MCINTYRE: A Monte Carlo Algorithm for Probabilistic Logic Programming

Fabrizio Riguzzi

ENDIF – Università di Ferrara – Via Saragat, 1 – 44122 Ferrara, Italy.
{fabrizio.riguzzi}@unife.it

Abstract. Probabilistic Logic Programming is receiving an increasing attention for its ability to model domains with complex and uncertain relations among entities. In this paper we concentrate on the problem of approximate inference in probabilistic logic programming languages based on the distribution semantics. A successful approximate approach is based on Monte Carlo sampling, that consists in verifying the truth of the query in a normal program sampled from the probabilistic program. The ProbLog system includes such an algorithm and so does the `cplint` suite. In this paper we propose an approach for Monte Carlo inference that is based on a program transformation that translates a probabilistic program into a normal program to which the query can be posed. In the transformation, auxiliary atoms are added to the body of rules for performing sampling and checking for the consistency of the sample. The current sample is stored in the internal database of the Yap Prolog engine. The resulting algorithm, called MCINTYRE for Monte Carlo INference wiTh Yap REcord, is evaluated on various problems: biological networks, artificial datasets and a hidden Markov model. MCINTYRE is compared with the Monte Carlo algorithms of ProbLog and `cplint` and with the exact inference of the PITA system. The results show that MCINTYRE is faster than the other Monte Carlo algorithms.

Keywords: Probabilistic Logic Programming, Monte Carlo Methods, Logic Programs with Annotated Disjunctions, ProbLog.

1 Introduction

Probabilistic Logic Programming (PLP) is an emerging field that has recently seen many proposals for the integration of probability in logic programming. Such an integration overcomes the limit of logic of dealing only with certain propositions and the limit of works in probability theory that consider mostly simple descriptions of domain entities instead of complex relational descriptions.

PLP is of interest also for its many application domains, the most promising of which is maybe Probabilistic Inductive Logic Programming [5] in which PLP languages are used to represent the theories that are induced from data. This allows a richer representation of the domains that often leads to increased modeling accuracy. This trend can be cast in a more general tendency in Machine

Learning to combine aspects of uncertainty with aspects of logic, as is testified by the development of the field of Statistical Relational Learning [7].

Many languages have been proposed in PLP. Among them, many share a common approach for defining the semantics, namely the so called distribution semantics [17]. This approach sees a probabilistic logic program as a description of a probability distribution over normal logic programs, from which the probability of queries is computed. Example of languages following the distribution semantics are Probabilistic Logic Programs [3], Probabilistic Horn Abduction [10], Independent Choice Logic [11], PRISM [17], Logic Programs with Annotated Disjunctions (LPADs) [21] and ProbLog [6]. These languages have essentially the same expressive power [20,4] and in this paper we consider only LPADs and ProbLog, because they stand at the extremes of syntax complexity, LPADs having the most complex syntax and ProbLog the simplest, and because most existing inference algorithms can be directly applied to them.

The problem of inference, i.e., the problem of computing the probability of a query from a probabilistic logic program, is very expensive, being #P complete [8]. Nevertheless, various exact inference algorithms have been proposed, such as the ProbLog system[1] [6], `cplint`[2] [12,13] and PITA[3] [14,16] and have been successfully applied to a variety of non-trivial problems. All of these algorithms find explanations for queries and then use Binary Decision Diagrams (BDDs) for computing the probability. This approach has been shown to be faster than algorithms not using BDDs. Reducing the time to answer a probabilistic query is important because in many applications, such as in Machine Learning, a high number of queries must be issued. To improve the speed, approximate inference algorithms have been proposed. Some compute a lower bound of the probability, as the k-best algorithm of ProbLog [8] which considers only the k most probable explanations for the query, while some compute an upper and a lower bound, as the bounded approximation algorithm of ProbLog [8] that builds an SLD tree only to a certain depth. A completely different approach for approximate inference is based on sampling the normal programs encoded by the probabilistic program and checking whether the query is true in them. This approach, called Monte Carlo, was first proposed in [8] for ProbLog, where a lazy sampling approach was used in order to avoid sampling unnecessary probabilistic facts. [1] presented algorithms for k-best, bounded approximation and Monte Carlo inference for LPADs that are all based on a meta-interpreter. In particular, the Monte Carlo approach uses the arguments of the meta-interpreter predicate to store the samples taken and to ensure consistency of the sample.

In this paper we present the algorithm MCINTYRE for Monte Carlo INference wiTh Yap REcord that computes the probability of queries by means of a program transformation technique. The disjunctive clauses of an LPAD are first transformed into normal clauses to which auxiliary atoms are added to the body for taking samples and storing the results. The internal database of the

[1] `http://dtai.cs.kuleuven.be/problog/`
[2] `http://www.ing.unife.it/software/cplint/`
[3] `https://sites.google.com/a/unife.it/ml/pita`

Yap Prolog engine is used to record all samples taken thus ensuring that samples are consistent. The truth of a query in a sampled program can be then tested by asking the query to the resulting normal program.

MCINTYRE is compared with the Monte Carlo algorithms of ProbLog and `cplint` and with the exact inference algorithm of the PITA system on various problems: biological networks, artificial datasets and a hidden Markov model. The results show that the performances of MCINTYRE overcome those of the other Monte Carlo algorithms.

The paper is organized as follows. In Section 2 we review the syntax and the semantics of PLP. Section 3 illustrates previous approaches for inference in PLP languages. Section 4 presents the MCINTYRE algorithm. Section 5 describes the experiments and Section 6 concludes the paper.

2 Probabilistic Logic Programming

One of the most interesting approaches to the integration of logic programming and probability is the distribution semantics [17], which was introduced for the PRISM language but is shared by many other languages.

A program in one of these languages defines a probability distribution over normal logic programs called *worlds*. This distribution is then extended to queries and the probability of a query is obtained by marginalizing the joint distribution of the query and the programs. We present the semantics for programs without function symbols but the semantics has been defined also for programs with function symbols [17,15].

The languages following the distribution semantics differ in the way they define the distribution over logic programs. Each language allows probabilistic choices among atoms in clauses: Probabilistic Logic Programs, Probabilistic Horn Abduction , Independent Choice Logic, PRISM and ProbLog allow probability distributions over facts, while LPADs allow probability distributions over the heads of disjunctive clauses. All these languages have the same expressive power: there are transformations with linear complexity that can convert each one into the others [20,4]. Next we will discuss LPADs and ProbLog.

Formally a *Logic Program with Annotated Disjunctions* T [21] consists of a finite set of annotated disjunctive clauses. An annotated disjunctive clause C_i is of the form $h_{i1} : \Pi_{i1}; \ldots; h_{in_i} : \Pi_{in_i} : -b_{i1}, \ldots, b_{im_i}$. In such a clause $h_{i1}, \ldots h_{in_i}$ are logical atoms and $b_{i1}, \ldots, b_{im_i}$ are logical literals, $\{\Pi_{i1}, \ldots, \Pi_{in_i}\}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. $b_{i1}, \ldots, b_{im_i}$ is called the *body* and is indicated with $body(C_i)$. Note that if $n_i = 1$ and $\Pi_{i1} = 1$ the clause corresponds to a non-disjunctive clause. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$ the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} \Pi_{ik}$. We denote by $ground(T)$ the grounding of an LPAD T.

An *atomic choice* is a triple (C_i, θ_j, k) where $C_i \in T$, θ_j is a substitution that grounds C_i and $k \in \{1, \ldots, n_i\}$. (C_i, θ_j, k) means that, for the ground clause $C_i\theta_j$, the head h_{ik} was chosen. In practice $C_i\theta_j$ corresponds to a random

variable X_{ij} and an atomic choice (C_i, θ_j, k) to an assignment $X_{ij} = k$. A set of atomic choices κ is *consistent* if $(C_i, \theta_j, k) \in \kappa, (C_i, \theta_j, l) \in \kappa \Rightarrow k = l$, i.e., only one head is selected for a ground clause. A *composite choice* κ is a consistent set of atomic choices. The *probability* $P(\kappa)$ *of a composite choice* κ is the product of the probabilities of the individual atomic choices, i.e. $P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik}$.

A *selection* σ is a composite choice that contains an atomic choice (C_i, θ_j, k) for each clause $C_i\theta_j$ in $ground(T)$. A selection σ identifies a normal logic program w_σ defined as $w_\sigma = \{(h_{ik} : -body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. w_σ is called a *world* of T. Since selections are composite choices, we can assign a probability to possible worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$.

Since the program does not have function symbols, the set of worlds is finite: $W_T = \{w_1, \ldots, w_m\}$ and, since the probabilities of the individual choices sum to 1, $P(w)$ is a distribution over worlds: $\sum_{w\in W_T} P(w) = 1$. We also assume that each world w has a two-valued well founded model $WFM(w)$. If a query Q is true in $WFM(w)$ we write $w \models Q$.

We can define the conditional probability of a query Q given a world: $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. The probability of the query can then be obtained by marginalizing over the query

$$P(Q) = \sum_w P(Q, w) = \sum_w P(Q|w)P(w) = \sum_{w\models Q} P(w)$$

Example 1. The following LPAD T encodes a very simple model of the development of an epidemic or a pandemic:

$$C_1 = epidemic : 0.6; pandemic : 0.3 : -flu(X), cold.$$
$$C_2 = cold : 0.7.$$
$$C_3 = flu(david).$$
$$C_4 = flu(robert).$$

This program models the fact that if somebody has the flu and the climate is cold, there is the possibility that an epidemic or a pandemic arises. We are uncertain about whether the climate is cold but we know for sure that David and Robert have the flu. Clause C_1 has two groundings, both with three atoms in the head, while clause C_2 has a single grounding with two atoms in the head, so overall there are $3 \times 3 \times 2 = 18$ worlds. The query *epidemic* is true in 5 of them and its probability is

$$\begin{aligned} P(epidemic) = {} & 0.6 \cdot 0.6 \cdot 0.7 + 0.6 \cdot 0.3 \cdot 0.7 + 0.6 \cdot 0.1 \cdot 0.7 + \\ & 0.3 \cdot 0.6 \cdot 0.7 + 0.1 \cdot 0.6 \cdot 0.7 = \\ & 0.588 \end{aligned}$$

A *ProbLog program* is composed by a set of normal clauses and a set of probabilistic facts, possibly non-ground. A probabilistic fact takes the form

$$\Pi :: f.$$

where Π is in [0,1] and f is an atom. The semantics of such program can be given by considering an equivalent LPAD containing, for each ProbLog normal

clause $h : -B$, a clause $h : 1 : -B$ and, for each probabilistic ProbLog fact, a clause

$$f : \Pi.$$

The semantics of the ProbLog program is the same as that of the equivalent LPAD.

It is also possible to translate an LPAD into a ProbLog program [4]. A clause C_i of the LPAD with variables $\overline{X}$

$$h_{i1} : \Pi_{i1}; \ldots; h_{in_i} : \Pi_{in_i} : -B_i$$

is translated into

$$\begin{array}{l} h_{i1} : -B_i, f_{i1}(\overline{X}). \\ h_{i2} : -B_i, problog\_not(f_{i1}(\overline{X})), f_{i2}(\overline{X}). \\ \vdots \\ h_{in_i-1} : -B_i, problog\_not(f_{i1}(\overline{X})), \ldots, problog\_not(f_{in_i-2}(\overline{X})), f_{in_i-1}(\overline{X}). \\ h_{in_i} : -B_i, problog\_not(f_{i1}(\overline{X})), \ldots, problog\_not(f_{in_i-1}(\overline{X})). \\ \\ \pi_{i1} :: f_{i1}(\overline{X}). \\ \vdots \\ \pi_{in-1} :: f_{in_i-1}(\overline{X}). \end{array}$$

where $problog\_not/1$ is a ProbLog builtin predicate that implements negation for probabilistic atoms and $\pi_{i1} = \Pi_{i1}$, $\pi_{i2} = \frac{\Pi_{i2}}{1-\pi_{i1}}$, $\pi_{i3} = \frac{\Pi_{i3}}{(1-\pi_{i1})(1-\pi_{i2})}, \ldots$. In general $\pi_{ij} = \frac{\Pi_{ij}}{\prod_{k=1}^{j-1}(1-\pi_{ik})}$.

Example 2. The ProbLog program equivalent to the LPAD of Example 1 is

$$\begin{array}{l} C_{11} = epidemic : -flu(X), cold, f1(X). \\ C_{12} = pandemic : -flu(X), cold, problog\_not(f1(X)), f2(X). \\ C_{13} = 0.6 :: f1(X). \\ C_{14} = 0.75 :: f2(X). \\ C_{21} = cold : -f3. \\ C_{22} = 0.7 :: f3. \\ C_3 \ = flu(david). \\ C_4 \ = flu(robert). \end{array}$$

3 Inference Algorithms

In order to compute the probability of a query from a probabilistic logic program, [6] proposed the ProbLog system that first finds a set of explanations for the query and then computes the probability from the set by using Binary Decision Diagrams. An explanation is a set of probabilistic facts used in a derivation of the query. The set of explanations can be seen as a Boolean DNF formula in which

the Boolean propositions are random variables. Computing the probability of the formula involves solving the disjoint sum problem which is #P-complete [19]. BDDs represent an approach for solving this problem that has been shown to work well in practice [6,13,14].

[8] proposed various approaches for approximate inference that are now included in the ProbLog system. The k-best algorithm finds only the k most probable explanations for a query and then builds a BDD from them. The resulting probability is only a lower bound but if k is sufficiently high it represents a good approximation. The bounded approximation algorithm computes a lower bound and an upper bound of the probability of the query by using iterative deepening to explore the SLD tree for the query. The SLD tree is built partially, the successful derivations it contains are used to build a BDD for computing the lower bound while the successful derivations plus the incomplete ones are used to compute the upper bound. If the difference between the upper and the lower bound is above the required precision, the SLD tree is built up to a greater depth. This process is repeated until the required precision is achieved. These algorithms are implemented by means of a program transformation technique applied to the probabilistic atoms: they are turned into clauses that, when the atom is called, add the probabilistic fact to the current explanation.

[1] presented an implementation of k-best and bounded approximation for LPADs that is based on a meta-interpreter and showed that in some cases this gives good results.

[8] also presented a Monte Carlo algorithm that samples the possible programs and tests the query in the samples. The probability of the query is then given by the fraction of programs where the query is true. The Monte Carlo algorithm for ProbLog is realized by using an array with an element for each ground probabilistic fact that stores one of three values: sampled true, sampled false and not yet sampled. When a probabilistic fact is called, the algorithm first checks whether the fact has already been sampled by looking at the array. If it has not been sampled, then it samples it and stores the result in the array. Probabilistic facts that are non-ground in the program are treated differently. A position in the array is not reserved for them since their grounding is not known at the start, rather samples for groundings of these facts are stored in the internal database of Yap and the sampled value is retrieved when they are called. If no sample has been taken for a grounding, a sample is taken and recorded in the database.

[1] presents a Monte Carlo algorithm for LPADs that is based on a meta-interpreter. In order to keep track of the samples taken, two arguments of the meta-interpreter predicate are used, one for keeping the input set of choices and one for the output set of choices. This algorithm is included in the `cplint` suite available in the source tree of Yap[4].

[4] `http://www.dcc.fc.up.pt/~vsc/Yap/downloads.html`

4 MCINTYRE

MCINTYRE first transforms the program and then queries the transformed program. The disjunctive clause $C_i = h_{i1} : \Pi_{i1} \vee \ldots \vee h_{in} : \Pi_{in_i} : -b_{i1}, \ldots, b_{im_i}.$ where the parameters sum to 1, is transformed into the set of clauses $MC(C_i)$:

$$\begin{aligned} MC(C_i, 1) = \; & h_{i1} : -b_{i1}, \ldots, b_{im_i}, \\ & sample\_head(ParList, i, VC, NH), NH = 1. \\ \ldots & \\ MC(C_i, n_i) = \; & h_{in_i} : -b_{i1}, \ldots, b_{im_i}, \\ & sample\_head(ParList, i, VC, NH), NH = n_i. \end{aligned}$$

where VC is a list containing each variable appearing in C_i and $ParList$ is $[\Pi_{i1}, \ldots, \Pi_{in_i}]$. If the parameters do not sum up to 1 the last clause (the one for *null*) is omitted. Basically, we create a clause for each head and we sample a head index at the end of the body with `sample_head/4`. If this index coincides with the head index, the derivation succeeds, otherwise it fails. Thus failure can occur either because one of the body literals fails or because the current clause is not part of the sample.

For example, clause C_1 of epidemic example becomes

$$\begin{aligned} MC(C_1, 1) = \; & epidemic : -flu(X), cold, \\ & sample\_head([0.6, 0.3, 0.1], 1, [X], NH), NH = 1. \\ MC(C_1, 2) = \; & pandemic : -flu(X), cold, \\ & sample\_head([0.6, 0.3, 0.1], 1, [X], NH), NH = 2. \end{aligned}$$

The predicate `sample_head/4` samples an index from the head of a clause and uses the builtin Yap predicates `recorded/3` and `recorda/3` for respectively retrieving or adding an entry to the internal database. Since `sample_head/4` is at the end of the body and since we assume the program to be range restricted, at that point all the variables of the clause have been grounded. A program is range restricted if all the variables appearing in the head also appear in positive literals in the body. If the rule instantiation had already been sampled, `sample_head/4` retrieves the head index with `recorded/3`, otherwise it samples a head index with `sample/2`:

```
sample_head(_ParList,R,VC,NH):-
  recorded(exp,(R,VC,NH),_),!.
sample_head(ParList,R,VC,NH):-
  sample(ParList,NH),
  recorda(exp,(R,VC,NH),_).

sample(ParList, HeadId) :-
  random(Prob),
  sample(ParList, 0, 0, Prob, HeadId).

sample([HeadProb|Tail], Index, Prev, Prob, HeadId) :-
  Succ is Index + 1,
  Next is Prev + HeadProb,
  (Prob =< Next ->
```

```
    HeadId = Index
  ;
    sample(Tail, Succ, Next, Prob, HeadId)
  ).
```

Tabling can be effectively used to avoid re-sampling the same atom. To take a sample from the program we use the following predicate

```
sample(Goal):-
  abolish_all_tables,
  eraseall(exp),
  call(Goal).
```

A fixed number of samples n is taken and the fraction $\hat{p}$ of samples in which the query succeds is computed. In order to compute the confidence interval of $\hat{p}$, we use the central limit theorem to approximate the binomial distribution with a normal distribution. Then the 95% binomial proportion confidence interval is calculated as

$$\hat{p} \pm z_{1-\alpha/2}\sqrt{\frac{\hat{p}\,(1-\hat{p})}{n}}$$

where $z_{1-\alpha/2}$ is the $1-\alpha/2$ percentile of a standard normal distribution and usually $\alpha = 0.05$. If the width of the interval is below a user defined threshold δ, we stop and we return the fraction of successful samples.

This estimate of the confidence interval is good for a sample size larger than 30 and if $\hat{p}$ is not too close to 0 or 1. The normal approximation fails totally when the sample proportion is exactly zero or exactly one. Empirically, it has been observed that the normal approximation works well as long as $n\hat{p} > 5$ and $n(1-\hat{p}) > 5$.

5 Experiments

We considered three sets of benchmarks: graphs of biological concepts from [6], artificial datasets from [9] and a hidden Markov model from [2]. On these dataset, we compare MCINTYRE, the Monte Carlo algorithm of ProbLog [8], the Monte Carlo algorithm of `cplint` [1] and the exact system PITA which has been shown to be particularly fast [14]. All the experiments have been performed on Linux machines with an Intel Core 2 Duo E6550 (2333 MHz) processor and 4 GB of RAM. The algorithms were run on the data for 24 hours or until the program ended for lack of memory. $\delta = 0.01$ was chosen as the maximum confidence interval width for Monte Carlo algorithms. The normal approximation tests $n\hat{p} > 5$ and $n(1-\hat{p}) > 5$ were disabled in MCINTYRE because they are not present in ProbLog. For each experiment we used tabling when it gave better results.

In the graphs of biological concepts the nodes encode biological entities such as genes, proteins, tissues, organisms, biological processes and molecular functions, and the edges conceptual and probabilistic relations among them. Edges

are thus represented by ground probabilistic facts. The programs have been sampled from the Biomine network [18] containing 1,000,000 nodes and 6,000,000 edges. The sampled programs contain 200, 400, ..., 10000 edges. Sampling was repeated ten times, to obtain ten series of programs of increasing size. In each program we query the probability that the two genes HGNC_620 and HGNC_983 are related.

For MCINTYRE and ProbLog we used the following definition of path

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

For MCINTYRE, we tabled `path/2` using Yap tabling with the directive

```
:- table path/2.
```

while for ProbLog we tabled the path predicate by means of ProbLog tabling with the command

```
problog_table(path/2),
```

For PITA we used the program

```
path(X,Y):-path(X,Y,[X],Z).
path(X,X,A,A).
path(X,Y,A,R):-X\==Y, arc(X,Z), \+ member(Z,A), path(Z,Y,[Z|A],R).
arc(X,Y):-edge(Y,X).
arc(X,Y):-edge(X,Y).
```

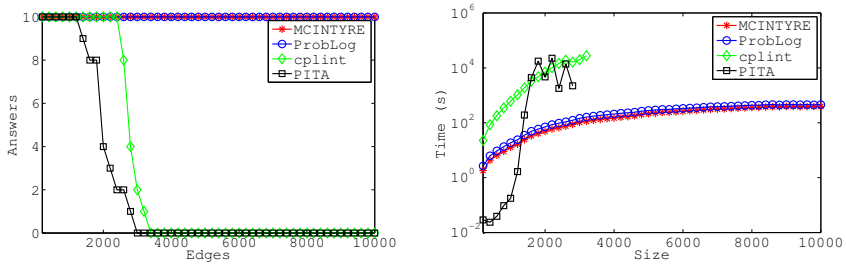
that performs loop checking by keeping a list of visited nodes rather than by using tabling because this approach gave the best results. We used the same program also for `cplint` because it does not allow to use tabling for loop checking.

Figure 1(a) shows the number of graphs for each size for which MCINTYRE, ProbLog, `cplint` and PITA were able to compute the probability. Figure 1(b) shows the execution times of the four algorithms as a function of graph size averaged over the graphs on which the algorithms succeeded.

MCINTYRE and ProbLog were able to solve all graphs, while PITA and `cplint` stopped much earlier. As regards speed, MCINTYRE is much faster than `cplint` and slightly faster than ProbLog. For non-small programs it is also faster than PITA.

The growing head dataset from [9] contains propositional programs in which the head of clauses are of increasing size. For example, the program for size 4 is

```
a0 :- a1.
a1:0.5.
a0:0.5; a1:0.5 :- a2.
a2:0.5.
a0:0.333333333333; a1:0.333333333333; a2:0.333333333333 :- a3.
a3:0.5.
```



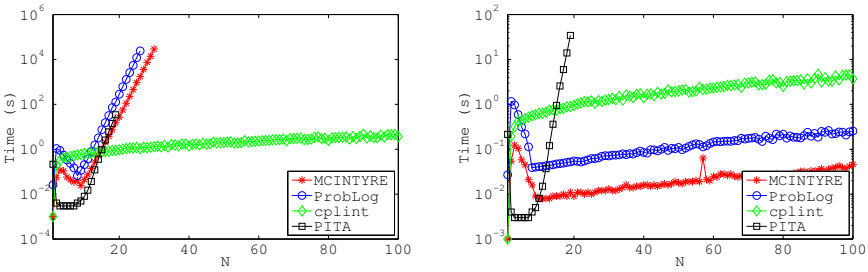
(a) Solved graphs. (b) Average execution times .

Fig. 1. Biological graph experiments.

The equivalent ProbLog program is

```
a0 :- a1.                    0.5::a1f.
a1:-a1f.                     0.5::a0_2.
a0:-a2,a0_2.
a1:-a2,problog_not(a0_2).    0.5::a2f.
a2:-a2f.
0.333333333333::a0_3.        0.5::a1_3.
a0:-a3,a0_3.
a1:-a3,problog_not(a0_3),a1_3.
a2:-a3,problog_not(a0_3),problog_not(a1_3).
0.5::a3f.
a3:-a3f.
```

In this dataset no predicate is tabled for both MCINTYRE and ProbLog. Figure 2(a) shows the time for computing the probability of `a0` as a function of the size. MCINTYRE is faster than ProbLog and PITA for non-small programs but all of them are much slower and less scalable than `cplint`. The reason why



(a) Sampling last. (b) Sampling first.

Fig. 2. Growing head from [9].

cplint performs so well is that the meta-interpreter checks for the consistency of the sample when choosing a clause to resolve with the goal, rather than after having resolved all the body literals as in MCINTYRE and ProbLog. However, since the clauses are ground, the sampling predicates of MCINTYRE can be put at the beginning of the body, simulating cplint behavior. Similarly, the probabilistic atoms can be put at the beginning of the body of ProbLog clauses. With this approach, we get the timings depicted in Figure 2(b) which shows that now MCINTYRE and ProbLog are faster than cplint and MCINTYRE is the fastest.

The blood type dataset from [9] determines the blood type of a person on the basis of her chromosomes that in turn depend on those of her parents. The blood type is given by clauses of the form

```
bloodtype(Person,a):0.90 ; bloodtype(Person,b):0.03 ;
bloodtype(Person,ab):0.03 ; bloodtype(Person,null):0.04 :-
  pchrom(Person,a),mchrom(Person,a).
```

where pchrom/2 indicates the chromosome inherited from the father and mchrom /2 that inherited from the mother. There is one such clause for every combination of the values {a, b, null} for the father and mother chromosomes. In turn, the chromosomes of a person depend from those of her parents, with clauses of the form

```
mchrom(Person,a):0.90 ; mchrom(Person,b):0.05 ;
mchrom(Person,null):0.05 :-
  mother(Mother,Person), pchrom(Mother,a), mchrom(Mother,a).
```

There is one such clause for every combination of the values {a, b, null} for the father and mother chromosomes of the mother and similarly for the father chromosome of a person. In this dataset we query the blood type of a person on the basis of that of its ancestors. We consider families with an increasing number of components: each program adds two persons to the previous one. The chromosomes of the parentless ancestors are given by disjunctive facts of the form

```
mchrom(p,a):0.3 ; mchrom(p,b):0.3 ; mchrom(p,null):0.4.
pchrom(p,a):0.3 ; pchrom(p,b):0.3 ; pchrom(p,null):0.4.
```

For both MCINTYRE and ProbLog all the predicates are tabled.

Figure 3 shows the execution times as a function of the family size. Here MCINTYRE is faster than ProbLog but slower than the exact inference of PITA. This is probably due to the fact that the bodies of clauses with the same atoms in the head are mutually exclusive in this dataset and the goals in the bodies are independent, making BDD operations particularly fast.

In the growing body dataset [9] the clauses have bodies of increasing size. For example, the program for size 4 is,

```
a0:0.5 :-  a1.
```

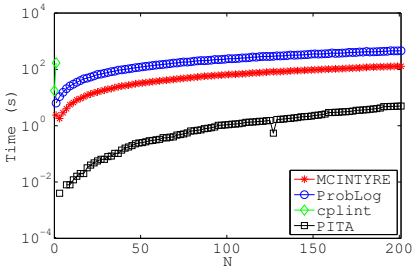
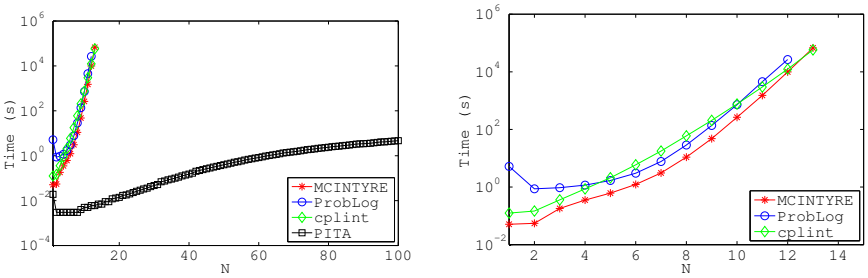


Fig. 3. [Bloodtype from [9].

```
a0:0.5 :- \+ a1,  a2.
a0:0.5 :- \+ a1, \+ a2,  a3.
a1:0.5 :-  a2.
a1:0.5 :- \+ a2,  a3.
a2:0.5 :-  a3.
a3:0.5.
```

In this dataset as well no predicate is tabled for both MCINTYRE and ProbLog and the sampling predicates of MCINTYRE and the probabilistic atoms of ProbLog have been put at the beginning of the body since the clauses are ground. Figure 4(a) shows the execution time for computing the probability of `a0`. Here PITA is faster and more scalable than Monte Carlo algorithms, again probably due to the fact that the bodies of clauses with the same heads are mutually exclusive thus simplifying BDD operations. Figure 4(b) shows the execution time of the Monte Carlo algorithms only, where it appears that MCINTYRE is faster than ProbLog and `cplint`.

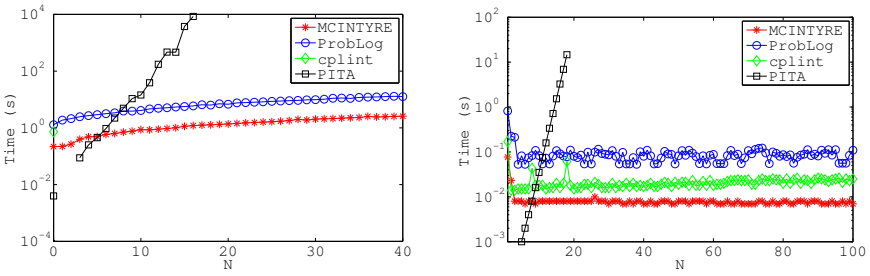


(a) All algorithms. (b) Monte Carlo algorithms.

Fig. 4. Growing body. from [9].

The UWCSE dataset [9] describes a university domain with predicates such as `taught_by/2`, `advised_by/2`, `course_level/2`, `phase/2`, `position/2`, `course/1`,

professor/1, student/1 and others. Programs of increasing size are considered by adding facts for the student/1 predicate, i.e., by considering an increasing number of students. For both MCINTYRE and ProbLog all the predicates are tabled. The time for computing the probability of the query taught_by(c1,p1) as a function of the number of students is shown in Figure 5(a). Here MCINTYRE is faster than ProbLog and both scale much better than PITA.



(a) UWCSE. (b) Hidden Markov Model.

Fig. 5. UWCSE and Hidden Markov Model

The last experiment involves the Hidden Markov model for DNA sequences from [2]: bases are the output symbols and three states are assumed, of which one is the end state. The following program generates base sequences.

```
hmm(O):-hmm1(_,O).
hmm1(S,O):-hmm(q1,[],S,O).
hmm(end,S,S,[]).
hmm(Q,S0,S,[L|O]):- Q\= end, next_state(Q,Q1,S0), letter(Q,L,S0),
  hmm(Q1,[Q|S0],S,O).
next_state(q1,q1,_S):1/3;next_state(q1,q2,_S):1/3;
  next_state(q1,end,_S):1/3.
next_state(q2,q1,_S):1/3;next_state(q2,q2,_S):1/3;
  next_state(q2,end,_S):1/3.
letter(q1,a,_S):0.25;letter(q1,c,_S):0.25;letter(q1,g,_S):0.25;
  letter(q1,t,_S):0.25.
letter(q2,a,_S):0.25;letter(q2,c,_S):0.25;letter(q2,g,_S):0.25;
  letter(q2,t,_S):0.25.
```

The algorithms are used to compute the probability of hmm(O) for random sequences O of increasing length. Tabling was not used for MCINTYRE nor for ProbLog.

Figure 5(b) show the time taken by the various algorithms as a function of the sequence length. Since the probability of such a sequence goes rapidly to zero, the derivations of the goal terminate mostly after a few steps only and

all Monte Carlo algorithms take constant time with MCINTYRE faster that ProbLog and `cplint`.

6 Conclusions

Probabilistic Logic Programming is of high interest for its many application fields. The distribution semantics is one of the most popular approaches to PLP and underlies many languages, such as LPADs and ProbLog. However, exact inference is very expensive, being #P complete and thus approximate approaches have to be investigated. In this paper we propose the algorithm MCINTYRE that performs approximate inference by means of a Monte Carlo technique, namely random sampling. MCINTYRE transforms an input LPAD into a normal program that contains a clause for each head of an LPAD clause. The resulting clauses contain in the body auxiliary predicates that perform sampling and check for the consistency of the sample.

MCINTYRE has been tested on graphs of biological concepts, on four artificial datasets from [9] and on a hidden Markov model. In all cases it turned out to be faster than the Monte Carlo algorithms of ProbLog and `cplint`. It is also faster and more scalable than exact inference except in two datasets, blood type and growing body, that however possess peculiar characteristics. MCINTYRE is available in the `cplint` package of the source tree of Yap and instructions on its use are available at `http://www.ing.unife.it/software/cplint/`.

In the future we plan to investigate other approximate inference techniques such as lifted belief propagation and variational methods.

References

1. Bragaglia, S., Riguzzi, F.: Approximate inference for logic programs with annotated disjunctions. In: International Conference on Inductive Logic Programming. LNAI, vol. 6489, pp. 30–37. Springer (2011)
2. Christiansen, H., Gallagher, J.P.: Non-discriminating arguments and their uses. In: International Conference on Logic Programming. LNCS, vol. 5649, pp. 55–69. Springer (2009)
3. Dantsin, E.: Probabilistic logic programs and their semantics. In: Russian Conference on Logic Programming. LNCS, vol. 592, pp. 152–164. Springer (1991)
4. De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., Kimmig, A., Landwehr, N., Mantadelis, T., Meert, W., Rocha, R., Santos Costa, V., Thon, I., Vennekens, J.: Towards digesting the alphabet-soup of statistical relational learning. In: Roy, D., Winn, J., McAllester, D., Mansinghka, V., Tenenbaum, J. (eds.) Proceedings of the 1st Workshop on Probabilistic Programming: Universal Languages, Systems and Applications, in NIPS (2008)
5. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic Inductive Logic Programming - Theory and Applications, LNCS, vol. 4911. Springer (2008)
6. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic prolog and its application in link discovery. In: International Joint Conference on Artificial Intelligence. pp. 2462–2467. AAAI Press (2007)

7. Getoor, L., Taskar, B. (eds.): Introduction to Statistical Relational Learning. MIT Press (2007)
8. Kimmig, A., Demoen, B., De Raedt, L., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming 11(2-3), 235–262 (2011)
9. Meert, W., Struyf, J., Blockeel, H.: CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In: International Conference on Inductive Logic Programming. LNCS, vol. 5989, pp. 96–109. Springer (2010)
10. Poole, D.: Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. New Generation Computing 11(3-4), 377–400 (1993)
11. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artificial Intelligence 94(1-2), 7–56 (1997)
12. Riguzzi, F.: A top-down interpreter for LPAD and CP-Logic. In: Congress of the Italian Association for Artificial Intelligence. LNCS, vol. 4733, pp. 109–120. Springer (2007)
13. Riguzzi, F.: Extended semantics and inference for the Independent Choice Logic. Logic Journal of the IGPL 17(6), 589–629 (2009)
14. Riguzzi, F., Swift, T.: Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In: International Conference on Logic Programming. LIPIcs, vol. 7, pp. 162–171. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2010)
15. Riguzzi, F., Swift, T.: An extended semantics for logic programs with annotated disjunctions and its efficient implementation. In: Italian Conference on Computational Logic. No. 598 in CEUR Workshop Proceedings, Sun SITE Central Europe (2010)
16. Riguzzi, F., Swift, T.: The PITA system: Tabling and answer subsumption for reasoning under uncertainty. Theory and Practice of Logic Programming, International Conference on Logic Programming Special Issue 11(4-5) (2011)
17. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: International Conference on Logic Programming. pp. 715–729. MIT Press (1995)
18. Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., Toivonen, H.: Link discovery in graphs derived from biological databases. In: International Workshop on Data Integration in the Life Sciences. LNCS, vol. 4075, pp. 35–49. Springer (2006)
19. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM Journal on Computing 8(3), 410–421 (1979)
20. Vennekens, J., Verbaeten, S.: Logic programs with annotated disjunctions. Tech. Rep. CW386, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (2003)
21. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: International Conference on Logic Programming. LNCS, vol. 3131, pp. 195–209. Springer (2004)