Prolog issues of an MCMC algorithm

Nicos Angelopoulos **James Cussens**

Department of Computer Science, University of York
Heslington, York, YO10 5DD, UK
{nicos,jc}@cs.york.ac.uk

Abstract. We present a Markov chain Mode Carlo algorithm that operates on generic model structures that are represented by terms found in the computed answers produced by stochastic logic programs. The objective of this paper is threefold (a) to show that SLD-trees are an elegant means for describing prior distributions over model structures (b) to sketch an implementation of the MCMC algorithm in Prolog, and (c) to provide insights on desirable properties for SLPs.

1 Introduction

A stochastic logic program (SLP) is a labelled logic program which defines, for any goal, a log-linear distribution over refutations of that goal and consequently a marginal distribution over instantiations of the variables in the goal. SLPs generalise hidden Markov models (HMMs) and stochastic context-free grammars (SCFGs) by replacing parsing with proof search and unification. They were introduced in (Muggleton, 1996) with a view to applications in inductive logic programming (ILP).

In the context of probabilistic models, SLPs belong to the 'generative' approach often closely related to grammars. A number of other formalisms in the same category also elaborate some underlying logical paradigm with probabilistic concepts. These include (Riezler, 1998; Kameya & Sato, 2000; Pfeffer, 2001).

The Markov chain Mode Carlo (MCMC) algorithm discussed here is analysed in (Angelopoulos & Cussens, 2001). There, SLPs generating Bayesian Networks (BNs) were investigated in relation to the posterior likelihood of certain structural characteristics given a body of observed data. For example, how likely is it that node A is a parent of node B.

In the current paper we will also use similar SLPs but will instead concentrate on programming issues related to our work. On one hand we describe some Prolog programming techniques used in implementing the MCMC algorithm, while on the other hand we discuss some desirable properties for SLPs, that improve the efficiency of the algorithm.

2 Probability distributions over SLPs

Let l_i denote the label of the ith clause (C_i) and let λ be the vector of logs of the labels: $\lambda = (\lambda_1, \ldots, \lambda_n) = (\log l_1, \ldots, \log l_n)$. We then define three distributions, ψ_λ, f_λ and p_λ.

Given a goal G, consider x an SLD-derivation which begins with G. $\psi_\lambda(x)$ is the product of the labels $(l_{x,1}, \ldots l_{x,k})$ for the clauses $(C_{x,1}, \ldots C_{x,k})$ used in the derivation. This is simply the statistically intuitive product of sampling from independent choices. Thus,

$$\psi_\lambda(x) = \prod_{i=1}^{k} l_{x,i} = \exp \sum_{i=1}^{k} \lambda_{x,i}$$

For refutations we are not only interested in their individual probability, but also in what proportion of the success space this probability represents. Thus we define f_λ over refutations (again of G) as :

$$f_\lambda(x) = \begin{cases} Z_\lambda^{-1}\psi_\lambda(x) & \text{if x is successful} \\ 0 & otherwise \end{cases}$$

where Z_λ is the normalising factor, and for x ranging over all refutations of G, is defined as $Z_\lambda = \sum_{\text{x}} \psi_\lambda(x)$. Z_λ can be thought as the *probability of success* and it is an indicator of the chances that a random sample will be a successful one. In statistical terms f_λ is ψ_λ given that x is a refutation, i.e. $\psi_\lambda(x \mid$ x is a refutation). It also defines a log-linear, or Gibbs, distribution. This becomes more apparent when we consider $\nu_i(x)$ the number of time clause C_i was used in x. Then,

$$f_\lambda(x) = Z_\lambda^{-1} \prod_i l_i^{\nu_i(x)} = Z_\lambda^{-1} e^{\sum_i \lambda_i \nu_i(x)}$$

Finally, we define distribution p_λ over yields, where a yield is the computed answer associated with refutation x. Intuitively speaking, we want to have a way of describing distributions over distinct yields. Thus for a yield M,

$$p_\lambda(M) = \sum_{\text{x yields M}} f_\lambda(x)$$

3 MCMC over SLD-Trees

MCMC (Gilks, Richardson, & Spiegelhlater, 1996) algorithms are commonly used to enable estimation of $P(M \mid D)$ (posterior probability of models given data D) from a prior distribution $p(M)$. Provided some weak conditions are met, and that the Markov chain is run for a sufficiently long period, then the visiting frequencies approach the true posterior probability of models.

In constructing the Markov chain we follow the Metropolis-Hasting algorithm

0. Set $i = 0$ and find M_0, an initial leaf of the SLD-tree, using the prior.
1. From M_i produce a candidate leaf M_*. Let the probability of reaching M_* be $q(M_i, M_*)$.

2. Let

$$\alpha(M_i, M_*) = \min \left\{ \frac{q(M_*, M_i) P(D|M_*) P(M_*)}{q(M_i, M_*) P(D|M_i) P(M_i)}, 1 \right\}$$

$$M_{i+1} = \begin{cases} M_* & \text{with probability } \alpha(M_i, M_*) \\ M_i & \text{with probability } 1 - \alpha(M_i, M_*) \end{cases}$$

3. If i reached limit then terminate, else set $i = i + 1$ and repeat from 1.

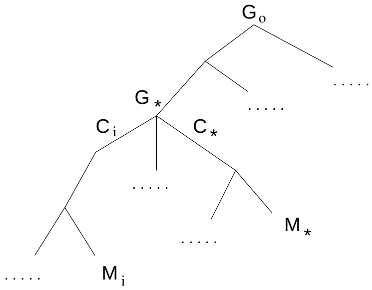

Fig. 1. Reaching M_* from M_i in the SLD-tree.

We use a version of SLD-trees where failures are explicitly represented. It may be that M_* is a leaf of the SLD-tree corresponding to unification failure; in that case $M_{i+1} = M_i$. All other leaves correspond to models (in the statistical sense) and for these we assume that means for calculating $P(D \mid M_*)/P(D \mid M_i)$, the likelihood ratio, are provided. In what follows, we concentrate on the procedure that leads from some M_i to a proposal M_* and by extension to calculating the quantity $q(M_*, M_i)/q(M_i, M_*)$. The essence of our approach, is to backtrack to an ancestor node in the SLD-tree and then sample from one of the alternative branches from this node. A general case of this is illustrated in Fig. 1. The algorithm is parametrised by the backtracking probability (p_b) and its formulated as follows.

0. Backtrack one step.
1. If at the top of the tree then go to 2, otherwise with probability p_b backtrack one more point and repeat 1, or, with probability $1 - p_b$, go to 2.
2. Sample for a new leaf M_* by selecting branches with probability proportional to clause labels. On the first choice, do not descend to the branch leading to M_i (branch C_i in Fig. 1).

4 A Bayesian Network Generator

The clauses of a simple SLP program for generating Bayesian Networks (the structure of a BN is a Directed Acyclic Graph) from a list of nodes is presented in Figure 2.

Querying (or sampling from) such a program with goal $|\ ?-bn([a,b,c],BN).$ will produce a BN represented as a list of Node-ParentsList pairs. For instance $BN =$ `[a-[],b-[a],c-[a]]` maps to BN, $b \leftarrow a \rightarrow c$. Note that here we transpose the usual representation of graphs in Prolog, which is a list of Node-ChildrenList pairs. In the particular program we can safely do this since any cycle in one representation will also be a cycle in its transpose.

BNs are built by adding a single node at a time. The node added at each stage has three possibilities in relation to each existing node in the partial BN. These choices are captured in the three clauses of the `choose_edges1/3` predicate. Their probability labels (or simply labels) give a slight preference to attaching children. The resulting graph is only accepted as a valid BN if there exists no cycle.

```
:- use_module(library(ugraphs)).
% top_sort/2, add_vertices/3, add_edges/3.

bn([],[]).
bn([RV|RVs],BN) :-
    bn(RVs,TailBN),
    connect(RV,RVs,TailBN,BN),
    no_cycle_introduced(RV,BN).

connect(RV,RVs,TailBN,BN) :-
    choose_edges(RVs,RV,Edges),
    add_vertices(TailBN,[RV],BN1),
    add_edges(BN1,Edges,BN).

%no cycles if there is a top sort
no_cycle_introduced(_RV,BN) :-
    top_sort(BN,_).
```

```
choose_edges([],RV,[]).
choose_edges([H|T],RV,Edges) :-
    choose_edges1([H|T],RV,Edges).

%RV parent of H
0.3 :: choose_edges1([H|T],RV,
                 [H-RV|Rest]) :-
    choose_edges(T,RV,Rest).
%RV child of H
0.4 :: choose_edges1([H|T],RV,
                 [RV-H|Rest]) :-
    choose_edges(T,RV,Rest).
%no direct connection
0.3 :: choose_edges1([_H|T],RV,Rest) :-
    choose_edges(T,RV,Rest).
```

Fig. 2. Simple BN generator.

In this paper we have assumed a way of calculating the likelihood ratio, thus completing the requirements for application of the presented MCMC. A method for calculating this quantity in the BN example was presented in (Angelopoulos & Cussens, 2001). Note that (a) in our current example D, the data set is a set of vectors, each holding a value for each variable in the BN (b) M can be effectively represented by the corresponding BN, and (c) we are only interested in the structure of the resulting BN and not in the conditional probability tables which parameterise it.

5 Constraints on the Prior

SLPs can express both soft and hard constraints about the prior distribution of models. Hard constraints are expressed as rules while soft constraints or preferences are expressed via the probabilistic labels on the clauses.

For example, the program in Fig. 3 expresses two hard constraints. Firstly, predicate `parents_pre/3` ensures that the only possible parents of the BN random variable `3` are `5` and `2`, and secondly the structure of `parents/2` constraints the maximum

```
:- use_module( library(ordsets) ).
% list_to_ord_set/2, ord_union/2,
% ord_add_element/3.

% Nodes are ordered.
bn( Nodes, BN )  :-
    bn(Nodes,[],BN).
bn([],_All,[]).
bn([RV|RVs],CandParents,BN) :-
    bn(RVs,[RV|CandParents],TailBN),
    parents_pre(RV,CandParents,Parents),
    list_to_ord_set(Parents,OrdPrnts),
    ord_add_element(TailBN,RV-
OrdPrnts,BN).

parents_pre(3,_RVPrnts,[2,5]).
parents_pre(RV,RVPrnts,Parents) :-
    RV =\= 3,
    parents(RVPrnts,Parents).

1/4::parents([A,B,C|TPrnts],Parents) :-
    select_parent([A,B,C|TPrnts],OneL,RestA),
    select_parent(RestA,TwoL,RestB),
    select_parent(RestB,ThreeL,_RestC),
    ord_union([OneL,TwoL,ThreeL],Parents).

1/4::parents([A,B|TPrnts],Parents) :-
   select_parent([A,B|TPrnts],OneL,RestCPs),
   select_parent(RestCPs,TwoL,_RemCps),
   ord_union( OneL, TwoL, Parents ).

1/4::parents([A|CndPrnts],ParentL) :-
   select_parent([A|CndPrnts],ParentL,_Rest).

1/4::parents(CandParents,[]).
```

Fig. 3. BN generator with Constraints.

number of parents a node can have. On the other hand the shown labels on `parents/2` distribute the number of parents uniformly over the ones available. Thus, we expect this program to generate BNs in which nodes have on average a little less than 1.5 parents, each.

Let the probability labels of `parents/2` be (l_6, l_7, l_8, l_9) then by changing these we can affect the edge density of the resulting BN. For instance, letting ($l_6 = 8/15, l_7 = 4/15, l_8 = 2/15, l_9 = 1/15$) the expected number of parents per node, becomes greater than two. Note that a soft constraint to its limit, becomes a hard one. So, for $l_6 = 0$ there will almost surely be no node with three parents in the resulting BN.

Unlike the program in Fig. 2 our current BN generator has another interesting feature. It is failure free, since instead of generating and testing, an auxiliary list is used to contain all possible parents which are consistent with the current BN. The task of maintaining such a list is eased in this example since we expect the input list, `Nodes`, to be topologically ordered in respect to the final BN. Such knowledge is not uncommon in practical applications of BNs. Having a failure free program ensures that the Markov chain converges much faster.

6 Prolog Sampling for MCMC

The standard control strategy of Prolog can be used for dealing with the calculation of the three distributions presented in Section 2 (provided some mild clause and query transformation take place and suitable meta-predicates are defined). Unfortunately, as it is apparent from Section 3, this is not the case for the MCMC sampling. To deal with the specific needs of the MCMC algorithm we introduce predicate transformations and an additional runtime meta-predicate.

The core control operation of the algorithm is shown in Fig. 1. The objective is to find proposing M_* when starting from M_i. As prescribed, M_{i+1} could then be either one of M_* or M_i. It is obvious that standard Prolog control does not match the task at hand very well. The main two reasons are : (a) the iterative nature of moving from M_i to M_{i+1}, and (b) the need to backtrack in a tentative manner and at arbitrary points of the execution path.

We achieve the desired backtracking behaviour by starting at the top of the tree and follow the execution path from G_0 to G_*. Then, we probabilistically choose C_* (different from C_i) and proceed until a leaf is reached (M_*). Since, this is repeated at each iteration it is crucial to minimise duplicated work. For this, we index each non-stochastic clause with a unique id and provide two extra arguments which hold different version of the path-so-far, as difference lists.

```
parents([A|B]/C, [D|E]/F, G, H) :-
   select_id([[[I,J,K|L],M],[[N,O|P],Q],[[R|S],T],[U,[]]], [G,H],
                                        [0.25,0.25,0.25,0.25], 18, A, B, D, V, W),
   (   V=:=18 ->
      [[I,J,K|L],M]=[G,H],
      select_parent(_, W/X, E/Y, [I,J,K|L], Z, A1),
      select_parent(_, X/B1, Y/C1, A1, D1, E1),
      select_parent(_, B1/F1, C1/G1, E1, H1, _),
      ordsets:ord_union([Z,D1,H1], M),
      F1=C, G1=F
      ;
      V=:=19 ->
         [[N,O|P],Q]=[G,H],
         select_parent(_, W/I1, E/J1, [N,O|P], K1, L1),
         select_parent(_, I1/M1, J1/N1, L1, O1, _),
         ordsets:ord_union(K1, O1, Q),
         M1=C, N1=F
         ;
         V=:=20 ->
            [[R|S],T]=[G,H],
            select_parent(_, W/C, E/F, [R|S], T, _)
            ;
            [U,[]]=[G,H], W=C, E=F
   ).
```

Fig. 4. Transformed clause corresponding to parents/2 of Fig.3.

Stochastic clauses of a single predicate are gathered into a single transformed clause. As an example consider the clause in Fig. 4, which is derived from predicate `parents/2` (Fig.3). The original parameters (variables G and H here) are funneled to `select_id/9` so that the probabilistic choice will only consider matching clauses. The ids corresponding to the original clauses are, $18, 19, 20$ and 21. `select_id/9` will choose one of these ids and thus determine the branch to follow. This is a probabilistic choice made in proportion to the clause probabilities corresponding to the unified sets of arguments.

In the current scenario of setting out from M_i to find a candidate M_* `select_id/9` will be called in three distinct modes. On the first leg and before reaching G_*, variable A will be instantiated to the choice already made for M_i, thus no unifications will be attempted. When G_* is reached an alternative to C_i is considered (restricted choice). Finally, after C_* has been chosen, A will be uninstantiated and an unrestricted probabilistic choice is made.

7 Efficiency Issues

The presented MCMC algorithm states that the probability of backtracking one more choice point is p_b. Having experimented with various programs and backtracking probabilities, we have found that a single values for p_b is rarely adequate to provide a good mixture of local and remote jumps. The main problem is that a single value will have an in-built bias towards, either local jumps, thus we run the danger of getting stuck in a single neighbourhood, or remote jumps, which may lead to slow mixing for the Markov chain. Fundamentally, this is due to the inability to have a single distribution over all the possible points. A more fruitful strategy has been to implement a cyclic transition kernel. Let d_i be the number of backtrack points when we are at M_i, then we define $p_{b,n} = 1 - 2^{-i}$, for $n = 1, \ldots, d_i$. At iteration i use $p_{b,k}$ such that $k = i$ MOD d_i. Thus at each iteration we make one part of the path highly likely. Effectively, we approximate a uniform distribution by shifting the focal point at each iteration.

Sampling from an SLP may lead to a failure. As already mentioned briefly, sampling a failure within the MCMC algorithm is a wasteful exercise since it leads to no possibility of moving from the current model (M_i) and thus to poor convergence for the chain. The situation can become even worst since the majority of constraints, are more easily expressed as failures (not permitted combinations). In this respect, SLPs show their origin as a generative probabilistic approach.

As demonstrated with example in Fig.3 it is often possible to write failure-free programs, or to at least reduce the frequency/probability of failed derivations. In general this is not dissimilar to the care needed when composing normal Prolog programs. For instance, generate-and-test programs are always the easier, more abstract alternatives, but more often than not, are orders of magnitude slower than more complicated programs for the same task.

8 Concluding remarks

The main benefit of the work presented here, is the combination of a specialised algorithm with a formalism of substantial knowledge representation capabilities. Experimental results pertaining to our approach were presented in (Angelopoulos & Cussens, 2001). Here we concentrated on issues related to Prolog. On one hand, SLPs are an extension to standard Prolog, while on the other hand, we presented some techniques used in implementing the algorithm and SLPs in Prolog.

Acknowledgements This work was supported by EPSRC research grant GR/N09152 *Induction of Stochastic Logic Programs*. Some of the software written for the project can be found in http://www-users.cs.york.ac.uk/˜nicos/slps/

References

Angelopoulos, N., & Cussens, J. (2001). Markov chain Monte Carlo using tree-based priors on model structure. In Breese, J., & Koller, D. (Eds.), *Proc. of the 17th Annual Conf. on Uncertainty in Artificial Intelligence (UAI–2001)*, pp. 16–23 Seattle. Morgan Kaufmann.

Gilks, W. R., Richardson, S., & Spiegelhlater, D. J. (Eds.). (1996). *Markov Chain Monte Carlo in Practise*. Chapman & Hall.

Kameya, Y., & Sato, T. (2000). Efficient EM learning with tabulation for parameterized logic programs. In *Proc. of the First International Conf. on Computational Logic (CL 2000)*, Vol. 1861 of *LNAI*, pp. 269–284 London. Springer.

Muggleton, S. (1996). Stochastic logic programs. In *Advances in ILP*, Vol. 32 of *Frontiers in AI and Applications*, pp. 254–264. IOS Press.

Pfeffer, A. (2001). IBAL: A probablistic rational programming language. In *Proc. of the Seventeenth International Joint Conference on AI (IJCAI-01)*.

Riezler, S. (1998). *Probabilistic Constraint Logic Programming*. Ph.D. thesis, Universität Tübingen. AIMS Report 5(1), 1999, IMS, Universität Stuttgart.