Expectation Maximization in Deep Probabilistic Logic Programming

Arnaud Nguembang Fadja[1], Fabrizio Riguzzi[2] and Evelina Lamma[1]

[1] Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[2] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[arnaud.nguembafadja,fabrizio.riguzzi,evelina.lamma]@unife.it

Abstract. Probabilistic Logic Programming (PLP) combines logic and probability for representing and reasoning over domains with uncertainty. Hierarchical probability Logic Programming (HPLP) is a recent language of PLP whose clauses are hierarchically organized forming a deep neural network or arithmetic circuit. Inference in HPLP is done by circuit evaluation and learning is therefore cheaper than any generic PLP language. We present in this paper an Expectation Maximization algorithm, called Expectation Maximization Parameter learning for HIerarchical Probabilistic Logic programs (EMPHIL), for learning HPLP parameters. The algorithm converts an arithmetic circuit into a Bayesian network and performs the belief propagation algorithm over the corresponding factor graph.

Keywords: Hierarchical Probabilistic Logic Programming, Arithmetic Circuits, Expectation Maximization, Factor Graph, Belief Propagation.

1 Introduction

Due to it expressiveness and intuitiveness, Probabilistic Logic Programming (PLP) has been recently used in many fields such as natural language processing [17,13], link prediction [9] and bioinformatics [10,3]. Hierarchical PLP (HPLP) [12] is a type of PLP where clauses and predicates are hierarchically organized forming deep neural networks or arithmetic circuits (AC). In this paper we present an algorithm, called "Expectation Maximization Parameter learning for HIerarchical Probabilistic Logic programs" (EMPHIL), that performs parameter learning of HPLP using Expectation Maximization. The algorithm computes the required expectations by performing two passes over ACs.

The paper is organized as follows: Section 2 describes PLP and hierarchical PLP. Sections 3 presents EMPHIL. Related work is discussed in Section 4 and Section 5 concludes the paper.

2 Probabilistic Logic Programming and hierarchical PLP

PLP languages under the distribution semantics [18] have been shown expressive enough to represent a wide variety of domains [2,16,1]. A program in PLP under the distribution semantics defines a probability distribution over normal logic programs called *instances*. We consider in this paper a PLP language with a general syntax called *Logic Programs with Annotated Disjunctions* (LPADs) [19] in which each clause head is a disjunction of atoms annotated with probabilities. Consider a program T with p clauses: $P = \{C_1, \ldots, C_p\}$. Each clause C_i takes the form:

$$h_{i1} : \pi_{i1}; \ldots ; h_{in_i} : \pi_{in_i} :-\ b_{i1}, \ldots , b_{im_i}$$

where $h_{i1}, \ldots , h_{in_i}$ are logical atoms, $b_{i1}, \ldots , b_{im_i}$ are logical literals and $\pi_{i1}, \ldots , \pi_{in_i}$ are real numbers in the interval $[0, 1]$ that sum up to 1. $b_{i1}, \ldots , b_{im_i}$ is indicated with $body(C_i)$. Note that if $n_i = 1$ the clause corresponds to a non-disjunctive clause. We denote by $ground(T)$ the grounding of an LPAD T. Each grounding $C_i\theta_j$ of a clause C_i corresponds to a random variable X_{ij} with values $\{1, \ldots , n_i\}$. The random variables X_{ij} are independent of each other. An *atomic choice* [15] is a triple (C_i, θ_j, k) where $C_i \in T$, θ_j is a substitution that grounds C_i and $k \in \{1, \ldots , n_i\}$ identifies one of the head atoms. In practice (C_i, θ_j, k) corresponds to an assignment $X_{ij} = k$.

A *selection* σ is a set of atomic choices that, for each clause $C_i\theta_j$ in $ground(T)$, contains an atomic choice (C_i, θ_j, k). A selection σ identifies a normal logic program l_σ defined as $l_\sigma = \{(h_{ik} :-\ body(C_i))\theta_j |(C_i, \theta_j, k) \in \sigma\}$. l_σ is called an *instance* of T. Since the random variables associated to ground clauses are independent, we can assign a probability to instances: $P(l_\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$.

We write $l_\sigma \models q$ to mean that the query q is true in the well-founded model of the program l_σ. We denote the set of all instances by L_T. Let $P(L_T)$ be the distribution over instances. The probability of a query q given an instance l is $P(q|l) = 1$ if $l \models q$ and 0 otherwise. The probability of a query q is given by

$$P(q) = \sum_{l\in L_T} P(q, l) = \sum_{l\in L_T} P(q|l)P(l) = \sum_{l\in L_T :l\models q} P(l) \quad (1)$$

In the hierarchical PLP language [12], clauses are restricted to the following form:

$$C = p(\boldsymbol{X}) : \pi :-\ \phi(\boldsymbol{X}, \boldsymbol{Y}), b_1(\boldsymbol{X}, \boldsymbol{Y}), \ldots , b_k(\boldsymbol{X}, \boldsymbol{Y})$$

where $p(\boldsymbol{X})$ is the single atom in the head annotated with the probability π, $\phi(\boldsymbol{X}, \boldsymbol{Y})$ is a conjunction of input literals (that is their definitions are given in input and are non-probabilistic) and $b_i(\boldsymbol{X}, \boldsymbol{Y})$ for $i = 1, \ldots , k$ literals for are *hidden predicate*. This means that clauses and predicates are hierarchically organized forming a tree that can be translated into a neural networks or Arithmetic Circuit (AC). Inference can be performed with HPLP programs by generating their groundings that, similarly to clauses, form a tree. Such a tree can be used for inference by translating it into an Arithmetic Circuit (AC). The AC has a $\times$ node for each clause computing the product of the values of its children, and a

$\bigoplus$ node for each clause head, computing the function $\bigoplus_i p_i = 1 - \prod_i (1 - p_i)$. Moreover, $\neg$ nodes are associated with negative literals in bodies, computing the function $1 - p$ where p is the value of their only child, Each leaf is associated to the Boolean random variable X_i of a clause and takes value π. The AC can be evaluated bottom-up from the leaves to the root. Because of the constraints that HPLP programs must respect, literals in bodies are mutually independent and bodies of different clauses are mutually independent as well, so the value that is computed at the root is the probability that the atom associated with the root is true according to the distribution semantics. Let us call v(N) the value of node N in the arithmetic circuit. Circuits generation and inference are described in [12].

Example 1. Consider the UW-CSE domain [7] where the objective is to predict the "advised by" relation between students and professors. An example of an HPLP program for $advisedby/2$ may be

$$\begin{aligned}
C_1 \quad &= advisedby(A, B) : 0.3 :- \\
&\quad student(A), professor(B), project(C, A), project(C, B), \\
&\quad r_{11}(A, B, C). \\
C_2 \quad &= advisedby(A, B) : 0.6 :- \\
&\quad student(A), professor(B), ta(C, A), taughtby(C, B). \\
C_{111} &= r_{11}(A, B, C) : 0.2 :- \\
&\quad publication(D, A, C), publication(D, B, C).
\end{aligned}$$

where $project(C, A)$ means that C is a project with participant A, $ta(C, A)$ means that A is a teaching assistant (TA) for course C and $taughtby(C, B)$ means that course C is taught by B. $publication(A, B, C)$ means that A is a publication with author B produced in project C. $student/1$, $professor/1$, $project/2$, $ta/2$, $taughtby/2$ and $publication/3$ are input predicates and $r_{11}/3$ is a hidden predicate.

The probability of $q = advisedby(harry, ben)$ depends on the number of joint courses and projects and on the number of joint publications from projects. The clause for $r_{11}(A, B, C)$ computes an aggregation over publications of a projects and the clause level above aggregates over projects. Supposing harry and ben have two joint courses c1 and c2, two joint projects pr1 and pr2, two joint publications p1 and p2 from project pr1 and two joint publications p3 and p4 from project pr2, the AC of such ground program is shown in Figure 1.

3 EMPHIL

EMPHIL performs parameter learning of HPLP using Expectation Maximization (EM). The parameter learning problem is: given an HPLP P and a training set of positive and negative examples, $E = \{e_1, \ldots, e_M, \textbf{not } e_{M+1}, \ldots, \textbf{not } e_N\}$, find the parameters Π of P that maximize the log-likelihood (LL):

$$\arg\max_{\Pi} \sum_{i=1}^{M} \log P(e_i) + \sum_{i=M+1}^{N} \log(\textbf{not } P(e_i)) \tag{2}$$

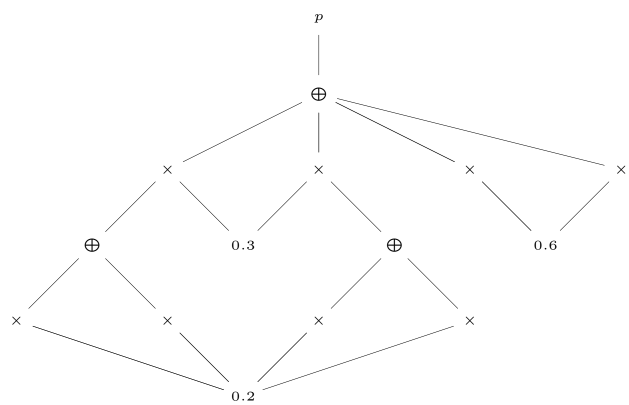


Fig. 1: Arithmetic circuit.
.

where $P(e_i)$ is the probability assigned to e_i by P. EMPHIL alternates between Expectation (E) and Maximization (M) steps. For a single example e, the Expectation step computes $\mathbf{E}[c_{i0}|e]$ and $\mathbf{E}[c_{i1}|e]$ for all rules C_i where c_{ix} is the number of times a variable X_{ij} takes value x for $x \in \{0, 1\}$ and for all $j \in g(i)$ i.e

$$\mathbf{E}[c_{ix}|e] = \sum_{j \in g(i)} P(X_{ij} = x|e)$$

where $g(i) = \{j|\theta_j$ is a substitution grounding $C_i\}$. These values are aggregated over all examples obtaining $E[c_{i0}] = \sum_{e \in E} \sum_{j \in g(i)} P(X_{ij} = 0|e)$ and $E[c_{i1}] = \sum_{e \in E} \sum_{j \in g(i)} P(X_{ij} = 1|e)$.

Then the Maximization computes

$$\pi_i = \frac{\mathbf{E}[c_{i1}]}{\mathbf{E}[c_{i0}] + \mathbf{E}[c_{i1}]}$$

For a single substitution θ_j of clause C_i we have that $P(X_{ij} = 0|e) + P(X_{ij} = 1|e) = 1$. So $E[c_{i0}] + E[c_{i1}] = \sum_{e \in E} |g(i)|$

Therefore to perform EMPHIL, we have to compute $P(X_{ij} = 1|e)$ for each example e. We do it using two passes over the AC, one bottom-up and one top-down. In order to illustrate the passes, we construct a graphical model associated with the AC and then apply the *belief propagation* (BP) algorithm [14].

A Bayesian Network (BN) can be obtained from the AC by replacing each node with a random variable. The variables associated with an $\bigoplus$ node have a conditional probabilistic table (CPT) that encodes an OR deterministic function, while variables associated with an $\times$ node have a CPT encoding an AND. Variables associated with a $\neg$ node have a CPT encoding the NOT function. Leaf nodes associated with the same parameter are split into as many nodes

X_{ij} as the groundings of the rule C_i, each associated with a CPT such that $P(X_{ij} = 1) = \pi_i$. We convert the BN into a Factor Graph (FG) using the standard translation because BN can be expressed in a simpler way for FGs. The FG corresponding to the AC of Figure 1 is shown in Figure 2.

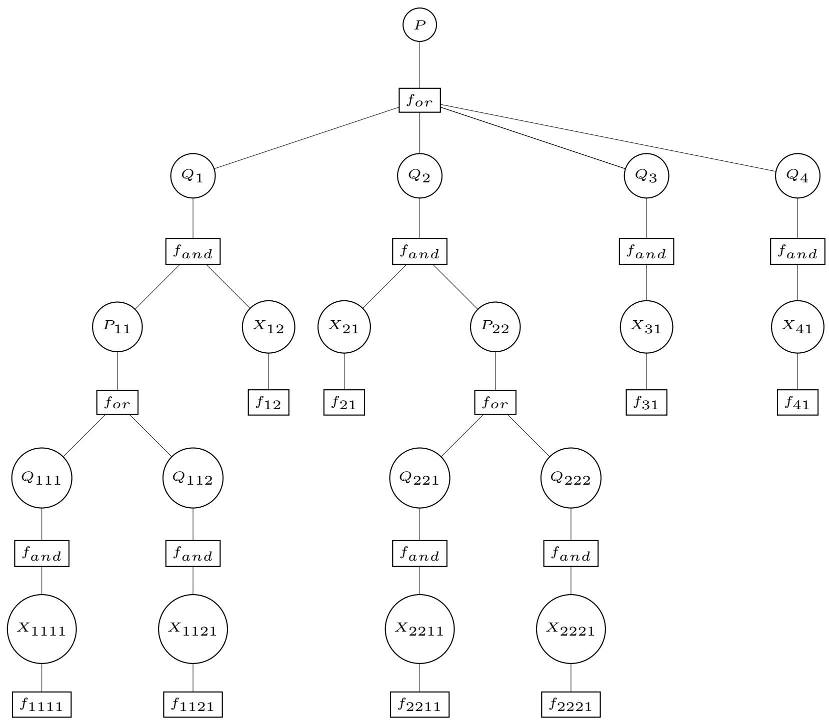


Fig. 2: Factor graph.

.

After constructing the FG, $P(X_{ij} = 0|e)$ and $P(X_{ij} = 1|e)$ are computed by exchanging messages among nodes and factors until convergence. In the case of FG obtained from an AC, the graph is a tree and it is sufficient to propagate the message first bottom-up and then top-down. The message from a variable N to a factor f is [14]

$$\mu_{N \to f}(n) = \prod_{h \in nb(N) \setminus f} \mu_{h \to N}(n) \tag{3}$$

where $nb(X)$ is the set of neighbor of X (the set of factors X appears in). The message from a factor f to a variable N is.

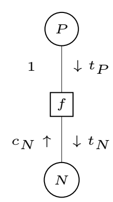
$$\mu_{f \to N}(n) = \sum_{\neg N} (f(n, \mathbf{s}) \prod_{Y \in nb(f) \setminus N} \mu_{Y \to f}(y)) \tag{4}$$

where $nb(f)$ is the set of arguments of f. After convergence, the belief of each variable N is defined as follows:

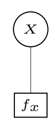
$$b(n) = \prod_{f \in nb(N)} \mu_{f \to N}(n) \tag{5}$$

that is the product of all incoming messages to the variable. By normalizing b(n) we obtain $P(N = n|e)$. Evidence is taken into account by setting the cells of the factor that are incompatible with evidence to 0. We want to develop an algorithm for computing $b(n)$ over the AC. So we want the AC nodes to send messages. We call c_N the normalized message, $\mu_{f \to N}(N = 1)$, in the bottom-up pass and t_N the normalized message, $\mu_{f \to N}(N = 1)$, in the top-down pass. Let us now compute the messages in the forward pass. Different cases can occur: the leaf, the inner and the root node. For a leaf node X, we have the factor graph in Figure 3b From Table 1d, the message from f_x to X is given by:

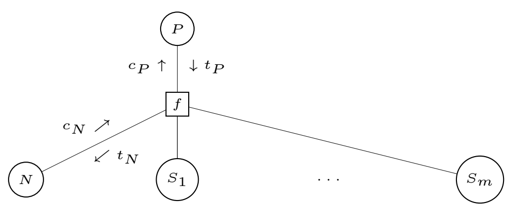
Fig. 3: Examples of factor graph



(a) Factor graph of not node.



(b) Factor graph for a leaf node.



(c) Factor graph for inner or root node.

$$\mu_{f_x \to X} = [\pi(x), 1 - \pi(x)] = [v(x), 1 - v(x)] \tag{6}$$

Note that the message is equal to the value of the node. Moreover, because of the construction of HPLP, for any variable node N

$$\mu_{f \to P}(p) = \mu_{P \to f}(p) \tag{7}$$

where P is the parent of N.

Let us consider a node P with children $N, S_1 \ldots S_m$ as shown in Figure 3c. We define $\mathbf{S} = S_1 \ldots S_m$ and $\mathbf{s} = s_1 \ldots s_m$. We prove by induction that $c_P = v(P)$.

Table 1: Cpts of factors

(a) P is an *or* node

p	$n=1$	$n=0$, $\mathbf{S}=\mathbf{0}$	$n=1$, $\neg\,(\mathbf{S}=\mathbf{0})$
0	0	1	0
1	1	0	1

(b) P is an *and* node

p	$n=0$	$n=1$, $\mathbf{S}=\mathbf{1}$	$n=1$, $\neg\,(\mathbf{S}=\mathbf{1})$
0	1	0	1
1	0	1	0

(c) P is a not node

p	$n=0$	$n=1$
0	0	1
1	1	0

(d) Leaf node $f_x = \pi(x)$

x	f_x
0	1-$\pi(x)$
1	$\pi(x)$

For leaf nodes it was proved above. Suppose that $c_C = v(C)$ for all children $N, S_1, ...S_m$:

If P is an $\times$ node, the cpt of P given its children is described in Table 1b and $\mu_{C\to f}(c) = v(c)$ for all children C. According to equation 4 we have:

$$\begin{aligned}\mu_{f\to P}(1) &= \sum_{\neg P} f(p,n,\mathbf{s}) \prod_{Y\in nb(f)\setminus P} \mu_{Y\to f}(y) \\ &= \sum_{n,\mathbf{s}} (f(p,n,\mathbf{s}) \prod_{Y\in\{N,\mathbf{S}\}} \mu_{Y\to f}(y) \\ &= \mu_{N\to f}(1)\cdot \prod_{S_k} \mu_{S_k\to f}(1) \\ &= v(N)\cdot \prod_{s_k} v(S_k) = v(P)\end{aligned} \tag{8}$$

In the same way, from Equation 8 we have:

$$\begin{aligned}\mu_{f\to P}(0) &= 1 - \mu_{N\to f}(1)\cdot \prod_{S_k} \mu_{S_k\to f}(1) \\ &= 1 - v(N)\cdot \prod_{s_k} v(S_k) = 1 - v(P)\end{aligned}$$

So $c_P = v(P)$

If P is an $\bigoplus$ node, the cpt of P given its children is described in Table 1a. From Equation 8 we have:

$$\begin{aligned}\mu_{f\to P}(1) &= \sum_{n,\mathbf{s}} f(p,n,\mathbf{s}) \prod_{Y\in\{N,\mathbf{S}\}} \mu_{Y\to f}(y)\\ &= 1-\mu_{N\to \cdot}(0)\cdot \prod_{S_k}\mu_{S_k\to f}(0) = 1-v(N)\cdot\prod_{S_k} v(S_k)\\ &= 1-(1-v(N))\cdot\prod_{S_k}(1-v(S_k)) = v(P)\end{aligned}$$

In the same way we have:

$$\begin{aligned}\mu_{f\to P}(0) &= \mu_{N\to f}(0).\prod_{S_k}\mu_{S_k\to f}(0) = v(N=0)\cdot\prod_{S_k} v(N=0)\\ &= 1-[1-(1-v(N)\cdot\prod_{S_k} v(S_k)] = 1-v(P)\end{aligned}$$

If P is a $\neg$ node with the single child N, its cpt us shown in table 1c and we have:

$$\begin{aligned}\mu_{f\to P}(1) &= \sum_{n} f(p,n) \prod_{Y\in\{N\}} \mu_{Y\to f}(y)\\ &= \mu_{N\to f}(0) = 1-v(N)\end{aligned}$$

and

$$\mu_{f\to P}(0) = \mu_{N\to f}(1) = v(N)$$

Overall, exchanging message in the forward pass means evaluating the value of each node in the AC. This leads to Algorithm 20.

Now let us compute the messages in the backward pass. Considering the factor graph in Figure 3c, we consider the message $t_P = \mu_{P\to f}(1)$ as known and we want to compute the message $t_N = \mu_{f\to N}(1)$.

If P is an *inner* $\bigoplus$ node (with children $N, S_1, ...S_m$), its cpt is shown in table 1a. Let us compute the messages $\mu_{f\to N}(1)$ and $\mu_{f\to N}(0)$:

$$\begin{aligned}\mu_{f\to N}(1) &= \sum_{\neg N}(f(p,n,\mathbf{s}) \prod_{Y\in nb(f)\backslash N} \mu_{Y\to f}(y))\\ &= [\sum_{p,\mathbf{s}}(f(p,n,\mathbf{s})\prod_{S} v(s)]\cdot[\mu_{P\to f}(1)]\\ &= \mu_{P\to f}(1) = t_P \qquad (9)\end{aligned}$$

In the same way

$$\begin{aligned}\mu_{f\to N}(0) &= \sum_{p,\mathbf{s}} f(p,n,\mathbf{s})\prod_{S} v(s)[\mu_{P\to f}(p)] \qquad (10)\\ &= [1-\prod_{S}(1-v(S))]\cdot[\mu_{P\to f}(1)] + \prod_{S}(1-v(S))[\mu_{P\to f}(0)]\\ &= v(P)\ominus v(N)\cdot t_P + (1-v(P)\ominus v(N))\cdot(1-t_P)\end{aligned}$$

Algorithm 1 Function Forward

```
function Forward(node)                                    ▷ node is an AC
    if node = not(n) then
        v(node) ← 1 − Forward(n)
        return v(node)
    else
        ▷ Compute the output example by recursively call Forward on its sub AC
        if node = ⨁(n_1, ... n_m) then                    ▷ ⨁ node
            for all n_j do
                v(n_j) ← Forward(n_j)
            end for
            v(node) ← v(n_1) ⊕ ... ⊕ v(n_m)
            return v(node)
        else                                              ▷ and Node
            if node = ×(π_i, n_1, ... n_m) then
                for all n_j do
                    v(n_j) ← Forward(n_j)
                end for
                v(node) ← π_i · v(n_1) · ... · v(n_m)
                return v(node)
            end if
        end if
    end if
end function
```

where the operator $\ominus$ is defined as follows:

$$v(p) \ominus v(n) = 1 - \prod_{\mathbf{s}} (1 - v(s)) = 1 - \frac{1 - v(p)}{1 - v(n)} \tag{11}$$

So we have

$$t_N = \frac{t_P}{t_P + v(P) \ominus v(N) \cdot t_P + (1 - v(P) \ominus v(n)) \cdot (1 - t_P)} \tag{12}$$

If P is a $\times$ node, its cpt is shown in table 1b and we have:

$$\begin{aligned}
\mu_{f \to N}(1) &= \sum_{\neg N} (f(p, n, \mathbf{s}) \prod_{Y \in nb(f) \setminus N} \mu_{Y \to f}(y)) \\
&= \mu_{P \to f}(P = 1) \cdot \prod_{S} \mu_{S \to f}(1) + \mu_{P \to f}(0) \cdot (1 - \prod_{S} \mu_{S \to f}(1)) \\
&= t_P \cdot \prod_{S} v(S) + (1 - t_P) \cdot (1 - \prod_{S} v(S)) \\
&= t_P \cdot \frac{v(P)}{v(N)} + (1 - t_P) \cdot (1 - \frac{v(P)}{v(N)})
\end{aligned}$$

In the same way,

$$\mu_{f \to N}(0) = \mu_{P \to f}(0) \cdot \sum_{\mathbf{S}} (f(p, n, \mathbf{s}) \prod_{\mathbf{S}} \mu_{S \to f}(s)) = 1 - t_P$$

So we have

$$t_N = \frac{t_P \cdot \frac{v(P)}{v(N)} + (1 - t_P) \cdot (1 - \frac{v(P)}{v(N)})}{t_P \cdot \frac{v(P)}{v(N)} + (1 - t_P).(1 - \frac{v(P)}{v(N)}) + (1 - t_P)} \tag{13}$$

If P is a $\neg$ node, its cpt is shown in Table 1c and we have:

$$\mu_{f \to N}(1) = \sum_{p} f(p, n) \prod_{Y \in \{P\}} \mu_{Y \to f}(y) = \mu_{P \to f}(0) = 1 - t_P$$

Equivalently

$$\mu_{f \to N}(0) = \sum_{p} f(p, n) \prod_{Y \in \{P\}} \mu_{Y \to f}(y) = \mu_{P \to f}(1) = t_P$$

And then

$$t_N = \frac{1 - t_P}{1 - t_P + t_P} = 1 - t_P \tag{14}$$

To take into account evidence, we consider $\mu_{P \to f} = [1, 0]$ as the initial messages in the backward pass (where P is the root) and use Equation 12 for $\bigoplus$ node. Overall, in the backward pass we have:

$$t_N = \begin{cases} \frac{t_P}{t_P + v(P) \ominus v(N) . t_P + (1 - v(P) \ominus v(N)) . (1 - t_p)} & \text{if } P \text{ is a } \oplus \text{ node} \\ \frac{t_P . \frac{v(P)}{v(P)} + (1 - t_P) . (1 - \frac{v(P)}{v(N)})}{t_P . \frac{v(P)}{v(N)} + (1 - t_P) . (1 - \frac{v(P)}{v(N)}) + (1 - t_P)} & \text{if } P \text{ is a } \times \text{ node} \\ 1 - t_P & \text{if } P \text{ is a } \neg \text{ node} \end{cases} \tag{15}$$

Since the belief propagation algorithm (for AC) converges after two passes, we can compute the unnormalized belief of each parameter during the backward pass by multiplying t_N by $v(N)$ (that is all incoming messages). Algorithm 2 perform the backward pass of belief propagation algorithm and computes the normalized belief of each parameter. It also counts the number of clause groundings using the parameter, lines 17–18.

We present EMPHIL in Algorithm 3. After building the ACs (sharing parameters) for positive and negative examples and initializing the parameters, the expectations and the counters, lines 2–5, EMPHIL proceeds by alternating between expectation step 8–13 and maximization step 13–17. The algorithm stops when the difference between the current value of the LL and the previous one is below a given threshold or when such a difference relative to the absolute value of the current one is below a given threshold. The theory is then updated and returned (lines 19–20).

Algorithm 2 PROCEDURE BACKWARD

procedure BACKWARD($t_p, node, B, Count$)
  if $node = not(n)$ **then**
    BACKWARD($1 - t_p, n, B, Count$)
  else
    if $node = \bigoplus(n_1, \dots n_m)$ **then** ▷ $\bigoplus$ node
      for all child n_i **do**
        $t_{n_i} \leftarrow \frac{t_p}{t_p + v(node) \ominus v(n_i).t_p + (1 - v(node) \ominus v(n_i)).(1 - t_p)}$
        BACKWARD($t_{n_i}, n_i, B, Count$)
      end for
    else
      if $node = \times(n_1, \dots n_m)$ **then** ▷ $\times$ node
        for all child n_i **do**
          $t_{n_i} \leftarrow \frac{t_p . \frac{v(node)}{v(n_i)} + (1 - t_p).(1 - \frac{v(node)}{v(n_i)})}{t_p . \frac{v(node)}{v(n_i)} + (1 - t_p).(1 - \frac{v(node)}{v(n_i)}) + (1 - t_p)}$
          BACKWARD($t_{n_i}, n_i, B, Count$)
        end for
      else ▷ leaf node π_i
        $B[i] \leftarrow B[i] + \frac{\pi_i t_p}{(\pi_i t_p + (1 - \pi_i)(1 - t_p))}$
        $Count[i] \leftarrow Count[i] + 1$
      end if
    end if
  end if
end procedure

4 Related Work

EMPHIL is related to Deep Parameter learning for HIerarchical probabilistic Logic programs (DPHIL) [4] that learns hierarchical PLP parameters using gradient descent and back-propagation. Similarly to EMPHIL, DPHIL perfoms two passes over the ACs: the Forward pass evaluates the AC, as EMPHIL, and the backward pass computes the gradient of the cross entropy error with respect to each parameter. A method for stochastic optimization, Adam [6], is used to update the parameters (shared over the ACs). Since EMPHIL is strongly related to EMPHIL we plan in our future work to implement EMPHIL and compare the performance of both algorithms.

Hierarchical PLP is also related to [8,5,11] where the probability of a query is computed by combining the contribution of different rules and grounding of rules with noisy-Or or Mean combining rules. In First-Order Probabilistic Logic (FOPL), [8] and Bayesian Logic Programs (BLP), [5], each ground atoms is considered as a random variables and rules have a single atom in the head and only positive literals in the body. Each rule is associated with a CPT defining the dependence of the head variable given the body ones. Similarly to HPLP, FOPL and BLP allow multiple layers of rules. Differently from FOPL and HPLP, BLP allows different combining rules. Like FOLP, BLP and hierarchical PLP, First-Order Conditional Influence Language (FOCIL) [11], uses probabilistic rules

Algorithm 3 Function EMPHIL.

function EMPHIL($Theory, \epsilon, \delta, Max$)
  $Examples \leftarrow$ BUILDACS($Theory$) ▷ Build the set of ACs
  for $i \leftarrow 1 \rightarrow |Theory|$ **do**
    $\Pi[i] \leftarrow random; B[i], Count[i] \leftarrow 0$ ▷ Initialize the parameters
  end for
  $LL \leftarrow -inf; Iter \leftarrow 0$
  repeat
    $LL_0 \leftarrow LL, LL \leftarrow 0$ ▷ Expectation step
    for all $node \in Examples$ **do**
      $P \leftarrow$ FORWARD($node$)
      BACKWARD($1, node, B, Count$)
      $LL \leftarrow LL + \log P$
    end for ▷ Maximization step
    for $i \leftarrow 1 \rightarrow |Theory|$ **do**
      $\Pi[i] \leftarrow \frac{B[i]}{Count[i]}$
      $B[i], Count[i] \leftarrow 0$
    end for
  until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL.\delta \vee Iter > Max$
  $FinalTheory \leftarrow$ UPDATETHEORY($Theory, \Pi$)
  return $FinalTheory$
end function

for compactly encoding probabilistic dependencies. The probability of a query is computed using two combining rules: the contributions of different groundings of the same rule with the same random variable in the head are combined by taking the *Mean* and the contributions of different rules are combined either with a weighted mean or with a *noisy-OR* combining rule. HPLP instead uses the noisy-OR combining rule for both cases. FOPL, BLP and FOCIL implement parameter learning using gradient descent, or Expectation Maximization, as EMPHIL. In this paper we specialized the results of [8,5,11] for the HPLP language obtaining formulas that can be easily implemented into an algorithm.

5 Conclusion

We present in this paper the algorithm EMPHIL for learning the parameters of hierarchical PLP using Expectation Maximization. The formula for expectations are obtained by converting an arithmetic circuit into a Bayesian network and performing belief propagation algorithm over the corresponding factor graph. We plan in our future work to implement and compare EMPHIL with DPHIL, an algorithm that performs parameter learning of HPLP using gradient descent. We also plan to design an algorithm for learning the structure of hierarchical PLP in order to search in the space of HPLP the program that best described the data using EMPHIL or DPHIL as sub-procedures.

References

1. Alberti, M., Bellodi, E., Cota, G., Riguzzi, F., Zese, R.: `cplint` on SWISH: Probabilistic logical inference with a web browser. Intell. Artif. 11(1), 47–64 (2017)
2. Alberti, M., Cota, G., Riguzzi, F., Zese, R.: Probabilistic logical inference on the web. In: Adorni, G., Cagnoni, S., Gori, M., Maratea, M. (eds.) AI*IA 2016. LNCS, vol. 10037, pp. 351–363. Springer International Publishing (2016)
3. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI 2007. vol. 7, pp. 2462–2467. AAAI Press/IJCAI (2007)
4. Fadja, A.N., Riguzzi, F., Lamma, E.: Learning the parameters of deep probabilistic logic programs. In: Bellodi, E., Schrijvers, T. (eds.) Probabilistic Logic Programming (PLP 2018). CEUR Workshop Proceedings, vol. 2219, pp. 9–14. Sun SITE Central Europe, Aachen, Germany (2018)
5. Kersting, K., De Raedt, L.: Basic principles of learning bayesian logic programs. In: Institute for Computer Science, University of Freiburg. Citeseer (2002)
6. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
7. Kok, S., Domingos, P.: Learning the structure of Markov Logic Networks. In: ICML 2005. pp. 441–448. ACM (2005)
8. Koller, D., Pfeffer, A.: Learning probabilities for noisy first-order rules. In: IJCAI. pp. 1316–1323 (1997)
9. Meert, W., Struyf, J., Blockeel, H.: CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In: De Raedt, L. (ed.) ILP 2009. LNCS, vol. 5989, pp. 96–109. Springer (2010)
10. Mørk, S., Holmes, I.: Evaluating bacterial gene-finding hmm structures as probabilistic logic programs. Bioinformatics 28(5), 636–642 (2012)
11. Natarajan, S., Tadepalli, P., Kunapuli, G., Shavlik, J.: Learning parameters for relational probabilistic models with noisy-or combining rule. In: Machine Learning and Applications, 2009. ICMLA'09. International Conference on. pp. 141–146. IEEE (2009)
12. Nguembang Fadja, A., Lamma, E., Riguzzi, F.: Deep probabilistic logic programming. In: Theil Have, C., Zese, R. (eds.) PLP 2017. CEUR-WS, vol. 1916, pp. 3–14. Sun SITE Central Europe (2017)
13. Nguembang Fadja, A., Riguzzi, F.: Probabilistic logic programming in action. In: Holzinger, A., Goebel, R., Ferri, M., Palade, V. (eds.) Towards Integrative Machine Learning and Knowledge Extraction, LNCS, vol. 10344. Springer (2017)
14. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1988)
15. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artif. Intell. 94, 7–56 (1997)
16. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., Cota, G.: Probabilistic logic programming on the web. Softw.-Pract. Exper. 46(10), 1381–1396 (10 2016)
17. Riguzzi, F., Lamma, E., Alberti, M., Bellodi, E., Zese, R., Cota, G.: Probabilistic logic programming for natural language processing. In: Chesani, F., Mello, P., Milano, M. (eds.) Workshop on Deep Understanding and Reasoning, URANIA 2016. CEUR Workshop Proceedings, vol. 1802, pp. 30–37. Sun SITE Central Europe (2017)
18. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP 1995. pp. 715–729. MIT Press (1995)

19. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions. In: ICLP 2004. LNCS, vol. 3132, pp. 431–445. Springer (2004)