Approximate Inference for Logic Programs with Annotated Disjunctions

Stefano Bragaglia and Fabrizio Riguzzi

DEIS – University of Bologna, ENDIF – University of Ferrara.
{stefano.bragaglia@unibo.it, fabrizio.riguzzi@unife.it}

Abstract. Logic Programs with Annotated Disjunctions (LPADs) are a promising language for Probabilistic Inductive Logic Programming. In order to develop efficient learning systems for LPADs, it is fundamental to have high-performing inference algorithms. The existing approaches take too long or fail for large problems. In this paper we adapt to LPAD the approaches for approximate inference that have been developed for ProbLog, namely k-best and Monte Carlo.
k-Best finds a lower bound of the probability of a query by identifying the k most probable explanations while Monte Carlo estimates the probability by smartly sampling the space of programs. The two techniques have been implemented in the `cplint` suite and have been tested on real and artificial datasets representing graphs. The results show that both algorithms are able to solve larger problems often in less time than the exact algorithm.

Keywords: Probabilistic Inductive Logic Programming, Logic Programs with Annotated Disjunctions, ProbLog.

1 Introduction

Statistical Relational Learning and Probabilistic Inductive Logic Programming provide successful techniques for learning from real world data. Such techniques usually require the execution of a high number of inferences in probabilistic logics, which are costly tasks. In order to reduce the computational load, we may resort to approximate inference that trades accuracy for speed. In this paper we present two approaches for computing the probability of queries from Logic Programs with Annotated Disjunctions (LPADs) [6] in an approximate way. LPADs are particularly interesting because of their sound semantics, of their intuitive syntax and because they allow to exploit many of the techniques developed in Logic Programming for probabilistic reasoning. We present two approaches inspired by those available for ProbLog [2]: k-best and Monte Carlo. The first finds a lower bound for the probability of a query by considering only the k most probable explanations, while the latter estimates the probability of the query by the fraction of sampled possible worlds where the query is true.

2 Logic Programs with Annotated Disjunctions

A Logic Programs with Annotated Disjunctions T [6] consists of a finite set of disjunctive clauses of the form $(H_1 : \alpha_1) \vee (H_2 : \alpha_2) \vee \ldots \vee (H_n : \alpha_n) \leftarrow B_1, B_2, \ldots B_m$ called *annotated disjunctive clauses*. The H_i, B_i and α_i that appear in such a clause are respectively logical atoms, logical literals and real numbers in the interval $[0, 1]$ such that $\sum_{i=1}^{n} \alpha_i \leq 1$. If $\sum_{i=1}^{n} \alpha_i < 1$, 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 $1 - \sum_{i=1}^{n} \alpha_i$. For a clause C of the form above, we define $head(C)$ as $\{(H_i : \alpha_i)|1 \leq i \leq n\}$ if $\sum_{i=1}^{n} \alpha_i = 1$ and as $\{(H_i : \alpha_i)|1 \leq i \leq n\} \cup \{(null : 1 - \sum_{i=1}^{n} \alpha_i)\}$ otherwise. Moreover, we define $body(C)$ as $\{B_i|1 \leq i \leq m\}$, $H_i(C)$ as H_i and $\alpha_i(C)$ as α_i.

In order to define the semantics of an LPAD T, we need to consider its grounding $ground(T)$ that must be finite, so T must not contain function symbols if it contains variables. More specifically, an *atomic choice* is a triple (C, θ, i) where $C \in T$, θ is a substitution for the variables of C and $i \in \{1, \ldots, |head(C)|\}$ meaning that the head $H_i(C)\theta : \alpha_i(C)$ was chosen for the clause $C\theta$. A *composite choice* κ is a set of atomic choices that are ground ($C\theta$ is ground) and consistent ($(C, \theta, i) \in \kappa, (C, \theta, j) \in \kappa \Rightarrow i = j$, meaning that only one head is selected for a ground clause) whose probability $P(\kappa)$ is given by $P(\kappa) = \prod_{(C,\theta,i) \in \kappa} \alpha_i(C)$. A *selection* σ is a composite choice containing an atomic choice (C, θ, i) in σ for each clause $C\theta$ in $ground(T)$ and identifies a normal logic program w_σ called a *possible world* (or simply *world*) of T and defined as follows $w_\sigma = \{(H_i(C)\theta \leftarrow body(C))\theta|(C, \theta, i) \in \sigma\}$.

$\mathcal{W}_T$ denotes the set of all the possible worlds of T. Since selections are composite choices, we can assign a probability to possible worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i) \in \sigma} \alpha_i(C)$. The probability of a closed formula ϕ according to an LPAD T is given by the sum of the probabilities of the possible worlds where the formula is true according to the WFS: $P(\phi) = \sum_{\sigma \in \mathcal{W}_T, w_\sigma \models \phi} P(\sigma)$. It is easy to see that P satisfies the axioms of probability.

In order to compute the probability of a query from a probabilistic logic program, [6] proposed to first find a covering set of explanations for the query and then compute the probability from the set by using Binary Decision Diagrams. An *explanation* is a composite choice κ such that the query is true in all the possible worlds consistent with κ. A set K of explanations is *covering* if each possible world where the query is true is consistent with at least one of the explanations in K.

The `cplint` system[1] [5] applied this approach to LPADs. `cplint` first computes a covering set of explanations for a query by using a Prolog meta-interpreter that performs resolution and keeps a set of atomic choices that represents a partial explanation. Each time the meta-interpreter resolves the selected goal with a disjunctive clause, it adds a (possibly non-ground) atomic choice to the partial explanation and checks for its consistency. If the program is range-restricted, when the meta-interpreter reaches the empty goal, every atomic choice in the

[1] `http://www.ing.unife.it/software/cplint/`

partial explanation becomes ground and an explanation is obtained. By enclosing the meta-interpreter in a `findall` call, a covering set K of explanations is found. Then `cplint` converts K into the following Disjunctive Normal Form (DNF) logical formula $F = \bigvee_{\kappa \in K} \bigwedge_{(C,\theta,i)\in\kappa} (X_{C\theta} = i)$. The probability of the query is then given by the probability of F taking value 1. F is converted to a Decision Diagram that is traversed by using a dynamic programming algorithm to compute the probability. Specifically, `cplint` uses Binary Decision Diagram (BDD) because of the availability of highly efficient packages for processing them. Since disjunctive clauses may contain any number of logical heads, multivalued variables are binary encoded by means of boolean variables to be used in BDDs.

3 Approximate Inference

In some domains, computing exactly the probability of a query may be impractical and it may be necessary to resort to some forms of approximations. [2,3] proposed various approaches for approximate inference. With *iterative deepening*, upper and lower bounds for the probability of the query are computed and their difference is gradually decreased by increasing the portion of the search tree that is explored. With the *k-best* algorithm, only the k most probable explanations are considered and a lower bound is found. With *Monte Carlo*, the possible worlds are sampled and the query is tested in the samples. An estimate of the probability of the query is given by the fraction of sampled worlds where the query succeeds. All three approaches have been adapted to LPADs and included in `cplint`. In the following we report only on the k-best and Monte Carlo, since iterative deepening was not giving clear advantages with respect to exact inference on the datasets tested.

3.1 *k*-best Algorithm

According to [3], using a fixed number of proofs to approximate the probability is fundamental when many queries have to be evaluated because it allows to control the overall complexity. The k-best algorithm uses branch and bound to find the k most probable explanations, where k is a user-defined parameter. The algorithm records the k best explanations. Given a partial explanation, its probability (obtained by multiplying the probability of each atomic choice it contains) is an upper bound on the probability that a complete explanation extending it can achieve. Therefore, a partial explanation can be pruned if its probability falls below the probability of the k-th best explanation. Our implementation of the k-best algorithm interleaves tree expansion and pruning: a set of partial explanations are kept and are iteratively expanded for some steps. Those whose upper bound is worse than the k-th best explanation are pruned. Once the proof tree has been completely expanded, the k best explanations are translated into a BDD to compute a lower bound of the probability of the query. This solution uses a meta-interpreter while ProbLog uses a form of iterative deepening that builds derivations up to a certain probability threshold and then increases the

Algorithm 1 Function SOLVE

```
function SOLVE($Goal$, $Explan$)
    if $Goal$ is empty then
        return 1
    else
        Let $Goal = [G|Tail]$
        if $G =(\backslash + \ Atom)$ then
            $Valid$ :=solve($[Atom]$, $Explan$)
            if $Valid = 0$ then
                return solve($Tail$, $Explan$)
            else
                return 0
            end if
        else
            Let $L$ be the list of couples $(GL, Step)$ where $GL$ is obtained by resolving
            $Goal$ on $G$ with a program clause $C$ on head $i$ with substitution $\theta$
            and $Step = (C, \theta, i)$
            return SAMPLE_CYCLE($L$, $Explan$)
        end if
    end if
end function
```

threshold if k explanations have not been found. The meta-interpreter approach has the advantages of avoiding to repeat resolution steps at the expense of a more complex bookkeeping.

3.2 Monte Carlo Algorithm

In [3] the Monte Carlo algorithm for ProbLog is realized by using a vector with an entry for every probabilistic fact. The entries store whether the facts have been sampled true, sampled false or not yet sampled. The vector is initialized with not yet sampled for all facts. Then a transformed ProbLog program is executed that derives the goal and updates the vector each time a new probabilistic fact is sampled.

ProbLog's algorithm requires all the probabilistic facts to be ground in the input program. While LPADs can be converted to ProbLog programs [1], the result of the conversion may contain non ground probabilistic facts so ProbLog's Monte Carlo algorithm may not always be used.

Our Monte Carlo algorithm for LPADs uses a meta-interpreter that keeps a partial explanation containing atomic choices for the disjunctive clauses sampled up to that point. The meta-interpreter is realized by Function SOLVE in Algorithm 1 and returns 1 if the list of atoms of the goal is derivable in the sample and 0 otherwise. In order to derive the selected literal G of the current goal, SOLVE finds all the matching clauses and builds a list of couples (new goal, atomic choice) for each matching clause. Then, it calls Function SAMPLE_CYCLE in Algorithm 2 whose aim is to perform sampling steps for the matching clauses

Algorithm 2 Function SAMPLE_CYCLE

```
 1: function SAMPLE_CYCLE($L, Explan$)
 2:     $Derivable = 0$
 3:     while $Derivable = 0$ and $L \neq \emptyset$ do
 4:         Remove the first element $(GL, (C, \theta, i))$ from $L$
 5:         repeat
 6:             if $C\theta$ is ground then
 7:                 if $(C, \theta)$ is already present in $Explan$ with head $j$ then
 8:                     $h := j$
 9:                 else
10:                     $h :=$SAMPLE($C$)
11:                 end if
12:             else
13:                 $h :=$SAMPLE($C$)
14:             end if
15:             $Explan := Explan \cup \{(C, \theta, h)\}$
16:             if $h = i$ then
17:                 $Derivable :=$SOLVE($GL, Explan$)
18:             else
19:                 $Derivable := 0$
20:             end if
21:         until CONSISTENT($Explan$)
22:     end while
23:     return $Derivable$
24: end function
```

until the truth of the selected literal is determined and a consistent set of ground atomic choices is obtained. Each matching clause is sampled independently and the resulting atomic choice is added to the partial explanation that is passed by reference to future calls of SOLVE and SAMPLE_CYCLE.

Since a matching clause may be sampled when it is still not completely ground, further grounding/sampling may lead to inconsistency in the partial explanation. To address this problem, sampling is repeated until a consistent partial explanation is found. The algorithm is guaranteed to terminate because the same head will be eventually sampled for each couple of identical groundings of a clause. Also note that the sampling distribution is not affected since inconsistency arises independently of the success or failure of a query.

In Algorithm 2, CONSISTENT($Explan$) returns true if $Explan$ is consistent while SAMPLE(C) samples a head index for clause C. SOLVE is called repeatedly to obtain the samples of truth values for the goal. The fraction of true values is an estimation of the probability of the query of interest. The confidence interval on those samples is computed every m samples and the simulation ends when its value drops below a user-defined δ.

4 Experiments

We considered three datasets: graphs of biological concepts from [2], artificial graphs and the UWCSE dataset from [4]. 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 implemented in YAP Prolog and run on the data for 24 hours or until the program ended for lack of memory. The values used for the parameters are $k = 64$ as the number of explanations to consider for k-best and $\delta = 0.01$ as the maximum confidence interval width for Monte Carlo algorithm because they represent a good compromise between speed and accuracy.

The biological networks represent relationships among biological entities. Each edge is associated with a probability value that expresses the strength of the relationship. Determining the probability of an indirect association among a couple of entities is the same as computing the probability that a path exists between their nodes. The datasets are obtained from a network containing 11530 edges and 5220 nodes built around four genes responsible of Alzheimer's disease. Ten samples were extracted from the whole network each containing 50 graphs of increasing size (from 200 to 5000 nodes). For our test purposes we queried the probability that the genes HGNC_620 and HGNC_983 are related. Figure 1 presents the results of the experiments: the number of graphs for which the computation succeeded is reported on Figure 1(a), while Figure 1(b) reports the CPU time in seconds averaged over the graphs on which the algorithms succeeded as a function of the number of edges. The experimental results suggest that k-best does not improve with respect to exact because of the cost of keeping partial explanations sorted in sparse graphs, but Monte Carlo can solve twice as much problems than exact (up to 4000 edges). In terms of time, each algorithm performs almost like its ProbLog counterpart. With regard to the average absolute error, both k-best algorithms show a value of about 0.9%. Monte Carlo's average absolute error, however, is 4.9% for our implementation and 6.7% for ProbLog.

The artificial networks were used to evaluate the effective speedup in specific scenarios. The datasets contain graphs of increasing size that have different complexity with respect to the branching ratio and the length of paths between the terminal nodes. The graphs are built iteratively and are named after their shape: *lanes*, *branches* and *parachutes*. Lanes graphs, for example, gain a new parallel path a node longer than the previous graph. Branches are more complex because every step adds a new set of paths a node longer than before by forking at each node. Parachutes graphs are a trade-off between the two: they fork but each step introduces only one node (open paths fall back on existing nodes). Each dataset has a probability 0.3 on the edges and the path definition of lanes and parachutes contain 300 graphs, while branches only 25. Figure 2 shows an example for each dataset.

Again, we queried the probability that a path exists between the terminal nodes (0 and 1) of the graphs. Figures 1(c), 1(d) and 1(e) show that in almost any case, our algorithms have performed better than their ProbLog equivalent, with Monte Carlo always being the fastest. The average absolute error for *k*-best and Monte Carlo is 0.001% and 3.170% respectively. ProbLog's Monte Carlo is not applicable because of the presence of a probability value in rules for *path*.

On the UWCSE dataset, Monte Carlo took 3.873 seconds to solve the problem with 20 students, while the algorithm CVE of [4] can solve at most the problem with 7 students and taking around 1000 seconds. For 7 students Monte Carlo takes 1.961 seconds and incurs in a 4.3% absolute error on the problem with 0 students, the only one for which we have the exact result (see Figure 1(f)). ProbLog's Monte Carlo was not applicable because the problem involves non ground probabilistic facts. ProbLog's *k*-best managed to solve the problem with 25 students thus resulting to be the fastest algorithm on this dataset. It incurs into an absolute error of 4.7% on the problem with 0 students.

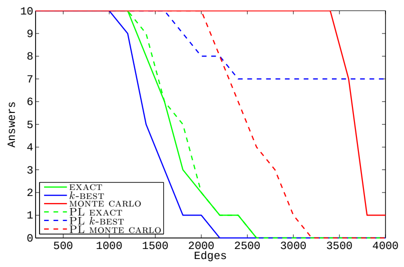
The source code of the algorithms together with more details on the datasets and the experiments are available at the address `http://sites.google.com/a/unife.it/ml/acplint`.

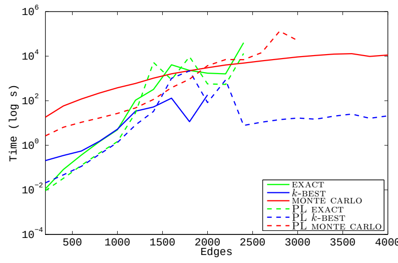
Acknowledgements

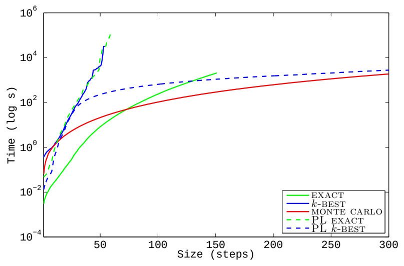
This work has been partially supported by the Italian MIUR PRIN 2007 project No. 20077WWCR8.

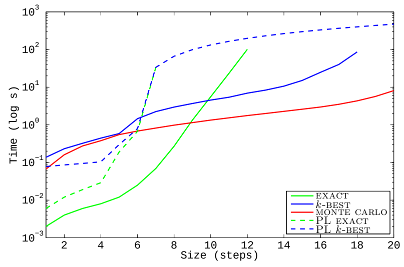
References

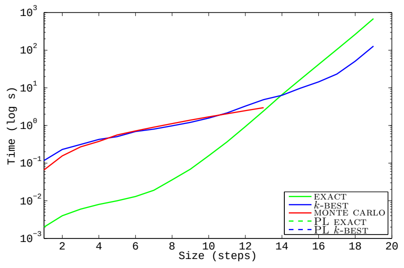
1. De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., Kimmig, A., Landwehr, N., Mantadelis, T., Meert, W., Rocha, R., Santos Costa, V., Thon, I., Vennekens, J.: Towards digesting the alphabet-soup of statistical relational learning. In: NIPS*2008 Workshop on Probabilistic Programming (2008)
2. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: 20th International Joint Conference on Artificial Intelligence. pp. 2468–2473. AAAI Press (2007)
3. Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., De Raedt, L.: On the efficient execution of problog programs. In: International Conference on Logic Programming. LNCS, vol. 5366, pp. 175–189. Springer (2008)
4. Meert, W., Struyf, J., Blockeel, H.: CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In: Inductive Logic Programming. LNCS, vol. 5989, pp. 96–109. Springer (2010)
5. Riguzzi, F.: A top down interpreter for LPAD and CP-Logic. In: Congress of the Italian Association for Artificial Intelligence. LNAI, vol. 4733, pp. 109–120. Springer (2007)
6. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: International Conference on Logic Programming. LNCS, vol. 3132, pp. 95–119. Springer (2004)


(a) Successes on biological graphs.


(b) Execution times on biological graphs.


(c) Execution times on Lanes graphs.


(d) Execution times on Branches graphs.


(e) Execution times on Parachutes graphs.

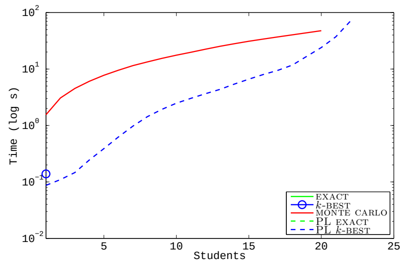
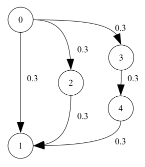
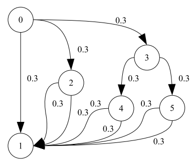

(f) Execution times on UWCSE graphs.

Fig. 1. Experimental results.


(a) Lanes.


(b) Branches.

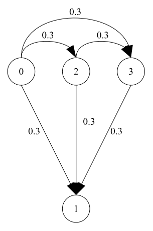

(c) Parachutes.

Fig. 2. Examples of artificial graphs.