

Improving the Efficiency of Gibbs Sampling for Probabilistic Logical Models by Means of Program Specialization

Daan Fierens

Report CW 581, April 2010



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Improving the Efficiency of Gibbs Sampling for Probabilistic Logical Models by Means of Program Specialization

Daan Fierens

Report CW 581, April 2010

Department of Computer Science, K.U.Leuven

Abstract

There is currently a large interest in probabilistic logical models. A popular algorithm for approximate probabilistic inference with such models is Gibbs sampling. From a computational perspective, Gibbs sampling boils down to repeatedly executing certain queries on a knowledge base composed of a static part and a dynamic part. The larger the static part, the more redundancy there is in these repeated calls. This is problematic since inefficient Gibbs sampling yields poor approximations.

We show how to apply program specialization to make Gibbs sampling more efficient. Concretely, we develop an algorithm that specializes the definitions of the query-predicates with respect to the static part of the knowledge base. In experiments on real-world benchmarks we obtain speedups of up to an order of magnitude.

Keywords : probabilistic logical models, probabilistic logic programming, program specialization, Gibbs sampling

CR Subject Classification : I.2.2, G.3, D.1.6

1 Introduction

In the field of artificial intelligence there is currently a large interest in *probabilistic logical models* (probabilistic extensions of logic programs and first-order logical extensions of probabilistic models such as Bayesian networks) [3, 10, 4]. *Probabilistic inference* with such a model is the task of answering various questions about the probability distribution specified by the model, usually conditioned on certain observations (the *evidence*). A variety of inference algorithms is currently being used. A popular algorithm for approximate probabilistic inference is *Gibbs sampling* [2, 12]. Gibbs sampling works by drawing samples from the considered probability distribution conditioned on the evidence. These samples are then used to compute an approximate answer to the probabilistic questions of interest. It is important that the process of drawing samples is efficient because the more samples can be drawn per time-unit, the more accurate the answers will be (i.e., the closer to the correct answer).

Computationally, Gibbs sampling boils down to repeatedly executing the same queries on a knowledge base composed of a static part (the evidence and background knowledge) and a highly dynamic part that changes at runtime because of the sampling. The more evidence, the larger the static part of the knowledge base, so the more redundancy there is in these repeated calls. Since it is important that the sampling process is efficient, this redundancy needs to be reduced as much as possible. In this paper we show how to do this by applying *program specialization* to the definitions of the query-predicates: we specialize these definitions with respect to the static part of the knowledge base. While a lot of work about logic program specialization is about exploiting static information about the input arguments of queries (partial deduction [5]), we instead exploit static information about the knowledge base on which the queries are executed.

While the above applies to all kinds of probabilistic logical models and programs, we focus in this paper on models that are first-order logical or “relational” extensions of Bayesian networks [3, 4]. Concretely, we use the general framework of *parameterized Bayesian networks* [10].

The *contributions of this paper* are the following. First, we show how to represent parameterized Bayesian networks in Prolog (Section 3). Second, we show how to implement Gibbs sampling in Prolog and show that doing this efficiently poses several challenges from the logic programming point of view (Sections 4 and 7). Third, we develop an algorithm for specializing the considered logic programs with respect to the evidence (Section 5). Fourth, we perform experiments on real-world benchmarks to investigate the influence of specialization on the efficiency of Gibbs sampling. Our results show that specialization yields speedups of up to an order of magnitude and that these speedups grow with the data-size (Section 6). The latter two are the main contributions of this paper, the first two are minor contributions.

We first give some background on probability theory and Bayesian networks.

2 Preliminaries: Probability Theory and Bayesian Networks

In probability theory [8] one models the world in terms of *random variables (RVs)*. Each state of the world corresponds to a joint state of all considered RVs. We use upper case letters to denote single RVs and boldface upper case letters to denote sets of RVs. We refer to the set of possible states of an RV X (i.e. the set of values that X can take) as the *range* of X , denoted $range(X)$. For now we consider only *discrete RVs*, i.e. RVs with a finite range (see Section 7).

A *probability distribution* on a finite set S is a function that maps each $x \in S$ to a number $P(x) \in [0, 1]$ such that $\sum_{x \in S} P(x) = 1$. A probability distribution for an RV X is a probability distribution on the set $range(X)$. A *conditional probability distribution (CPD)* for an RV X conditioned on a set of other RVs \mathbf{Y} is a function that maps each possible joint state of \mathbf{Y} to a probability distribution for X .

Syntactically, a *Bayesian network* [8] for a set of RVs \mathbf{X} is a set of CPDs: for each $X \in \mathbf{X}$ there is one CPD for X conditioned on a (possibly empty) set of RVs called the *parents* of X .

Intuitively, the CPD for X specifies the direct probabilistic influence of X 's parents on X . The probability distribution for X conditioned on its parents $\text{pa}(X)$, as determined by the CPD for X , is denoted $P(X \mid \text{pa}(X))$.

Semantically, a Bayesian network represents a probability distribution $P(\mathbf{X})$ on the set of all possible joint states of \mathbf{X} . Concretely, $P(\mathbf{X})$ is the product of all the CPDs in the Bayesian network: $P(\mathbf{X}) = \prod_{X \in \mathbf{X}} P(X \mid \text{pa}(X))$. It can be shown that $P(\mathbf{X})$ is a proper probability distribution provided that the parent relation is acyclic (the parent relation is often visualized as a directed acyclic graph but given the CPDs this graph is redundant).

3 Parameterized Bayesian Networks

Bayesian networks essentially use a propositional representation. Several ways of lifting them to a first-order representation have been proposed [3, Ch.6,7,13] [4]. There also exist several probabilistic extensions of logic programming, such as PRISM, Independent Choice Logic and ProbLog [3, Ch.5,8]. Both kinds of probabilistic logical models (probabilistic logic programs and the extensions of Bayesian networks) essentially serve the same purpose. In this paper we focus on the Bayesian network approach. Our main motivation for this choice is that this paper is about Gibbs sampling and this has been well-studied in the context of Bayesian networks.

There are many different representation languages for first-order logical (or “relational”) extensions of Bayesian networks. We use the general framework of *parameterized Bayesian networks* [10]. While this framework is perhaps not a full-fledged knowledge representation language, it does offer a representation that is suited to implement probabilistic inference algorithms on. One possible approach is to first construct a model in a suitable representation language, and then transform or “map” the model to a parameterized Bayesian network when probabilistic inference needs to be performed (mappings between different kinds of probabilistic logical models are an active research topic [3, Ch.12,13]).

We now briefly introduce parameterized Bayesian networks.

3.1 Essentials of Parameterized Bayesian Networks

Like Bayesian networks use RVs, parameterized Bayesian networks use so-called *parameterized RVs* [10]. Parameterized RVs have a number of typed parameters ranging over certain populations. When each parameter in a parameterized RV is instantiated or “grounded” to a particular element of its population we obtain a regular or “concrete” RV. To each parameterized RV we associate a *parameterized CPD* (see below) with the same parameters as the parameterized RV.

Syntactically, a parameterized Bayesian network is a set of parameterized CPDs, one for each parameterized RV. Semantically, a parameterized Bayesian network \mathcal{B} , in combination with a given population for each type, specifies a probability distribution. Let \mathbf{X} denote the set of all concrete RVs that can be obtained by grounding all parameterized RVs in \mathcal{B} with respect to their populations. The probability distribution specified by \mathcal{B} is then the following distribution on the set of all possible joint states of \mathbf{X} : $P(\mathbf{X}) = \prod_{X \in \mathbf{X}} P(X \mid \text{pa}(X))$, where $P(X \mid \text{pa}(X))$ denotes the probability distribution for X as determined by its parameterized CPD.

Rather than providing a further formal discussion of parameterized Bayesian networks we show how they can be represented in Prolog (as far as we know this has not been done before).

3.2 Representing Parameterized Bayesian Networks in Prolog

To deal with parameterized RVs in Prolog we associate to each of them a unique predicate: for a parameterized RV with n parameters we use a $(n+1)$ -ary predicate, the first n arguments correspond to the parameters, the last argument represents the state of the RV. We refer to these predicates as *state predicates*.

Syntactically a parameterized Bayesian network is a set of parameterized CPDs. To deal with parameterized CPDs we also associate to each of them a unique predicate, the last argument now represents a probability distribution on the range of the associated RV. We refer to these predicates as *CPD-predicates*. In this paper we assume that each CPD-predicate is defined by a decision list. A *decision list* is an ordered set of rules such that there is always at least one rule that applies, and of all rules that apply only the first one fires (in Prolog this is achieved by putting a cut at the end of each body and having a last clause with *true* as the body).

Example 1 (Representing a parameterized Bayesian network in Prolog) *Consider a university with students and courses. Suppose that we use the following parameterized RVs: level (with a parameter from the population of courses), iq and graduates (each with a student parameter) and grade (with a student parameter and a course parameter). To represent the state of the RVs we use the state predicates level/2, iq/2, graduates/2 and grade/3. The meaning of level/2 is that the atom level(C, L) is true if the parameterized RV level for the course C is in state L. To represent the parameterized CPDs we use CPD-predicates cpd_level/2, cpd_iq/2, cpd_grade/3 and cpd_graduates/2. If the level RVs do not have parents, their parameterized CPD could for instance be defined as follows.*

```
cpd_level(_C, [intro:0.4, advanced:0.6]).
```

We use lists like [intro:0.4, advanced:0.6] to represent probability distributions. The other parameterized CPDs could for instance be defined as follows.

```
cpd_iq(_S, [high:0.5, low:0.5]).
```

```
cpd_grade(S, C, [a:0.7, b:0.2, c:0.1]) :- iq(S, high), level(C, intro), !.
cpd_grade(S, C, [a:0.2, b:0.2, c:0.6]) :- iq(S, low), level(C, advanced), !.
cpd_grade(S, C, [a:0.3, b:0.4, c:0.3]).
```

```
cpd_graduates(S, [yes:0.2, no:0.8]) :- grade(S, _C, c), !.
cpd_graduates(S, [yes:0.5, no:0.5]) :- findall(C, grade(S, C, a), L),
                                     length(L, N), N < 2, !.
cpd_graduates(S, [yes:0.9, no:0.1]).
```

In the bodies of the clauses defining the CPD-predicates we allow the use of state predicates (e.g. *iq/2* and *level/2* in the clauses for *cpd_grade/3*) and of background predicates, but not of CPD-predicates. With *background predicates* we mean auxiliary predicates that do not depend on the state of RVs (this includes built-ins such as *length/2*). We assume that the definitions of the background predicates are available in a *background knowledge base*. We also allow the use of meta-predicates (such as *findall/3*) but not of predicates with side-effects (such as *assert/1*).

When we know the population for each type (e.g. we know the set of students and the set of courses) we also know the set of concrete RVs X . Suppose that in addition we also know the state of these concrete RVs because we are given a knowledge base with facts defining the state predicates (e.g. a fact *grade(s1, c1, a)* indicates that student *s1* has grade ‘a’ for course *c1*). We can then obtain the probability distribution for a concrete RV conditioned on its parents by simply calling the associated CPD-predicate on this knowledge base. For instance, we obtain the probability that the student *s1* will graduate conditioned on her grades by calling *cpd_graduates(s1, Distribution)*. We refer to this as *calling the CPD for that concrete RV*. Since we represent each parameterized CPD as a decision list it is guaranteed that this always returns exactly one probability distribution.¹

As we explain in the next section, calling a CPD is an operation that needs to be performed frequently during probabilistic inference. Another such operation is *setting a concrete RV to a*

¹Some CPD-predicates are defined by non-ground facts (e.g. *cpd_level/2*). This does not cause problems because we always call CPD-predicates with all arguments except the last instantiated.

given state. This is done by modifying the corresponding fact in the knowledge base (e.g. the fact $grade(s1, c1, a)$ is turned into $grade(s1, c1, b)$).²

4 Probabilistic Inference with Parameterized Bayesian Networks

Given the population for each type, a parameterized Bayesian network defines a probability distribution $P(\mathbf{X})$ on the set of all possible joint states of the concrete RVs \mathbf{X} . In a typical inference scenario, the state of a subset of all these RVs is observed. This information is called the *evidence*. *Probabilistic inference* is the task of answering certain questions about the probability distribution $P(\mathbf{X})$ conditioned on the evidence. The most common inference task is to compute marginal probabilities. A *marginal probability* is the probability that a particular RV is in a particular state. For instance, given the level of all courses and the grades of all students for all courses (the evidence), we might want to compute for each student the probability that she has a high IQ. In theory such probabilities can be computed by performing a series of sum and product operations on the probability distributions specified by the parameterized CPDs. Unfortunately, for real-world population sizes this is computationally intractable (inference with Bayesian networks is NP-hard [8]). Hence, one often uses *approximate probabilistic inference* instead. An important class of approximate inference algorithms are *Monte Carlo algorithms* that draw samples from the given distribution conditioned on the evidence. Various algorithms are being used, a very popular one is *Gibbs sampling* [2, 12].

4.1 Gibbs Sampling for Parameterized Bayesian Networks

Let \mathbf{O} denote the set of all observed concrete RVs (i.e. the RVs for which we have evidence), and \mathbf{U} the set of all unobserved ones ($\mathbf{U} = \mathbf{X} \setminus \mathbf{O}$). Below we assume that we need to compute marginal probabilities for all unobserved RVs. Pseudocode for the Gibbs sampling algorithm is shown in Figure 1. We now explain this further.

<pre> procedure GIBBS_SAMPLING(\mathbf{O}, \mathbf{U}) 1 for each $O \in \mathbf{O}$ 2 set O to its known state 3 for each $U \in \mathbf{U}$ 4 set U to random state $\in range(U)$ 5 initialize all counters for U 6 repeat until enough samples 7 for each $U \in \mathbf{U}$ 8 RESAMPLE(U) 9 compute estimates from counters </pre>	<pre> procedure RESAMPLE(U) 1 call the CPD for U 2 for each $u \in range(U)$ 3 set U to state u 4 for each child X of U 5 call the CPD for X 6 calculate $P_{resample}(U)$ 7 sample u_{new} from $P_{resample}(U)$ 8 set U to u_{new} 9 increment counter for (U, u_{new}) </pre>
---	--

Figure 1: The Gibbs sampling algorithm (left) and its RESAMPLE procedure (right).

Before the start of the sampling process all observed RVs are instantiated to their known state and all unobserved RVs are instantiated to a random state. In terms of our implementation in Prolog, this is done by creating a knowledge base defining all the state predicates: for each $RV \in \mathbf{O} \cup \mathbf{U}$ there is one fact for the corresponding state predicate. Before we start sampling, we also create a number of counters: for each $U \in \mathbf{U}$ and each $u \in range(U)$ we create a counter to store the number of samples in which U is in state u . All counters are initialized to zero.

Let us now consider the sampling process itself. To create one sample, we visit (in an arbitrary but fixed order) all unobserved RVs. When we visit an RV U , we “resample” it. The idea is to sample the new state from the probability distribution for U conditioned on the current

²For efficiency reasons we do not use assert/retract to update the knowledge base. Instead we make use of an internal hash table to store the values of the RVs.

state of *all* other RVs. For details on how to construct this distribution $P_{resample}(U)$ we refer to Bidyuk and Dechter [1], here we focus on the main computations that this requires (see the RESAMPLE procedure in Figure 1): first we need to call the CPD for U , then we loop over all possible states of U and for each state u we set U to u and call the CPDs of each of the children of U .³ Based on the information returned by all these CPD-calls it is straightforward to construct the distribution $P_{resample}(U)$. We then randomly sample a state from this distribution, set U to this new state and increment the appropriate counter for U .

The above is done for all unobserved RVs, yielding one sample.⁴ Note that observed RVs are clamped to their known state, hence the generated sample is guaranteed to be consistent with the evidence. This entire procedure is repeated N times, yielding N samples. It is then straightforward to construct an estimate of all required marginal probabilities based on the computed counts. For instance, the estimated probability that student $s1$ has a high IQ conditioned on the evidence is the number of samples in which the RV iq for $s1$ was in the state ‘high’, divided by N .

4.2 Efficient Gibbs Sampling in Prolog

The higher the number of samples N , the closer the estimated marginal probabilities will be to their correct values.⁵ Gibbs sampling is often used by giving the sampling process a fixed time to run before computing the estimates. In this case, the less time it takes to draw a single sample, the more samples can be drawn in the given time, so the higher the accuracy of the estimates. In other words: any gain in efficiency of the sampling process might lead to a gain in accuracy of the estimates. Hence it is crucial to implement the sampling process as efficiently as possible.

The Gibbs sampling algorithm uses several operations, but there is one operation that we clearly found to be the computational bottleneck, namely *calling the CPDs*. This operation occurs inside several nested loops (see line 5 of the RESAMPLE procedure in Figure 1) and is hence performed many times. The knowledge base on which these CPD-queries are called is highly dynamic: the state of the unobserved RVs changes continuously because they are being resampled. This is only one part of the knowledge base, however. The part that is about the observed RVs (the evidence) stays constant during the entire sampling process. This static part of the knowledge base causes redundancy in the repeated calls of the CPD-queries since part of the computations are performed over and over again. Since we want the sampling process to be as efficient as possible, this redundancy needs to be removed.

The more evidence we have, the larger the redundancy. In many practical cases, the amount of evidence is considerable. One typical inference scenario is prediction or *classification*. The standard classification setting is that all concrete RVs associated to one particular parameterized RV (the class) are unobserved and need to be predicted based on observations of all the other RVs. For instance, we can predict the class of web pages based on information about these web pages such as their word-counts and mutual hyperlinks [11]. Another typical inference scenario is dealing with *missing data*. If we have a database in which a fraction of all entries are missing (for instance due to measurement errors), and we have a probabilistic model of the domain, then we can use probabilistic inference to fill in the missing entries based on the observed ones. This is for instance often done in the context of machine learning from incomplete data [7]. In both scenarios there are typically more observed RVs than unobserved ones.

To summarize, when performing inference we often have a large amount of evidence and this causes redundancy in the Gibbs sampling algorithm. In the next section we show how this redundancy can be removed, and Gibbs sampling be made more efficient, by means of program specialization.

³ X is called a child of U in a parameterized Bayesian network if U is a parent of X .

⁴In practice we use a slight variation of this procedure which includes a number of common optimizations (such as making use of the ‘support network’ [3, Ch.7]).

⁵With N going to infinity the estimates provably converge to the correct values under the condition that none of the probabilities in the CPDs equal zero [1].

5 Applying Logic Program Specialization to Parameterized CPDs

The main idea is to *specialize the definitions of the CPD-predicates with respect to the static part of the knowledge base*. Recall that we define each CPD-predicate in Prolog by means of a decision list (Example 1, p. 3). Our specialization approach is a source-to-source transformation that takes three inputs: 1) the decision lists for all the CPD-predicates, 2) the evidence (i.e. the observed RVs with their observed states), and 3) the background knowledge base. The output of the transformation is a specialized version of the decision lists. The transformation is such that Gibbs sampling produces exactly the same sequence of samples with the specialized decision lists as with the original ones (but in a more efficient way).

We use the term *CPD-query* to refer to any atom for a CPD-predicate with the last argument uninstantiated and all other arguments instantiated to elements of the proper populations. For instance, $cpd_grade(s, c, Distribution)$ is a CPD-query if s is in the considered population of students and c in the population of courses. All calls to CPD-predicates that occur during Gibbs sampling are calls of CPD-queries. Moreover, there is only a fixed set of CPD-queries that are ever called during Gibbs sampling, and this set can be determined before the start of the sampling process. Concretely, by examining the RESAMPLE procedure (Figure 1) one can see that the only CPD-queries that are ever called are those associated to an unobserved RV (line 1 of RESAMPLE) or to an RV with an unobserved parent (line 5). As long as the specialized decision lists that we construct behave exactly the same with respect to this fixed set of CPD-queries as the original decision lists do, Gibbs sampling will indeed produce exactly the same samples with specialization as without.

There is a lot of existing work on transformation or specialization of logic programs that has the same end-goal as our work, namely transforming a given program to an “equivalent” but more efficient program [9]. However, we are not aware of any work that considers the same setting as we do, namely that of executing a fixed set of queries on a knowledge base with a static and a dynamic part, and specializing with respect to the static part. In particular, this setting makes our work different from the work on *partial deduction* for logic programs [5, 6]. In our setting, we know all input arguments of the queries but we know only part of the knowledge base on which they will be executed. In contrast, in the partial deduction setting, one knows only some of the input arguments of the queries but one knows the entire knowledge base. Hence, existing off-the-shelf systems for partial deduction (see e.g. Leuschel et al. [6]) are, as far as we see, not optimal for our setting.

5.1 Outline of the Specialization Algorithm

The CPD-predicates are defined in terms of the state predicates. The evidence is a partial interpretation of these state predicates (specifying the known state for a subset \mathbf{O} of all concrete RVs). We now want to specialize the definitions of the CPD-predicates with respect to the evidence. Since the evidence is defined at the ground level but the definitions of the CPD-predicates are at the non-ground level, we first have to (partially) ground these definitions before we can specialize them. This is the main idea behind our specialization algorithm, which is shown in Figure 2.

The outer-loop of our algorithm (line 1 of the SPECIALIZE procedure) is over all the CPD-predicates: we specialize each CPD-predicate p in turn. To do so, we first collect all CPD-queries for p . As explained before, the only CPD-queries that we need are the ones associated to an RV that is unobserved or has an unobserved parent. The set of all such CPD-queries is denoted $AllQueries(p, \mathbf{U}, \mathbf{O})$ (line 3 of the SPECIALIZE procedure). We then loop over this set: for each CPD-query q we apply the SPEC_DECISION_LIST procedure. We explain this procedure by means of an example.

Example 2 (Specializing a decision list with respect to a CPD-query) *Let p be $cpd_graduates/2$, let the decision list D that defines p be the same as given earlier in Example 1 (p. 3), and let*

<pre> procedure SPECIALIZE(U, O, o) 1 for each CPD-predicate p 2 let D be the decision list for p 3 for each $q \in AllQueries(p, U, O)$ 4 SPEC_DECISION_LIST(D, q, U, O, o) function SPECIALIZE_BODY(B, U, O, o) 1 $B_1 = GROUND_BODY(B, U \cup O)$ 2 $B_2 = SPECIALIZE_LITERALS(B_1, U, O, o)$ 3 $B_3 = SIMPLIFY_BODY(B_2)$ 4 if B_3 is identical to B_1 5 return B 6 else return B_3 </pre>	<pre> procedure SPEC_DECISION_LIST(D, q, U, O, o) 1 if D is non-empty 2 let C be the first clause in D and D_{rest} be the other clauses in D 3 $C_q = GROUND_HEAD(C, q)$ 4 let $Head$ be the head and B_q the body of C_q 5 $Body = SPECIALIZE_BODY(B_q, U, O, o)$ 6 if $Body = true$ 7 ASSERT_FACT($Head$) 8 else 9 if $Body \neq false$ 10 ASSERT_CLAUSE($Head, Body$) 11 SPEC_DECISION_LIST(D_{rest}, q, U, O, o) </pre>
---	---

Figure 2: The specialization algorithm for the decision lists that define the CPD-predicates (U contains the unobserved RVs, O the observed RVs and o their observed values).

the CPD-query q be $cpd_graduates(s1, Distr)$. The SPEC_DECISION_LIST procedure starts by processing the first clause C in D :

```
cpd_graduates( $S, [yes:0.2, no:0.8]$ ) :- grade( $S, _C, c$ ), !.
```

First we ground the head variables of C with respect to q (line 3 of SPEC_DECISION_LIST) yielding the clause C_q :

```
cpd_graduates( $s1, [yes:0.2, no:0.8]$ ) :- grade( $s1, _C, c$ ), !.
```

Next, we apply the function SPECIALIZE_BODY to the body of C_q (line 5), yielding $Body$. There are three possible cases.

- If $Body$ equals $true$, we assert a fact $cpd_graduates(s1, [yes:0.2, no:0.8])$ (line 7). We can then discard the remaining clauses in D with respect to q (these clauses will never be reached for q since only the first applicable clause in a decision list fires).
- If $Body$ equals $false$, we discard C_q and continue by processing the next clause in D (line 11).
- Otherwise, we assert a clause of the form

```
cpd_graduates( $s1, [yes:0.2, no:0.8]$ ) :-  $Body$ , !.
```

(line 10), and we again continue by processing the next clause in D (line 11).

□

The function SPECIALIZE_BODY (Figure 2) performs three steps which we explain in the next sections. For comprehensibility we already give a simple example.

Example 3 (Specializing the body of a clause in a decision list) Let B , the body to be specialized, be $grade(s1, C, c)$ (this is the situation of our previous example). First we ground the free variable C in B (line 1 of SPECIALIZE_BODY), yielding a disjunction B_1 , namely $grade(s1, c1, c) ; \dots ; grade(s1, cn, c)$. Then we specialize each of the literals in B_1 with respect to the evidence (line 2). Consider the first literal, $grade(s1, c1, c)$. If we have evidence that $s1$ obtained grade ‘ c ’ for course $c1$ then we replace the literal by $true$, if we have different evidence we replace it by $false$, if we have no evidence we leave it unchanged. Doing this for each literal yields a disjunction B_2 . Finally, we simplify B_2 using logical propagation rules (e.g. a disjunction is true if one of its disjuncts is true), yielding B_3 (line 3).

□

There is one exception to the approach illustrated in the example. If there is no evidence at all about B , then B_3 would be identical to B_1 , i.e. we would ground without specializing and simplifying, resulting in code explosion. In preliminary experiments we found that in such cases it is better for efficiency if we do not ground at all (Leuschel and Bruynooghe [5] have similar observations about code explosion). This special case is taken care of by lines 4 and 5 of SPECIALIZE_BODY.

From the perspective of efficiency of the specialization process, our algorithm is clearly not optimal: the specialization time can easily be reduced (for instance by merging the three different steps of SPECIALIZE_BODY). However, in our experiments we observed that the specialization time is negligible as compared to the runtime of Gibbs sampling with the specialized decision lists (see Section 6.2). Hence, we keep our specialization algorithm as simple as possible, rather than complicating it in order to reduce specialization time. This also makes it easier to see that specialization indeed preserves the semantics of the CPD-predicates (and hence that Gibbs sampling produces the same sequence of samples as without specialization).

Note that the above remark is about the optimality of the specialization *process*, not of its *output*. As far as we can judge, the output (the specialized decision lists) is close to the optimal one that can be obtained by means of specialization.

We now explain the three steps in the function SPECIALIZE_BODY of Figure 2.

5.2 Step 1: Compact Grounding of Conjunctions

The first step in SPECIALIZE_BODY is carried out by the function GROUND_BODY. This function takes as input a conjunction of literals (that forms the body of a clause) and partially grounds it. Remember that the rationale for grounding is that we can only specialize a literal with respect to the evidence if the literal is ground (because the evidence is ground as well). Also remember that we use three kinds of predicates in the bodies: state predicates, background predicates, and meta-predicates (Section 3.2). We first consider conjunctions without meta-predicates.

Let us first explain how we deal with a conjunction consisting of a *single literal* L . If L is a state literal (a literal built from a state predicate; case1), we ground it as follows. If the last argument of L is a free variable, we ground that variable with respect to the range of the state predicate. We ground all other arguments of L that are variables with respect to their population (for instance, for the variable C in $grade(s1, C, b)$ this is the population of courses). If L is a background literal (case2), we ground it if possible. Concretely, we first check whether L can be called (sometimes this is not possible for instance because some of its arguments are not yet properly instantiated, see Example 4).⁶ If L can be called, we collect all answer substitutions for the free variables in L by calling L , and we ground with respect to these substitutions. If L cannot be called, we leave it unchanged (non-ground).

The result of grounding a conjunction with a single literal is a disjunction of literals (since free variables in the body are existentially quantified). For instance, grounding $grade(s1, C, c)$ gives a disjunction $grade(s1, c1, c) ; \dots ; grade(s1, cn, c)$.

Let us now explain how we deal with a conjunction of *multiple literals*. We traverse the conjunction and whenever we encounter a literal L with free variables, we ground these variables as above (we ground them in all literals that contain them). In principle, the result would be a disjunction of conjunctions. However, we try to represent the grounding as compactly as possible by means of an (arbitrarily complex) formula with nested disjunctions and conjunctions. For instance, we do not ground the conjunction $p, q(X)$ as $(p, q(x1) ; \dots ; p, q(xn))$ but as $p, (q(x1) ; \dots ; q(xn))$. To do this we recursively decompose the given conjunction into independent components before we ground it. We do this decomposition in the same way as Santos Costa et al. [13] and Struyf [14, Ch.3] in their “once-transformation”.

⁶Currently we use input from the user to know when this is the case. The input is similar to the “rescall” annotations of Leuschel et al. [6].

Let us now consider conjunctions with meta-predicates. Our specialization algorithm currently only supports *findall*/3 as we found this to be sufficient for real-world models. To deal efficiently with *findall* atoms we distinguish two possible ways in which they can be used (we use input from the user to make the distinction). We speak of a *count-findall* if the function of the *findall* is only to count how many solutions there are. We speak of a *collect-findall* otherwise. The reason why we give special attention to *findall*'s (and especially *count-findall*'s) is that they are used in most of our real-world models. Without specialization *findall*'s are computationally expensive. Hence, it is important that our specialization algorithm deals with them efficiently.

To ground a *count-findall* we ground its second argument like we ground any other conjunction (hence this argument will become a disjunction). We also replace the first argument of the *findall* by a dummy element (to count the number of solutions, see below). To ground a *collect-findall* we also ground its second argument as usual, except that we do not ground any variables that also occur in the first argument (since they are needed to collect the solutions). We illustrate this with an example.

Example 4 (Grounding a conjunction that contains a *findall*) Consider the following conjunction with a *count-findall*.

```
findall(C,grade(s1,C,a),L), length(L,N), N<2
```

We ground this conjunction as follows (the constant *d* denotes a dummy element).

```
findall(d,( grade(s1,c1,a) ; ... ; grade(s1,cn,a) ),L),
        length(L,N), N<2
```

Note that we cannot easily ground *L*, and hence *N* neither.

Now consider the following conjunction with a *collect-findall*.

```
findall(V,(grade(s1,C,c),level(C,V)),L), ...
```

We ground this conjunction as follows.

```
findall(V,( grade(s1,c1,c),level(c1,V) ;
        ... ; grade(s1,cn,G),level(cn,V) ),L), ...
```

□

To summarize, `GROUND_BODY` takes as input a conjunction of literals and partially grounds it, yielding a formula with nested disjunctions and conjunctions.

5.3 Step 2: Specialization of Literals

The (partial) grounding step performed by `GROUND_BODY` paves the way for the actual specialization with respect to the evidence. This happens in two steps. We now discuss the first step, which is carried out by the function `SPECIALIZE_LITERALS`.

In the function `SPECIALIZE_LITERALS`, we traverse the formula constructed by `GROUND_BODY` and for each literal *L* that we encounter (also literals inside the second argument of a *findall*) we apply the function `SPEC_LITERAL` of Figure 3.

As mentioned in the previous section, we only specialize a literal if it has been fully grounded by `GROUND_BODY`. Hence, when a literal *L* is non-ground we leave it unchanged (line 7 of `SPEC_LITERAL`). If *L* is a ground state atom, we specialize it with respect to the evidence by using the function `SPEC_STATE_ATOM` (line 3). If *L* is a negated ground state atom, we instead use the function `SPEC_NEG_STATE_ATOM` (line 5), which is the same but with the roles of true and false reversed. The only remaining possibility is that *L* is a ground background literal. In that case, we specialize *L* with respect to the background knowledge base: we simply call *L*; if this succeeds, then *L* is replaced by *true*, otherwise by *false* (line 6).

```

function SPEC_LITERAL( $L, U, O, o$ )
1 if  $L$  is ground
2   if  $L$  is a state atom
3     return SPEC_STATE_ATOM( $L, U, O, o$ )
4   else if  $L$  is a negated state atom
5     return SPEC_NEG_STATE_ATOM( $L, U, O, o$ )
6   else return CALL_LITERAL( $L$ )
7 else return  $L$ 

function SPEC_STATE_ATOM( $A, U, O, o$ )
1 if  $A$  has an associated RV  $X \in O \cup U$ 
2   if  $X \in O$  // there is evidence about  $X$ 
3     if  $A$  is consistent with the evidence in  $o$ 
4       return true
5     else return false
6   else return  $A$  // no evidence, so no specialization
7 else return false // non-existent RV

```

Figure 3: Specializing a literal L with respect to evidence (observed states o for RVs O).

5.4 Step 3: Simplification of Specialized Formulas

The previous step does not change the structure of the formula that it operates on (the nested disjunctions and conjunctions) but only the individual literals in the formula: some literals are replaced by *true*, some by *false*, the others are left unchanged. In the third step (SIMPLIFY_BODY), we simplify the resulting formula.

Let us first consider formulas without *findall*'s. We traverse the given formula as shown in Figure 4. For a disjunction we first recursively simplify each of the disjuncts (line 3). Then we simplify the disjunction itself by two simple logical propagation rules: 1) if a disjunct is *true*, the entire disjunction is *true*; 2) disjuncts that are *false* can be dropped (line 4). We deal with conjunctions in a similar way.

```

function SIMPLIFY_BODY( $F$ )
1 if  $F$  is a disjunction
2   for each disjunct  $F_i$  of  $F$ 
3      $S_i =$ SIMPLIFY_BODY( $F_i$ )
4   return SIMPLIFY_DISJUNCTION( $\{S_1, \dots, S_n\}$ )
5 else if  $F$  is a conjunction
6   for each conjunct  $F_i$  of  $F$ 
7      $S_i =$ SIMPLIFY_BODY( $F_i$ )
8   return SIMPLIFY_CONJUNCTION( $\{S_1, \dots, S_n\}$ )
9 else return  $F$  // base case:  $F$  is a single literal

```

Figure 4: Simplifying a formula F (for formulas without *findall*).

Let us now consider formulas that contain *findall*'s. We traverse the formula as above. At some point we will encounter a literal of the form *findall*(C, Q, L). Note that Q will always be a disjunction (because Q was the result of a grounding step in GROUND_BODY). We first recursively simplify each of its disjuncts, yielding Q_2 .

- When dealing with a *collect-findall* we know that each disjunct in Q_2 is either *false* or some formula (but not *true*⁷). We remove all disjuncts that are *false* from Q_2 , yielding Q_3 . The end-result is a literal *findall*(C, Q_3, L).
- With a *count-findall*, each disjunct in Q_2 is either *false* or *true* or some formula. We remove all disjuncts that are *false* or *true* from Q_2 , yielding Q_3 . To preserve correctness, we adapt the corresponding count based on the number of *true* disjuncts that were removed, and we propagate this information. Let us explain this further with a somewhat extreme but illustrative example.

Example 5 (Simplifying a formula that contains a count-*findall*) Consider the following conjunction (d is the dummy of Example 4, p. 9).

```

findall( $d, ( \text{grade}(s1, c1, a) ; \text{true} ; \text{grade}(s1, c3, a) ; \text{false} ; \text{true} ), L$ ),
length(L, N), N < 2.

```

⁷The result of applying GROUND_BODY to the second argument of a collect-*findall* is a disjunction of conjunctions each containing at least one non-ground atom (Section 5.2). Since non-ground atoms are left unspecialized, no disjunct will be reduced to *true* during specialization.

We can safely remove the false disjunct in the *findall*. We can also remove the two true disjuncts if we take them into account in the computation of the count N .

```
findall(d, (grade(s1, c1, a) ; grade(s1, c3, a)), L), length(L, M),
      N is M+2, N<2.
```

Using simple arithmetic, this conjunction is further simplified to the following.

```
findall(d, (grade(s1, c1, a) ; grade(s1, c3, a)), L), length(L, M), M<0.
```

Since M is at least 0, the entire conjunction is further simplified to false. \square

6 Experiments

We now experimentally analyze the influence of specializing the definitions of the CPD-predicates on the efficiency of the Gibbs sampling algorithm.

6.1 Experimental Setup

We test our algorithms on three real-world datasets: IMDB, UWCSE and WebKB. These datasets are common benchmarks in the area of probabilistic logical models [3]. In previous work we have applied machine learning algorithms to these datasets [4]. For each dataset we converted the learned model to a parameterized Bayesian network. Table 1 gives some statistics about the models and the data (see Fierens et al. [4] for more information).

Table 1: Statistics about the data (number of parameterized and concrete RVs) and the models (number of clauses in the decision lists and usage of *findall*'s).

Dataset	Parameterized RVs	Concrete RVs	Clauses	Count- <i>findall</i> 's	Collect- <i>findall</i> 's
IMDB	7	2852	13	yes	no
UWCSE	10	9607	32	yes	yes
WebKB	5	78132	12	no	no

We use two inference scenarios, corresponding to the two scenarios of Section 4.2. The first scenario is *'prediction'*: there is one parameterized RV that we want to predict, all concrete RVs associated to that parameterized RV are unobserved, all others are observed. For each dataset we do multiple experiments, each time with a different parameterized RV as the prediction target.⁸ The second scenario is *'missing data'*: a random fraction f of all concrete RVs is unobserved ('missing'), the others are observed. We use several values of f , ranging from 5% to 50%. For each value we repeat each experiment 5 times, each time with different unobserved RVs. We report the mean and standard deviation of the runtime across these 5 repetitions.

We measure the time to draw 10000 samples with our Gibbs sampling algorithm.⁹ We report runtimes in minutes. The *runtime without specialization* is the runtime of Gibbs sampling with parameterized CPDs that have not been grounded or specialized. The *runtime with specialization* is the sum of the specialization time and the runtime of Gibbs sampling with the specialized CPDs. Recall that both settings produce exactly the same sequence of samples.

⁸For more than half of all parameterized RVs, predicting them is trivial from a probabilistic perspective (it does not require Gibbs sampling). We exclude such parameterized RVs as targets. We also exclude the prediction of the parameterized RV 'prof' on the WebKB dataset since these experiments timed-out.

⁹Since our main goal is to investigate the *relative* efficiency of the different settings (with versus without specialization), the choice of the number of samples does not heavily influence our conclusions.

6.2 Main Results

The results for the ‘missing data’ scenario are shown in Figure 5. The most important result is that using specialization always yields a speedup. The magnitude of the speedup of course greatly depends on the amount of evidence. On WebKB, the dataset that is by far the most computationally demanding, we get a speedup of an order of magnitude when there are 5% unobserved RVs. On the smaller datasets (IMDB and UWCSE), the speedups are more modest.

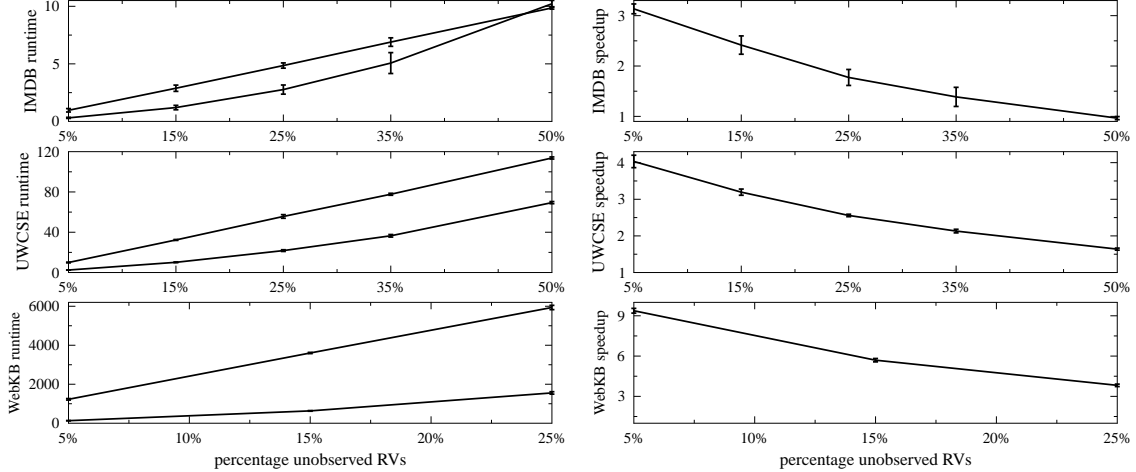


Figure 5: Results for the ‘missing data’ scenario. Left subgraphs show the runtime without (upper line) and with specialization (lower line); right subgraphs show the corresponding speedup-factor achieved due to specialization. Error bars indicate the standard deviation.

The results for the ‘prediction’ scenario are shown in Table 2. For half of the prediction targets, specialization yields significant speedups of a factor 4 to 7. For the other targets, the speedup is small to negligible (≤ 1.5). These are mostly cases where the state predicate that forms the computational bottleneck (e.g. because it is involved in a *findall*) is unobserved and hence cannot be specialized on.

Table 2: Results for the ‘prediction’ scenario: runtime without specialization, runtime with specialization and speedup-factor achieved due to specialization.

Data/Target	No spec.	Spec.	Speedup	Data/Target	No spec.	Spec.	Speedup
IMDB/acts	16.1	14.9	1.08	UWCSE/phase	12.2	2.1	5.87
IMDB/directs	2.6	1.7	1.51	UWCSE/teaches	71.8	15.8	4.55
UWCSE/advisedby	75.1	17.4	4.31	WebKB/hasproject	2628	406	6.48
UWCSE/coauthor	10.9	10.4	1.05				

Recall that we compute the runtime with specialization as the specialization time (t_{spec}) plus the runtime for Gibbs sampling with specialized CPDs (t_{run}). We also measured the fraction of time spent on specialization: $t_{spec}/(t_{spec} + t_{run})$. We found that this fraction is typically very low (on average 2.3%). This shows that there is no point in making the specialization process itself faster (Section 5.1).

6.3 Influence of the Size of the Data

In the above results (especially for the ‘missing data’ scenario), the speedups are the lowest on the smallest dataset (IMDB) and the highest on the largest one (WebKB). This suggests a correlation between the speedup due to specialization and the data-size. To investigate this, we performed additional experiments in which we varied the size of the datasets. Figure 6 shows the

results for the ‘missing data’ scenario with 15% unobserved RVs (results for the other settings are very similar). The trend in the speedup is clear: the larger the dataset, the higher the speedup. This is a positive result: speedups are more necessary on large datasets than on small ones.

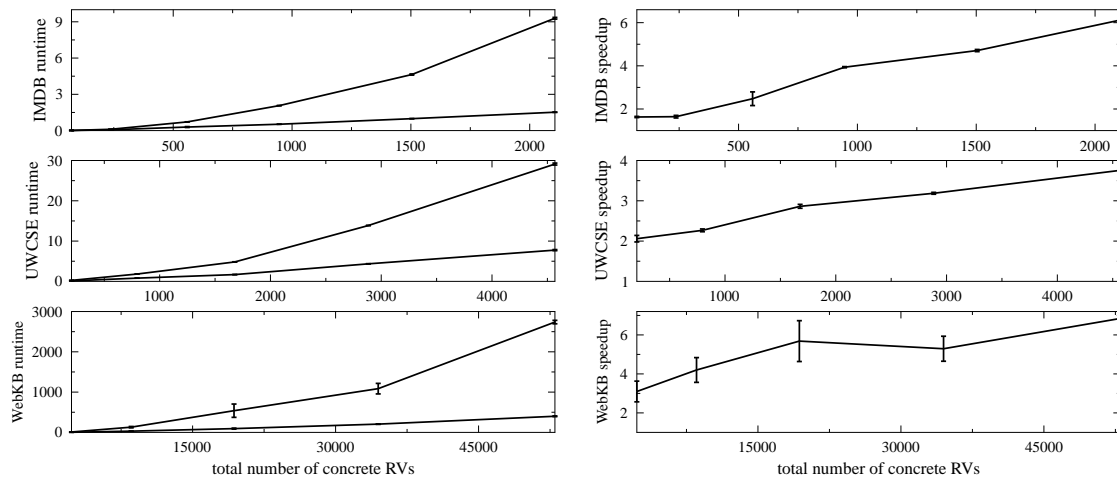


Figure 6: Influence of the data-size (number of concrete RVs) for the ‘missing data’ scenario with 15% unobserved RVs. (The meaning of each subgraph is the same as in Figure 5.)

7 Directions for Future Work

There are several interesting directions for future research. One direction is to try to further speed up Gibbs sampling by applying other optimization techniques (besides program specialization) from the field of logic programming. A promising kind of techniques are *query transformations*. The idea would be to transform the bodies of the decision lists to equivalent expressions that can be evaluated more efficiently. One kind of query transformation is *query reordering*: the idea is to put the most selective literals in a conjunction first [15]. The main challenge is to estimate which literals are most selective. It would be interesting to see whether the probability distributions specified by the CPDs can be used to provide useful clues about this. Also other kinds of query transformations, such as the *once-* and *theta-*transformations of Santos Costa et al. [13], seem promising.

A second interesting direction for future research is to extend our representation. One useful extension is to allow also other formats for defining the CPD-predicates than decision lists (under the restriction that each CPD-predicate remains functional: each CPD-query should return exactly one probability distribution). Another extension is to allow RVs with an infinite range, such as numerical RVs. Such extensions of course require adaptations to our specialization algorithm.

8 Conclusions

We considered the task of performing approximate probabilistic inference with probabilistic logical models by means of Gibbs sampling. We used the general framework of parameterized Bayesian networks. We showed how to represent the considered models and how to implement a Gibbs sampling algorithm for such models in Prolog. We argued that several techniques from the field of logic programming, such as program specialization and query transformations, are suited to make this algorithm more efficient, which can in turn make the obtained inference answers more accurate. We developed an algorithm for specializing our logic programs with respect to the evidence, and experimentally investigated the influence of specialization on the

efficiency of Gibbs sampling. Our results show that specialization yields speedups of up to an order of magnitude and that these speedups grow with the data-size.

Acknowledgements.

This research is supported by Research Foundation-Flanders (FWO Vlaanderen), GOA/08/008 ‘Probabilistic Logic Learning’ and Research Fund K.U.Leuven.

References

- [1] B. Bidyuk and R. Dechter. Cutset sampling for Bayesian networks. *Journal of Artificial Intelligence Research*, 28:1–48, 2007.
- [2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [3] L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton. *Probabilistic Inductive Logic Programming*. Springer, 2008.
- [4] D. Fierens, J. Ramon, M. Bruynooghe, and H. Blockeel. Learning directed probabilistic logical models: Ordering-search versus structure-search. *Annals of Mathematics and Artificial Intelligence*, 54(1):99–133, 2008.
- [5] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4-5):461–515, 2002.
- [6] M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, volume 3094 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.
- [7] X.-L. Li and Z.-H. Zhou. Structure learning of probabilistic relational models from incomplete relational data. In *Proceedings of the 18th European Conference on Machine Learning (ECML 2007)*, volume 4701 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2007.
- [8] R. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, New Jersey, 2003.
- [9] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19-20:261–320, 1994.
- [10] D. Poole. First-order probabilistic inference. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 1997)*, pages 985–991. Morgan Kaufmann, 2003.
- [11] H. Poon and P. Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, pages 214–225. AAAI Press, 2006.
- [12] V. Santos Costa. On the implementation of the CLP(BN) language. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, volume 5937 of *Lecture Notes in Artificial Intelligence*, pages 234–248. Springer, 2010.
- [13] V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2003.

- [14] J. Struyf. *Improving the efficiency of inductive logic programming in the context of relational data mining*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, December 2004.
- [15] J. Struyf and H. Blockeel. Query optimization in inductive logic programming by re-ordering literals. In *Proceedings of the 13th International Conference on Inductive Logic Programming (ILP 2003)*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 329–346. Springer-Verlag, 2003.