IDENTIFYING INFLUENTIAL INPUTS IN PROBABILISTIC LOGIC PROGRAMMING

A Thesis
submitted to the Faculty of the
Graduate School of Arts and Sciences
of Georgetown University
in partial fulfillment of the requirements for the
degree of
Master of Science
in Computer Science

By

Jiachi Zhang, B.S.

Washington, DC
April 20, 2020
Identifying Influential Inputs in Probabilistic Logic Programming

Jiachi Zhang, B.S.

Thesis Advisor: Wenchao Zhou

Abstract

Combining probabilistic theory and statistic reasoning, probabilistic logic programming (PLP) has various applications on many data-driven problems. Notable examples include machine learning models such as probabilistic soft logic (PSL) and Markov Logic Network (MLN). Despite the prevalence of PLP, there are few tools for us to debug and analyze PLP programs. In our prior work, we developed a PLP program debugging system which we call P3 for ProbLog. In this thesis, we further extend the system to more general PLP programs where the rule weights represent not only probabilities but also parameters in Markov Random Fields. Our PLP analyzing systems are based on probabilistic provenance which records the derivations of queried tuples. Based on the probabilistic provenance, we provide 3 types of queries: explanation query for inference derivation, probability query for calculating conditional probabilities, and influence query for identifying influential inputs. We evaluate our system on several PLP cases, such as Smoke and Visual Question Answering (VQA), and the evaluation results demonstrate the effectiveness of our system.

Index words: Data Provenance, Probabilistic Logic Programming, Markov Logic Network
ACKNOWLEDGMENTS

I wish to express my appreciation to my advisor, Professor Wenchao Zhou who patiently guided and encouraged me to finish this thesis. I'm also grateful to Professor Lisa Singh and Professor Micah Sherr for joining my thesis committee and their helpful suggestions on this thesis.
# Table of Contents

## Chapter

1. Introduction ......................................................... 1

2. Background ......................................................... 5
   2.1 Probabilistic Logic Programming ............................. 5
   2.2 Data Provenance .............................................. 7
   2.3 Markov Logic Network ....................................... 9

3. Provenance Model .................................................. 15
   3.1 Provenance Format ........................................... 15
   3.2 Provenance Maintenance .................................... 22
   3.3 Provenance Query ........................................... 23

4. Provenance Analysis .............................................. 27
   4.1 Explanation Query .......................................... 28
   4.2 Probability Query .......................................... 29
   4.3 Influence Query ............................................ 35

5. Experimental Study ............................................... 37
   5.1 Smoke .......................................................... 37
   5.2 Visual Question Answering .................................. 44

6. Conclusion ......................................................... 50

REFERENCES .......................................................... 52
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example of Markov Random Field and its cliques</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of Markov Blanket of node D</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>An example of Smoke</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Provenance graph of $\text{know}(\text{&quot;Steve&quot;,&quot;Elena&quot;)}$</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Provenance graph of $\text{cancer}(1)$</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Provenance graph of $\text{cancer}(5)$</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Provenance graph of integrated Markov Logic Network</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Complete Markov Logic Network graph for a test case of Smoke</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Small Markov Logic Network graph from smoke cases</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Influence of $\text{smoke}(1)$ on $\text{smoke}(2)$</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Influence of $\text{friends}(1,2)$ on $\text{smoke}(2)$</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>An abstract Visual Question Answering provenance graph</td>
<td>45</td>
</tr>
<tr>
<td>5.6</td>
<td>Visual Question Answering test image with wrong answer</td>
<td>46</td>
</tr>
<tr>
<td>5.7</td>
<td>Visual Question Answering results</td>
<td>48</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Probability query results. .................................................. 39
5.2 Influence query results on smoke(5). ............................... 40
In many data intensive applications, there has been a paradigm shift towards probabilistic and statistical reasoning. In some cases, it is in support of programs that rely on probability distributions. Probabilistic reasoning is used as a basis to trade off performance and accuracy, when collecting and aggregating readings from sensors. In other cases, probabilistic reasoning is dictated by use of external libraries, typically involving programs that rely on outputs of machine learning libraries which are intrinsically probabilistic. Consequently, over the past few years, there is an emergence of *probabilistic logic programming* (PLP) languages. Many of these languages use database-style declarative conjunctive rules that are loosely based on Datalog semantics (i.e., first-order logic [23]) with extensions to handle probability. Such logic-based approach simulates a world where different predicates are connected with logic rules; the program can then draw certain conclusions with deductive reasoning [9]. Written as declarative conjunctive rules, PLP further extends the syntax and semantics to handle probabilities where both the input data and logic rules are probabilistic. Notable examples of PLP include SLP [15], PRISM [21] and ProbLog [18].

PLP has been used as many machine learning models, such as Probabilistic Soft Logic (PSL) [2] and Markov Logic Network (MLN) [20]. These machine learning models are commonly related to Probabilistic Graphical Model (PGM). PGMs are probabilistic models which consist of special graphs where the nodes in the graph are random variables. They can be classified into two types by the variable dependency:
directed PGM, such as Bayesian Network and undirected PGM, such as Markov Random Field (MRF) [11]. PLP programmers need to design different learning and inference algorithms for different applications.

While PLP plays an important role in many different areas, people have few tools to debug or get more insights from PLP programs. Most PLP programs only give users a conclusion, such as the probability of a queried tuple, however, we know little about how it is inferred or which are the deciding factors that lead to such conclusion. In order to collect the inference derivation, we explore the possibility of using data provenance for PLP program debugging.

Data provenance is an important feature in database systems. The data provenance is maintained to answer questions, such as how a tuple is derived and where it comes from [3]. It has been shown that data provenance can be applied in a variety of different areas, such as network and distributed systems. However, prior provenance work falls short in enabling debugging capabilities for PLP programs because they are geared primarily towards traditional relational databases. Researchers have applied data provenance in the field of probabilistic databases [19], where the proposed provenance model can handle probabilistic inputs. However, it does not work for PLP because the SQL operators in probabilistic database are still deterministic. This is unlike PLP programs where the rules (and hence the operators used for executing these rules) are probabilistic. Consequently, systems that support provenance in probabilistic databases do not work on queries or programs with uncertainty built into the algorithm rather than the underlying data.

In our prior work, we introduced a model and system called P3 (Provenance for Probabilistic logic Programs) [25], which enabled probabilistic provenance. The provenance model is probabilistic in that both the rule tuples and base tuples in the provenance are probabilistic. The probabilistic provenance can help users query
the derivation of an inferred tuple with all probabilities of base inputs as well as
probabilities of rule derivations. Based on the probabilistic provenance model, we
also showed that the collected provenance can be used to debug and analyze PLP
programs by querying the most important derivations and the most influential inputs.

In this thesis, we extend the system to handle more complex situations where the
rule weights correspond to different semantics; more concretely, these rule weights
encode parameters used in Markov Random Field and can potential vary in the range
\((-\infty, \infty)\). Given the change of rule weight semantics, we also need to adjust both the
provenance model and inference algorithms.

This thesis targets MLN as a concrete example: MLN is based on undirected
probabilistic graphical model MRF which consists of “cliques” which represent the
logical and statistical correlations between derived results and the inputs that are
used to derive them. In MLN, the weighted rules are converted into cliques and rule
weights vary in the range \((-\infty, \infty)\). The rule weights are used to compute clique
potentials, and given all the computed clique potentials, we can calculate the joint
probability of a grounded MLN graph and the conditional probabilities of a queried
tuples. In practice, MLN is widely regarded as an effective machine learning model.

In the following chapters, we present how we design the system based on MLN of
provenance generation, provenance parsing and provenance analysis to build a more
generalized version of P3.

The key contributions of this thesis include:

1) **Probabilistic provenance model.** Based on P3, we design a new form of
probabilistic provenance for MLN. Compared to previous provenance model,
this model is easier to read and more convenient to parse for MLN.
2) **Provenance visualization.** In order to give users more intuitive view of PLP models. We develop a visualization tool to parse the provenance and present the model in graphs. We provide 2 types of visualization graph. One is in derivation form which is used for ProbLog. Another is in MRF form which is used for MLN.

3) **Probabilistic inference methods.** Researchers have developed many different types of MLN inference algorithms, such as Gibbs sampling, MC-SAT and belief propagation [20][4]. However, the classic inference algorithms cannot handle probabilistic input data. Although the rules are soft with weights, the input data only have 2 states: true and false. In order to handle probabilistic input, we design probabilistic inference algorithms based on Gibss Sampling and MC-SAT.

4) **Influence Query.** In ProbLog cases, We define the literal influence as the partial derivative of success probability of literal. In MLN cases, the literal influence is also defined as the partial derivative of queried probability of input literal. The difference is that literal influence in ProbLog is not influenced by literal probability. However, the literal influence in MLN varies when the probability of literal changes. We will present how we compute the approximate literal influence then identify influential inputs in PLP programs.

In order to evaluate our system, we perform a set of case studies, consisting of both small cases (for demonstrating the correctness of probabilistic provenance and revealing mathematical intuitions), and large cases (for demonstrating the power of probabilistic provenance in solving complex, practical problems). The evaluation results are presented in Chapter 5.
In this chapter, we introduce three key fundamental concepts we use in this paper: PLP languages, data provenance and Markov logic network (MLN). We will present their basic concepts as background knowledge, and discuss how they are related to this thesis.

2.1 Probabilistic Logic Programming

Most of the PLP languages, such as ProbLog [18] and Datalog$_P$ [6] use Datalog-style syntax with semantic extension to probability. In P3, we use ProbLog syntax and semantics. The following program statements show the syntax of ProbLog:

\begin{verbatim}
rid p1: H() :- B1(), B2(), ..., Bn().
tid p2: B1().
\end{verbatim}

The statements in ProbLog can be classified into two types: rule statements and base tuple statements. In a rule statement, \texttt{rid} is the rule identifier, \texttt{p1} is the rule probability, \texttt{H()} is rule head and \texttt{B1()}, \texttt{B2()}, ..., \texttt{Bn()} are rule body tuples. The rule statement means if the rule body tuples \texttt{B1()}, \texttt{B2()}, ..., \texttt{Bn()} are all true, rule head \texttt{H()} is also true with probability \texttt{p1}. In the base tuple statement, \texttt{tid} is the tuple identifier, \texttt{p2} represents the independent probability of \texttt{B1()} being true.
2.1.1 Success probability

In ProbLog, some of the tuples, such as base tuples, have probabilities specified by users, while the probabilities of others, particularly derived tuples, depend on the base tuples and how they are derived. In such cases, we need to compute their success probability. The success probability is defined as the probability that a queried tuple can be derived given the observed probabilistic tuples. For example, in the rule statement above, assume that \( H() \) has only one derivation (through this very rule), and \( H() \) is derived directly from base tuples (i.e., the tuple in the rule body are all base tuples). The success probability of \( H() \) can be directly computed by multiplying all probabilities of rule body tuples and rule probability \( p_1 \), since we assume all base tuples are independent. However, when the tuple is derived from other derived tuples, its success probability is difficult to compute (as the tuples in the rule body may no longer be independent to each other); in fact, it involves evaluate the success probability of a boolean formula which is NP-hard.

2.1.2 Weighted Datalog rules

As a special case of PLP, ProbLog does not support cases where the rule weights do not represent probabilities and vary in range \((-\infty, \infty)\). Therefore we can not compute the success probability of derived tuples in ProbLog rule statements. Because of the change of language semantics, we call these statements weighted Datalog rules:

\[
\text{rid } w: H() :- B1(),B2(),...,Bn().
\]

\[
\text{tid } p: B1().
\]

The base tuple statement in weighted Datalog rules has the same syntax and semantics to ProbLog where \( p \) is the independent probability of \( B1() \). However, in rule statement, \( w \) is a float number which is not probability. Although the rule also means
if rule body tuples $B_1(), B_2(), \ldots, B_n()$ are all true, rule head tuple $H()$ becomes true, the rule weight cannot tell what the probability the rule is. The calculation of the probability of derived tuples relies on the semantics of the PLP. We discuss in Section 2.3 the probability calculation of MLN.

With weighted Datalog rules, we design the corresponding provenance model, probability and influence query algorithms and inference query algorithm on MLN. We will discuss them in following chapters.

2.2 Data Provenance

Data provenance, as known as data lineage, plays an important role in database systems. It is defined as a description of data origins and is designed to query where a piece of data comes from and how it arrives in the database [3]. Identifying data provenance is essential when we want to debug the database system or programs in other areas because data provenance records the history of program execution. Data provenance has been widely used in many declarative logic program debugging, such as Datalog [6], NDlog (Network Datalog) [12][13][14] and so on.

Traditionally, the provenance mainly has two forms: graph representation and algebraic representation, which is commonly encoded as provenance polynomials [7][27]. The provenance graph is an acyclic and unidirectional graph which records the dependency between different vertices. Provenance algebraic is also known as provenance polynomial. The provenance polynomials vary in their formats and semantics. People will build different provenance models when they use data provenance on different applications.

In our previous work P3, we wrote ProbLog-like programs and designed our provenance model both in graph representation and algebraic representation. In a prove-
nance graph $G(V,E)$, the vertices $V$ in the graph consist of rule vertices and tuple vertices. The directed edges $E$ represent dependency relationships between tuples which are derived from rules. For example, the provenance graph for a ProbLog rule $r_1 \ p_1 \ H() \ :- \ B_1(), B_2(), \ldots, B_n()$ consists of tuple vertices $H(), B_1(), \ldots, B_n()$ and rule vertex $r_1$. Rule head tuple $H()$ is a tuple vertex and it depends on all the rule vertices where $H()$ derives from, such as $r_1$. A rule vertex depends on its rule body tuples, such as $B_1(), B_2(), \ldots, B_n()$.

The provenance graph can be straightforwardly converted into its corresponding algebraic representation. It employs two binary operators: "+" which indicates disjunctive connection and "*" which indicates conjunctive connection in the provenance. Considering the ProbLog rule $r_1 \ p_1 \ H() \ :- \ B_1(), B_2(), \ldots, B_n()$, if rule head tuple $H()$ derives from different rules, the rule tuples are connected by disjunctive operator "+". This polynomial means if one of the derivation of $H()$ is true, $H()$ is true. On the other hand, the rule body tuples $B_1(), B_2(), \ldots, B_n()$ are connected by conjunctive operator "*". This polynomial means if all the rule body tuples are true, this rule is true. Although both operators are logic operators, the base tuples and rule tuples are probabilistic. We will discuss the semantics of provenance polynomial and show how to compute the success probability of provenance polynomial in Section 3.3.

In P3, the provenance of a queried tuple is encoded in algebraic form by traversing the provenance graph. The provenance polynomials become the basis for answering complex queries, such as computing the success probability of provenance, and finding the most influential derivations and the most influential base tuples.
2.3 Markov Logic Network

Markov Logic Network is a machine learning model which combines first-order logic with Markov random field (MRF) [20]. An MRF is an undirected probabilistic graphical model for joint probability of a set of random variables $X$. MRF consists of an undirected graph $G$ where each fully connected subgraph makes a clique and each clique has its potential function $\varphi$.

![Figure 2.1: An example of Markov Random Field and its cliques.](image)

Figure 2.1 gives an example of MRF and the cliques in example MRF. In this figure, an MRF consists of 5 nodes: A, B, C, D and E. It also includes three cliques: the first consists of node A, B and D; the second consists of node D and E, and the last consists of node C and E.

2.3.1 Cliques and joint probability

In MLN, researchers consider first-order logic sentences as cliques where each predicate is a node then assign weights to the cliques. Given the clique weights and random variable values, we can calculate clique $k$’s potential function $\varphi_k$ by

$$
\varphi_k(x_{\{k\}}) = \exp(w_k f(x_{\{k\}}))
$$

(2.1)
where $w_k$ is the clique weight, $f(x_{\{k\}})$ is a binary feature which indicates whether the clique can be satisfied by random variable states $x_{\{k\}}$, $f(x) \in \{0, 1\}$.

Combining all cliques, the joint probability of a grounding $x$ of a random variable $X$ is given by the following formula:

$$P(X = x) = \frac{1}{Z} \prod_k \varphi_k(x_{\{k\}})$$

(2.2)

where $Z$ is the summation of all possible groundings of $X$ and it is defined as

$$Z = \sum_x \prod_k \varphi_k(x_{\{k\}})$$

(2.3)

### 2.3.2 MLN inference

Researches on MLN mainly include MLN learning and inference. Given a branch of observed data, MLN learning algorithms try to maximize the likelihood of observed data. MLN inference tasks include inferring the most possible grounding of random variables, inferring the probabilities of unobserved variables given some observed data, etc [5]. In this paper, we focus on the conditional probability inference. In other word, given a queried tuple $q$ and a branch of observed data $D$, we want to query the conditional probability $P(q|D)$. The provenance can help us query both the unobserved and observed variables which are required for inference. Therefore, it becomes a powerful tool for the probability inference.

Traditional probability inference algorithms include Gibbs sampling [20], MC-SAT [17], belief propagation [22] and so on. Gibbs sampling is a classic inference algorithm which is based on Markov property and has been widely used in many machine learning models. In MLN, we assume that the probability of an queried variable only depends on its Markov Blanket (MB) and a MB only includes variables in the cliques which contain the queried variable [20].
Figure 2.2: An example of Markov Blanket of node D.

Figure 2.2 shows an example of a Markov Blanket of a node in an MRF. In this figure, there are two cliques which contain node D. The upper circle and the lower circle surround them separately. Therefore, the Markov Blanket of node D includes node A, B and E.

The conditional probability of variable $X_l$ given the states of its MB $B_l$ is defined by the following formula:

$$P(X_l = x_l|B_l = b_l) = \frac{\exp(\sum w_i f_i(X_l = x_l|B_l = b_l))}{\exp(\sum w_i f_i(X_l = 0|B_l = b_l)) + \exp(\sum w_i f_i(X_l = 1|B_l = b_l))}$$

(2.4)

where $f$ indicates whether the clique $i$, which includes the queried variable $X_l$, can be satisfied and $w_i$ is the weight of clique $i$ [20]. Based on this formula, Gibbs sampling is running iteratively until its convergence.

Similar to Gibbs sampling, MC-SAT is also an iterative sampling algorithm. The difference is that MC-SAT samples on satisfied cliques by the current states and uses
SampleSAT [26] to generate next states. For each grounded clique $c_k$, it is inserted into a sebset of cliques $M$ with probability $1 - e^{-w_k}$ if it can be satisfied by the previous states. Given a set of cliques $M$, SampleSAT algorithm helps us uniformly sample from all possible groundings of variables $SAT(M)$ which satisfy $M$. Compared to Gibbs sampling, MC-SAT is faster on large scale MRFs and is guaranteed to be sound [17]. Therefore, it is more preferred in many MLN implementations, such as Alchemy [20], Tuffy [16] and Slimshot [8].

Most MLN implementations only support binary inputs which only have two possible states: true and false. However, in many PLP cases, the inputs are probabilistic. Therefore, we implement probabilistic versions of Gibbs sampling and MC-SAT for the inference with probabilistic inputs. We will discuss this in Chapter 4.

![Figure 2.3: An example of Smoke.](image)

2.3.3 Running example

We select a MLN program Smoke from Alchemy as a running example. From the original MLN program, we choose two implication rules and rewrite them in first-
order logic sentences with rule identifiers and rule weights. From the input data, we select four tuples and rewrite them in base tuple statements of weighted Datalog rules. The MLN program is presented in the following statements:

\begin{align*}
\text{r1} & \quad 0.89: \text{smoke}(P) \implies \text{cancer}(P). \\
\text{r2} & \quad 0.88: \text{friends}(A,B) \implies (\text{smoke}(A) \iff \text{smoke}(B)). \\
\text{t1} & \quad 1.0: \text{smoke}(2). \\
\text{t2} & \quad 1.0: \text{friends}(1,2). \\
\text{t3} & \quad 1.0: \text{smoke}(3). \\
\text{t4} & \quad 1.0: \text{friends}(1,3).
\end{align*}

In this MLN program, Statements \text{r1} and \text{r2} are two rule statements. Rule \text{r1} means someone smokes is likely to have cancer. Rule \text{r2} means if two persons are friends, they are likely to have same smoking habit. Statements \text{t1}, \text{t2}, \text{t3} and \text{t4} are four tuple statements. In this program, we observe that person 2 and person 3 smoke, person 1 and person 2 are friends, and person 1 and person 3 are friends.

Figure 2.3 shows the MLN graph of this case. In this figure, blue circles indicate observed tuples which are from input data. Red circles are derived tuples from rule statements. In MLN, we convert every rule into a clique. For example, \text{cancer}(1) is derived from \text{smoke}(1) through \text{r1}. And they compose a clique and the clique weight is 0.89 which is the rule weight of \text{r1}. Similarly, \text{smoke}(1), \text{smoke}(2) and \text{friends}(1,2) are from rule \text{r2} and they compose a clique. Its clique weight is 0.88 which is from \text{r2}. Based on the MLN graph and cliques built from rules, we can compute the joint probability of any ground MLN graph and infer the conditional probability of any unobserved tuples.
In this paper, we also use Smoke as the test case for our PLP program in weighted Datalog rules. We will discuss how we rewrite this MLN program in weighted Datalog rules in the next chapter.
Chapter 3

Provenance Model

Chapter 2 describes the provenance model in P3. In this paper, our provenance model uses the same graph representation with P3 but differs in format and semantics of provenance polynomials. In this chapter, we will first present how we change the format of provenance polynomial for MLN with the help of visualization tools. Then we will describe our provenance maintenance and querying method in addition to cycle handling method.

3.1 Provenance Format

Given a provenance graph $G(V, E)$ and queried tuple, we want to build a provenance polynomial in string which consists of based tuples, rule tuples and operators. The expected provenance format should be consistent to the PLP language’s semantics and easy to parse. This section will use examples to show the formats of our provenance polynomials both for ProbLog and MLN programs.

3.1.1 Provenance for ProbLog

The following PLP program is called Simple-acquaintances which is from PSL examples. We rewrite the program in ProbLog.

r1 0.8: know(P1,P2) :- live(P1,C), live(P2,C), P1!=P2.
r2 0.4: know(P1,P2) :- like(P1,L), like(P2,L), P1!=P2.

15
In Simple-acquaintances, there are three rule statements and four tuple statements. Rule \( r_1 \) means if two persons live in the same place, they are likely to know each other. Rule \( r_2 \) means if two persons have same hobby, they are likely to know each other. Rule \( r_3 \) is an acquaintance transition rule which means if person 1 knows person 2 and person 2 knows person 3, person 1 is likely to know person 3.

Tuple statements \( t_1, t_2, t_3 \) and \( t_4 \) give a concrete scenario of this test case. Tuple \( \text{live("Steve","DC")} \) means Steve lives in DC. Tuple \( \text{live("Elena","DC")} \) means Elena lives in DC. Tuple \( \text{like("Steve","Veggies")} \) means Steve likes veggies. Finally, tuple \( \text{like("Elena","Veggies")} \) means Elena likes veggies.

Given the base tuples, the queried provenance of \( \text{know("Steve", "Elena")} \) should contains all the base tuple which directly or indirectly derives the queried tuple. The base tuples and rule tuples are connected by boolean operators, the operator "*" indicates conjunctive connection of tuples and "+" indicates disjunctive connection of derivations. The provenance of \( \text{know("Steve", "Elena")} \) is

\[
\text{r1(live("Steve","DC")*live("Elena","DC")+}
\]

\[
\text{r2(like("Steve","Veggies")*like("Elena","Veggies"))}
\]

This provenance explains that the queried tuple \( \text{know("Steve", "Elena")} \) has two derivations. One is from rule \( r_1 \) and another is from rule \( r_2 \). These two derivations are connected by disjunctive operator "+" and each derivation has its conjunctive base tuples of rule body.
We developed a visualization tool for ProbLog programs in derivation form which consists of a directed graph showing all the derivation of queried tuple. The visualized graph is shown in Figure 3.1. In this figure, red circles indicate rule tuples, blue boxes indicate based tuples and derived tuples. Clearly, there are two rule vertices r1 and r2 pointing to the queried tuple vertex \texttt{know("Steve","Elena")} and they represent the queried tuple’s two derivations. Each rule vertex also has two base tuple vertices pointing to itself and the base tuple vertices are the rule body tuples of the rule.

3.1.2 Provenance for MLN

In this paper, we design the provenance for MLN. And we will explain it using case Smoke which is from Alchemy. We present the original MLN program as a running example in Section 2.3. The following program shows the rewritten statements in weighted Datalog rules which we mentioned in Section 2.1:

\begin{verbatim}
r1 0.89: cancer(P) :- smoke(P).
r2 0.44: smoke(A) :- friends(A,B), smoke(B), A!=B.
r3 0.44: smoke(B) :- friends(A,B), smoke(A), A!=B.
\end{verbatim}
t1 1.0: smoke(2).
t2 1.0: friends(1,2).
t3 1.0: smoke(3).
t4 1.0: friends(1,3).

There are two types of statements in this program: rule statements and tuple statements. Statements \( r_1, r_2 \) and \( r_3 \) are the rule statements. Rule \( r_1 \) means someone smokes is likely to have cancer. Rule \( r_2 \) means if two different persons A and B are friends and person B smokes, then person A is likely to smoke. Rule \( r_3 \) means if two different persons A and B are friends and person A smokes, then person B is likely to smoke. Although the rule weights in this program are within \([0, 1]\), they do not represent probabilities. In this thesis, we convert each weighted Datalog rule into MLN cliques and interpret rule weight as clique weight.

Statements \( t_1, t_2, t_3 \) and \( t_4 \) are four tuple statements which are from input data set. Statement \( t_1 \) means person 2 smokes. Statement \( t_2 \) means person 1 and person 2 are friends. Statement \( t_3 \) means person 3 smokes. Finally, statement \( t_4 \) means person 1 and person 3 are friends. Probabilities of these four independent tuples are all 1.

Different to ProbLog, we use the provenance to build the MLN graph for a queried tuple. Therefore, the provenance should contain all tuples in the queried tuple’s Markov Blanket. In other word, the provenance must contain both rule body tuples and derived tuples to build MLN graph. Since the provenance of ProbLog only contains based tuples, we need to design a new form of provenance for MLN.

In our new provenance model for MLN, the provenance consists of base tuples, rule tuples, operator and derived tuples. The base tuples, rule tuples and operators play a similar role to ProbLog. Derived tuples are the rule head tuples and they are
always ignored by ProbLog’s provenance model. For example, if we try to query the provenance of cancer(1). The provenance we design is

\[
\text{cancer(1)}(\text{r1(smoke(1)}(\text{r2(smoke(2)*friends(1,2))}+\text{r2(smoke(3)*friends(1,3))}))\]

In this provenance, both tuples of rule head and rule body are included and the rule head and rule body are connected by rule tuples. For example, the intermediate derived tuple smoke(1) is included in the provenance. Furthermore, the queried tuple cancer(1) is also a derived tuple and it appears in the provenance. Tuples of rule head and rule body compose cliques and the rule tuples are related to the clique weights. In this format, the provenance can be easily parsed into MLN by converting the provenance string into a clique table.

We also developed a visualization tool drawing undirected MRF graph for MLN. The graph which visualizes the provenance above is shown in Figure 3.2. In this

**Figure 3.2:** Provenance graph of cancer(1).
figure, blue circles indicate observed tuples from input data and red circle indicate
the unobserved tuples which are derived from observed tuples and weighted Datalog
rules. Although the provenance graph is directed, we present the provenance in an
MLN graph which is undirected. We will discuss the semantics of this provenance
polynomial and MLN graph in Section 3.3.

3.1.3 Bidirectional Implication

In the original MLN program in Section 2.3, there are only tow rules:

r1 0.89: smoke(P) => cancer(P).

r2 0.88: friends(A, B) => (smoke(A) <=> smoke(B)).

However, in our PLP program in weighted Datalog rules, we write three rules:

r1 0.89: cancer(P) :- smoke(P).

r2 0.44: smoke(A) :- friends(A,B), smoke(B), A!=B.

r3 0.44: smoke(B) :- friends(A,B), smoke(A), A!=B.

In these two programs, the first rules are the same. The difference is that we
rewrite r2 in MLN program into two rules in weighted Datalog rules. The reason we
do this is that r2 in MLN program contains bidirectional implication operator and
PLP cannot handle this operator. Therefore, we divide this rule into two rules which
only contains implication operator and divide the original weight to the divided rules.
We also prove that the original rule is equivalent to the divided rules.

In this paper, the PLP rules we write are all implication sentence in first-order
logic. For example, given an implication sentence

friends(A, B) ∧ smoke(B) ⇒ smoke(A)  (3.1)
which means that if person A and person B are friends and B smokes, A smokes, we rewrite it in PLP language with extension to rule weight:

\[ r2 \ 0.44: \ smoke(A) :- \ friends(A,B), \ smoke(B), \ A!B. \]

In the case of Smoke, the original first-order logic sentence is

\[ friends(A,B) \Rightarrow (\ smoke(A) \Leftrightarrow \ smoke(B)) \]  \hspace{1cm} (3.2)

which means if person A and person B are friends, they have same smoking habits. And it is written in rule statement:

\[ r2 \ 0.88: \ friends(A, B) \Rightarrow (\ smoke(A) \Leftrightarrow \ smoke(B)). \]

In MLN, every first-order logic sentence is converted into a clause. And the literals in the clause compose a clique. Therefore, Boolean operator, such as bidirectional implication does not impact on MLN graph building. However, since PLP rules are in pattern of rule head and rule body, we can not rewrite the bidirectional implication sentence directly.

In order to solve this problem, we first convert formula 3.2 into CNF:

\[ (!friends(A,B) \lor \neg smoke(B) \lor \ smoke(A)) \land (!friends(A,B) \lor \neg smoke(A) \lor \ smoke(B)) \]  \hspace{1cm} (3.3)

which is equivalent to

\[ (friends(A,B) \land \ smoke(B) \Rightarrow \ smoke(A)) \land (friends(A,B) \land \ smoke(A) \Rightarrow \ smoke(B)) \]  \hspace{1cm} (3.4)

In this way, we can rewrite the bidirectional implication sentence into two divided implication sentences

\[ friends(A,B) \land \ smoke(B) \Rightarrow \ smoke(A) \]  \hspace{1cm} (3.5)

\[ friends(A,B) \land \ smoke(A) \Rightarrow \ smoke(B) \]  \hspace{1cm} (3.6)
In weighted Datalog rules, these two implication sentences are written in two rules

\[ r2 \ 0.44: \ smoke(A) :- \ friends(A,B), \ smoke(B), \ A\neq B. \]

\[ r3 \ 0.44: \ smoke(B) :- \ friends(A,B), \ smoke(A), \ A\neq B. \]

After the rule rewriting, another question is that what the rule weights of divided rules should be. Assuming that the rule weight of original bidirectional rule is \( w \) and rule weights of divided rules are \( w_1 \) and \( w_2 \). Considering the potential function 2.1, if the original statement 3.2 is satisfied, its potential should be \( \exp(w) \) and the potential of the two divided statements 3.5 and 3.6 should be \( \exp(w_1 + w_2) \). On the other hand, if the original rule is not satisfied, two divided rule are neither satisfied and their potentials are both \( \exp(0) \). Therefore, we only need to ensure that

\[ w = w_1 + w_2 \tag{3.7} \]

to make the divided rules equivalent to the original bidirectional rule both in first-order logic and MLN. In practice, we simply let \( w_1 = w_2 \). Then the rule weights of \( r2 \) and \( r3 \) in weighted Datalog rules are both 0.44 while the rule weight of original bidirectional rule is 0.88.

### 3.2 Provenance Maintenance

In order to maintain the provenance, we perform an automatic rule rewrite of PLP program to record the direct dependencies of tuples and rule executions. More specifically, given a rule

\[ \text{rid} \ \text{w}: \ H() :- \ B1(),B2(),...,Bn(). \]

we rewrite this rule into three rules:

\[ H() :- \ B1(),...,Bn(). \]

\[ \text{prov}(H(),w,\text{rid}) :- \ B1(),...,Bn(). \]

\[ \text{rule}(\text{rid},(B1(),...,Bn())) :- \ B1(),...,Bn(). \]
The first rule is the original rule which will generate rule head tuple given rule body tuples. The second rule records the dependency between rule execution and rule head. Rule execution includes information about the rule, such as rule identifiers and rule weights. The last rule records dependency between rule execution with rule body. By using the second and the third rules, the system builds the provenance graph by connecting rule body, the rule head that derived from the rule body, and the rule execution that enables the derivation of the rule head.

Although the rule weights appear in provenance graph, it is not necessary for the provenance polynomial to include rule weights. The reason is that we can maintain an additional map from rule identifiers to rule weights and use it when we parse the provenance polynomial.

During the runtime of PLP program, the provenance will be automatically maintained in provenance graph. Compared to the original PLP program, provenance maintenance adds reasonable constant overhead for each rule and have limited influence on the program.

3.3 Provenance Query

Provenance query builds a provenance polynomial by traversing the provenance graph given a queried tuple. We can traverse the graph following edges starting from the queried tuple until we reach the base tuples.

3.3.1 Provenance Polynomial

In ProbLog, the provenance polynomial is designed to query the success probability of the queried tuple \( q \). We consider the provenance polynomial as a Boolean formula \( \lambda(q) \) which consists of base tuples, rule executions and boolean operators that connect
them. In the provenance polynomial, operator "*" indicates conjunctive connection and "+" indicates disjunctive connection. For example, given a provenance:

\[
\text{r1(live("Steve", "DC")*live("Elena", "DC")+}
\]

\[
\text{r2(like("Steve", "Veggies")*like("Elena", "Veggies"))}
\]

we expand it by adding conjunctive operators after rule tuples:

\[
\text{r1*live("Steve", "DC")*live("Elena", "DC")+}
\]

\[
\text{r2*like("Steve", "Veggies")*like("Elena", "Veggies")}
\]

Since we assume that base tuples are independent, we can compute the success probability of tuples which are connected by operator "+" by calculating the product of their probabilities. However, the success probability of polynomials which are connected by operator "+" is not the summation of each derivations’ success probabilities. In fact, computing a success probability of arbitrary Boolean polynomial is an NP-hard problem [24]. Therefore, we use Monte-Carlo simulation [10] to compute the approximate probability values.

In MLN, the provenance polynomial is not only designed to predict the probability of queried tuple \( q \), but is also used to build the MLN graph which may contain derived tuples. For example, in provenance

\[
\text{cancer(1)(}
\]

\[
\text{r1(smoke(1)(}
\]

\[
\text{(r2(smoke(2)*friends(1,2)))+}
\]

\[
\text{(r2(smoke(3)*friends(1,3))))},
\]

the first tuple cancer(1) and second tuple smoke(1) are the derived tuples. Others are base tuples.

The provenance polynomial for MLN consists of rule executions, base tuples, derived tuples and boolean operators. Rule executions perform similarly to operators
and it connects rule head tuple and rule body tuples. For example, in the provenance above, \( r_1 \) is the rule tuple which connects rule head tuple \( \text{cancer}(1) \) and rule body tuple \( \text{smoke}(1) \) while \( \text{smoke}(1) \) is the rule head tuple of another rule. Operator "*" connects rule body tuples and operator "+" connects rule executions that contribute to alternative derivations of a derived tuple. In this way, the provenance will be converted into a table of cliques where each clique contains rule head tuple, rule body tuples and the corresponding rule execution. For example, we can extract three rule-based cliques from the provenance above:

\[
\begin{align*}
\text{cancer}(1), r_1, \text{smoke}(1) \\
\text{smoke}(1), r_2, \text{smoke}(2), \text{friends}(1,2) \\
\text{smoke}(1), r_2, \text{smoke}(3), \text{friends}(1,3)
\end{align*}
\]

then these three cliques compose the MLN graph of queried tuple \( \text{cancer}(1) \).

### 3.3.2 Singular Cliques

In addition to rule-based cliques which are built from provenance, MLN also includes singular cliques where each contains only one tuple. For example, \( \text{cancer}(1) \) makes up a singular clique and this clique also has a weight which is bound with the predicate name "cancer". The singular cliques play an important role; it is similar to prior probability. For example, given an arbitrary person (within the network), we can calculate an evaluation on how probable the person contracts cancer. Although we do not include them in running examples, the singular cliques are also included in the clique table when it is running in practice.

### 3.3.3 Handling Cycles

The provenance graph in ProbLog cases and MLN cases are both acyclic and unidirectional. However, when the PLP program contains recursive rules such as
r3 0.2: know(P1, P3) :- know(P1, P2), know(P2, P3), P1 ≠ P3.

from Simple-acquaintance case, and

r2 0.44: smoke(A) :- friends(A, B), smoke(B), A ≠ B.

r3 0.44: smoke(B) :- friends(A, B), smoke(A), A ≠ B.

from Smoke case, same predicate appears both in rule head and rule body, such as
predicate "know" in the first case and predicate "smoke" in the second case. When a
PLP program has such recursive rules, the queried tuple may appear in its derivation
in the provenance graph. Since we assume the graph is acyclic, traversing the cyclic
graph will cause endless querying because it cannot reach base tuples.

In order to avoid endless querying, we maintain a traversing path which records
the visited tuples in order while traversing the provenance graph. When we find that
the current tuple is in the traversing path, we stop searching and consider the current
node as base tuple. In this way, we can generate a complete provenance even though
it is sometimes redundant.

In ProbLog cases, we proved that removing the cyclic subgraph does not influence
the success probability of queried tuple [25]. Therefore, we prune the derivations which
contains cycles to simplify the provenance after provenance query.

In MLN cases, we use the same provenance query algorithm to handle cycles.
However, since the provenance may contain duplicate tuples, we check for duplicate
cliques whenever we insert a new clique in MLN’s clique table.
The previous chapter describes the provenance format and how we build the provenance. In this chapter, we will focus on analyzing and debugging programs written in MLN, which serves as a special case of PLP programs. Given a MLN program, users may want to know what the queried tuple’s MLN graph looks like, what the probability of a queried tuple is and which base tuple has the most influence on the queried tuple. In this thesis, we provide three types of queries: explanation query, probability query and influence query for PLP analyzing and debugging.

Figure 4.1: Provenance graph of cancer(5).
4.1 Explanation Query

The explanation query returns the derivations of the queried tuple in a PLP program. It shows how an unobserved tuple is derived through weighted Datalog rules we defined. For example, Figure 4.1 depicts an example provenance graph. In this example provenance, the queried tuple cancer(5) is derived from tuple smoke(5). smoke(5), in turn, has two derivations of smoke(6), friends(5,6) and smoke(1), friends(1,5). However, the explanation results cannot represent the complete MLN graph. Moreover, sometimes the explanation result is not sufficient for accurate probability inference. We consider a subgraph is sufficient for probability inference only when it contains all tuples and cliques which Gibbs sampling and MC-SAT require. For example, in Figure 4.1, the probability inference of smoke(2) is not accurate because this graph lacks a tuple cancer(2) and the clique of smoke(2), cancer(2). We can only query the derivations of a queried tuple while traversing the provenance graph; cancer(2) is derived from the queried tuple smoke(2) and therefore it is not included in the provenance.

Graph Integration. The simplest method to make the MLN graph sufficient for all queried tuples is building the complete MLN graph because the complete MLN graph contains all tuples. In order to build the complete MLN graph, we need all the observed tuples and unobserved tuples which are derived from observed tuples. Therefore, we query the provenances of all unobserved tuples, parse the provenance and delete duplicate cliques then integrate all subgraphs parsed from these provenances. And we call this method graph integration. The integrated graph is shown in Figure 4.2. We can see that the graph in Figure 4.1 is a subgraph of the integrated graph. Obviously the integrated graph is sufficient for probability inference of any unobserved tuples because it contains all tuples in a case.
4.2 Probability Query

Given the explanation query results, we want to predict the conditional probability $P(q|D)$ where $q$ is the queried tuple and $D$ is the input data which contains both probabilities of observed tuples and weights of cliques.

In this paper, our probability query algorithm is based on MLN’s classic inference algorithms: Gibbs sampling and MC-SAT. We implement probabilistic versions of
Gibbs sampling and MC-SAT to handle probabilistic input data; in addition, in order to accelerate inference, we perform graph partition to find the optimal MLN subgraph of a queried tuple.

4.2.1 Graph Partition

Before the probability inference algorithms, we want to find the optimal MLN subgraph for a queried tuple. Although the complete MLN graph is sufficient for all queried tuples, it is often overly and unnecessarily large. The MLN subgraph is optimal when it is the minimal and sufficient for a queried tuple. In order to accelerate probability inference for a queried tuple, we need to implement a graph partition algorithm to find the optimal subgraph.

In MLN, the probability of a queried tuple $q$ only depends on its Markov Blanket $MB(q)$. Effectively, we follow a recursive depth-first-search in the provenance graph. If there exists an unobserved tuple $q'$ in $MB(q)$, the optimal subgraph should also contain optimal subgraph of $q'$. Therefore, the subgraph for $q$ should be $MB(q) \cup MB(q')$. This process continues until it reaches a tuple where its Markov Blanket consists of only observed tuples.

For example, if we want to find the optimal MLN subgraph for $cancer(2)$, we start from $cancer(2)$’s Markov Blanket. Tuple $smoke(2)$ is in $cancer(2)$’s Markov Blanket. Since $smoke(2)$ is an unobserved tuple, we continue searching $smoke(2)$’s Markov Blanket. Despite of $cancer(2)$, there are two tuples in $smoke(2)$’s Markov Blanket and they are $friends(1,2)$ and $smoke(1)$. Since both of them are observed tuples, we stop searching and return the final result.
Algorithm 1 Find the optimal subgraph

<table>
<thead>
<tr>
<th><strong>Input</strong></th>
<th>q: queried tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>G: the complete MLN graph</td>
</tr>
</tbody>
</table>

| **Output** | G*: Optimal subgraph  |

| **Initialize** | G* ← ∅: empty subgraph to return  |
|               | s ← {q}: stack for depth-first search  |
|               | S ← ∅: empty clique set  |

while s! = ∅ do
    t ← s.pop()
    for each clique c of t do
        if c is not in S then
            S.insert(c)
            for each tuple l in c do
                add l to G*
                if l is unobserved tuple then
                    st.push(l)
                end
            end
        end
    end
end

Algorithm 1 shows the pseudocode of the algorithm: the input q is the queried tuple and G is the complete MLN graph integrated from all provenances. We initialize the optimal subgraph G* as an empty set of tuples. During the depth-first search, if the current tuple l is an unobserved tuple, we insert it into G* and continue searching its Markov Blanket. If l is an observed tuple, we insert it into G* and stop searching.

4.2.2 Probabilistic Gibbs Sampling

Gibbs sampling is a MCMC algorithm which generates a series of posterior samples to approximate the real probability distribution. In MLN, Gibbs sampling samples on a the queried tuple’s Markov Blanket. The probability of queried tuple X_l given its Markov Blanket B_l is shown in Equation 2.4.
In this formula, \( f(X_l = x_l|B_l = b_l) \) indicates whether the clique \( f \) can be satisfied when the binary state of \( X_l \) is \( x_l \) and binary states of \( B_l \) are \( b_l \). In many real-world cases, the observed data is probabilistic and we cannot decide whether a clique \( f \) is true or false. One solution is using Monte Carlo simulation to generate observed data states based on their probabilities. Another solution is computing the probability of a clique being satisfied and use it as potential of the clique. Experimental results show that second solution is more stable and faster to converge.

Considering a clique \( c \) which is defined as a set of tuples \( \{h, w, b_1, ..., b_n\} \) where \( h \) indicates rule head tuple, \( w \) indicates rule weight and \( b_1, ..., b_n \) are rule body tuples, we can convert it into clausal form \( \{h, !b_1, ..., !b_n\} \) where all the literals are connected by disjunctive operators. If the probability of rule head tuple is \( p_h \) and the probabilities of rule body tuples are \( p_1, ..., p_n \), the probability of this clause being satisfied is

\[
p = 1 - (1 - p_h) \prod_{i=1}^{n} p_i
\]  

Equation 4.1

Therefore, the probability of \( X_l \) becomes

\[
P(X_l = x_l|B_l = b_l) = \frac{\exp(\sum p_i w_ip_i(X_l = x_l|B_l = b_l))}{\exp(\sum p_i w_ip_i(X_l = 0|B_l = b_l)) + \exp(\sum p_i w_ip_i(X_l = 1|B_l = b_l))}
\]

Equation 4.2

where both \( x_l \) and \( b_l \) are all probabilistic values and \( p_i \) indicates probability of the \( i \)th clique of variable \( l \) and it is defined in Equation 4.1.
Algorithm 2 Probabilistic Gibbs Sampling

**Input** : \( r \): number of sample times

\[ G \]: an MLN graph

**Output**: \( P \): probabilities of unobserved tuples in \( G \)

initialize \( x^{(0)} \) randomly

for iteration \( i = 1, 2, \ldots, r \) do

initialize \( x^{(i)} \leftarrow \emptyset \)

for each unobserved tuple \( l \) in \( G \) do

sample \( x^{(i)}_l \) using probability \( P(X_l = x^{(i-1)}_l|B_l = b^{(i-1)}_l) \)

add \( x^{(i)}_l \) to \( x^{(i)} \)

end

end

compute \( P \) given \( x^{(1)}, x^{(2)}, \ldots, x^{(r)} \)

Algorithm 2 shows the pseudocode of the Gibbs Sampling: the input \( r \) is the total sample times and \( G \) is an MLN graph. In our experiments, the graph \( G \) is always the optimal MLN subgraph returned from Algorithm 1. The beginning state \( x^{(0)} \) is initialized randomly. Then for each iteration, we sample the current state \( x^{(i)} \) using Equation 4.2 with the previous state \( x^{(i-1)} \). Then the output probability of a queried tuple \( P \) is computed from a series of states \( x^{(1)}, x^{(2)}, \ldots, x^{(r)} \).

4.2.3 Probabilistic MC-SAT

Different to Gibbs sampling which samples tuples based on its Markov Blanket, MC-SAT uses slice sampling and introduces an auxiliary variable \( u_k \) for each clique \( c_k \). Assume the clique weight \( w_k \) is always non-negative. If \( c_k \) is not satisfied by current state \( x^{(i)} \), we draw \( u_k \) uniformly from \([0,1]\). Therefore, \( u_k \leq 1 \) and there is no requirement for the next state to satisfy \( c_k \). When \( c_k \) is satisfied by \( x^i \), we draw \( u_k \) uniformly from \([0,e^{w_k}]\). With probability \( 1 - e^{-w_k} \), \( u_k \) is larger than \( e^{w_k} \), in which case \( c_k \) must be satisfied by next state \( x^{(i+1)} \). Following the process mentioned above, we can build
a subset of cliques $M$ and take a uniform sample on next states which satisfies all cliques of $M$ using SampleSAT algorithm.

When the input data is probabilistic, we cannot determine whether a clique is satisfied or not. However, we can calculate the probability of a clique being satisfied using Equation 4.1. Therefore, the potential of a clique $c_k$ becomes $e^{w_k p_k(x(i))}$. We draw $u_k$ uniformly from $[0, e^{w_k p_k(x(i))}]$ and with probability $1 - e^{-w_k p_k(x(i))}$, clique $c_k$ will be added to $M$.

Despite of the soften clique sampling, another adjustment is in SampleSAT. In SampleSAT, we want to minimize the sum of costs of cliques. The cost of clique $c_k$ is define as

$$cost(c_k) = \begin{cases} 1 & \text{if } c_k \text{ is not satisfied} \\ 0 & \text{if } c_k \text{ is satisfied} \end{cases} \quad (4.3)$$

We also soften this formula so that the cost of $c_k$ becomes

$$cost(c_k) = 1 - p_k \quad (4.4)$$

where $p_k$ is from Equation 4.1 and indicates the probability of clique $k$ being satisfied.

<table>
<thead>
<tr>
<th>Algorithm 3 Probabilistic MC-SAT</th>
</tr>
</thead>
</table>
| **Input**: $r$: number of sample times  
               $G$: an MLN graph  
| **Output**: $P$: probabilities of unobserved tuples in $G$ |
| initialize $x(0)$ randomly |
| for iteration $i=1,2,...,r$ do |
|   initialize $M \leftarrow \phi$ |
|   for each clique $c$ in $G$ do |
|     with probability $1 - e^{-w_k p_k(x(i-1))}$, add $c$ to $M$ |
|   end |
|   sample $x(i) \sim U_{SAT}(M)$ |
| end |
| compute $P$ given $x(1), x(2), ... x(r)$ |
Algorithm 3 shows pseudo code of probabilistic MC-SAT: the input \( r \) is the number of sample times and \( G \) is an MLN graph. Similar to Algorithm 2, \( G \) is also the optimal MLN subgraph returned from Algorithm 1. The starting state \( x^{(0)} \) is initialized randomly. Then for each iteration, we initialize an empty set of cliques \( M \). For each clique \( c \), we insert it into \( M \) with the probability \( 1 - e^{-w_k p_k(x^{(i-1)})} \) where \( x^{(i-1)} \) is the previous state. Then we use SampleSAT algorithm to generate the current state \( x^{(i)} \) which makes the cost of \( M \) the minimum. Finally, the output probability is computed from a series of states \( x^{(1)}, x^{(2)}, ..., x^{(r)} \).

4.3 Influence Query

In addition to the probability of queried tuple, users may want to know the influence of each observed tuple on the queried tuple. In other word, which tuple will change the probability of queried tuple the most if we change its probability? In this section, we will discuss how we define and compute the influence of observed tuples in MLN-based PLP programs.

The influence of an observed tuple evaluates the impact on the probability of queried tuple when the probability of observed tuple changes. Let \( p_q \) be the probability of a queried tuple \( q \), and \( p_o \) be the probability of an observed tuple \( o \). We define the influence of \( o \) on \( q \) as the partial derivative:

\[
\text{Inf}_o(q) = \frac{\partial p_q}{\partial p_o}
\]  

(4.5)

where \( p_o \) is a constant number from input data and \( p_q \) is calculated using probability query method from the previous section.

Assuming we have a function between \( p_q \) and \( p_o \) in which situation \( p_o \) is a variable of \( p_q \), such as equation 4.2, the partial derivative can be calculated directly only when there are no other unobserved tuples in \( q \)’s Markov Blanket \( B_q \). However, if there exist
other unobserved tuples in \( B_q \), it is impossible to calculate the partial derivative with chain rule because the probabilities of unobserved tuples depends on each other. In order to calculate the approximate influence value, we choose a small value \( \Delta \in (0, 1] \) and the influence can be calculated with

\[
Inf_{\Delta}(q) = \frac{p_q(p_o + \Delta_1) - p_q(p_o - \Delta_2)}{\Delta_1 + \Delta_2}
\] (4.6)

where \( \Delta_1 = min(\Delta, 1 - p_o) \) and \( \Delta_2 = min(\Delta, p_o) \) to ensure \( p_o \in [0, 1] \).

In order to find the most influence tuples, we can calculate the influence of every pair of \( p_q \) and \( p_o \) using Equation 4.6. However, not all observed tuples in an MLN have influence on the queried tuple. For example, friends(1,2) has no influence on cancer(5) in Figure 4.1 because they are connected through smoke(1) which is the observed tuple. Therefore we can find the optimal subgraph first using graph partition algorithm from Section 4.2.1 and it will reduce the overhead significantly.
In this chapter, we take two real-world cases as examples to demonstrate the feasibility of our system. The first case is Smoke, which infers the probability of a person smoking and having cancer. The second is called Visual Question Answering (VQA), which answers users’ questions related to a picture. It returns the most probable answers from a candidate list.

5.1 Smoke

Smoke is a classic case from the MLN literature. This case can be briefly described in two sentences:

- Someone who smokes is likely to have cancer.
- If two persons are friends, they are likely to have similar smoking habits.

As it is mentioned in Section 3.1, we use the pre-trained MLN model from Alchemy and rewrite it in weighted Datalog rules:

\[
\begin{align*}
    & r_1 0.89 : \text{cancer}(P) :- \text{smoke}(P). \\
    & r_2 0.44 : \text{smoke}(A) :- \text{friends}(A,B), \text{smoke}(B), A!=B. \\
    & r_3 0.44 : \text{smoke}(B) :- \text{friends}(A,B), \text{smoke}(A), A!=B.
\end{align*}
\]
Figure 5.1: Complete Markov Logic Network graph for a test case of Smoke.

5.1.1 Integrated provenance graph

In a PLP program, users may want to know what tuples can be derived from observed tuples and how we build the MLN graph for these tuples. Therefore, we use the method of graph integration mentioned in Section 4.1 and generate the complete MLN graph for a test case. The complete graph is shown in Figure 5.1. From this graph, we can
figure out the unobserved tuples which are in red cycles and how they are connected with observed tuples which are in blue circles.

5.1.2 Probability Query

Given a complete MLN graph, we want to query the probabilities of every unobserved tuples. In this case, the unobserved tuples include smoke(2), cancer(2), cancer(4), smoke(5), cancer(5), smoke(6), cancer(6), cancer(7), smoke(8) and cancer(8). The probabilities of input data are all set to 1.0. Table 5.1 shows the probability query results using both Gibbs sampling and MC-SAT when they all sample 100,000 rounds.

<table>
<thead>
<tr>
<th>Tuple name</th>
<th>Probability with Gibbs Sampling</th>
<th>Probability with MC-SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>smoke(2)</td>
<td>0.581</td>
<td>0.588</td>
</tr>
<tr>
<td>cancer(2)</td>
<td>0.315</td>
<td>0.310</td>
</tr>
<tr>
<td>cancer(4)</td>
<td>0.393</td>
<td>0.395</td>
</tr>
<tr>
<td>smoke(5)</td>
<td>0.491</td>
<td>0.503</td>
</tr>
<tr>
<td>cancer(5)</td>
<td>0.296</td>
<td>0.299</td>
</tr>
<tr>
<td>smoke(6)</td>
<td>0.478</td>
<td>0.485</td>
</tr>
<tr>
<td>cancer(6)</td>
<td>0.298</td>
<td>0.288</td>
</tr>
<tr>
<td>cancer(7)</td>
<td>0.390</td>
<td>0.391</td>
</tr>
<tr>
<td>smoke(8)</td>
<td>0.587</td>
<td>0.601</td>
</tr>
<tr>
<td>cancer(8)</td>
<td>0.317</td>
<td>0.304</td>
</tr>
</tbody>
</table>

Table 5.1: Probability query results.

From that table, we observe that Gibbs sampling and MC-SAT return similar results. Experiments also show that MC-SAT is faster than Gibbs sampling even through it is not obvious in this small test case; we plan to evaluate the performance more systematically in our future work.

Another observation is that the probability of smoke tuple increases when it is closer to the observed smoke tuple. For example, smoke(6), smoke(5) and smoke(2) are connected and smoke(2) is directly connected to the observed tuple smoke(1).
The probabilities of these three tuples are in ascending order. Since the probabilities of all input tuples are all 1, the closer an unobserved tuple is to the observed tuple, the more probable it has smoking habit.

5.1.3 Influence Query

In addition to calculating the probabilities of queried tuples, we also want to know how much influence the observed tuples have on the queried tuples’ probabilities. Take smoke(5) as an example, in the MLN graph shown in Figure 5.1, there are four observed tuples which may have influence on smoke(5). Identified by graph partition algorithm from Section 4.2.1, they are smoke(1), friends(1,2), friends(2,5) and friends(5,6). Using method described in Section 4.5, Table 5.2 shows the influence of these four observed tuples on smoke(5).

<table>
<thead>
<tr>
<th>Tuple name</th>
<th>Influence value</th>
</tr>
</thead>
<tbody>
<tr>
<td>smoke(1)</td>
<td>0.069</td>
</tr>
<tr>
<td>friends(1,2)</td>
<td>0.023</td>
</tr>
<tr>
<td>friends(2,5)</td>
<td>0.011</td>
</tr>
<tr>
<td>friends(5,6)</td>
<td>≈ 0</td>
</tr>
</tbody>
</table>

Table 5.2: Influence query results on smoke(5).

From this table, we can see that smoke(1) is the most influential tuple and the clique consisting of smoke(1), friends(1,2) and smoke(2) is the most important clique for smoke(5). It is reasonable because the probability of smoke(5) depends on smoke(2) and smoke(2) depends on smoke(1). In other word, smoke(1) is the probability source of all tuples which depend on it. Consider an extreme case, if person 1 does not smoke, person 5 is also unlikely to smoke. On the other hand, if the probability of smoke(1) is 1, the probability of smoke(5) will become very high.

Another interesting observation is that friends(5,6) nearly has almost no influence on smoke(5). In this MLN graph, smoke(6) is not grounded, its probability
relies solely on smoke(5). Therefore, the probability of friends(5,6) will not influence smoke(5) because friends(5,6) decides how closely smoke(5) and smoke(6) are correlated but smoke(6) does not contribute to smoke(5).

\[
\begin{array}{c}
cancer1 \\
smoke1 \\
smoke2 \\
friends1_2
\end{array}
\]

**Figure 5.2:** Small Markov Logic Network graph from smoke cases.

In order to have deeper insight of influence in this case, we use a smaller MLN graph and draw 3D graphs to show how the influence change when the input probabilities change. Figure 5.2 shows the small MLN graph we use. In this MLN graph, smoke(2) is the tuple we query and it is only influenced by smoke(1) and friends(1,2). We try to fit the curves of smoke(1)’s influence and friends(1,2)’s influence on smoke(2) while the probabilities of smoke(1) and friends(1,2) change. The influence is calculated with probabilistic Gibbs sampling with \( r = 100,000 \) and \( \Delta = 0.1 \). We also increase the rule weights of \( r2 \) and \( r3 \) so that the rule weights of \( r2 \) and \( r3 \) are much higher than \( r1 \) and other singular cliques. For example, we do not change the rule weights of other rules and change rule weights of \( r2 \) and \( r3 \) to 10. The reason we increase these two rule weights are as follows:

- Enhance the influence from rules \( r2 \) and \( r3 \), such as smoke(1) on smoke(2).

Therefore, the influence will in be a larger scale and easier to plot.
• Suppress the influence from other rules, such as $r1$ and other singular cliques. Therefore, influence values will be dominated by $r2$ and $r3$ which evaluate the correlation between smoke tuples and friends tuples.

Figure 5.3: Influence of smoke(1) on smoke(2).

Figure 5.3 shows the influence of smoke(1) on smoke(2). It is a 3D graph with three axes. In this graph, the axis labeled with "smoke1" represents the probability of tuple smoke(1). The axis labeled with "friend1_2" represents the probability of tuple friends(1,2) and the last axis represents the influence value of smoke(1) on smoke(2). From Figure 5.3, we can observe that the influence of smoke(1) is always positive. It is reasonable because the rule weight is positive then smoke(2) is always trying to catch smoke(1). The higher smoke(1) is, the higher smoke(2) will become.
Another interesting observation is that the influence becomes the highest when probability of smoke(1) is around 0.5. Intuitively, when the probability of smoke(1) is 0.5, smoke(1) is completely random. When it is slightly changed, it becomes deterministic instead of complete randomness. This change is more significant than situations where the probability of smoke(1) is around 0 or 1.

Figure 5.4: Influence of friends(1,2) on smoke(2).

Figure 5.4 shows the influence of friends(1,2) on smoke(2). The axes are similar to Figure 5.3. From this graph, we can observe that the influence of friends(1,2) is negative when smoke(1) is less than 0.5 and becomes positive when smoke(1) is greater than 0.5. Assuming that the probability of friends(1,2) is 0, smoke(1) does not depend on smoke(2) and is initialized randomly. In other word, the initialized
probability of smoke(2) is 0.5. As the probability of friends(1,2) rises, smoke(2) will move closer to smoke(1). Therefore, when probability of smoke(1) is less than 0.5, probability of smoke(2) drops in which situation the influence of friends(1,2) is negative. On the other hand, when probability of smoke(1) is greater than 0.5, probability of smoke(2) rises in which situation the influence of friends(1,2) is positive.

5.2 Visual Question Answering

Visual question answering (VQA) is a multi-modality machine learning system combining computer vision, natural language processing (NLP) and probabilistic logic reasoning. The target in VQA is answering some questions based on an image, such as identifying the building in the background. Our experiments are based on prior PSL programs [1]. Instead of predicting answers directly with deep neural networks, they collect tuples from machine learning models then reason the answer tuples with PLP program. Some input tuples are from computer vision models, such as hasImg which captures image information. Some are from NLP models, such as hasQ which contains question’s keywords. Others are word similarities (sim) from language models. All the tuples are probabilistic and we collect their probabilities from machine learning models and other sources.

Based on the PSL-version of the VQA program, we rewrite rules in ProbLog:

r1 w1: hasImgAns(V,Z,X1,R1,Y1) :- word(V,Z), hasImg(V,X1,R1,Y1), sim(Z,X1), sim(Z,Y1).

r2 w2: candidate(V,Z) :- word(V,Z).

r3 w3: candidate(V,Z) :- word(V,Z), hasQ(V,X,R,Y), hasImgAns(V,Z,X1,R1,Y1), sim(R,R1), sim(Y,Y1), sim(X,X1).
Figure 5.5: An abstract Visual Question Answering provenance graph.

\[ r4 \ w4: \text{ans}(V,Z) \leftarrow \text{candidate}(V,Z), \text{hasQ}(V,X,R,"WHAT"), \]
\[ \text{hasImg}(V,Z1,R1,X1), \text{sim}(Z,Z1), \text{sim}(R,R1), \text{sim}(X,X1). \]

In this program, \( r1 \) means we can extend \text{hasImg} tuples by replacing words in the tuples with their synonyms. \( r2 \) considers every word from our dictionary is possible to become a candidate for the question. Despite of \( r2 \), \( r3 \) is another source of question candidates where the candidate is derived from question context and word similarity. The last rule takes all information from images, questions and candidates then reasons the final answers. The rule weights \( w1 \), \( w2 \), \( w3 \) and \( w4 \) represent the probabilities of the rule head tuples being derived.

In this program, the observed tuple \text{word} connects every word in our dictionary with an image. Observed tuple \text{sim} represents the word similarity and is from language model. Observed tuple \text{hasImg} is from computer vision models and it contains extracted information from the image, such as "Horses are in the background.". Observed tuple \text{hasQ} is from NLP models and it contains extracted information about the question, such as "What is the building in the background?".
5.2.1 Explanation for correct prediction

Figure 5.5 [25] gives an example which shows how the program reasons an answer tuple \texttt{ans("ID1","barn")}. In this case, "ID1" is the image identifier and "barn" is the answer to the question "What is the building in the background?". We collect answer tuples inferred by this PLP program then find out \texttt{ans("ID1","barn")} is with highest probability, which means the building in the background is most likely to be a barn.

We queried the provenance of \texttt{ans("ID1","barn")} and used P3 to explain this result. In Figure 5.5, we show the most influential derivation of \texttt{ans("ID1","barn")}. In this derivation, word "barn" is selected to be candidate through rule \texttt{r2}. Then it derives \texttt{ans("ID1","barn")} because of its highly correlation with word "horse" which appears in the image. This result is also intuitively reasonable because barn is one of the most correlated building to horses.
P3 can also help identify the most influential base tuples. In this case of \texttt{ans("ID1","barn")}, we want to know which base tuple will affect the probability of \texttt{ans("ID1","barn")} the most if we change its probability. After running influence query on this case, we found that \texttt{word("ID1","barn")} is the most influential base tuple. This is reasonable because a word becomes an answer only when it is selected to be a candidate. Then the most influential \texttt{hasImg} tuple is \texttt{hasImg("ID1","horse","in","background")} which indicates knowing there are horses in the image is essential for the answer. The most influential word similarity tuple is \texttt{sim("barn","horse")}. This also confirms that horse plays an important role in this answer's inference.

5.2.2 Debugging unexpected prediction

In P3, identifying most influential base tuples can help us debug the program if some unexpected results occur. For example, in Figure 5.6 [25], we still want to know "What is the building in the background?". Figure 5.6 is similar to Figure 5.5. Both of them contain horses and a building in the background. From the cross at the top of the building in Figure 5.6, we can recognize that the building in Figure 5.6 should be a church. However, the VQA system predicts that the building is most likely to be a barn.

In this experiment, we included the information that "There are cross in the background." by replacing the image tuple \texttt{hasImg("ID1","horse","in","background")} with \texttt{hasImg("ID1","cross","in","background")}. Since we've confirmed that the program includes all required information, the most possible reason VQA predicts a wrong answer is that the probability values of input data are unreasonable.

In order to identify possible unreasonable tuples, we run influence query to find out the most influential tuples which contribute to the answer of "barn" and answer.
of "church". After the influence query, we noticed that not only word "barn" has much higher similarities to other objects, such as "horse" and "cross" than word "church", but also the word similarity score between "cross" and "church" is much lower than the one between "cross" and "barn". To be specific, the similarity between "cross" and "church" is 0.09 but the similarity between "cross" and "barn" is 0.3. Apparently, the similarity values of "cross" with other objects are too low. After we correct this error, the VQA system returns "church" as the most probable answer which is what we expected.
5.2.3 More experimental results

Figure 5.7 shows more experimental results of VQA. The first three images are with correct answers. We show tuples which are in the most influential derivations with their probabilities. For example, in the first image, the VQA system predicts it is terminal which is the expected answer. The most influential observation tuple is hasImg("planes", "are", "parked") which indicates that there are planes parked in the background of this image. The most influential word similarity tuples also include sim("terminal", "planes") which shows the highly correlation between terminal and planes.

In Figure 5.7, the lower three images are with wrong answers. We show top three unique influential base tuples of the expected answer with their influence values. The last lines show us that we can make the expected answer the most probable by modifying the probability of its most influential tuple. For example, in the first image, the VQA system predicts that the building in the background is house while the correct answer is hotel. We use influence query to find the three most influential tuples for ans("ID", "hotel") with their influence values. In this case, the most influential tuple to the correct answer is sim("hotel", "building"). By increasing the similarity value of sim("hotel", "building") by around 0.3, VQA system returns "hotel" as the most probable answer.
In this thesis, we propose and develop a debugging system based on data provenance for PLP programs. We use data provenance to record the inference derivations of PLP programs. Based on the provenance polynomials, we can query some complex problems, such as identifying influential inputs in PLP programs.

In our previous work P3, we rewrite PLP programs in ProbLog-like rules and compute the success probabilities of provenance polynomials. In this paper, we extend P3 to more general PLP programs where the rule weights do not only represent probabilities with the help of MLN. We convert the provenance polynomial into MLN cliques and compute both the joint probabilities of a grounded MLN and the conditional probability of unobserved tuples.

We design a new format of provenance polynomials which is easier to parse for MLN and we use it to build a MLN graph. However, the provenance of a queried tuple is not always sufficient to build the complete MLN graph. In order to ensure the MLN graph is sufficient for the queried tuple’s inference, we query the provenances of all unobserved tuples and integrate them into a complete MLN graph and we call this method graph integration.

Since the original MLN does not support probabilistic input data, we implement probabilistic versions of Gibbs sampling and MC-SAT inference algorithms. We use the inference algorithms to compute the conditional probabilities of queried tuples.
given some observed data. We also calculate the influence of observed tuples on queried tuples by computing the approximate partial derivatives.

We ran several experiments on classic MLN case Smoke. Querying results prove the effectiveness of our method and we also come up with some interesting conclusions from probability query and influential results. VQA is a more complex and practical case. Based on our previous experiments with P3, we ran more experiments with influence query. Analyzing VQA query results gives us valuable information for debugging VQA programs when it predicts unexpected answers.

Moving forward, we are planing to finish the following tasks:

- Implement a more complex and practical case from MLN programs. Hyper-text classification is a good choice. People can infer the class of a document given the words included in it. Compared to the case Smoke, this test case contains more observed data, has larger MLN graphs, and can be used on more practical applications.

- Evaluate the performance our provenance query, probability query and influence query algorithms. With larger MLN graphs, the running time of our query algorithms may become unacceptable and we want to explore optimizations and evaluate the effectiveness of these algorithms.

- Accelerate our probabilistic Gibbs sampling and MC-SAT algorithms. Accelerating these two algorithms is the key of our system acceleration because they are also the basis of our influence query algorithm. Besides optimizing their implementations, we are also planning to parallelize them by leveraging hardware accelerations such as deploying the algorithms on GPUs.
References


