MCINTYRE: A Monte Carlo System for Probabilistic Logic Programming

Fabrizio Riguzzi
Dipartimento di Matematica e Informatica, 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. The current sample is stored in the internal database of the Yap Prolog engine. The resulting system, 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 systems.

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 allows logic to deal also with uncertain propositions and probability theory to considers complex relational descriptions of domain entities.

PLP is of interest for many application domains, such as biological networks [11], environmental assessment [14] or ontology engineering [32]. The most promising domain is Probabilistic Inductive

Address for correspondence: Dipartimento di Matematica e Informatica, Università di Ferrara, Via Saragat, 1, 44122 Ferrara, Italy, fabrizio.riguzzi@unife.it, Tel/Fax +390532974836

Logic Programming [10] in which PLP languages are used to represent the theories that are induced from data. This allows a rich 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 [15].

Many languages have been proposed in PLP. Among them, many share a common approach for defining the semantics, namely the so called distribution semantics [40]. 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 (in chronological order) Probabilistic Logic Programs [8], Probabilistic Horn Abduction [21], Independent Choice Logic [22], PRISM [40], pD [13], Logic Programs with Annotated Disjunctions (LPADs) [46], ProbLog [11] and CP-logic [44]. These languages have essentially the same expressive power [45, 9] 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 systems 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 [19]. Nevertheless, various exact inference algorithms have been proposed, such as the ones in the systems PRISM[1] [41], ProbLog [2] [11], `cplint`[3] [26, 29, 28, 30] and PITA[4] [35, 36, 37, 38, 31] and have been successfully applied to a variety of non-trivial problems. All of these algorithms find explanations for queries and then all except PRISM use Binary Decision Diagrams (BDDs) for computing the probability. This approach has been shown to be faster than previous algorithms. Recently, weighted model counting using deterministic, decomposable negation normal forms has been applied with success to inference in PLP [12].

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 [24, 25, 27, 23, 16, 18, 33, 3, 1, 2, 4]. 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 [19] 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 [19] 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 [19] for ProbLog, where a lazy sampling approach was used in order to avoid sampling unnecessary probabilistic facts. Bragaglia and Riguzzi [5] present 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 system 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

[1] `http://sato-www.cs.titech.ac.jp/prism/`
[2] `http://dtai.cs.kuleuven.be/problog/`
[3] `http://sites.unife.it/ml/cplint`
[4] `https://sites.unife.it/ml/pita`

body for taking samples and storing the results. The internal database of the Yap Prolog engine[5] [7] 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 PITA 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 system. 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 [40], 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 [40, 34, 38].

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 [45, 9]. We will discuss here LPADs and ProbLog because LPADs have the most liberal syntax and ProbLog the most restrictive, thus they can be considered as the two ends of a spectrum.

2.1. Logic Programs with Annotated Disjunctions

Formally a *Logic Program with Annotated Disjunctions* [46] T 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 $body(C_i)$. If it is empty, the $:\!-$ symbol is omitted. 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.

[5] `http://www.dcc.fc.up.pt/~vsc/Yap/`

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 $k \in \{1, \dots, n_i\}$. 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 $\forall i, j, k, j, l\ (C_i, \theta_j, k) \in \kappa, (C_i, \theta_j, l) \in \kappa \Rightarrow k = l$. A *composite choice* κ is a consistent set of atomic choices. The *probability* $P(\kappa)$ *of a composite choice* κ is $P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik}$. 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 $w_\sigma = \{(h_{ik} \text{ :- } body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. w_σ is called a *world* of T. Since selections are composite choices, we can assign a probability to possible worlds: $P(w_\sigma) = P(\sigma)$.

The programs we consider do not have function symbols so the set of worlds is finite: $W_T = \{w_1, \dots, w_m\}$. 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 worlds:

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

Example 2.1. The following LPAD T encodes a very simple model of the development of an epidemic or a pandemic:

$$\begin{aligned} C_1 &= epidemic : 0.6 \;;\; pandemic : 0.3 \text{ :- } flu(X), cold. \\ C_2 &= cold : 0.7. \\ C_3 &= flu(david). \\ C_4 &= flu(robert). \end{aligned}$$

This program models the fact that, if somebody has the flu and the climate is cold, there is the possibility that an epidemic arises, a pandemic arises or neither of the two. 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

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

2.2. ProbLog

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 $h \text{ :- } B$, a clause $h : 1 \text{ :- } B$ 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 [9]. A clause C_i of the LPAD with variables $\overline{X}$ $h_{i1} : \Pi_{i1}; \dots ; h_{in_i} : \Pi_{in_i} \text{ :- } B_i$ 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 built-in 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 $\pi_{ij} = \frac{\Pi_{ij}}{\prod_{k=1}^{j-1}(1-\pi_{ik})}$. Recent versions of ProbLog allow clauses with annotated disjunctive heads as well [17] and treat them by translating into basic ProbLog as illustrated above.

Example 2.2. The ProbLog program equivalent to the LPAD of Example 2.1 is

$$\begin{array}{lcl}
C_{11} & = & epidemic :- flu(X), cold, f_1(X).\\
C_{12} & = & pandemic :- flu(X), cold, problog\_not(f_1(X)), f_2(X).\\
C_{13} & = & 0.6 :: f_1(X).\\
C_{14} & = & 0.75 :: f_2(X).\\
C_{21} & = & cold :- f_3.\\
C_{22} & = & 0.7 :: f_3.\\
C_3 & = & flu(david).\\
C_4 & = & flu(robert).
\end{array}$$

In this program, clause C_1 is translated into two clauses, C_{11} and C_{12}, one for each head of C_1. The first, C_{11}, has a positive probabilistic literal for which the program contains the probabilistic fact $0.6 :: f_1(X)$, the latter instead, C_{12}, contains a negative probabilistic literal and a positive one, for which the program contains the probabilistic fact $0.75 :: f_2(X)$. Thus, the head $pandemic$ is derived from clause C_{12} when $f_1(X)$ is false and $f_2(X)$ is true. This happens with probability $0.4 \cdot 0.75 = 0.3$, the probability associated to $pandemic$ in C_1.

3. Inference Algorithms

One of the first systems for computing the probability of a query from a probabilistic logic program was PRISM [41] that uses tabling to find derivations for the query. PRISM, however, requires goals in a disjunction to be mutually exclusive and goals in a conjunction to be independent, strong requirements that are not easy to satisfy.

De Raedt et al. [11] proposed the ProbLog system that overcomes these limitations by first finding a set of explanations for the query and then computing 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 [43]. BDDs represent an approach for solving this problem that has been shown to work well in practice [11, 29, 35].

Kimmig et al. [19] 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 represents a good approximation if k is sufficiently high. 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: these are turned into clauses that add the probabilistic fact to the current explanation.

Bragaglia and Riguzzi [5] 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. They also presented 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.

ProbLog [19] also contains 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. Figure 1 shows the overall algorithm: a fixed number of samples n is taken and the fraction $\hat{p}$ of samples in which the query succeeds is computed. In order to compute the confidence interval of $\hat{p}$, the central limit theorem is used to approximate the binomial distribution with a normal distribution. Then the binomial proportion confidence interval is calculated as [39] $\hat{p} \pm z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{Samples}}$ where $Samples$ is the number of samples, $z_{1-\alpha/2}$ is the $1-\alpha/2$ percentile of a standard normal distribution (usually $\alpha = 0.05$ so $z_{1-\alpha/2} = 1.96$). If the width of the interval is below a user defined threshold δ, the algorithm stops and returns the fraction of successful samples, otherwise another batch of n samples is taken. In Figure 1 SAMPLE(Q) is used to take a sample of the program and to test the query in the sample. The algorithm converges because the $Samples$ variables is always increasing and thus the condition $2z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{Samples}} < \delta$ in line 15 of Figure 1 will eventually become true.

Sampling in ProbLog is realized by asking the query over a transformed program in which the probabilistic facts are replaced by rules. Moreover, ProbLog uses an array with an element for each ground probabilistic fact that stores one of three values: sampled true, sampled false or not yet sampled. When a literal matching a probabilistic fact is called, ProbLog 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: 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. No position in the array is reserved for them since their grounding is not known at the start.

```
function MonteCarlo($T, Q, n, \delta$)
    Input: Program $T$, query $Q$, number of batch samples $n$, precision $\delta$
    Output: $P(Q)$
    Transform $T$
    $Samples \leftarrow 0$
    $TrueSamples \leftarrow 0$
    repeat
        for $i = 1 \rightarrow n$ do
            $Samples \leftarrow Samples + 1$
            if SAMPLE($Q$) succeeds then
                $TrueSamples \leftarrow TrueSamples + 1$
            end if
        end for
        $\hat{p} \leftarrow \frac{TrueSamples}{Samples}$
    until $2z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{Samples}} < \delta$
    return $\hat{p}$
end function
```

Figure 1. Monte Carlo algorithm.

4. MCINTYRE

MCINTYRE follows the algorithm in Figure 1 and differs from ProbLog in the transformation (line 4 in Figure 1), in the sampling process (line 10) and in the exit condition in the loop (line 15).

MCINTYRE applies to *range restricted* programs, i.e., programs in which all the variables appearing in the head of a clause also appear in positive literals in the body. MCINTYRE applies the following transformation: 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 $MC(C_i)$:

$$MC(C_i, 1) = \quad h_{i1} :- b_{i1}, \ldots, b_{im_i}, sample\_head([\Pi_{i1}, \ldots, \Pi_{in_i}], i, V, NH), NH = 1.$$

$$\ldots$$

$$MC(C_i, n_i) = \quad h_{in_i} :- b_{i1}, \ldots, b_{im_i}, sample\_head([\Pi_{i1}, \ldots, \Pi_{in_i}], i, V, NH), NH = n_i.$$

where V is a list containing each variable appearing in C_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 Example 2.1 becomes

$$MC(C_1, 1) = \quad epidemic :- flu(X), cold, sample\_head([0.6, 0.3, 0.1], 1, [X], NH), NH = 1.$$

$$MC(C_1, 2) = \quad pandemic :- flu(X), cold, sample\_head([0.6, 0.3, 0.1], 1, [X], NH), NH = 2.$$

The predicate `sample_head/4` samples an index from the head of a clause and uses the built-in 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 programs to be range restricted, at that point all the variables of the clause have been grounded. 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(samples,(R,VC,NH),_),!.
sample_head(ParList,R,VC,NH):-
  sample(ParList,NH),
  recorda(samples,(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)
  ).
```

Thus `sample_head/4` samples a new head only if one had not been sampled for the ground clause under consideration. If a head had already been sampled, then the index of the head is retrieved from the Yap internal database. In this way, at most one head is sampled for each ground clauses and the sample obtained is consistent. It is not necessary to sample the head of clauses not involved in the derivation as whatever sample is taken this does not influence the success or failure of the goal.

It is often convenient to anticipate as much as possible the sampling and comparison predicates in the body so that if a different head was sampled for that clause the derivation stops early. The sampling and comparison predicates can be called as soon as all the variables in the clause have been instantiated. In Section 5 we show an application of this technique.

Tabling can be used in the transformed program since it does not interfere with the sampling process: in fact, even if the result of calls to the `random/1` predicate are non deterministic, the samples are taken only once for each grounding of each clause.

To take a sample from the program we use the following predicate

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

For example, if the query is $epidemic$, resolution matches the goal with the head of clause $MC(C_1, 1)$. Suppose $flu(X)$ succeeds with $X/david$ and $cold$ succeeds as well. Then $sample\_head([0.6, 0.3, 0.1],$ $1, [david], NH)$ is called. Since clause 1 with X replaced by $david$ has not yet been sampled, a number between 1 and 3 is sampled according to the distribution in $[0.6, 0.3, 0.1]$ and stored in NH. If $NH = 1$, the derivation succeeds and the goal is true in the sample, if $NH = 2$ or $NH = 3$ then the derivation fails and backtracking is performed. This involves finding the solution $X/robert$ for $flu(X)$. $cold$ was sampled as true before so it remains true, now $sample\_head([0.6, 0.3, 0.1], 1, [robert], NH)$ is called to take another sample.

MCINTYRE takes also into account the validity of the binomial proportion confidence interval. The normal approximation is good for a sample size larger than 30 and if $\hat{p}$ is not too close to 0 or 1, while it 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 $Sample \cdot \hat{p} > 5$ and $Sample \cdot (1 - \hat{p}) > 5$ [39]. Thus MCINTYRE changes the condition in line 15 of Figure 1 to

$$2z_{1-\alpha/2}\sqrt{\frac{\hat{p}\,(1-\hat{p})}{Samples}} < \delta \wedge Samples \cdot \hat{p} > 5 \wedge Samples \cdot (1-\hat{p}) > 5$$

The differences between MCINTYRE and ProbLog thus regard both the algorithms and the implementations. As regards the algorithms, to deal with LPADs (i.e., clauses with more than two heads), the clauses are translated into ProbLog by introducing $n-1$ Boolean variables if the clause has n heads (see Example 2.2). Then ProbLog samples these Boolean variables and has to sample a different number of variables depending on the clause, while MCINTYRE always samples a single integer variable between 1 and n. As regards the implementation, ProbLog uses an array for ground probabilistic facts instead of the Yap internal database and a larger number of predicate calls to sample a value.

5. Experiments

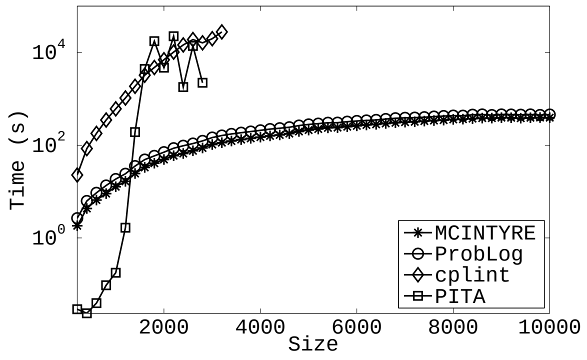
We considered three sets of benchmarks: graphs of biological concepts from [11], artificial datasets from [20][6] and a hidden Markov model from [6]. On these dataset, we mainly compare MCINTYRE with the Monte Carlo algorithm of ProbLog [19] but for reference we also report the results of the Monte Carlo algorithm of `cplint` [5] and the exact system PITA which has been shown to be particularly fast [35]. For dataset whose ProbLog version contains ground probabilistic facts we also compare MCINTYRE with a manually crafted version of ProbLog that we call ProbLogNG and that treats all facts as non ground, to highlight the differences in the two implementations.

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 $Samples \cdot \hat{p} > 5$ and $Samples \cdot (1 - \hat{p}) > 5$ were disabled in MCINTYRE because they are not present in ProbLog. For each experiment we used tabling when it gave better results.

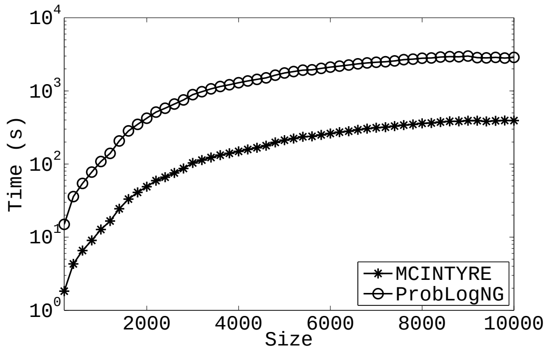
In the graphs of biological concepts, the nodes encode biological entities such as genes, proteins, tissues, organisms, biological processes and molecular functions, and the edges conceptual and probabilistic relations among them. Edges are thus represented by ground probabilistic facts. The programs have been sampled from the Biomine network [42] 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

[6]Available at `http://dtai.cs.kuleuven.be/cplve/ilp09/`



(a) Average execution times.



(b) Average execution times for MCINTYRE and ProbLogNG only.

Figure 2. Biological graph experiments.

```
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 2(a) shows the execution times of the four algorithms as a function of graph size averaged over the graphs on which the algorithms succeeded. Table 1 shows the average execution times of MCINTYRE and ProbLog/ProbLogNG in tabular form, together with the ratio between MCINTYRE time and ProbLog/ProbLogNG time.

MCINTYRE and ProbLog are able to solve all graphs, while PITA and `cplint` stop much earlier. MCINTYRE and ProbLog are much faster than `cplint` and than PITA from size 1400 onwards. MCINTYRE is faster than ProbLog but its gain reduces with the size of the graphs: MCINTYRE time goes from 68% to 86% of the ProbLog time.

Figure 2(b) and Table 1 show the comparison between MCINTYRE and ProbLogNG. As can be seen, the difference between MCINTYRE and ProbLog is much greater, showing that the MCINTYRE implementation is much leaner and that it could also benefit from the use of an array for ground probabilistic facts. Also here the gain factor reduces with the size of the graphs.

The growing head dataset from [20] contains propositional programs in which the head of clauses are of increasing size. For example, the program for size 3 is

Size	MCINTYRE	ProbLog	M/P	ProbLogNG	M/PNG
1000	12.72	18.61	0.6835	108.36	0.1174
2000	49.05	70.53	0.6955	423.54	0.1158
3000	103.47	145.74	0.7100	889.41	0.1163
4000	149.02	208.81	0.7137	1293.01	0.1153
5000	211.76	280.69	0.7544	1760.60	0.1203
6000	261.70	332.96	0.7860	2116.15	0.1237
7000	313.94	389.94	0.8051	2469.80	0.1271
8000	360.09	434.99	0.8278	2801.63	0.1285
9000	392.26	459.40	0.8539	2996.42	0.1309
10000	393.26	458.23	0.8582	2875.38	0.1368

Table 1. Average execution times for MCINTYRE, ProbLog and ProbLogNG on the biological graphs. Columns M/P and M/PNG report the ratio of the MCINTYRE time over the ProbLog and ProbLogNG time respectively.

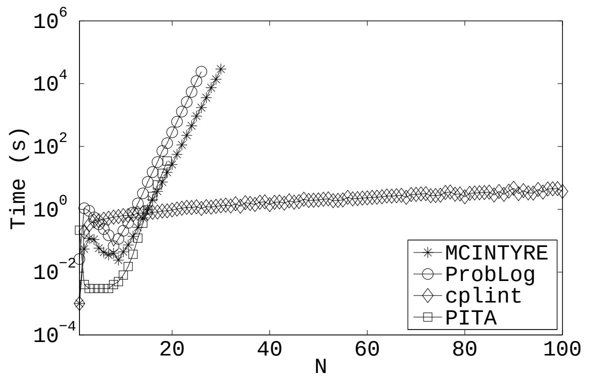


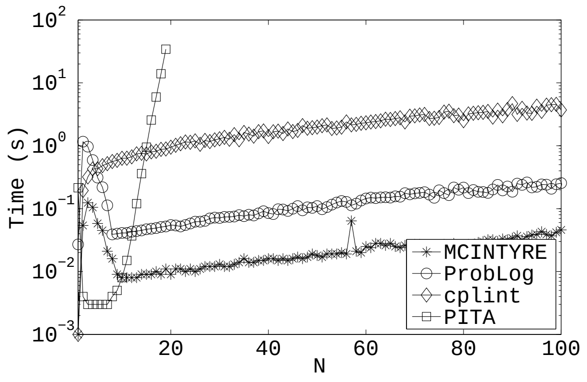
Figure 3. Execution times for the growing head dataset, sampling last.

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

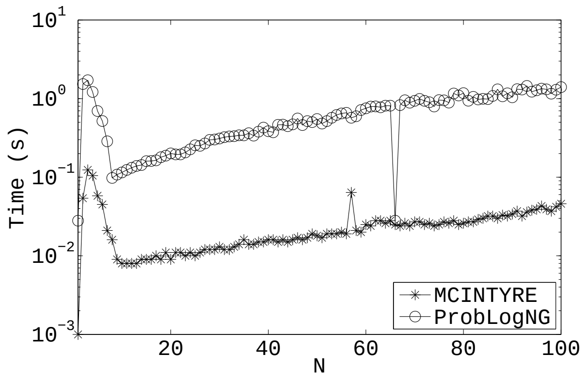
The equivalent ProbLog program is

```
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.
```

In this dataset no predicate is tabled for both MCINTYRE and ProbLog. Figure 3 shows the time for computing the probability of `a0` as a function of the size. MCINTYRE is faster than ProbLog and PITA but all of them are much slower and less scalable than `cplint`. The reason why `cplint` performs so well is that the meta-interpreter checks for the consistency of the sample when choosing a clause



(a) Execution times.



(b) Execution times for MCINTYRE and ProbLogNG only.

Figure 4. Growing head dataset, sampling first.

Size	MCINTYRE	ProbLog	M/P	ProbLogNG	M/PNG
10	0.008	0.041	0.1951	0.115	0.0696
20	0.009	0.055	0.1636	0.201	0.0448
30	0.013	0.071	0.1831	0.311	0.0418
40	0.016	0.085	0.1882	0.385	0.0416
50	0.018	0.110	0.1636	0.550	0.0327
60	0.025	0.145	0.1724	0.754	0.0332
70	0.027	0.175	0.1543	0.938	0.0288
80	0.026	0.216	0.1204	1.174	0.0221
90	0.034	0.186	0.1828	1.040	0.0327
100	0.046	0.255	0.1804	1.403	0.0328

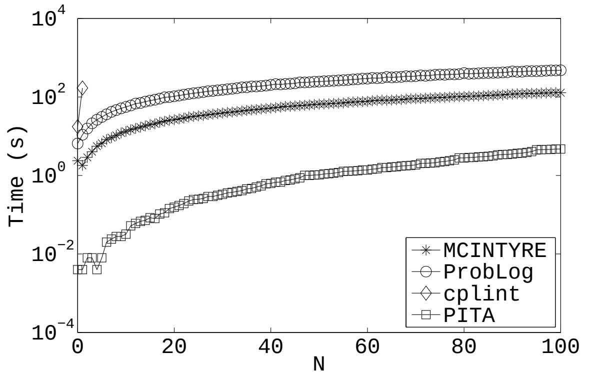
Table 2. Execution times for MCINTYRE, ProbLog and ProbLogNG on the growing head dataset with sampling first. Columns M/P and M/PNG report the ratio of the MCINTYRE time over the ProbLog and ProbLogNG time respectively.

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 4(a). Table 2 and Figure 4(b) compare MCINTYRE with ProbLog and ProbLogNG. Now MCINTYRE and ProbLog are faster than `cplint`. MCINTYRE is also faster than ProbLog and ProbLogNG by a constant factor, taking 17.04% of ProbLog and 3.80% of ProbLogNG time on average.

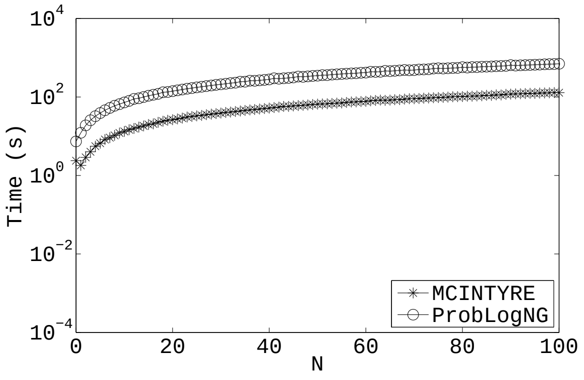
The blood type dataset from [20] determines the blood type of a person on the basis of her chromosomes that in turn depend on those of her parents. The blood type is given by clauses of the form

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

where `P` stands for a person and `pchrom/2` indicates the chromosome inherited from the father and



(a) Execution times.



(b) Execution times for MCINTYRE and ProbLogNG only.

Figure 5. Blood type dataset .

`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(P,a):0.90 ; mchrom(P,b):0.05 ; mchrom(P,null):0.05 :-
  mother(Mother,P), 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 parent-less 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.

Figures 5(a) and 5(b) shows the execution times as a function of the family size. Here MCINTYRE is faster than ProbLog and ProbLogNG but slower than the exact inference of PITA. This is probably due to the fact that, in this dataset, the bodies of clauses with the same atoms in the head are mutually exclusive and the goals in the bodies are independent, making BDD operations particularly fast. Table 3 shows the execution times of MCINTYRE, ProbLog and ProbLogNG together with time ratios: MCINTYRE is faster than ProbLog/ProbLogNG by a nearly constant factor.

In the growing body dataset [20] the clauses have bodies of increasing size. For example, the program for size 4 is,

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

Size	MCINTYRE	ProbLog	M/P	ProbLogNG	M/PNG
10	13.2520	55.7350	0.2378	72.7920	0.1821
20	26.2930	104.2250	0.2523	140.9390	0.1866
30	38.9870	152.1470	0.2562	209.9120	0.1857
40	52.0210	203.2610	0.2559	279.5900	0.1861
50	65.9060	245.9480	0.2680	349.4330	0.1886
60	77.3020	293.8190	0.2631	418.8950	0.1845
70	90.1520	339.3580	0.2657	484.8050	0.1860
80	102.2910	406.9530	0.2514	572.9660	0.1785
90	118.1940	447.7490	0.2640	647.6960	0.1825
100	128.9810	480.3920	0.2685	700.9790	0.1840

Table 3. Execution times for MCINTYRE, ProbLog and ProbLogNG on the blood type dataset. Columns M/P and M/PNG report the ratio of the MCINTYRE time over the ProbLog and ProbLogNG time respectively.

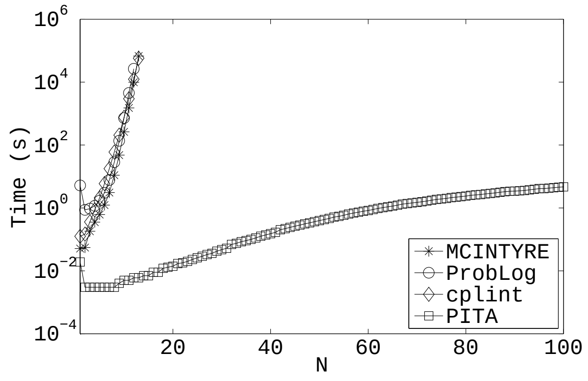


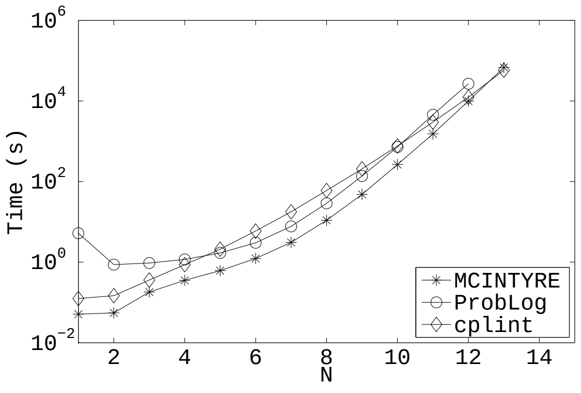
Figure 6. Execution times for the growing body dataset.

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.

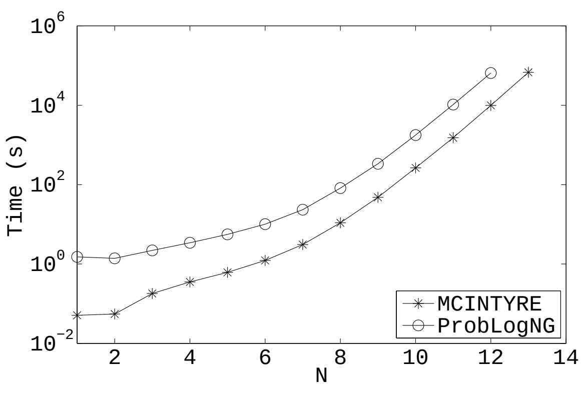
Figure 6 shows the execution time for computing the probability of `a0`. Here PITA is faster and more scalable than Monte Carlo algorithms, again probably due to the fact that the bodies of clauses with the same heads are mutually exclusive thus simplifying BDD operations. Figure 7(a) shows the execution time of the Monte Carlo algorithms only, where it appears that MCINTYRE is faster than ProbLog and `cplint`. Figure 7(b) compares MCINTYRE and ProbLogNG. Looking a the times of MCINTYRE, ProbLog and ProbLogNG in Table 4, we can observe that again MCINTYRE is faster by a roughly constant factor.

The UWCSE dataset [20] 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. The definitions of these predicates take 36 clauses[7] Programs of increasing size are consid-

[7]Available at `http://dtai.cs.kuleuven.be/cplve/ilp09/`

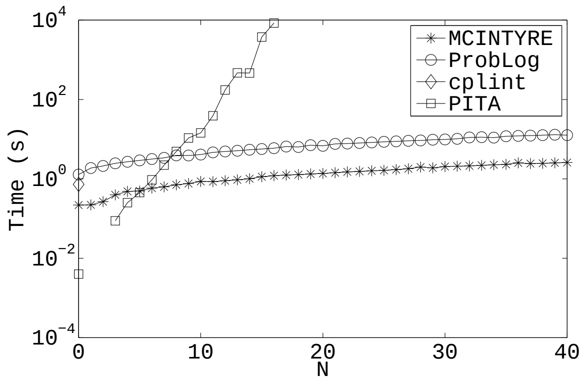


(a) Execution times for Monte Carlo algorithms only.

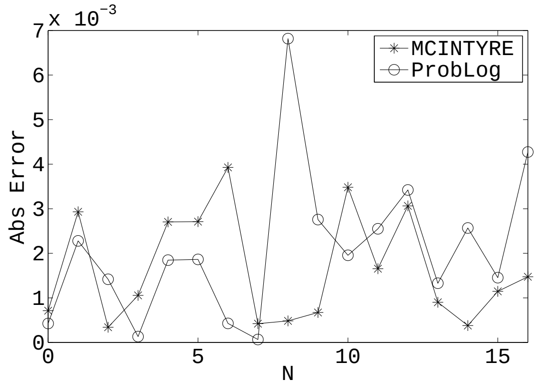


(b) Execution times for MCINTYRE and ProbLogNG only.

Figure 7. Growing body dataset.



(a) Execution times.



(b) Absolute error.

Figure 8. UWCSE dataset.

ered by adding facts for the `student/1` predicate, i.e., by considering an increasing number of students. All the predicates are tabled for both MCINTYRE and ProbLog.

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 8(a) and Table 5[8]. This is a difficult dataset in which exact inference fails for more than 16 students due to a lack of memory error. An idea of the complexity can be obtained by counting the average number of clauses sampled during a single sample call: for 0 students, it is 2.734, then it grows linearly up to 47.342 for 40 students. Here again MCINTYRE is faster than ProbLog by a roughly constant factor and both scale much better than PITA.

Figure 8(b) shows the absolute error of the probability computed by MCINTYRE and ProbLog as a function of the number of students, i.e., the quantity $AE(Q) = |P_{mc}(Q) - P_e(Q)|$, where $P_{mc}(Q)$ is the probability of the query computed by the Monte Carlo algorithm and $P_e(Q)$ is the probability of the

[8]ProbLogNG is not shown as there are no ground probabilistic facts in the ProbLog version of this dataset, so there is no difference with plain ProbLog.

Size	MCINTYRE	ProbLog	M/P	ProbLogNG	M/PNG
1	0.051	5.201	0.0099	1.514	0.0337
2	0.055	0.870	0.0632	1.391	0.0395
3	0.181	0.949	0.1907	2.205	0.0821
4	0.354	1.165	0.3039	3.428	0.1033
5	0.615	1.691	0.3637	5.581	0.1102
6	1.231	3.016	0.4082	10.082	0.1221
7	3.086	7.722	0.3996	23.321	0.1323
8	10.938	28.683	0.3813	81.929	0.1335
9	47.986	137	0.3503	335.73	0.1429
10	264	721	0.3658	1772	0.1488
11	1518	4542	0.3343	10418	0.1457
12	9950	26617	0.3738	64725	0.1537
13	67609	-	-	-	-

Table 4. Execution times for MCINTYRE, ProbLog and ProbLogNG on the growing body dataset. Columns M/P and M/PNG report the ratio of the MCINTYRE time over the ProbLog and ProbLogNG time respectively.

query computed by an exact method, in this case PITA. The figure shows that the error is always below the 0.01 value of the threshold δ, thus demonstrating that convergence to the correct value was always achieved.

The last experiment involves the Hidden Markov model for DNA sequences from [6]: bases are 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.
```

Basically, a sequence is generated starting from state `q1` with the call `hmm(q1,[],S,O)`. `hmm/4` then stops returning the empty symbol list if the state is `end`, otherwise it samples a new state and a new letter (output symbol) and calls itself recursively.

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 9 and Table 6[9] 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, all Monte Carlo algorithms stop after the first batch of samples and thus take constant time with MCINTYRE faster that ProbLog and `cplint`.

[9]ProbLogNG is not shown as there no ground probabilistic facts in the ProbLog version of this dataset.

Size	MCINTYRE	ProbLog	M/P
5	0.494	2.922	0.1690
10	0.862	4.106	0.2099
15	1.137	5.657	0.2010
20	1.375	6.815	0.2018
25	1.643	8.590	0.1913
30	2.043	9.868	0.2070
35	2.315	11.859	0.1952
40	2.588	12.633	0.2049

Table 5. Execution times for MCINTYRE and ProbLog on the UWCSE dataset. Column M/P reports the ratio of the MCINTYRE time over the ProbLog time.

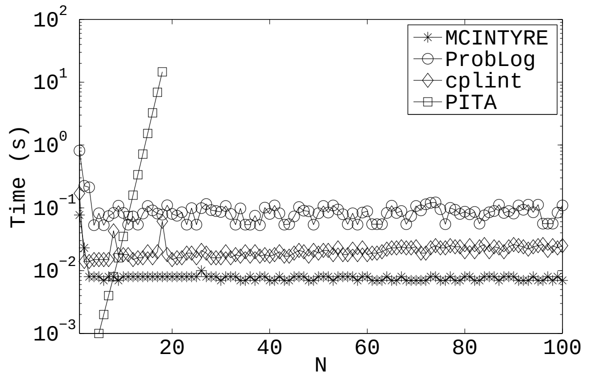


Figure 9. Execution times on the Hidden Markov Model dataset.

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 system 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 auxiliary predicates in the body 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 [20] and on a hidden Markov model. In all cases it turned out to be faster than the Monte Carlo algorithms of `cplint` and ProbLog. The comparison with the latter shows that an ad hoc system can have better performances than a system that encompasses various types of exact and approximate inference.

MCINTYRE is also faster and more scalable than exact inference except in two datasets, blood type and growing body, that however possess peculiar characteristics. This shows that approximate inference

Size	MCINTYRE	ProbLog	M/P
10	0.008	0.083	0.0964
20	0.008	0.080	0.1000
30	0.007	0.086	0.0814
40	0.007	0.080	0.0875
50	0.008	0.082	0.0976
60	0.008	0.088	0.0910
70	0.007	0.107	0.0654
80	0.008	0.087	0.0920
90	0.008	0.081	0.0988
100	0.007	0.109	0.0642

Table 6. Execution times for MCINTYRE and ProbLog on the HMM dataset. Column M/P reports the ratio of the MCINTYRE time over the ProbLog time.

can be more convenient when the accuracy in the probability of the query is not of foremost importance.

MCINTYRE is available in the `cplint` package of the source tree of Yap and instructions on its use are available at `http://sites.unife.it/ml/cplint`.

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

References

[1] Bellodi, E., Riguzzi, F.: EM over Binary Decision Diagrams for Probabilistic Logic Programs, *Italian Conference on Computational Logic*, vol. 810 of *CEUR Workshop Proceedings*, Sun SITE Central Europe, 2011, ISSN 1613-0073.

[2] Bellodi, E., Riguzzi, F.: Experimentation of an Expectation Maximization Algorithm for Probabilistic Logic Programs, *Intelligenza Artificiale*, **8**(1), 2012, 3–18, doi:10.3233/IA-2012-0027.

[3] Bellodi, E., Riguzzi, F.: Learning the Structure of Probabilistic Logic Programs, *International Conference on Inductive Logic Programming*, vol. 7207 of *LNCS*, Springer, 2012, doi:10.1007/978-3-642-31951-8_10.

[4] Bellodi, E., Riguzzi, F.: Expectation Maximization over Binary Decision Diagrams for Probabilistic Logic Programs, *Intelligent Data Analysis*, **17**(2), 2013.

[5] Bragaglia, S., Riguzzi, F.: Approximate Inference for Logic Programs with Annotated Disjunctions, *International Conference on Inductive Logic Programming*, vol. 6489 of *LNCS*, Springer, 2011, doi:10.1007/978-3-642-21295-6_7.

[6] Christiansen, H., Gallagher, J. P.: Non-discriminating Arguments and Their Uses, *International Conference on Logic Programming*, vol. 5649 of *LNCS*, Springer, 2009, doi:10.1007/978-3-642-02846-5_10.

[7] Costa, V. S., Damas, L., Rocha, R.: The YAP Prolog System, *Theory and Practice of Logic Programming*, **12**(1-2), 2012, 5–34, doi:10.1017/S1471068411000512.

[8] Dantsin, E.: Probabilistic Logic Programs and their Semantics, *Russian Conference on Logic Programming*, vol. 592 of *LNCS*, Springer, 1991, doi:10.1007/3-540-55460-2_11.

[9] 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, *Workshop on Probabilistic Programming: Universal Languages, Systems and Applications, in NIPS* (D. Roy, J. Winn, D. McAllester, V. Mansinghka, J. Tenenbaum, Eds.), 2008.

[10] De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S., Eds.: *Probabilistic Inductive Logic Programming - Theory and Applications*, vol. 4911 of *LNCS*, Springer, 2008.

[11] De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A Probabilistic Prolog and Its Application in Link Discovery., *International Joint Conference on Artificial Intelligence*, AAAI Press, 2007.

[12] Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., De Raedt, L.: Inference in probabilistic logic programs using weighted CNF's, *Conference on Uncertainty in Artificial Intelligence*, AUAI Press, 2011.

[13] Fuhr, N.: Probabilistic datalog: Implementing logical information retrieval for advanced applications, *Journal of the American Society for Information Science*, **51**(2), 2000, 95–110.

[14] Gavanelli, M., Riguzzi, F., Milano, M., Cagnoli, P.: Logic-Based Decision Support for Strategic Environmental Assessment, *Theory and Practice of Logic Programming, International Conference on Logic Programming Special Issue*, **10**(4-6), July 2010, 643–658, doi:10.1017/S1471068410000335.

[15] Getoor, L., Taskar, B., Eds.: *Introduction to Statistical Relational Learning*, MIT Press, 2007.

[16] Gutmann, B., Kimmig, A., Kersting, K., Raedt, L. D.: Parameter Learning in Probabilistic Databases: A Least Squares Approach, *European Conference on Machine Learning*, vol. 5211 of *LNCS*, Springer, 2008, doi:10.1007/978-3-540-87479-9_49.

[17] Gutmann, B., Thon, I., Kimmig, A., Bruynooghe, M., De Raedt, L.: The magic of logical inference in probabilistic programming, *Theory and Practice of Logic Programming*, **11**(4-5), 2011, 663–680, doi:10.1017/S1471068411000238.

[18] Gutmann, B., Thon, I., Raedt, L. D.: Learning the Parameters of Probabilistic Logic Programs from Interpretations, *European Conference on Machine Learning and Knowledge Discovery in Databases*, vol. 6911 of *LNCS*, Springer, 2011, doi:10.1007/978-3-642-23780-5_47.

[19] 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), 2011, 235–262, doi:10.1017/S1471068410000566.

[20] Meert, W., Struyf, J., Blockeel, H.: CP-Logic Theory Inference with Contextual Variable Elimination and Comparison to BDD Based Inference Methods, *International Conference on Inductive Logic Programming*, vol. 5989 of *LNCS*, Springer, 2010, doi:10.1007/978-3-642-13840-9_10.

[21] Poole, D.: Logic Programming, Abduction and Probability - A Top-Down Anytime Algorithm for Estimating Prior and Posterior Probabilities, *New Generation Computing*, **11**(3-4), 1993, 377–400, doi:10.1007/BF03037184.

[22] Poole, D.: The Independent Choice Logic for Modelling Multiple Agents under Uncertainty, *Artificial Intelligence*, **94**(1-2), 1997, 7–56, doi:10.1016/S0004-3702(97)00027-1.

[23] Raedt, L. D., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic Prolog programs, *Machine Learning*, **70**(2-3), 2008, 151–168, doi:10.1007/s10994-007-5030-x.

[24] Riguzzi, F.: Learning Logic Programs with Annotated Disjunctions, *International Conference on Inductive Logic Programming*, vol. 3194 of *LNCS*, Springer, September 2004, doi:10.1007/978-3-540-30109-7_21.

[25] Riguzzi, F.: ALLPAD: Approximate Learning of Logic Programs with Annotated Disjunctions, *International Conference on Inductive Logic Programming*, vol. 4455 of *LNCS*, Springer, 2007, doi:10.1007/978-3-540-73847-3_11.

[26] Riguzzi, F.: A Top Down Interpreter for LPAD and CP-logic, *Congress of the Italian Association for Artificial Intelligence*, vol. 4733 of *LNCS*, Springer, 2007, doi:10.1007/978-3-540-74782-6_11.

[27] Riguzzi, F.: ALLPAD: Approximate Learning of Logic Programs with Annotated Disjunctions, *Machine Learning*, **70**(2-3), March 2008, 207–223, doi:10.1007/s10994-007-5032-8.

[28] Riguzzi, F.: Inference with Logic Programs with Annotated Disjunctions under the Well Founded Semantics, *International Conference on Logic Programming*, vol. 5366 of *LNCS*, Springer, 2008, doi:10.1007/978-3-540-89982-2_54.

[29] Riguzzi, F.: Extended Semantics and Inference for the Independent Choice Logic, *Logic Journal of the IGPL*, **17**(6), 2009, 589–629, doi:10.1093/jigpal/jzp025.

[30] Riguzzi, F.: SLGAD Resolution for Inference on Logic Programs with Annotated Disjunctions, *Fundamenta Informaticae*, **102**(3-4), October 2010, 429–466, doi:10.3233/FI-2010-392.

[31] Riguzzi, F.: Optimizing Inference for Probabilistic Logic Programs Exploiting Independence and Exclusiveness, *Italian Conference on Computational Logic*, vol. 857 of *CEUR Workshop Proceedings*, Sun SITE Central Europe, 2012, ISSN 1613-0073.

[32] Riguzzi, F., Bellodi, E., Lamma, E.: Probabilistic Datalog+/- under the Distribution Semantics, *International Workshop on Description Logics*, vol. 846 of *CEUR Workshop Proceedings*, Sun SITE Central Europe, 2012, ISSN 1613-0073.

[33] Riguzzi, F., Di Mauro, N.: Applying the Information Bottleneck to Statistical Relational Learning, *Machine Learning*, **86**(1), 2012, 89–114, doi:10.1007/s10994-011-5247-6.

[34] Riguzzi, F., Swift, T.: An Extended Semantics for Logic Programs with Annotated Disjunctions and its Efficient Implementation, *Italian Conference on Computational Logic*, vol. 598 of *CEUR Workshop Proceedings*, Sun SITE Central Europe, Aachen, Germany, 2010, ISSN 1613-0073.

[35] Riguzzi, F., Swift, T.: Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions, *International Conference on Logic Programming*, vol. 7 of *LIPIcs*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, July 2010, doi:10.4230/LIPIcs.ICLP.2010.162.

[36] 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, 433–449, doi:10.1017/S147106841100010X.

[37] Riguzzi, F., Swift, T.: The PITA System for Logical-Probabilistic Inference, *Latest Advances in Inductive Logic Programming, Inductive Logic Programming, 21th International Conference*, Imperial College Press, 2012.

[38] Riguzzi, F., Swift, T.: Well-Definedness and Efficient Inference for Probabilistic Logic Programming under the Distribution Semantics, *Theory and Practice of Logic Programming, Convegno Italiano di Logica Computazionale Special Issue*, 2013, doi:10.1017/S1471068411000664.

[39] Ryan, T. P.: *Modern Engineering Statistics*, John Wiley & Sons, 2007.

[40] Sato, T.: A Statistical Learning Method for Logic Programs with Distribution Semantics, *International Conference on Logic Programming*, MIT Press, 1995.

[41] Sato, T., Kameya, Y.: Parameter Learning of Logic Programs for Symbolic-Statistical Modeling, *Journal of Artificial Intelligence Research*, **15**, 2001, 391–454.

[42] Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., Toivonen, H.: Link Discovery in Graphs Derived from Biological Databases, *International Workshop on Data Integration in the Life Sciences*, vol. 4075 of *LNCS*, Springer, 2006, doi:10.1007/11799511_5.

[43] Valiant, L. G.: The Complexity of Enumeration and Reliability Problems, *SIAM Journal on Computing*, **8**(3), 1979, 410–421, doi:10.1137/0208032.

[44] Vennekens, J., Denecker, M., Bruynooghe, M.: CP-logic: A language of causal probabilistic events and its relation to logic programming, *Theory and Practice of Logic Programming*, **9**(3), 2009, 245–308, doi:10.1017/S1471068409003767.

[45] Vennekens, J., Verbaeten, S.: *Logic Programs With Annotated Disjunctions*, Technical Report CW386, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2003.

[46] Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions, *International Conference on Logic Programming*, vol. 3131 of *LNCS*, Springer, 2004, doi:10.1007/978-3-540-27775-0_30.