Abstract

ProbLog is a recent probabilistic extension of Prolog, where facts can be labeled with mutually independent probabilities that they belong to a randomly sampled program. The implementation of ProbLog on top of the YAP-Prolog system provides various inference algorithms that calculate the success probability of a query, i.e. the probability that the query is provable in a randomly sampled program. We discuss extensions of these algorithms with tabling that broaden the class of problems that can be handled. First, exploiting structure sharing can speed up inference in domains where different proofs of a query share many subgoals. Second, we extend exact inference to deal with negated ground subgoals in clause bodies.

1. Introduction

In the past few years, a multitude of different formalisms combining probabilistic reasoning with logic, databases, or logic programming has been developed. To use such formalisms in statistical relational learning, efficient inference algorithms are crucial. ProbLog (De Raedt et al., 2007) is a simple extension of Prolog defining the success probability of a query in terms of random subprograms. Efficient inference algorithms for ProbLog have been implemented on top of the YAP-Prolog system (Kimmig et al., 2008). However, as these algorithms rely on exploring the space of all proofs, they suffer from redundant computations for domains where different proofs of a query share many subgoals, such as HMMs.

Inspired by PRISM (Sato & Kameya, 2001), where tabling is successfully applied for efficient probability calculations in such settings, we explore whether similar results could be obtained for ProbLog. We also discuss how tabling can serve to extend exact ProbLog inference to (certain types of) programs with negation.

As a motivating example, we consider a ProbLog encoding of the well-known bloodtype model, where a person’s blood type probabilistically depends on a single gene, which in turn probabilistically depends on the corresponding gene of the person’s parents:

\[
\text{bloodtype}(\text{Pers}, \text{B}) :- \\
\text{pchrom}(\text{Pers}, \text{P}), \text{mchrom}(\text{Pers}, \text{M}), \text{b}(\text{B}, \text{P}, \text{M}).
\]

\[
\text{pchrom}(\text{Pers}, \text{P}) :- \\
\text{father}(\text{Fa}, \text{Pers}), \\
\text{pchrom}(\text{Fa}, \text{Pf}), \text{mchrom}(\text{Fa}, \text{Mf}), \text{p}(\text{P}, \text{Pf}, \text{Mf}).
\]

\[
\text{mchrom}(\text{Pers}, \text{M}) :- \\
\text{mother}(\text{Mo}, \text{Pers}), \\
\text{pchrom}(\text{Mo}, \text{Pm}), \text{mchrom}(\text{Mo}, \text{Mm}), \text{m}(\text{M}, \text{Pm}, \text{Mm}).
\]

Here, the predicates \text{father} and \text{mother} encode the genealogical tree. The predicates \text{b}, \text{p} and \text{m} model the conditional probability distributions in a PRISM-like switch style. Here, the first argument denotes the random variable, i.e. for fixed second and third argument, exactly one instance of such a fact is true. More precisely, the first predicate, \text{bloodtype}, encodes \text{Pers}' bloodtype \text{B} as depending on a chromosome from each parent through the switch \text{b}. The other two predicates declare that a single chromosome is inherited from the parent’s, and that the chromosome originates from one of the grandparent’s through the switches \text{p} and \text{m}.

Notice that in this problem each gene can take one of three different values. If the ancestor structure forms a tree for each person (we assume no common ancestors), the number of proofs for each possible blood type is \(n(0) = 3 \cdot 3\) for persons without known ancestors (\text{pchrom} and \text{mchrom} take random values), and \(n(g) = (3 \cdot n(g - 1))^2\) for generations \(g > 0\) (each parent gene can have 3 values, and each of these has
n(g − 1) possible proofs, as one generation of ancestors less is known for parents). This amounts to about 5 · 10^6 for two generations of ancestors known, and 205 · 10^{12} for three generations. However, these proofs share many common subgoals. For instance, the query `bloodtype(p1,a)` can be proven from nine different gene combinations, as both P and M can take values a, b or null. Standard backtracking search will thus re-examine each instance of the `mchrom` fact for each instance of the `pchrom` fact, and similar repetitions occur in the clauses for these facts in each generation of the ancestor tree. Systems such as PRISM address this problem by using tabling to avoid recomputing intermediate answers\(^1\). Next, we investigate how the same principles can apply to ProbLog.

2. ProbLog Basics

A ProbLog program consists of a set of labeled facts \( p_i :: c_i \) together with a set of definite clauses. Each ground instance (that is, each instance not containing variables) of such a fact \( c_i \) is true with probability \( p_i \), where all probabilities are assumed mutually independent. The definite clauses allow to add arbitrary background knowledge (BK). A ProbLog program \( T = \{ p_1 :: c_1, \ldots, p_n :: c_n \} \cup BK \) defines a probability distribution over subprograms \( L \subseteq L_T = \{ c_1, \ldots, c_n \} \):

\[
P(L|T) = \prod_{c_i \in L} p_i \prod_{c_i \notin L_T \setminus L} (1 - p_i). \tag{1}
\]

The success probability \( P_s(q|T) \) of a query \( q \) in a ProbLog program \( T \) is defined as

\[
P_s(q|T) = \sum_{L \subseteq L_T} P(q|L) \cdot P(L|T), \tag{2}
\]

where \( P(q|L) = 1 \) if there exists a \( \theta \) such that \( L \cup BK = q\theta \), and \( P(q|L) = 0 \) otherwise. In other words, the success probability of query \( q \) is the probability that the query \( q \) is provable in a randomly sampled logic program. The definition of success probabilities employs non-probabilistic Prolog programs only. Therefore, it generalizes directly to background clauses with negated ground body literals as long as negation (seen as negation as failure) is not involved in cycles, which we will assume here. This observation allows one to encode a switch predicate with \( n \) possible values in ProbLog as a sequence of \( n - 1 \) probabilistic facts. Consider for example the last switch \( m \) in the bloodtype example, with values \( a, b \) and null. The first probabilistic fact decides whether it is \( a \) or not; if it is not \( a \), the second fact says whether it is \( b \) or not; finally, if it is neither \( a \) nor \( b \), it has to be \( null \). The following logic program realizes this:

\[
\begin{align*}
m(a,P,M) & :- mprob_a(P,M).
m(b,P,M) & :-\quad probnot\ mprob_a(P,M),\ mprob_b(P,M).
m(null,P,M) & :-\quad probnot\ mprob_a(P,M),\ probnot\ mprob_b(P,M).
\end{align*}
\]

Probabilistic facts `mprob_a(P,M)` and `mprob_b(P,M)` are needed for all combinations of argument values\(^2\). The `probnot` operator denotes negation as failure on ground queries. For a ground probabilistic fact \( Q \), `probnot Q` succeeds if \( Q \) is not in the program's model. Therefore, if \( Q \) succeeds with probability \( P \), `probnot Q` succeeds with probability \( 1 - P \). Our implementation of ProbLog relies on representing proofs as sets of probabilistic facts, leading to a straightforward extension to `probnot` for probabilistic facts: negated facts are included in the set as well, and proving fails if both cases occur for the same fact. Note that while `probnot` on probabilistic facts is sufficient to realize the switch encoding, implementing `probnot` on ground goals in general is more involved; we will return to this in the following sections.

3. Approximate Inference

In (Kimmig et al., 2008), we proposed a Monte Carlo method for computing the probability of a ProbLog query. The algorithm works by repeatedly sampling a logic program from the ProbLog program and checking whether the sample satisfies the query of interest. The fraction of samples where the query is provable is taken as an estimate of the query probability. This approach has shown itself to be quite robust and effective in practice, therefore it is interesting to investigate whether we can use tabling to improve performance.

For very large programs, generating a full sample may be more expensive than checking for satisfiability. As discussed in (Kimmig et al., 2008), we can take advantage of independence between facts to generate the sample lazily as needed during search for a proof: we verify whether a fact is in the sample only when we need it for a proof. To do so, samples are represented as the sample array with possible values corresponding to “true”, “false” and “unknown”. Notice that this implementation trick also provides a natural implementation for the `probnot` operator implementing negation as failure on (arbitrary) ground goals, as it follows standard Prolog inference with backtracking, filling up the array as needed on the way.

Whatever implementation we choose, it should be clear

---

\(^1\)We encoded bloodtype in PRISM; queries for three generations were solved in a fraction of a second.

\(^2\)Note that the probabilities of the facts have to be adjusted such that the product of the sequential outcomes equals the desired probabilities. For ease of use, switches are automatically compiled into the sequence encoding.
that the sample is a definite logic program, where any
ground call to probnot Q can be replaced as a call to
a new ground goal probnot,Q, such that either Q or
probnot,Q hold in the sample. Such logic programs
have a single minimal model, and are quite amenable
to SLG-execution. This suggests a first straightforward
algorithm for tabling, where for every sample program
we 1) reset the Table Space and 2) execute the
tabled program. Notice that SLG-refutation is of
interest only if the same subgoal is encountered multi-
times during a proof for a query, as then the same
sequence of array entries will be checked repeatedly.
Table 1 shows average execution time for the blood-
type example as a function of family size on an artifi-
cial database of individuals. The results were obtained
of the same variable in a complete proof. This makes
the Prolog part conceptually simple, although further
investigation is needed to obtain a clear idea of the
price to be paid in the form of memory requirements.

5. Discussion and Future Work
Tabling is an important feature of modern logic pro-
gramming systems. It is a key to efficient execution
in PRISM (Sato & Kameya, 2001), and has become
fundamental to understand the operation of PRISM
programs. Motivated by this work, we study whether
similar results can be obtained for ProbLog. Our first
experiment is promising: we show that tabling can
be introduced naturally in approximative inference of
ProbLog programs, with very significant benefits. Our
results raise a number of important related questions
to be answered in future work.

Can exact inference for ProbLog programs benefit from
tabling? As the formula encoding all proofs is cru-
ical for ProbLog inference, tabling requires encoding
the proofs themselves. Our initial implementation of
the probnot operator suggests that using linked tries
might be a way of realizing this that can easily be
extended to negation as well.

Can we improve MonteCarlo inference? The Monte-
Carlo algorithm has to reset the tables, only to con-
struct a very similar tree next. Given that we can
establish a clear connection between the trie table and
the sampling array, it may be worthwhile to exper-
iment with variational methods, such as MCMC. In
this case, each move would just consist of flipping ran-
dom variables and verifying whether the trie changes.

Acknowledgements A.K. and B.G. are supported
by FWO Vlaanderen. This work is partially supported
by GOA/08/008 Probabilistic Logic Learning, JEDI
(PTDC/EIA/66924/2006), STAMPA (PTDC/EIA/67738/
2006) and funds granted to LIACC, CRACS and IT via
the Programa de Financiamento Plurianual, Fundação
para a Ciência e Tecnologia and Programa POSC.

References
ProbLog: A probabilistic Prolog and its application
in link discovery. *IJCAI* (pp. 2462–2467).

Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B.,
of ProbLog Programs. *ICLP* (pp. 175–189). Springer.

Sato, T., & Kameya, Y. (2001). Parameter learning of
logic programs for symbolic-statistical modeling. *J.

<table>
<thead>
<tr>
<th>Ancestor tree depth</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLD</td>
<td>1.0</td>
<td>9.5</td>
<td>11.7</td>
<td>14.0</td>
</tr>
<tr>
<td>SLG</td>
<td>0.6</td>
<td>1.0</td>
<td>2.0</td>
<td>4.3</td>
</tr>
</tbody>
</table>

Table 1. MonteCarlo Execution Times (in secs).