots, p_n' :: msw(1, V_1, \ldots, V_k)\}$$

where $V_1, \ldots, V_k$ are all variables appearing in the disjunctive rule. Furthermore for each rule h_i one clause is added to the program as follows

h_1 :- $b_1, \ldots, b_m, msw(1, V_1, \ldots, V_k)$.

h_2 :- $b_1, \ldots, b_m, msw(2, V_1, \ldots, V_k), not\ msw(1, V_1, \ldots, V_k)$.

h_3 :- $b_1, \ldots, b_m, msw(3, V_1, \ldots, V_k), not\ msw(2, V_1, \ldots, V_k), not\ msw(1, V_1, \ldots, V_k)$.

h_i :- $b_1, \ldots, b_m, msw(i, V_1, \ldots, V_k), not\ msw(i-1, V_1, \ldots, V_k), \ldots,$

$not\ msw(1, V_1, \ldots, V_k)$.

The probability p_1' is defined as p_1 and for $i > 1$ as

$$p_i' = \begin{cases} p_i \cdot (1 - \sum_{j=1}^{i-1} p_j)^{-1} & \text{if } p_i > 0 \\ 0 & \text{if } p_i = 0 \end{cases}.$$

Based on the above translation, we can translate the original program into the following Problog encoding

```
0.5::throws(suzy).
throws(billy).
```

```
0.8::msw(1,1).
msw(1,2).
broken :- throws(suzy), msw(1,1).
miss :- throws(suzy), msw(1,2), \+msw(1,1).

0.6::msw(2,1).
msw(2,2).
broken :- throws(billy), msw(2,1).
miss :- throws(billy), msw(2,2),\+msw(2,1).

query(broken).
```

The equivalent LPMLN2ASP encoding according to the rules mentioned at the beginning of Section 4.5 is

```
0 throws(suzy).
throws(billy).

@log(0.8/0.2) msw(1,1).
msw(1,2).
broken :- throws(suzy), msw(1,1).
miss :- throws(suzy), msw(1,2), not msw(1,1).

@log(0.6/0.4) msw(2,1).
msw(2,2).
broken :- throws(billy), msw(2,1).
miss :- throws(billy), msw(2,2), not msw(2,1).
```

On executing

```
lpmln2asp -i problog2-tight.lp -q broken
```

the output is

```
broken 0.760005204855
```

This example is a Problog encoding of a program in the language of CP-Logic. The following figure [10] is a probability tree of the above encoding. $P(broken)$ is given by

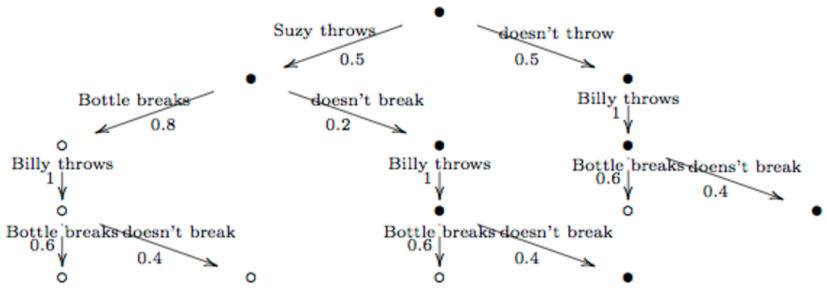

Figure 4.3: Probability Tree for Example 20

$$P(broken) = 0.5 * 0.8 * 1.0 * 0.6 + 0.5 * 0.8 * 1.0 * 0.4 +$$
$$0.5 * 0.2 * 1.0 * 0.6 + 0.5 * 1.0 * 0.6 = 0.76$$

which corresponds to the value computed using LPMLN2ASP .

4.6 Debugging inconsistent Answer Set Programs

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. When the given CLINGO program is inconsistent, one can call LPMLN2ASP for the same input to find out which rules cause inconsistency. For this use-case, it is necessary to translate all hard rules in the ASP program. Probabilistic stable models of the program are used to identify which rules are causing inconsistency.

[10] The Figure 4.3 is taken from https://dtai.cs.kuleuven.be/problog/tutorial/various/16_cplogic.html

Example 21. Consider the same example as Example 6 from which the last two soft rules are made hard.

```
bird(X) :- residentbird(X).                                    (r1)
bird(X) :- migratorybird(X).                                   (r2)
:- residentbird(X) , migratorybird(X).                         (r3)
residentbird(jo).                                              (r4)
migratorybird(jo).                                             (r5)
```

Clearly the encoding is *unsatisfiable*. This is also the case when the encoding is run with the command line

```
lpmln2asp -i input.lp
```

By default, LPMLN2ASP does not translate hard rules but we can instruct LPMLN2ASP to translate hard rules as well by giving the `-hr` option to the program. We also use the `-all` option since we want to get a verbose output for analyzing the stable models to remove inconsistencies in the program. On executing

```
lpmln2asp -i input.lp -hr -all
```

LPMLN2ASP outputs

```
Answer: 1
residentbird(jo) migratorybird(jo) unsat(1,"a",jo) unsat(2,"a",jo)
   unsat(3,"a",jo)
Optimization: 3
Answer: 2
unsat(4,"a") migratorybird(jo) unsat(2,"a",jo)
Optimization: 2
Answer: 3
residentbird(jo) unsat(5,"a") unsat(1,"a",jo)
```

```
Optimization: 2
Answer: 4
unsat(4,"a") unsat(5,"a")
Optimization: 2
Answer: 5
residentbird(jo) migratorybird(jo) bird(jo) unsat(3,"a",jo)
Optimization: 1
Answer: 6
unsat(4,"a") migratorybird(jo) bird(jo)
Optimization: 1
Answer: 7
residentbird(jo) unsat(5,"a") bird(jo)
Optimization: 1

Probability of Answer 5 : 0.333333333333
Probability of Answer 6 : 0.333333333333
Probability of Answer 7 : 0.333333333333
```

The output shows that either of the answers 5,6 or 7 can be used to identify the inconsistencies in the program and change the program such that minimal rules have to be modified to resolve inconsistency (since they are the most probable stable models). Answer 5 shows that rule 3 in the original program is unsatisfied. This is evident from the presence of atom `unsat(3,"a",jo)` in the stable model where the first argument 3 is the index of the unsatisfied rule. In the original program rule 3 is the constraint

```
:- residentbird(X) , migratorybird(X).
```

Removing this rule from the original program makes the program consistent. Similarly, according to Answer 6, removing rule `(r4)`, or according to Answer 7 removing rule `(r5)` would make the program consistent.

After examining the most probable stable models above, we can see that at most we need to modify one rule in the original program to resolve inconsistency. If we consider any other stable models except the most probable stable models we can see that we need to remove more than one rule. For instance, consider Answer 4 which has two $unsat$ atoms `unsat(4,"a")` and `unsat(5,"a")`. This means we need to remove rule 4 and rule 5 to resolve inconsistency in the program. A similar argument can be made for other stable models. In order to make the program consistent we can either

- remove one rule following Answers 5, 6 or 7,
- remove two rules following Answers 2, 3 or 4,
- remove three rules following Answer 1.

Chapter 5

LPMLN2MLN SYSTEM

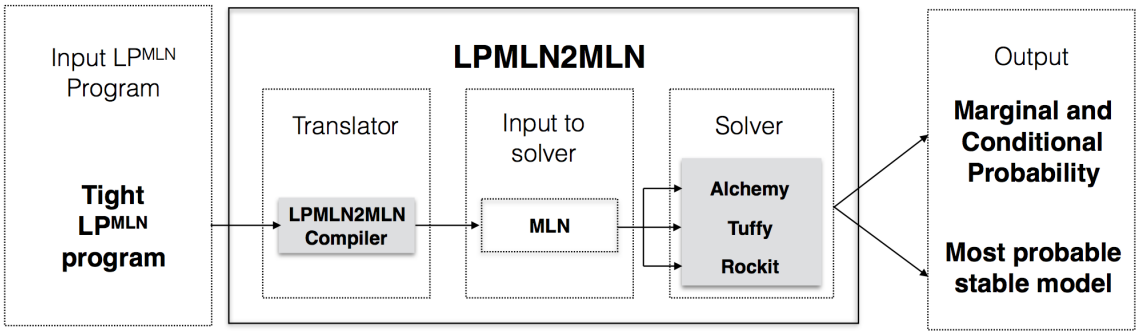

Figure 5.1: LPMLN2MLN System

5.1 Introduction

In this chapter we use *completion* as defined in (Lee and Wang, 2016b) to compute LP^{MLN} programs using MLN solvers like ALCHEMY, TUFFY and ROCKIT. This is realized in the implementation of the LPMLN2MLN system as shown in Figure 5.1. This chapter describes the syntax for the input language of LPMLN2MLN, the completion algorithm implemented, Tseitin's transformation usage in completion and the usage of the system. The chapter also explains the differences among the underlying solvers and their respective weaknesses and capabilities. This implementation, however, is restricted to tight programs only.

5.2 Completion of non-ground rules in LPMLN2MLN

The stable models of a *tight* logic program coincide with the models of the program's completion (Erdem and Lifschitz, 2003). This result allows for computing stable models of

a program using SAT solvers. This idea can be extended to compute $\mathrm{LP}^{\mathrm{MLN}}$ programs using MLN solvers (Lee and Wang, 2016b). System LPMLN2MLN translates a *tight* $\mathrm{LP}^{\mathrm{MLN}}$ program into an equivalent MLN program by computing the completion of $\mathrm{LP}^{\mathrm{MLN}}$ program. In theory, using loop formulas (Lin and Zhao, 2004) *tight* as well as *non-tight* $\mathrm{LP}^{\mathrm{MLN}}$ programs can be converted to MLN programs. However, translating non-tight programs using *loop formulas* does not yield effective computation using MLN solvers since the number of ground loop formulas that are required for computation can be exponential in the worst case, and therefore, we focus only on tight programs in this chapter.

Review: Tight programs (Fages, 1994)

This is a review of *tight* programs from (Lee and Lifschitz, 2003). We assume the same signature σ as defined in Section 2.1.

Let Π be a ground logic program consisting of the rules of the form (2.1). The *positive dependency graph* of Π is the directed graph G such that

- the vertices G are the atoms occurring in Π, and
- for every rule of the form (2.1) in Π, G has an edge from each A_i such that $1 \leq i \leq k$ to each atom in A_j such that $k+1 \leq j \leq l$.

A nonempty set L of atoms is called a *loop* of Π if, for every pair a_1, a_2 of atoms in L, there exists a path of nonzero length from a_1 to a_2 in the positive dependency graph of Π. such that all vertices in this path belong to L.

For example, consider the program Π_1

$$p \leftarrow q, not\ r$$

$$q \leftarrow r, not\ p$$

$$r \leftarrow not\ p$$

the positive dependency graph G_1 for the above program is

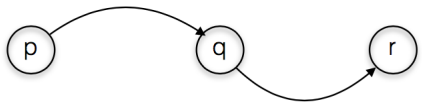

Figure 5.2: Positive Dependency Graph for Π_1

Consider another program Π_2

$$p \leftarrow q$$

$$q \leftarrow p$$

$$p \leftarrow not\ r$$

the positive dependency graph G_2 for the above program is

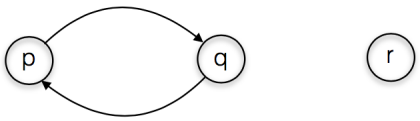

Figure 5.3: Positive Dependency Graph for Π_2

We say that a program is a *tight* program if Π has no loops. For instance, program Π_1 above is a tight program, and program Π_2 is not.

Review: Clark's completion (Clark, 1978)

The definition of completion explained here is based on Clark's completion. Let σ be a finite first-order signature that has no function constants of arity > 0. A rule is a first-order

formula of σ that has the form

$$F \to P(\mathbf{t})$$

Where F is a formula, P is a predicate constant and $\mathbf{t}$ is a tuple of terms. In logic programming, we write this rule as

$$P(\mathbf{t}) \leftarrow F$$

and call $P(\mathbf{t})$ the *head* of the rule, and F as the *body* of the rule. A logic program Π is a finite set of such rules. The *completion formula* for an n-ary predicate constant P relative to program Π is the sentence obtained as follows

1. Choose *n* variables that are pairwise distinct and do not occur in the program Π.

2. For each rule of the form

   $$P(t_1, ..., t_n) \leftarrow F$$

   create the rule

   $$P(x_1, ..., x_n) \leftarrow F \wedge x_1 = t_1 \wedge ... \wedge x_n = t_n$$

3. For each of the rules

   $$P(x_1, ..., x_n) \leftarrow F \wedge x_1 = t_1 \wedge ... \wedge x_n = t_n$$

   obtained in the previous step, make a list $\mathbf{y}$ of variables that occur in the body F but not in its head and replace the formula in body by

   $$P(x_1, ..., x_n) \leftarrow \exists \mathbf{y}(F \wedge x_1 = t_1 \wedge ... \wedge x_n = t_n)$$

4. For each rule obtained from step (3), write the rule

$$\forall x_1, ..., x_n(P(x_1, ..., x_n) \leftrightarrow (\exists \mathbf{y_1}(F_1 \wedge x_1 = t_1 \wedge ... \wedge x_n = t_n)) \vee$$
$$...$$
$$\vee (\exists \mathbf{y_m}(F_m \wedge x_1 = t_1 \wedge ... \wedge x_l = t_l))$$

Completion of rules in LPMLN2MLN

We extend the completion from (Lee and Wang, 2016b) to non-ground $\mathrm{LP}^{\mathrm{MLN}}$ rules. An $\mathrm{LP}^{\mathrm{MLN}}$ program Π comprises of rules of the form $w : R$ where R is a rule of the form (2.1). We assume the same signature σ as defined in Section 2.1. Each R can be identified with the rule

$$head(\mathbf{t_1}) \leftarrow body(\mathbf{t_2}) \tag{5.1}$$

where $\mathbf{t_1}$ is a set of terms occurring in the $head$ and $\mathbf{t_2}$ is a set of terms occurring in the $body$ of the rule. Each of the terms in $\mathbf{t_1}$ and $\mathbf{t_2}$ can be either a variable or an object constant.

Each rule R is translated into R' such that all atoms in *head* in Π consists of the same list of terms. The translation is as follows

1. We replace $\mathbf{t_1}$ by $\mathbf{v}$ where $\mathbf{v}$ is a *list* of n variables $v_1, ..., v_n$ for the n-ary predicate $head$. Variables $v_1, ..., v_n$ are pairwise-distinct and do no occur in Π.

2. Rewrite R as

$$head(\mathbf{v}) \leftarrow body(\mathbf{t'_2}) \wedge \bigwedge_{c_i \text{is an object constant in } \mathbf{t_1}} v_i = c_i$$

   where $\mathbf{t'_2}$ is obtained from $\mathbf{t_2}$ by replacing every variable in $\mathbf{t_2}$ that was present in $\mathbf{t_1}$ by its corresponding substitute $\mathbf{v_i}$ from step (1).

The completion of the program $\mathrm{LP}^{\mathrm{MLN}}$ Π denoted by $comp(\Pi)$ consists of rules

$$\alpha \ : \ head(\mathbf{v}) \rightarrow \bigvee_{\substack{w:head(\mathbf{v}) \leftarrow body(\mathbf{t'_2}) \in \Pi \\ \mathbf{z} \in \mathbf{t'_2} \setminus \mathbf{v}}} \left(\exists \mathbf{z} \ body(\mathbf{t'_2}) \right) . \tag{5.2}$$

for each predicate constant $head$ in Π. The equivalent MLN program for an $\mathrm{LP}^{\mathrm{MLN}}$ program Π consist of rules $\Pi \cup comp(\Pi)$.

Example 22. Consider the encoding Example 6

$$\alpha \ : Bird(X) \leftarrow ResidentBird(X) \tag{r1}$$

$$\alpha \ : Bird(X) \leftarrow MigratoryBird(X) \tag{r2}$$

$$\alpha \ : \leftarrow ResidentBird(X)\ ,\ MigratoryBird(X) \tag{r3}$$

$$2 \ : ResidentBird(Jo) \tag{r4}$$

$$1 \ : MigratoryBird(Jo) \tag{r5}$$

After step 1 the translated program obtained is

$$\alpha \ : Bird(v_1) \leftarrow ResidentBird(X) \tag{r1}$$

$$\alpha \ : Bird(v_2) \leftarrow MigratoryBird(X) \tag{r2}$$

$$\alpha \ : \leftarrow ResidentBird(X)\ ,\ MigratoryBird(X) \tag{r3}$$

$$2 \ : ResidentBird(v_1) \tag{r4}$$

$$1 \ : MigratoryBird(v_2) \tag{r5}$$

After step 2 the translated program obtained is

$$\alpha \ : Bird(v_1) \leftarrow ResidentBird(v_1) \tag{r1}$$

$$\alpha \ : Bird(v_1) \leftarrow MigratoryBird(v_1) \tag{r2}$$

$$\alpha \ : \leftarrow ResidentBird(X)\ ,\ MigratoryBird(X) \tag{r3}$$

$$2 \ : ResidentBird(v_1) \leftarrow v_1 = Jo \tag{r4}$$

$$1 \ : MigratoryBird(v_1) \leftarrow v_1 = Bob \tag{r5}$$

The completion of the above program $comp(\Pi)$ is given by

$$\alpha \ : Bird(v_1) \rightarrow ResidentBird(v_1) \vee MigratoryBird(v_1)$$

$$\alpha \ : ResidentBird(v_1) \rightarrow v_1 = Jo$$

$$\alpha \ : MigratoryBird(v_1) \rightarrow v_1 = Bob$$

The following theorem justifies using completion of tight programs for computing $\mathrm{LP}^{\mathrm{MLN}}$

Theorem 3. *(Lee and Wang, 2016b) For any tight* $\mathrm{LP}^{\mathrm{MLN}}$ *program* Π *such that the* $SM'[\Pi]$ *is not empty, stable models of* Π *under the* $\mathrm{LP}^{\mathrm{MLN}}$ *semantics and the models of* $comp(\Pi)$ *under the MLN semantics have the same probability distribution over all interpretations.*

5.3 Tseitin's transformation for completion formulas

Markov Logic Network solvers ALCHEMY, TUFFY convert the formulas to its CNF representation before proceeding with the inference. The completion rule as yielded by Equation (5.2) may blow up the rule size exponentially in the worst case when it is converted to its CNF after grounding. Tseitin (Tseitin, 1968) shows a way to reduce complexity by introducing proxy variables for subformulas. This keeps the number of clauses linear in the size of the input rule. Tseitin's transformation is an equisatisfiable transformation i.e. the transformed formula is satisfiable iff the original formula is satisfiable.

Review: Tseitin's transformation

Given a formula F, let $sub(F)$ be the set of all subformulas of F including F itself and $p_{sub(F)}$ represent a new variable introduced for each of the subformulas of F. The Tseitin's transformation of F is the formula

$$p_F \wedge \bigwedge_{F_1 \odot F_2 \in sub(F)} p_{F_1 \odot F_2} \leftrightarrow p_{F_1} \odot p_{F_2} \tag{5.3}$$

where $\odot$ is an arbitrary boolean connective. Since $p_{F_1 \odot F_2} \leftrightarrow p_{F_1} \odot p_{F_2}$ contains at most three literals and two connectives, the size of this formula in CNF is bound by a constant.

Example 23. Let F be the formula

$$(a \wedge b) \vee (c \wedge d) \tag{5.4}$$

converting it into CNF would yield

$$(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d).$$

The conversion blows the formula size exponentially in the number of literals in the original formula. The Tseitin's transformation $T(F)$ of (5.4) is

$$x_1 \leftrightarrow a \wedge b$$

$$x_2 \leftrightarrow c \wedge d$$

$$x_3 \leftrightarrow x_1 \vee x_2$$

$$T(F) \text{ is } x_3 \wedge (x_3 \leftrightarrow x_1 \vee x_2) \wedge (x_2 \leftrightarrow c \wedge d) \wedge (x_1 \leftrightarrow a \wedge b)$$

where each of the substitutions can be converted into CNF,

$$\begin{aligned} x_1 \leftrightarrow a \wedge b &\Leftrightarrow x_1 \rightarrow (a \wedge b) \wedge ((a \wedge b) \rightarrow x_1) \\ &\Leftrightarrow (x_1 \rightarrow a) \wedge (x_1 \rightarrow b) \wedge (\neg a \vee \neg b \vee x_1) \\ &\Leftrightarrow (\neg x_1 \vee a) \wedge (\neg x_1 \vee b) \wedge (\neg a \vee \neg b \vee x_1) \end{aligned}$$

Using Tseitin's transformation in LPMLN2MLN

System LPMLN2MLN does not use the Tseitin's transformation as described in Equation (5.3) as is but uses a simplified version of the translation. The original translation considers all sub-formulas of the original formula to be replaced by auxiliary variables. However,

the simplified algorithm only considers disjunctive terms in the completion of a rule as described in Equation (5.2) as a sub-formula

Given the completion of Π as described in (5.2), for every disjunctive term $body(\mathbf{t'_2})$ in the rule we add the rule

$$\alpha : \tilde{\forall} \left(Aux_{body(\mathbf{t'_2})}(\mathbf{t'_2}) \leftrightarrow body(\mathbf{t'_2}) \right) \tag{5.5}$$

to Π where $Aux_{body(\mathbf{t'_2})}(\mathbf{t'_2})$ is an atom introduced for the subformula $body(\mathbf{t'_2})$ and rewrite the completion rule as

$$\alpha \ : \ head(\mathbf{v}) \rightarrow \bigvee_{\substack{w:R' \in \Pi \\ \mathbf{z} \ \in \ \mathbf{t'_2} \setminus \mathbf{v}}} \left(\exists \mathbf{z} \ Aux_{body(\mathbf{t'_2})}(\mathbf{t'_2}) \right) . \tag{5.6}$$

The following theorem justifies the equivalent rewriting using Aux atoms

Theorem 4. *(Lee* et al.*, 2017, Proposition 1) For any MLN $\mathbb{L}$ of signature σ, let $F(\mathbf{x})$ be a subformula 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*

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

For any interpretation I of $\mathbb{L}$, let I_{aux} be the extension of I of signature $\sigma \cup \{Aux\}$ defined by $I_{Aux}(Aux(\mathbf{c})) = (F(c))^I$ for every list $\mathbf{c}$ of ground terms. We have

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

5.4 Completion of disjunction in Head of rule

$\mathrm{LP}^{\mathrm{MLN}}$ allows rules of the form (2.1) which consists of disjunction in head. Rule R considered in Equation (5.1) consists of head with a single positive literal. We use the result

described in (Lee and Lifschitz, 2003) to extend the definition of completion to disjunction in head. Proposition 2 in (Lee and Lifschitz, 2003) states that for any tight program Π whose rules have the form as in (2.1) and any set X of atoms, X is an answer set for Π iff X satisfies $Comp(\Pi)$. This proposition shows that the method of computing answer sets based on completion can be extended to tight programs whose rules have the form (2.1).

Consider for instance a rule with disjunction in head

$$w: \ P_1(\mathbf{t}_1) \vee ... \vee P_n(\mathbf{t}_n) \leftarrow Body$$

According to the Proposition stated above, the rule can be transformed into n rules of the form (5.1) as

$$w: \ P_i(\mathbf{t}_i) \leftarrow Body \wedge \bigwedge_{j:j\in\{1...n\}\setminus i} \neg P_j(\mathbf{t}_j)$$

for each $i \in \{1 \dots n\}$.

Choice rules in $head$ are handled in a similar way. A choice rule given by

$$w: \ \{P(\mathbf{t})\}^{ch} \leftarrow Body$$

is nothing but a case of disjunction in head

$$w: \ P(\mathbf{t}) \vee \neg P(\mathbf{t}) \leftarrow Body.$$

5.5 LPMLN2MLN System Architecture

Figure 5.1 shows the architecture of the LPMLN2MLN system. LPMLN2MLN is an encompassing system comprising of the LPMLN2MLN compiler and the underlying solver ALCHEMY, TUFFY and ROCKIT. The input to the system is an $\mathrm{LP}^{\mathrm{MLN}}$ program. The syntax of the input program is detailed in Section 5.6. The input is a set of weighted logic

programming rules given to the LPMLN2MLN compiler. The compiler outputs an MLN encoding according to the translation defined in Sections 5.2 and 5.3 which is given as input to the solvers ALCHEMY, TUFFY or ROCKIT.

The complier outputs three different (but equivalent) encodings based on the underlying solver selected by the user. Each of these three solvers support different input language syntax and the compiler takes care of these differences. From the user's perspective, the input is the same irrespective of the solver selected.

Each of the solvers expect input in first-order logic syntax (ignoring the minor differences in the input language syntax amongst these solvers). The compiler translates each rule of the form (5.1) into an equivalent non-ground rule. For instance, each rule of the form

$$head(\mathbf{x}) \leftarrow body(\mathbf{y})$$

is translated by the compiler into

$$body(\mathbf{y}) \rightarrow head(\mathbf{x})$$

defined as $trans(\Pi)$. This is a trivial $1-1$ translation performed for each rule R in Π. The compiler then computes the completion $comp(\Pi)$ of the program and adds the result to $trans(\Pi)$. The input to the respective MLN solvers is $trans(\Pi) \cup comp(\Pi)$.

There are three modes of computation in LPMLN2MLN: Most probable stable model, marginal probability and conditional probability. The mode of computation is determined by the arguments provided to the underlying solvers. Each of these underlying solvers have various options that can be used to control the solving parameters which determine the mode of computation, the accuracy of the answers, the solving algorithms used, etc. These options are passed as it is by LPMLN2MLN to the solver selected.

5.6 LPMLN2MLN Input Language Syntax

This section describes the input language syntax of the LPMLN2MLN system. Users familiar with ALCHEMY syntax would find that LPMLN2MLN syntax follows ALCHEMY syntax except one difference where rules are written in the form as described in Equation (2.1).

5.6.1 Logical connectives

The syntax for logical connectives is as follows:

- `not` for negation,
- `^` for conjunction,
- `v` for disjunction,
- <= for implication,
- = for equality between two identifiers and
- `!=` for inequality between two identifiers.

Operator precedence is as follows: negation $>$conjunction $>$disjunction $>$implication $>$equality $=$ inequality. Associativity for all the operators is left to right.

5.6.2 Identifiers and Variables

A legal identifier is a sequence of letters, digits and underscores that begin with a letter. Identifier for an object constants should begin with an uppercase letter, and it should begin with a lowercase letter for an object variable.

5.6.3 Declarations

Declarations are similar to ALCHEMY syntax. The signature of an atom needs to be declared before it can be used in the program. The declaration of an atom with predicate constant P and n sorts $s_1, \ldots, s_n$ is given by

$$P(s_1, ..., s_n)$$

The domain of each sort $s_1, \ldots, s_n$ also needs to be declared before the predicate itself is declared. If a sort s has n objects $o_1, \ldots, o_n$ it is represented as

$$s_i = \{o_1, \ldots, o_n\}$$

All the sorts that are required for defining the predicates need to be defined before the predicate declaration. A sort may not be empty. All object constants for a sort s_i that are used in the input program need to be declared to belong to s_i. MLN solvers can infer the domain of a sort from the evidence file as well. Therefore, for programs with large domains where most of the object constants are used in the evidence file, the user can declare a sort containing only the object constants that are used in the input program. These object constants are required for completion in accordance with Equation (5.2).

5.6.4 Rules

Rules are of two types, soft rules and hard rules. A soft rule is written as

$$w\ head <= body$$

where $head$ is a disjunction of atoms or empty, $body$ is a conjunction of literals, equality terms or its negation, or empty and w is the weight of the rule. A weight is a whole number or a real number in the usual decimal notation. Equality terms or its negation can be used in the $body$ of a rule as

```
j =  k
j != k
```

where j is a variable and k can be a variable or an object constant. A hard rule is written as

$$head <= body.$$

Notice the period at end of the hard rule. If the $body$ of a rule is empty, <= is dropped. Such a rule is written as

$$w <= body$$
$$<= body.$$

A rule with both $head$ and $body$ empty cannot be written.

Choice rule

Choice rules can be written as

$$w \{h_i\} <= body$$
$$\{h_i\} <= body.$$

where h_i is a single atom in the head of a rule.

5.6.5 Comments

All comments start with // and start on a new line.

LPMLN2MLN Rule examples

This section describes the different types of rules that can be used as input for the system LPMLN2MLN. Each of the rules below is a *Hard rule* (notice the period at the end). Each of these *Hard rules* can be written as a *Soft rule* by removing the period and adding a weight

at the start of the rule. For all the examples below $T, F, C2$ are object constants and x, y are variables.

- Simple formula:

```
male(T) <= not intervene(T).
```

- Disjunction in Head:

```
male(T) v male(F) <= not intervene(T).
```

- Conjunction in Body:

```
human(T, C2) <= male(x) ^ human(T, x).
```

- Constraint:

```
<= male(x) ^ male(y).
```

- Comparison Operators:

```
female(x) <= male(x) ^ male(y) ^ x!=y.
```

```
female(x) <= male(x) ^ male(y) ^ x=y.
```

- Choice Rules:

```
{load(x, y)} <= step(x, y).
```

Example 24. Encoding of Example 6 in the input language of LPMLN2MLN

```
entity = {Jo}
Bird(entity)
MigratoryBird(entity)
ResidentBird(entity)
```

```
Bird(x) <= ResidentBird(x).

Bird(x) <= MigratoryBird(x).

<= ResidentBird(x) ^ MigratoryBird(x).

2 ResidentBird(Jo)

1 MigratoryBird(Jo)
```

5.7 LPMLN2MLN Usage

The basic usage of LPMLN2MLN is

```
lpmln2mln  -i <input_file> [-e <evid_file>] [-r <output_file>]
   [-q <query_predicates>] [-a] [-t] [-ro] [-mln <mln_options>]


optional arguments:
-h, --help              show this help message and exit
-i <input_file>         input file. [REQUIRED]
-e <evidence_file>      evidence file
-r <output_file>        output file. [REQUIRED]
-q <query_predicates>   Multiple comma separated query
                        predicates.
-al, -alchemy           [DEFAULT] Compile for Alchemy
-tu, -tuffy             Compile for Tuffy
-ro, -rockit            Compile for rockit
-mln " <mln_options>"   Extra options for the respective
                        solvers.Passed as it is to the solvers.
                        Options are enclosed in quotes. [1]
```

[1] There should be a space between the first quote and the MLN options that are required to be passed to the respective MLN solver. If the space is not provided, the MLN options are parsed as options of the

Command line usage

This section describes the command line usage of LPMLN2MLN and some examples of usage [2] . We use the input as described in Example 24 for this section.

- MAP inference

  ```
  lpmln2mln -i <input_file> -r <output_file> -q <query_predicates>
  ```

  By default LPMLN2MLN uses ALCHEMY. All the MLN solvers used in LPMLN2MLN require an output file to store the output and a query predicate. Therefore these options are required by LPMLN2MLN as well. For example the command

  ```
  lpmln2mln -i bird.lpmln -r out.txt -q Bird -mln " -m"
  ```

  computes MAP inference using ALCHEMY. Here we are providing `-a` as an argument to the `-mln` option which is required to instruct ALCHEMY to compute MAP inference. Corresponding option needs to be provided for TUFFY if it is used as a solver. Solver ROCKIT computes MAP inference by default.

  Output:

  ```
  Bird(Jo)
  ResidentBird(Jo)
  ```

- Marginal Probability of query predicates

  ```
  lpmln2mln -i <input_file> -r <output_file> -q <query_predicates>
  ```

  For example the command

  ```
  lpmln2mln -i bird.lpmln -r out.txt -q Bird
  ```

LPMLN2ASP system. This is a limitation of the platform Ubuntu on which the system is developed. Example usage in the further sections would make the usage of this option clear.

[2]The filename is birds.lpmln for the usage

computes marginal probability of `Bird` using ALCHEMY. Since ALCHEMY'S default operation is to compute marginal probability no other options are required.

Output:

```
Bird(Jo) 0.90296
```

- Marginal Probability of query predicates using TUFFY as the solver

```
lpmln2mln -i <input_file> -r <output_file> -q <query_predicates> -
    tuffy
```

For example the command

```
lpmln2mln -i bird.lpmln -r out.txt -q Bird -tuffy -mln " -marginal"
```

computes marginal probability of `Bird` using TUFFY. Since TUFFY'S default operation is to compute MAP inference, we need to add `-mln " -marginal"` to instruct TUFFY to compute marginal probability. This outputs a command line that should be executed in the location where TUFFY is installed. An example output for this mode is

```
java -jar tuffy.jar -i input.mln -r out.txt -q Bird  -marginal
```

Output after executing the command generated above in the installation location of TUFFY:

```
1.0000  Bird("Jo")
```
[3]

- Conditional probability of query predicates given evidence e

```
lpmln2mln -i <input_file> -r <output_file> -q <query_predicates> -e
    <evidence_file>
```

[3]Note that the accuracy of the solvers depend on the parameters provided to the solver. In this case, TUFFY is run with default parameters which are different than ALCHEMY'S default parameters and hence we get different results.

Since evidence is provided, conditional probability of the query given evidence is computed. For example

```
lpmln2mln -i bird.lpmln -r out.txt -q ResidentBird -e evid.db
```

computes conditional probability of `Bird` where `evid.db` contains

```
Bird(Jo)
```

System LPMLN2MLN uses the evidence file by passing it as it is to the solver selected by the user and therefore the user needs to make sure that the syntax of the rules in evidence file conforms to that of the respective solver. Here we are calculating $P(ResidentBird(Jo) \mid Bird(Jo))$, that is, probability of $ResidentBird$ being Jo given that Jo is a $Bird$.

Output:

```
ResidentBird(Jo) 0.724978
```

5.8 Target systems

The system LPMLN2MLN translates the input program into weighted first order formulas, which can be run on ALCHEMY, TUFFY or ROCKIT. The following section describes the respective systems and their limitations.

5.8.1 ALCHEMY

The input language of ALCHEMY allows us to write first-order formulas prepended with weights. Any first-order logic formula can be written in ALCHEMY. ALCHEMY performs all computation in memory. This limits its usage to smaller domains which can be computed in memory. Larger domains most likely leads to a segment fault with this system. When using LPMLN2MLN system to compute $\mathrm{LP}^{\mathrm{MLN}}$ program with ALCHEMY, the user needs

to be wary of some limitations of ALCHEMY. The characters @ and $ are reserved and should not be used in the input program to LPMLN2MLN. Due to the internal processing of functions in ALCHEMY, variable names should not start with "funcVar" and predicate names should not start with "isReturnValueOf". The completion rules as obtained from Equation (5.2) and Equation (5.6) can be used directly in ALCHEMY.

5.8.2 TUFFY

TUFFY is another MLN solver that takes as input weighted rules in the first-order logic syntax. Solver ALCHEMY cannot scale well to real-world datasets due to all its computation being in-memory. TUFFY achieves scalability via three factors: (i) bottom up approach to grounding; (ii) hybrid architecture that performs local search using a relational database management system; (iii) partitioning, loading and parallel algorithms for solving. It has been shown in (Niu *et al.*, 2011) that TUFFY is more efficient and scalable than ALCHEMY in grounding of larger domains which makes a TUFFY translation desirable.

Like ALCHEMY, TUFFY takes as input weighted first order formulas. However, the input language of TUFFY is not as general as the input language of ALCHEMY.

Differences between the input languages of TUFFY and ALCHEMY

- TUFFY does not support bi-directional implication <=> which is supported by ALCHEMY. LPMLN2MLN handles it by translating a rule with bi-directional implication such as

  ```
  a <=> b.
  ```

  to

  ```
  a => b.
  b => a.
  ```

- TUFFY uses different syntax when comparing variables or variables with constants. While we can directly write

  $x = y \wedge x = Jo$

  $x = y \vee x = Jo$

  in ALCHEMY, for TUFFY the equality needs to be encoded as

  [x = y AND x = Jo]

  [x = y OR x = Jo]

  and grouped together. Here, x and y are variables and Jo is a constant.

- Exist clauses are used differently in TUFFY. A rule of the form,

  ```
  p(x) => Exist y q(x,y) v Exist z r(x,z)
  ```

  is written in Tuffy as

  ```
  Exist y,z   p(x) => q(x,y) v r(x,z)
  ```

- TUFFY does not accept predicates without any arguments. For example, consider the following LPMLN2MLN input

  ```
  P
  Q
  R
  1 P <= Q
  1 Q <= P
  2 P <= not R
  3 R <= not P
  ```

  While this input is valid for ALCHEMY, for TUFFY the user needs to encode the program as

```
temp = {X}
P(temp)
Q(temp)
R(temp)


1 P(X) <= Q(X)
1 Q(X) <= P(X)
2 P(X) <= not R(X)
3 R(X) <= not P(X)
```

Here we introduce a sort `temp` for all the predicates without arguments in the original program. Since `temp` cannot be used without declaring it first and since a sort cannot be empty, we add a dummy object constant `1` to the sort `temp`. Although the input language of LPMLN2MLN allows literals without any terms, the user needs to make sure that the output program generated is runnable by encoding it in the way shown above.

- POSTGRES is used as the database which is used for grounding and solving by TUFFY and needs to be installed separately.

TUFFY Translation

This section describes the translations used in order to make the input program compatible with TUFFY.

- Each rule of the form as described in Equation (5.5) is translated as

$$\begin{aligned} &\alpha : \tilde{\forall} \left(Aux_{body(\mathbf{t}'_\mathbf{2})}(\mathbf{t}'_\mathbf{2}) \rightarrow body(\mathbf{t}'_\mathbf{2}) \right) \\ &\alpha : \tilde{\forall} \left(body(\mathbf{t}'_\mathbf{2}) \rightarrow Aux_{body(\mathbf{t}'_\mathbf{2})}(\mathbf{t}'_\mathbf{2}) \right) \end{aligned} \tag{5.7}$$

  This is an equivalent translation.

- Each rule of the form as described in Equation (5.2) is translated as

$$\alpha \; : \; \exists \mathbf{z} \Bigg(head(\mathbf{v}) \rightarrow \bigvee_{\substack{w : head(\mathbf{v}) \leftarrow body(\mathbf{t_2'}) \in \Pi \\ \mathbf{z} \; \in \; \mathbf{t_2'} \setminus \mathbf{v}}} body(\mathbf{t_2'}) \Bigg) \; . \tag{5.8}$$

  This is again an equivalent translation since $\mathbf{z}$ only contains variables from $\mathbf{t_2'} \setminus \mathbf{v}$.

- Each rule containing equality term is post-processed to make them compatible with TUFFY syntax. For example consider the program in the input language of ALCHEMY

```
set = {1,2}
p(set)
q(set)
p(x) => x=1 v q(x) v x=2.
```

  In the input language of TUFFY the program is

```
set = {1,2}
p(set)
q(set)
p(x) => q(x) v [x=1 OR x=2]
```

5.8.3 ROCKIT

ROCKIT is yet another MLN solver that reduces MAP inference in graphical models as an optimization problem in *Integer Linear Programming*(ILP). ROCKIT uses *Cutting Plane Aggregation* [4] which is a novel meta-algorithm to compute MAP estimates on ILP instances. ROCKIT parallelizes the MAP inference pipeline taking advantage of shared-memory multi-core architectures which makes it faster than both ALCHEMY and TUFFY as shown in (Noessner *et al.*, 2013).

[4] Cutting plane aggregation is disabled when existential formulas are used

ROCKIT language syntax is even more limited than that of TUFFY. This section lists down some of these limitations of ROCKIT syntax. The current implementation is limited in supporting ROCKIT syntax. ROCKIT does not allow equality terms and hard rules with empty body and therefore shouldn't be used in LPMLN2MLN when using ROCKIT as the solver.

Differences between ROCKIT, ALCHEMY and TUFFY

- In ROCKIT, it is not possible to use implications (=¿) and conjunctions (). The user has to transform the formula to CNF.
- Exist syntax is different from TUFFY and ALCHEMY.

  ```
  Exists y friends(x,y)
  ```

  becomes

  ```
  |y| friends(x,y) >= 1
  ```

- All constants should be within " " . Rest all are considered variables in a formula.
- Gurobi is the internal solver. It is a commercially available ILP solver with free academic licenses and needs to be installed separately.
- The evidence file cannot be empty. In comparison, ALCHEMY supports an empty evidence files and TUFFY can be executed without providing the evidence file.
- MySQL is used as the database which is used for grounding and needs to be installed separately.

ROCKIT Translation

The translation of LP^{MLN} to the input language if ROCKIT is similar to the translation to the input language of TUFFY.

- All rules in $trans(\Pi)$

  $$body(\mathbf{y}) \rightarrow head(\mathbf{x})$$

  are translated to $trans_{ro}(\Pi)$ as

  $$\neg body(\mathbf{y}) \vee head(\mathbf{x}). \tag{5.9}$$

  This is an equivalent translation.

- Rules obtained from (5.7) and (5.8) can be used in ROCKIT after applying the translation as described in (5.9).

- Each rule with "Exist" is post-processed to make it compatible with ROCKIT syntax. For example consider the program in ALCHEMY

  ```
  set = {1,2}
  p(set)
  q(set)
  Exist x p(x).
  ```

  In ROCKIT, the program is written as

  ```
  set = {1,2}
  p(set)
  q(set)
  |x| p(x) >=1.
  ```

Chapter 6

COMPUTING OTHER FORMALISMS IN LPMLN2MLN

It has been shown that formalisms like Pearl's Causal Models, Bayes Net, P-Log and Problog can be embedded in LP^{MLN} (Lee and Wang, 2016b; Lee and Yang, 2017b; Lee *et al.*, 2015). In this chapter we demonstrate how to use LPMLN2MLN to compute the tight fragments of these formalisms. We use ALCHEMY as the solver for computing each of the formalisms below.

6.1 Computing P-log in LPMLN2MLN

We refer the reader to Section 4.2 for a description of P-Log formalism.

Example 25. We use the same example as described in Example 16. The following is an encoding of Example 16 in the input language of LPMLN2MLN .

```
doors = {1,2,3}
numbers = {2,3}
boolean = {T,F}
attributes = {Attropen ,Attrselected ,Attrprize}

Open(doors)
Selected(doors)
Prize(doors)
CanOpen(doors,boolean)
Obs(attributes ,doors)
UnObs(attributes ,doors)
Intervene(attributes)
```

```
NumDefProb(attributes ,numbers)
PossWithDefProb(attributes ,doors)
PossWithAssProb(attributes ,doors)

CanOpen(d,F) <= Selected(d).
CanOpen(d,F) <= Prize(d).
CanOpen(d,T) <= !CanOpen(d,F).

<= CanOpen(d,T) ^ CanOpen(d,F).
<= Prize(d1) ^ Prize(d2) ^ d1 != d2.
<= Selected(d1) ^ Selected(d2) ^ d1 != d2.
<= Open(d1) ^ Open(d2) ^ d1 != d2.

Prize(1) v Prize(2) v Prize(3) <= !Intervene(Attrprize).
Selected(1) v Selected(2) v Selected(3) <= !Intervene(Attrselected).
Open(1) v Open(2) v Open(3) <= !Intervene(Attropen).
<= Open(d) ^ !CanOpen(d,T) ^ !Intervene(Attropen).

PossWithDefProb(Attrprize ,d) <= !PossWithAssProb(Attrprize ,d) ^!
   Intervene(Attrprize).
NumDefProb(Attrprize ,2) <= Prize(d1) ^ PossWithDefProb(Attrprize ,d1)
   ^PossWithDefProb(Attrprize ,d2) ^ d1!=d2.
NumDefProb(Attrprize ,3) <= Prize(d1) ^ PossWithDefProb(Attrprize ,d1)
   ^PossWithDefProb(Attrprize ,d2) ^ PossWithDefProb(Attrprize ,d3) ^d1
   !=d2 ^ d1!=d3 ^ d2!=d3.

-0.6931 not not NumDefProb(Attrprize ,2)
```

```
-0.4055 not not NumDefProb(Attrprize ,3)

PossWithDefProb(Attrselected ,d) <= !PossWithAssProb(Attrselected ,d)
   ^!Intervene(Attrselected).
NumDefProb(Attrselected ,2) <= Selected(d1) ^PossWithDefProb(
   Attrselected ,d1) ^ PossWithDefProb(Attrprize ,d2) ^ d1!=d2.
NumDefProb(Attrselected ,3) <= Selected(d1) ^PossWithDefProb(
   Attrselected ,d1) ^ PossWithDefProb(Attrselected ,d2) ^
   PossWithDefProb(Attrselected ,d3) ^ d1!=d2 ^ d1!=d3 ^ d2!=d3.

-0.6931 not not NumDefProb(Attrselected ,2)
-0.4055 not not NumDefProb(Attrselected ,3)

PossWithDefProb(Attropen ,d) <= !PossWithAssProb(Attropen ,d) ^ !
   Intervene(Attropen) ^ CanOpen(d,T).
NumDefProb(Attropen ,2) <= Open(d1) ^ PossWithDefProb(Attropen ,d1) ^
   PossWithDefProb(Attropen ,d2) ^ d1!=d2.
NumDefProb(Attropen ,3) <= Open(d1) ^ PossWithDefProb(Attropen ,d1) ^
   PossWithDefProb(Attropen ,d2) ^ PossWithDefProb(Attropen ,d3) ^ d1!=
   d2 ^ d1!=d3 ^ d2!=d3.

-0.6931 not not NumDefProb(Attropen ,2)
-0.4055 not not NumDefProb(Attropen ,3)

Obs(Attrselected ,1).
<= Obs(Attrselected ,1) ^ !Selected(1).
Obs(Attropen ,2).
```

```
<= Obs(Attropen ,2) ^ !Open(2).


UnObs(Attrprize ,2).
<= UnObs(Attrprize ,2) ^ Prize(2).
```

On executing

```
lpmln2mln -i input.lp -r out.txt -q Prize
```

the output is

```
Prize(1) 0.286021
Prize(2) 4.9995e-05
Prize(3) 0.713979
```

which corresponds to the output of the P-log program. Note that the accuracy of the output can be improved by giving parameters to solver ALCHEMY such as `-maxsteps` using the `-mln` options in LPMLN2MLN like

```
lpmln2mln -i input.lp -r out.txt -q Prize -mln " -maxSteps 10000"
```

6.2 Computing Pearl's Causal Model in LPMLN2MLN

We refer the reader to Section 4.3 for a description of the formalism.

Example 26. We use the same example as used in Example 17. The following is an encoding of Example 17 in the input language of LPMLN2MLN . This translation is represented in the input language of LPMLN2MLN as follows

```
events={A0,A1,B0,B1,C0,C1,D0,D1}


do(events)
a
b
```

```
c
d
u
w
sa
bs
cs
ds

0.8472  u
-1.3862  w

c <= u.
a <= c.
a <= w.
b <= c.
d <= a.
d <= b.

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).

cs <= do(C1).
```

```
as <= do(A1).
bs <= do(B1).
ds <= do(D1).
```

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.

The different types of inference that can be computed are:

- *Prediction*: To represent prediction, the evidence file contains

  ```
  !a
  ```

  On executing

  ```
  lpmln2mln -i pcm.lp -e evid.db -q d
  ```

  the output is

  ```
  d 4.9995e-05
  ```

  which means that if rifleman A did not shoot, the prisonser is certainly alive.

- *Abduction*: To represent abduction, the evidence file contains

  ```
  !d
  ```

  On executing

  ```
  lpmln2mln -i pcm.lp -e evid.db -q c
  ```

  the output is

  ```
  c 4.9995e-05
  ```

  which means that if the prisoner is alive then the captain did not order execution.

- *Transduction*: To represent transduction, the evidence file contains

```
a
```
[1]

On executing

```
lpmln2mln -i pcm.lp -e evid.db -q b
```

the output is

```
b 0.877962
```

which means there is a 87.7% chance that rifleman B shot as well.

- *Action*: To represent an action, the evidence file contains

```
!c
do(A1)
```

On executing

```
lpmln2mln -i pcm.lp -e evid.db -q ds,bs
```

outputs

```
ds 0.99995
bs 4.9995e-05
```

which means that the prisoner will die and rifleman B will not shoot.

- *Counterfactual*: To represent the counterfactual query, the evidence file contains

```
do(A0)
d
```

Here `d` is an *observation* and `do(a0)` is an *intervention*. On executing

[1]Note that in LPMLN2ASP the evidence file contains `:- not a` but since double negation is invalid syntax for ALCHEMY, implicitly double negation is added.

```
lpmln2mln -i pcm.lp -e evid.db -q ds
```

LPMLN2MLN outputs

```
ds 0.916958
```

which means there is around 8.4% chance that the prisoner would be alive.

6.3 Computing Bayes Net in LPMLN2MLN

We refer the reader to Section 4.4 for a description of the formalism.

Example 27. We use the same example as used in Example 18. The following is an encoding of Example 18 in the input language of LPMLN2MLN .

```
parent = {A,S,L,R,None}
combination = {T,F,T1F1,T1F0,T0F1,T0F0,F1,F0,A1,A0,L1,L0}


tampering
fire
alarm
smoke
leaving
report
pf(parent, combination)


-3.8918 pf(None,T)
-4.5951 pf(None,F)
0 pf(A,T1F1)
1.7346 pf(A,T1F0)
4.5951 pf(A,T0F1)
```

```
-9.2102 pf(A,T0F0)

2.1972 pf(S,F1)

-4.5951 pf(S,F0)

1.9924 pf(L,A1)

-6.9067 pf(L,A0)

1.0986 pf(R,L1)

-4.5951 pf(R,L0)


tampering <= pf(None,T).


fire <= pf(None,F).


alarm <= tampering ^ fire ^ pf(A,T1F1).

alarm <= tampering ^ not fire ^ pf(A,T1F0).

alarm <= not tampering ^ fire ^ pf(A,T0F1).

alarm <= not tampering ^ not fire ^ pf(A,T0F0).


smoke <= fire ^ pf(S,F1).

smoke <= not fire ^ pf(S,F0).


leaving <= alarm ^ pf(L,A1).

leaving <= not alarm ^ pf (L,A0).


report <= leaving ^ pf(R,L1).

report <= not leaving ^ pf(R,L0).
```

The different types of inferences that can be computed are:

- *Diagnostic Inference*: To compute $P(fire = \mathbf{t} \mid leaving = \mathbf{t})$, the user can invoke

```
lpmln2mln -i fire-bayes.lpmln -e evid.db -q fire
```

where `evid.db` contains the line

```
leaving
```

This outputs

```
fire 0.328017
```

- *Predictive Inference*: To compute $P(leaving = \mathbf{t} \mid fire = \mathbf{t})$, the user can invoke

```
lpmln2mln -i fire-bayes.lpmln -e evid.db -q leaving
```

where `evid.db` contains the line

```
fire
```

This outputs

```
leaving 0.886961
```

- *Mixed Inference*: To compute $P(alarm = \mathbf{t} \mid fire = \mathbf{f}, leaving = \mathbf{t})$, the user can invoke

```
lpmln2mln -i fire-bayes.lpmln -e evid.db -q alarm
```

where `evid.db` contains two lines

```
!fire
leaving
```

This outputs

```
alarm 0.950955
```

- *Intercausal Inference*: To compute $P(tampering = \mathbf{t} \mid fire = \mathbf{t}, alarm = \mathbf{t})$, the user can invoke

  ```
  lpmln2mln -i fire-bayes.lpmln -e evid.db -q tampering
  ```

  where `evid.db` contains two lines

  ```
  fire
  alarm
  ```

  This outputs

  ```
  tampering 0.0080492
  ```

- *Explaining away*: Lets compute $P(tampering = \mathbf{t} \mid alarm = \mathbf{t})$. The user can invoke

  ```
  lpmln2mln -i fire-bayes.lpmln -e evid.db  -q tampering
  ```

  where `evid.db` contains line

  ```
  alarm
  ```

  This outputs

  ```
  tampering 0.478002
  ```

  If this result is compared with the previous result, we can see that $P(tampering = \mathbf{t} \mid alarm = \mathbf{t}) > P(tampering = \mathbf{t} \mid fire = \mathbf{t}, alarm = \mathbf{t})$. Observing the value of $fire$ explains away the $tampering$ i.e. the probability of $tampering$ decreases.

6.4 Computing Problog in LPMLN2MLN

We refer the reader to Section 4.5 for a description of the formalism. Tightness in Problog is defined similarly to that of $\mathrm{LP}^{\mathrm{MLN}}$. Consider a ground problog rule of the form

$A \leftarrow B_1, ..., B_m, not\ B_{m+1}, ..., not\ B_n$ where $A, B_1, ..., B_n$ are atoms from σ ($0 \leq m \leq n$), and A is not a probabilistic atom. A set of rules is called *tight* if its dependency graph is acyclic. An atom A depends on an atom B_i if B_i occurs in a ground rule with head A.

It is easy to see that any tight problog program can be converted into a tight $\mathrm{LP}^{\mathrm{MLN}}$ program using the translation as described in Section 4.5 and similarly any non-tight problog program can be converted into an non-tight $\mathrm{LP}^{\mathrm{MLN}}$ program. Example 19 that was used to compute Problog in LPMLN2ASP is non-tight and therefore cannot be computed by LPMLN2MLN. Example 20, however, is tight and therefore can be computed by LPMLN2MLN.

Example 28. We use the same example as used in Example 20. The following is an LPMLN2MLN encoding of Example 20 corresponding to its LPMLN2ASP encoding

```
p = {Suzy, Billy}
x = {1,2}
y = {1,2}
throws(p)
broken
miss
msw(x,y)

0 throws(Suzy)
throws(Billy).

1.3862 msw(1,1)
msw(1,2).
broken <= throws(Suzy) ^ msw(1,1).
miss <= throws(Suzy) ^ msw(1,2) ^ not msw(1,1).
```

```
0.4054 msw(2,1)

msw(2,2).

broken <= throws(Billy) ^ msw(2,1).

miss <= throws(Billy) ^ msw(2,2) ^ not msw(2,1).
```

On executing

```
lpmln2mln -i input.lp -e empty.db -r out.txt -q broken
```

the output is

```
broken 0.752975
```

which corresponds to the value computed using PROBLOG2 and LPMLN2ASP.

Chapter 7

EXPERIMENTS

7.1 Maximal Relaxed Clique

We experiment on the problem of finding a *Maximal relaxed clique* in a graph. The goal is to create a subgraph such that maximum number of nodes in the graph and maximum number of edges in the graph are selected. For every subgraph, we assign a reward to every pair of connected nodes and a reward for every node included in the subgraph. A maximal relaxed clique is a subgraph that maximizes the reward that can be given to the subgraph.

The LPMLN2ASP encoding of the above problem is

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

Rule 1 states that every node X can be **in** the subgraph. Rule 3 states that if a node is not **in** the subgraph in an interpretation I, we give the interpretation I a penalty of 5. According to Rule 2, Given an interpretation I representing a subgraph, a pair of nodes in the subgraph is **disconnected** if there is no **edge** between them. Rule 4 states that if two nodes X and Y in an interpretation I are **disconnected**, we give a penalty of 5 to I. The command line used to run this program is

```
lpmln2asp -i input -e evidence -r output
```

Similarly the LPMLN2MLN encoding of the above problem is

```
NodeSet = {1}
```

```
In(NodeSet)

Node(NodeSet)

Edge(NodeSet, NodeSet)

Disconnected(NodeSet, NodeSet)


{In(x)} <= Node(x).

Disconnected(x, y) <= In(x) ^ In(y) ^ !Edge(x, y) ^ x!=y.

5  <= !In(x) ^ Node(x)

5  <= Disconnected(x, y)
```

We declare the sort `NodeSet` containing just 1 node. Additional nodes can be present in the evidence file. The command line used to run this program is:

```
lpmln2mln -i input -e evidence -r output -mln "<MAP inference option
   for the respective solvers>"
```

Example 29. Consider the graph and its LPMLN2ASP encoding as given above.

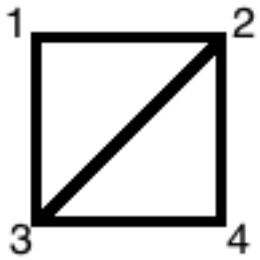

Figure 7.1: Maximal Relaxed Clique Example

For every interpretation I we calculate its penalty pnt_0^I as defined in Equation (3.5). Interpretation I with the lowest pnt_0^I is the optimal stable model. Consider an interpretation $I = \{in(1), in(2), in(3)\}$. For this interpretation, pnt_0^I is 5 given by $(w_3 * 1) + (w_4 * 0) = 5$ where w_i represents the weight of the i^{th} rule in the LPMLN2ASP encoding. Consider another interpretation $I = \{in(1), in(2), in(3), in(4)\}$. For this interpretation, pnt_0^I is $(w_3 *$

$0) + (w_4 * 2) = 10$. The pnt_0^I in the latter case is more even though all nodes are included because it has a pair of disconnected nodes: $(1, 4)$ and $(4, 1)$. Since the interpretation $I = \{in(1), in(2), in(3)\}$ results in the minimum pnt_0^I, I is the maximal relaxed clique. Note that $I = \{in(2), in(3), in(4)\}$ is another maximal relaxed clique of the same graph.

For this experiment, we generate a graph by randomly generating edges between nodes with probability $\{0.5, 0.8, 0.9, 1\}$ and different number of nodes $\{10, 20, 50, 100, 200, 300, 400, 500\}$ at each probability. For each problem instance, we perform MAP inferences to find maximal relaxed cliques with both LPMLN2ASP and LPMLN2MLN. The timeout is set to 20 minutes. The experiments are performed on a machine powered by 4 Intel(R) Core(TM) i5-2400 CPU with OS Ubuntu 14.04 LTS and 8G memory.

Instance	LPMLN2ASP **w.** CLINGO **4.5**	LPMLN2MLN **w.** ALCHEMY **2**	LPMLN2MLN **w.** TUFFY **0.3**	LPMLN2MLN **w.** ROCKIT **0.5**
p50n10 [1]	0.004[2] [205/239][3]	0.11 + 2.84 [4] [310/788][5]	3 + 6.083[6] [310/786][7]	0.693[8] [71][9]
p50n20	0.017 [769/887]	0.34 + 4.85 [1,220/3,136]	3 + 27.345 [1,220/3,132]	0.884 [242]
p50n50	Timeout [4,511/5,185]	2.5 + 11.18 [7,550/19,466]	3 + Timeout [7,550/19,438]	1.046 [1320]
p50n100	Timeout [17,744/20,294]	17.01 + 36.88 [30,100/77,643]	5 + Timeout [30,100/77,593]	1.539 [5229]
p50n200	Timeout [70,570/80,696]	154.75 + 235.62 [120,200/310,395]	13 + Timeout [120,200/310,382]	6.903 [20,376]
p50n300	Timeout [135,924/181,247]	564.62 + 799.61 [270,300/698,022]	36 + Timeout [270,300/697,868]	20.239 [45,501]

p50n400	Timeout [241,400/321,999]	Timeout	Timeout	Timeout [80,472]
p50n500	Timeout [376,7332/502,465]	Timeout	Timeout	Timeout
p80n10	0.004 [219/235]	0.1 + 2.27 [310/802]	3 + 5.568 [310/800]	0.755 [102]
p80n20	0.014 [802/808]	0.31 + 3.4 [1,220/3,169]	3 + 19.001 [1,220/3,165]	0.759 [366]
p80n50	2.730 [4,739/4,785]	2.45 + 12.2 [7,550/19,658]	3 + 712.472 [7,550/19,648]	1.11 [2,099]
p80n100	Timeout [18,717/19,193]	17.35 + 42.11 [30,100/78,552]	5 + Timeout [30,100/78,534]	1.941 [8,261]
p80n200	Timeout [74,174/75,913]	157.82 + 261.28 [120,200/313,845]	12 + Timeout [120,200/313,809]	179.944 [32,400]
p80n300	Timeout [108,969/127,337]	564.6 + 878.63 [270,300/705,886]	30 + Timeout [270,300/705,831]	Timeout [72,427]
p80n400	Timeout [193,151/225,501]	Timeout	Timeout	Timeout [129,097]
p80n500	Timeout [461,326/471,714]	Timeout	Timeout	Timeout [201,312]
p90n10	0.004 [223/233]	0.06 + 0.07 [310/802]	3 + 3.967 [310/802]	0.741 [109]
p90n20	0.015 [824/844]	0.28 + 1.71 [1,220/3,187]	3 + 4.084 [1,220/3,185]	0.806 [402]

p90n50	0.961 [4,932/5,000]	2.51 + 11.53 [7,550/19,833]	3 + 457.61 [7,550/19,832]	1.129 [2,350]
p90n100	Timeout [19,464/19,682]	17.34 + 42.02 [30,100/79,281]	5 + Timeout [30,100/79,272]	2.875 [9,215]
p90n200	Timeout [76,909/77,467]	160.18 + 267.17 [120,200/316,546]	20 + Timeout [120,200/316,527]	278.734 [36,482]
p90n300	Timeout [100,051/109,501]	567.73 + 928.94 [270,300/712,354]	34 + Timeout [270,300/712,324]	Timeout [81,759]
p90n400	Timeout [177,147/193,493]	Timeout	Timeout	Timeout [144,864]
p90n500	Timeout [478,931/481,977]	Timeout	Timeout	Timeout [225,901]
p100n10	0.007 [131/141]	0.1 + 0.11 [310/810]	3 + 3.839 [310/810]	0.588 [120]
p100n20	0.013 [461/481]	0.31 + 0.35 [1,220/3,220]	3 + 4.166 [1,220/3,220]	0.740 [440]
p100n50	0.046 [2,651/2,701]	2.52 + 3.11 [7,550/20,050]	3 + 6.088 [7,550/20,050]	0.955 [2,600]
p100n100	0.141 [10,301/104,01]	17.4 + 24.19 [30,100/80,100]	5 + 14.438 [30,100/80,100]	1.464 [10,200]
p100n200	0.384 [40,601/40,801]	159.96 + 259.22 [120,200/320,200]	15 + 77.492 [120,200/320,200]	2.084 [40,400]
p100n300	0.804 [90,901/91,201]	565.46 + 921.05 [270,300/720,300]	33 + 257.52 [270,300/720,300]	3.322 [90,600]

p100n400	1.395 [161,201/161,201]	Timeout	Timeout	4.772 [160,800]
p100n500	2.175 [251,501/252,001]	Timeout	Timeout	6.115 [251,000]

Figure 7.2: Running Statistics on Finding Maximal Relaxed Clique (MAP Inference)

Figure 7.2 gives the results of running maximal relaxed clique with various graph instances on each of the four underlying solvers. The graph instances range from a small size of 10 nodes to a large size of 500 nodes. We compare the system based on the results of the experiment primarily on the performance of the respective solvers. Note that while LPMLN2ASP and LPMLN2MLN with ROCKIT gives exact solutions, LPMLN2MLN with ALCHEMY and TUFFY may return sub-optimal solutions. The quality of answers for these solvers based on different parameters is discussed in the next experiment.

The naive grounding (grounding of formulas + the MRF creation time) of ALCHEMY is a primary bottleneck for the solver. Even after the compact encoding based on Equation (5.6) used in the translation for ALCHEMY, it times out during grounding for $N > 340$. Solver TUFFY uses database for grounding and noticeably has better grounding times than ALCHEMY for most graph instances ignoring the constant time it takes for TUFFY to connect to POSTGRES database server. In spite of better grounding mechanisms than ALCHEMY, solver TUFFY times out while grounding with $N > 370$ in our experiments. Although using database optimizes the grounding process in MLN solvers it is still not good enough when compared to the grounding process of CLINGO and ROCKIT. CLINGO uses GRINGO for grounding while ROCKIT uses MYSQL in conjunction with GUROBI for grounding. Both CLINGO and ROCKIT can ground all instances of the graph. The grounding time

for ALCHEMY and TUFFY is comparable to solving time while it is negligible compared to solving time for CLINGO and ROCKIT.

Grounding time of all solvers constantly increases as the number of nodes increases. Interestingly, this increase in grounding time for bigger graph instances does not correlate with the increase in solving time. For MLN solvers ALCHEMY and TUFFY, solving time increases constantly with graph size regardless of the sparsity of graph. A graph instance where all the nodes are connected $p = 1$ to each other, a fully connected graph, is solved much faster than all other instances by CLINGO and ROCKIT. For CLINGO , the running time is sensitive to particular problem instance due to the exact optimization algorithm CDNL-OPT (Gebser *et al.*, 2011) used in CLINGO. 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 for instances where $p \neq 1$, as the size of the graph increases, the search space gets larger, thus the running time increases. Both ALCHEMY and TUFFY use MaxWalkSat for MAP inference which allows the solver to return sub-optimal solutions. The approximate nature of the method allows relatively consistent running time for different problem instances, as long as parameters such as the maximum number of iterations/tries are fixed among all experiments. The running time was also not affected much by the edge probability. System ROCKIT uses Cutting Plane Inference (CPI) (Riedel, 2012) along with Cutting Plane Aggregation (CPA) (Noessner *et al.*, 2013) for inference. Using CPI, ROCKIT iteratively solves a partial version of the complete ground network based on the Cutting Plane approach. At each iteration, it checks for all the constraints that are unsatisfied and adds them to the ILP (Integer Linear Programming) solver GUROBI. In a fully connected graph instance where all of the nodes are connected, the number of rules violated due to the last rule of the program is 0, and therefore, ROCKIT runs faster .

Performance wise LPMLN2ASP outperforms LPMLN2MLN in a fully connected graph ($p = 1$) and in all other instances LPMLN2MLN with ROCKIT clearly outperforms others. One factor that aids in ROCKIT's performance is the internal solver GUROBI's multi-core architecture. GUROBI uses all the cores available on the machine for computation while CLINGO , ALCHEMY and TUFFY use a single core for computation [1] . GUROBI is also the fastest commercial ILP solver according to some benchmark results [2] .

Answer Quality of LPMLN2MLN in Maximal Relaxed Clique

We modify the relaxed clique example such that for every graph instance we know what the output is. We create graph instances such that each instance has N nodes and $N + 1$ edges. Every i^{th} node has an edge to the $(i + 1)^{th}$ node. N^{th} node is connected the first node. We then add an edge between the first node and $(\lfloor N/2 \rfloor) + 1$ node. The base case considered for this experiment is $N = 4$.

For $N = 4$, the optimal stable models are $I = \{in(1), in(2), in(3)\}$ and $I = \{in(1), in(4), in(3)\}$. Both the interpretations I have the lowest pnt_0^I of 5 because of the one node that is not included in either interpretation. It is easy to check that these interpretations have the lowest penalty by enumerating over all the 2^4 interpretations.

For $N = 5$, for $I = \{in(1), in(2), in(3)\}$, $pnt_0^I = 10$ due to the two nodes not considered in the interpretation. Interpretation without any nodes included has $pnt_0^I = 25$. Interpretations with any one node included has $pnt_0^I = 20$. Interpretations with any two nodes included has $pnt_0^I = 15 + 5 \times k$ where $k = 0$ if the two nodes are connected and $k = 2$ otherwise. Interpretations with any four nodes included has $pnt_0^I = 5 + 5 \times k$ where $k \geq 2$ because there has to be two nodes in the selection which are disconnected.

[1] TUFFY grounding utilizes POSTGRES which has a multi-core architecture but this speed-up only affects grounding and not solving

[2] The benchmark results are available on the GUROBI website at www.gurobi.com

Interpretation with all five nodes included has a $pnt_0^I = 40$ because of the four pairs of disconnected nodes. Therefore, $I = \{in(1), in(2), in(3)\}$ is the optimal stable model.

For $N > 5$, the optimal stable model is $\{in(1), in(\lfloor N/2 \rfloor + 1)\}$. When $N > 5$ the interpretation containing the subgraph with nodes $\{in(1), in(\lfloor N/2 \rfloor + 1)\}$ has a pnt_0^I value of $5 \times (N-2)$ for the $N-2$ nodes not present in the interpretation and 0 from disconnected edges. An empty interpretation has a pnt_0^I value of $5 \times N$ since no nodes are included. An interpretation with $in(K)$ where $K \in 1, \ldots, N$ gets a penalty of $5 \times (N-1)$. An interpretation with K nodes where $K \geq 3$ would get a penalty of $5 \times (N-K) + 5 \times D$ where D is the number of disconnected edges. Adding any node to the interpretation other than $\{in(1), in(\lfloor N/2 \rfloor + 1)\}$ would result in $D \geq 2$ because no three nodes are connected to each other. Also, any interpretation with nodes $\{in(i), in(i+1)\}$ where $i \in \{1, \ldots, N-1\}$ and $\{in(N), in(1)\}$ would result in penalty $5 \times (N-2)$ and thus it is also an optimal stable model. The input program is the same as maximal relaxed clique program as given in the above section. We use LPMLN2MLN with ALCHEMY and TUFFY to compare the quality of answers with different parameters.

To check the quality of the answer, the MAP inference output from LPMLN2MLN is fed as evidence to LPMLN2ASP . This results in a single interpretation I. We can then check the penalty of this interpretation by examining the $unsat$ atoms present in I. The penalty is the exponentiated negative sum of weight where weight is obtained from the $unsat$ atoms. Therefore, the lower the weight of an interpretation the worse it is compared to the gold standard result.

Table 7.1 compares the quality of answers using various configuration parameters of the underlying solvers. For TUFFY the default number of tries is set to 1. Therefore, we experiment TUFFY with MAXTRIES $= 10$. TUFFY improves search speed by using MRF partitioning. If the number of optimal components increases, it becomes likely that one step of WalkSat "breaks" an optimal sub-solution instead of fixing the sub-optimal component.

Therefore, we set DONTBREAK switch as one of the configuration option to see how it affects the accuracy of the result. DONTBREAK forbids WalkSat steps which break hard rules and that speeds up the computation in TUFFY.

Instance	Solver	Configuration	Grounding size	Time	Penalty
$N = 4$	LPMLN2ASP	default	27;33[4]	0.002	30
	ALCHEMY	default	30;100	1.53	30
	TUFFY	default	18;52	2.371	30
	TUFFY	maxTries=10	18;52	2.396	30
	TUFFY	dontBreak	18;52	2.15	30
	TUFFY	maxTries=10,dontBreak	18;52	2.505	30
	ROCKIT	default	22[5]	0.004	30
$N = 10$	LPMLN2ASP	default	230;327	0.002	45
	ALCHEMY	default	132;484	1.53	45
	TUFFY	default	108;338	2.371	45
	TUFFY	maxTries=10	108;338	2.396	50
	TUFFY	dontBreak	108;338	2.15	90
	TUFFY	maxTries=10,dontBreak	108;338	2.505	65
	ROCKIT	default	46	0.004	45
$N = 30$	LPMLN2ASP	default	1890;2787	0.3	145
	ALCHEMY	default	992;3844	6.46	160
	TUFFY	default	928;3588	31.915	150
	TUFFY	maxTries=10	928;3588	106.275	150
	TUFFY	dontBreak	928;3588	2.934	1175
	TUFFY	maxTries=10,dontBreak	928;3588	3.317	620
	ROCKIT	default	126	3.327	145

$N = 60$	LPMLN2ASP	default	7230;20540	621.97	295
	ALCHEMY	default	3782;14884	12	415
	TUFFY	default	3658;14388	180.69	300
	TUFFY	maxTries=10	3658;14389	Timeout	300
	TUFFY	dontBreak	3658;14390	5.085	6905
	TUFFY	maxTries=10,dontBreak	3658;14391	5.603	4955
	ROCKIT	default	246	56.04	295

Table 7.1: Answer Quality Maximal Relaxed Clique

Solver LPMLN2ASP gives the gold standard result and we compare the performance of other solvers against this result. The penalty is the sum of the weights of all unsatisfied rules in the grounded program (the lower the better). All the solvers' solving time increases as the size of graph increases, which is expected. ROCKIT with the default configuration gives the same answer as LPMLN2ASP while being significantly faster than LPMLN2ASP. ALCHEMY gives worse answers than TUFFY in default configuration while being faster than TUFFY. Since the graph instances are much smaller ALCHEMY is expected to be faster than TUFFY. Option MAXTRIES does not improve the quality of answers in TUFFY while taking significantly more time to compute. Although using the DONTBREAK option fastens the computation, it considerably worsens the answer quality to the point that DONTBREAK results in the highest penalty in all cases. Using DONTBREAK in conjunction with MAXTRIES results in second worse results. Interestingly, using MAXTRIES with DONTBREAK results in only a marginal increase in computation time.

[4]Number of atoms; Number of rules

[5]Number of evidence atoms.

7.2 Link Prediction in Biological Networks - A performance comparison with PROBLOG2

Public biological databases contain huge amounts of rich data, such as annotated sequences, proteins, domains, and orthology groups, genes and gene expressions, gene and protein interactions, scientific articles, and ontologies. Biomine (Eronen and Toivonen, 2012) is a system that integrates cross-references from several biological databases into a graphical model with multiple types of edges. Edges are weighted based on their type, reliability, and informativeness.

We use graphs extracted from the Biomine network [3] . The graphs are extracted around genes known to be connected to the Alzheimer's disease (HGNC ids 620, 582, 983, and 8744). A typical query on such a database of biological concepts is whether a given gene is connected to a given disease. In a probabilistic graph, the importance of the connection can be measured as the probability that a path exists between the two given nodes, assuming that each edge is true with the specified probability, and that edges are mutually independent (De Raedt *et al.*, 2007; Sevon *et al.*, 2006). Nodes in the graph correspond to different concepts such as gene, protein, domain, phenotype, biological process, tissue, and edges connect related concepts. Such a program can be expressed in the language of Problog as (Mantadelis *et al.*, 2015)

```
p(X,Y) :- drc(X,Y).
p(X,Y) :-
        drc(X, Z),
        Z \== Y,
        p(Z, Y).
```

The LPMLN2ASP encoding for the same program is

[3] We thank Theofrastos Mantadelis for providing us with the dataset

```
p(X,Y) :- drc(X,Y).
p(X,Y) :- drc(X, Z), Z != Y, p(Z, Y).
```

The evidence file contains weighted edges `drc/2` encoded as

```
0.942915444848::drc('hgnc_983','pubmed_11749053').
0.492799999825::drc('pubmed_10075692','hgnc_620').
0.434774330065::drc('hgnc_620','pubmed_10460257').
.
.
.
```

The same evidence used for Problog is processed to work with the syntax of LPMLN2ASP as

```
0.942915444848 drc('hgnc_983','pubmed_11749053').
0.492799999825 drc('pubmed_10075692','hgnc_620').
0.434774330065 drc('hgnc_620','pubmed_10460257').
.
.
.
```

We test the systems on varying graph sizes ranging from 366 nodes 363 edges to 5646 nodes 64579 edges. The experiment was run on a 40 core Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz machine with 128 GB of RAM. The timeout for the experiment was set to 20 minutes.

Nodes	**Edges**	LPMLN2ASP	PROBLOG2
366	363	0.37	0.152
1677	2086	9.77	1.7406

1982	4143	14	Timeout
2291	6528	19.71	Timeout
2588	9229	25.92	Timeout
2881	12248	33.05	Timeout
3168	15583	42.21	Timeout
3435	19204	49.91	Timeout
3724	23135	59.56	Timeout
3989	27370	69.72	Timeout
4252	31891	82.04	Timeout
4501	36690	93.23	Timeout
4750	41761	105.4	Timeout
4983	47094	116.79	Timeout
5200	52673	129.27	Timeout
5431	58506	142.2	Timeout
5646	64579	157.77	Timeout

Table 7.2: Problog2 vs LPMLN2ASP Comparison on Biomine Network (MAP Inference)

We perform MAP inference for comparison. Table 7.1 shows the results of the experiment. Apart from the smaller graph instances where Problog is faster than LPMLN2ASP, LPMLN2ASP significantly outperforms Problog for medium to large graphs for MAP inference. In fact, for graphs with nodes greater than 1980 Problog times out. For Marginal inference, to check for the probability of path between two genes, LPMLN2ASP times out with just 25 nodes and therefore it is infeasible to experiment for marginal probability on LPMLN2ASP . The sampling based approach of Problog computes the probability of a path

from 'hgnc_983' to 'hgnc_620' in 13 seconds. This experiment shows 2that for MAP inference, our implementation far outperforms the current implementation of Problog while being significantly slower in computing Marginal and Conditional probabilities.

7.3 Social influence of smokers - Computing MLN using LPMLN2ASP

We use Example 15 used in Section 4.1 to compare the scalability of LPMLN2ASP for MAP inference on MLN encodings and compare with the MLN solvers ALCHEMY, TUFFY and ROCKIT used in LPMLN2MLN. We scale the example by increasing the number of people and relationships among them.

The LPMLN2ASP encoding of the example used in the experiment is

```
1.1 cancer(X) :- smokes(X).
1.5 smokes(Y) :- smokes(X), influences(X, Y).
{smokes(X)} :- person(X).
{cancer(X)} :- person(X).
```

The ALCHEMY encoding of the example is

```
smokes(node)
influences(node,node)
cancer(node)


1.1 smokes(x) => cancer(x)
1.5 smokes(x) ^ influences(x,y) => smokes(y)
```

and is run with the command line [4]

```
infer -m -i input -e evidence -r output -q cancer -ow smokes,cancer
```

The TUFFY encoding of the example is [5]

[4] Option -ow is provided to alchemy to denote which predicates are under open world assumption.

[5] * makes the predicate closed world assumption

```
smokes(node)
*influences(node,node)
cancer(node)

1.1 smokes(x) => cancer(x)
1.5 smokes(x) , influences(x,y) => smokes(y)
```

and is run with the command line

```
java -jar tuffy.jar -i input -e evidence -r output -q cancer
```

The ROCKIT encoding of the example is [6]

```
smokes(node)
*influences(node,node)
cancer(node)

1.1 !smokes(x) v cancer(x)
1.5 !smokes(x) v !influences(x,y) v smokes(y)
```

and is run with the command line

```
java -jar rockit.jar -input input -data evidence -output output
```

The data was generated such that for each person p, the person *smokes* with an 80% probability, and p *influences* every other person with a 60% probability. We generate evidence instances based on different number of persons ranging from 10 to 1000. We compare the performance of the solvers based on the time it takes to compute the MAP estimate. The experiment was run on a 40 core Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz machine with 128 GB of RAM. The timeout for the experiment was set to 20 minutes.

[6] * makes the predicate closed world assumption

# of Persons	LPMLN2ASP **w.** CLINGO **4.5**	ALCHEMY **2.0**	TUFFY **0.3**	ROCKIT **0.5**
10	0	0.04	1.014	0.465
50	0.03	1.35	1.525	0.676
100	0.10	18.87	1.560	0.931
200	0.32	435.71	2.672	1.196
300	0.7	Timeout	4.054	1.660
400	1.070	Timeout	4.505	1.914
500	1.730	Timeout	5.935	2.380
600	2.760	Timeout	7.683	2.822
700	3.560	Timeout	10.390	3.274
800	4.72	Timeout	11.384	3.727
900	Timeout	Timeout	12.056	4.012
1000	Timeout	Timeout	12.958	4.678

Table 7.3: Performance of Solvers on MLN Program

Table 7.3 lists the computation time in seconds for each of the four solvers on instances of domains of varying size. LPMLN2ASP is the best performer for number of people till 600 but times out when number of people are greater than 800. ALCHEMY is the worst performer out of all 4 and for instances with number of people greater than 200 it times out. As expected, for ALCHEMY, grounding is the major bottleneck. For the instance with 200 persons, ALCHEMY grounds it in 422.85 seconds and only takes 9 seconds to compute the MAP estimate. Since the problem of grounding has been addressed in TUFFY and ROCKIT, these solvers are able to perform better than ALCHEMY. ROCKIT has the best results amongst all the solvers. This experiment shows that for small to medium sized instances, our implementation is the fastest and the most accurate solver while for larger instances ROCKIT is the fastest.

Chapter 8

SUMMARY AND CONCLUSION

8.1 Summary of the two $\mathrm{LP}^{\mathrm{MLN}}$ solvers

While both LPMLN2MLN and LPMLN2ASP can compute marginal/conditional probability and MAP inference on an $\mathrm{LP}^{\mathrm{MLN}}$ program, they are quite different from the context of the underlying solvers. LPMLN2ASP makes use of an ASP solver CLINGO while LPMLN2MLN makes use of MLN solvers ALCHEMY, TUFFY and ROCKIT. Both these systems treat the respective solvers as black-boxes and therefore the performance of the system is very much dependent on the underlying solvers. LPMLN2ASP can compute the full fragment of $\mathrm{LP}^{\mathrm{MLN}}$ programs unlike LPMLN2MLN which can compute only the tight fragments. Therefore $\mathrm{LP}^{\mathrm{MLN}}$ programs with inductive definitions can be run only on LPMLN2ASP. The below table summarizes some of the key differences of these two systems.

LPMLN2ASP computes exact results which are easy to understand and provides the gold-standard results, however, this affects the scalability of the system when calculating marginal and conditional probabilities. The system relies on enumerating *all* stable models of a program to compute the marginal and conditional probabilities. For MAP inference, not all the stable models needs to be enumerated and as a result MAP inference is almost always faster than probability calculation. What is interesting to note here is that the MAP estimate of an $\mathrm{LP}^{\mathrm{MLN}}$ program is directly associated with the optimal answer set computation using the weak constraint semantics of ASP, and also how the newly added feature of CLINGO 4 which exposes the CLINGO internals through a PYTHON library can be used to aid probability computation in an ASP solver.

Solver	Input Program	Inference Type	Aggregates	Probability Computation Scalability
LPMLN2ASP	Tight & Non-tight	Exact	Available	No
LPMLN2MLN **w.** ALCHEMY	Tight	Approximate	Not Available	No
LPMLN2MLN **w.** TUFFY	Tight	Approximate	Not Available	Yes
LPMLN2MLN **w.** ROCKIT	Tight	Approximate/Exact	Not Available	Yes

Table 8.1: Comparison Between LPMLN2ASP and LPMLN2MLN

LPMLN2MLN computes approximate results because all the underlying MLN solvers use approximate sampling-based inference methods. Use of approximate methods makes this implementation highly scalable when compared to LPMLN2ASP. All the MLN solvers trade accuracy for faster computation. Accuracy can be increased in ALCHEMY, TUFFY and ROCKIT by adjusting certain parameters that are available for the respective systems like `maxSteps` for ALCHEMY, `mcsatSamples` for TUFFY and `gap` for ROCKIT.

Another factor that aids in the scalability of these systems is the *grounding* of input programs. Although grounding in MLN solvers is naive, they do not need to ground the entire network. An essential part of the Markov networks relevant to the query can be constructed from the Markov blankets. In contrast, LPMLN2ASP needs to ground the entire program before it can begin computing stable models. However, CLINGO's grounder is very efficient at grounding and the grounding time with LPMLN2ASP is negligible when compared to probability computation time. One more thing to factor in while considering scalability is the architecture of LPMLN2ASP. In LPMLN2ASP system, every time a stable

model is computed by CLINGO, the control is handed over to the probability computation module. This is an interrupting call and adds up to the overall running time of computation on domains with a large number of stable models. Solvers TUFFY and ROCKIT makes use of RDBMS's POSTGRES and MYSQL respectively for grounding of Markov networks by executing a series of SQL calls. For grounding, ALCHEMY grounds everything in-memory as opposed to database approaches taken by TUFFY and ROCKIT. TUFFY also employs bottom up grounding to take advantage of the relational optimizer used by POSTGRES.

LPMLN2ASP adapts the syntax of CLINGO which makes it easier to introduce probability into ASP programs by just appending weights to existing rules. Since any valid and a safe CLINGO rule can be converted to a weighted $\mathrm{LP}^{\mathrm{MLN}}$ rule, ASP constructs like aggregates, python/lua code can be used in LPMLN2ASP program without any issues. On the other hand, LPMLN2MLN defines its own syntax which is based on ALCHEMY's first-order logic syntax written in the style of logic programs. Users familiar with ALCHEMY syntax can easily write programs in LPMLN2MLN. However, it does not support constructs like aggregates, or even simple constructs like basic arithmetic operations on integer variables. For example a rule of the form

$$a(X) \leftarrow b(Y), X = Y - 1$$

cannot be expressed in LPMLN2MLN but can be easily expressed in LPMLN2ASP .

LPMLN2MLN internally uses three different solvers ALCHEMY, TUFFY and ROCKIT. While each of them is an MLN solver, there is a lot of difference in the solvers' input syntax, internal architecture and computation algorithms. ALCHEMY is the only solver that supports the full first-order logic syntax in the input. TUFFY's language syntax derives from ALCHEMY'S language syntax, however, it supports a smaller fragment of the ALCHEMY language syntax. ROCKIT's language syntax derives from ALCHEMY, however, it supports a smaller set of logic operators and requires users to write formulas in its CNF form. TUFFY

uses a hybrid architecture which allows it to perform stochastic search in the database. TUFFY also uses parallel algorithms to optimize the process of MAP inference. ROCKIT uses *Integer linear Programming* solver GUROBI for computation. This makes ROCKIT's implementation completely different from that of TUFFY and ALCHEMY. ROCKIT has an advantage that GUROBI is the fastest and the most efficient available commercial ILP solver according to various benchmarks. Therefore, the performance of ROCKIT in terms of time and accuracy is the best amongst all the solvers.

8.2 Conclusion

We presented two implementations of $\mathrm{LP}^{\mathrm{MLN}}$ using ASP and MLN solvers. System LPMLN2ASP translates non-ground weighted $\mathrm{LP}^{\mathrm{MLN}}$ rules to non-ground ASP rules using weak constraints. System LPMLN2MLN translates $\mathrm{LP}^{\mathrm{MLN}}$ rules to weighted first-order formulas. Both the systems, LPMLN2ASP and LPMLN2MLN, are based on extending translations that turn $\mathrm{LP}^{\mathrm{MLN}}$ into answer set programs and Markov logic respectively.

We used LPMLN2ASP to translate $\mathrm{LP}^{\mathrm{MLN}}$ programs to ASP programs with weak constraints to perform MAP inference on $\mathrm{LP}^{\mathrm{MLN}}$ programs and added a probability computation module to compute marginal and conditional probability. While CLINGO does not have a built in notion of probability, using optimal answer set finding algorithm that uses weak constraints, ASP can be used for MAP inference, and using stable models enumeration we can effectively compute exact marginal and conditional probabilities of query atoms. For LPMLN2MLN, we introduce an input language which is adapted from the input language of ALCHEMY. We perform completion on the input $\mathrm{LP}^{\mathrm{MLN}}$ program to translate it to an MLN programs and use a simplified version of Tseitin's transformation for a compact encoding of the problem.

We showed that both these implementations can be used to compute other formalisms such as Pearl's Causal models, Bayes Net, Problog, and P-log by translating these for-

malisms to $\mathrm{LP}^{\mathrm{MLN}}$. We showed that ASP solvers like CLINGO can be used to compute MLN as well. We also showed how LPMLN2ASP can be used to resolve inconsistencies in an ASP program by translating hard rules.

The two $\mathrm{LP}^{\mathrm{MLN}}$ implementations show the contrasting properties of the two approaches to solving LPMLN programs and also how different implementations of MLN solvers affect the performance of computation of $\mathrm{LP}^{\mathrm{MLN}}$ programs. While LPMLN2ASP gives exact "gold standard" results it cannot actually scale well for marginal and conditional probability computation. On the other hand, MAP computation performance is comparable to the best performing MLN solvers for small to medium sized domains. The three MLN solvers used in LPMLN2MLN have different characteristics as well. Solver ROCKIT'S usage of GUROBI shows how integer linear programming tools can be used to solve $\mathrm{LP}^{\mathrm{MLN}}$ problems much faster and with a better accuracy than traditionally available tools. The optimizations and improvements in the underlying solvers directly helps the two $\mathrm{LP}^{\mathrm{MLN}}$ solvers and by extension helps in computing the other formalisms as well.

The two $\mathrm{LP}^{\mathrm{MLN}}$ implementations, however, do not have any native grounding and solving capacities and rely on other implementations for both. This is in contrast with the solvers of the other formalisms discussed such as ProbLog, P-Log and MLN which have their own native inference and learning algorithms. $\mathrm{LP}^{\mathrm{MLN}}$ systems also do not support weight learning of $\mathrm{LP}^{\mathrm{MLN}}$ rules. The native solvers of these formalisms are relatively mature softwares whereas LPMLN2ASP and LPMLN2MLN are both prototype systems. Future work includes building native grounding, solving and weight learning algorithms for $\mathrm{LP}^{\mathrm{MLN}}$ borrowing the techniques from the related systems.

REFERENCES

Baral, C., M. Gelfond and J. N. Rushton, "Probabilistic reasoning with answer sets", TPLP **9**, 1, 57–144 (2009a).

Baral, C., M. Gelfond and J. N. Rushton, "Probabilistic reasoning with answer sets", TPLP **9**, 1, 57–144 (2009b).

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

Calimeri, F., W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca and T. Schaub, "Asp-core-2: Input language format", ASP Standardization Working Group, Tech. Rep (2012).

Calimeri, F., W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca and T. Schaub, "Asp-core-2 input language format", (2013).

Clark, K., "Negation as failure", in "Logic and Data Bases", edited by H. Gallaire and J. Minker, pp. 293–322 (Plenum Press, New York, 1978).

De Raedt, L., A. Kimmig and H. Toivonen, "ProbLog: A probabilistic prolog and its application in link discovery.", in "IJCAI", vol. 7, pp. 2462–2467 (2007).

Erdem, E. and V. Lifschitz, "Tight logic programs", Theory and Practice of Logic Programming **3**, 499–518 (2003).

Eronen, L. and H. Toivonen, "Biomine: predicting links between biological entities using network models of heterogeneous databases", BMC bioinformatics **13**, 1, 119 (2012).

Fages, F., "Consistency of Clark's completion and existence of stable models", Journal of Methods of Logic in Computer Science **1**, 51–60 (1994).

Fierens, D., G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens and L. De Raedt, "Inference and learning in probabilistic logic programs using weighted boolean formulas", Theory and Practice of Logic Programming pp. 1–44 (2013).

Gebser, M., A. Harrison, R. Kaminski, V. Lifschitz and T. Schaub, "Abstract gringo", Theory and Practice of Logic Programming **15**, 4-5, 449–463 (2015).

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

Gelfond, M. and V. Lifschitz, "The stable model semantics for logic programming", in "Proceedings of International Logic Programming Conference and Symposium", edited by R. Kowalski and K. Bowen, pp. 1070–1080 (MIT Press, 1988).

Gutmann, B., *On continuous distributions and parameter estimation in probabilistic logic programs*, Ph.D. thesis, Ph. D thesis, KULeuven (2011).

Lee, J. and V. Lifschitz, "Describing additive fluents in action language $\mathcal{C}$+", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", pp. 1079–1084 (2003).

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

Lee, J., S. Talsania and Y. Wang, "Computing LPMLN using ASP and MLN solvers", Unpublished (2017).

Lee, J. and Y. Wang, "A probabilistic extension of the stable model semantics", in "International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2015 Spring Symposium Series", (2015).

Lee, J. and Y. Wang, "Weighted rules under the stable model semantics", in "Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)", (2016a).

Lee, J. and Y. Wang, "Weighted rules under the stable model semantics", in "Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)", (2016b).

Lee, J. and Z. Yang, "Lp mln, weak constraints, and p-log", (2017a).

Lee, J. and Z. Yang, "LPMLN, weak constraints, and p-log", in "Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)", (2017b).

Lin, F. and J. Zhao, "On tight logic programs and yet another translation from normal logic programs to propositional logic", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", pp. 853–864 (2003).

Lin, F. and Y. Zhao, "ASSAT: Computing answer sets of a logic program by SAT solvers", Artificial Intelligence **157**, 115–137 (2004).

Mantadelis, T., D. Shterionov and G. Janssens, "Compacting boolean formulae for inference in probabilistic logic programming", in "International Conference on Logic Programming and Nonmonotonic Reasoning", pp. 425–438 (Springer, 2015).

Niu, F., C. Ré, A. Doan and J. Shavlik, "Tuffy: Scaling up statistical inference in markov logic networks using an rdbms", Proceedings of the VLDB Endowment **4**, 6, 373–384 (2011).

Noessner, J., M. Niepert and H. Stuckenschmidt, "Rockit: Exploiting parallelism and symmetry for map inference in statistical relational models", arXiv preprint arXiv:1304.4379 (2013).

Pearl, J., *Causality: models, reasoning and inference*, vol. 29 (Cambridge Univ Press, 2000).

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

Riedel, S., "Improving the accuracy and efficiency of map inference for markov logic", arXiv preprint arXiv:1206.3282 (2012).

Sang, T., P. Beame and H. Kautz, "Solving bayesian networks by weighted model counting", in "Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)", vol. 1, pp. 475–482 (2005).

Sevon, P., L. Eronen, P. Hintsanen, K. Kulovesi and H. Toivonen, "Link discovery in graphs derived from biological databases", in "International Workshop on Data Integration in the Life Sciences", pp. 35–49 (Springer, 2006).

Tseitin, G., "On the complexity of derivation in the propositional calculus", Studies in Constructive Mathematics and Mathematical Logic **Part II** (1968).

Vennekens, J., S. Verbaeten and M. Bruynooghe, "Logic programs with annotated disjunctions", in "International Conference on Logic Programming", pp. 431–445 (Springer, 2004).