# CLP($\mathcal{BN}$): Constraint Logic Programming for Probabilistic Knowledge

Vítor Santos Costa[1], David Page[2], and James Cussens[3]

[1] DCC-FCUP and LIACC
Universidade do Porto
Portugal
vsc@dcc.fc.up.pt

[2] Dept. of Biostatistics and Medical Informatics
University of Wisconsin-Madison
USA
page@biostat.wisc.edu

[3] Department of Computer Science
University of York UK
jc@cs.york.ac.uk

**Abstract.** In Datalog, missing values are represented by Skolem constants. More generally, in logic programming missing values, or existentially quantified variables, are represented by terms built from Skolem functors. The CLP($\mathcal{BN}$) language represents the joint probability distribution over missing values in a database or logic program by using constraints to represent Skolem functions. Algorithms from inductive logic programming (ILP) can be used with only minor modification to learn CLP($\mathcal{BN}$) programs. An implementation of CLP($\mathcal{BN}$) is publicly available as part of YAP Prolog at `http://www.ncc.up.pt/~vsc/Yap`.

## 1 Introduction

One of the major issues in knowledge representation is how to deal with incomplete information. One approach to this problem is to use probability theory in order to represent the likelihood of an event. More specifically, advances in representation and inference with Bayesian networks have generated much interest and resulted in practical systems, with significant industrial applications [1]. A Bayesian network represents a joint distribution over a set of random variables where the network structure encapsulates conditional independence relations between the variables.

A Bayesian network may be seen as establishing a set of relations between events. This presents a clear analogy with propositional calculus, as widely discussed in the literature [2], and raises the question of whether one could move one step forward towards a Bayesian network system based on the more powerful predicate calculus. Arguably, a more concise representation of Bayesian Networks would avoid wasted work and possible mistakes. Moreover, it would make it easier to learn interesting patterns in data. Work such as Koller's Probabilistic Relational Models (PRMs) [3], Sato's PRISM [4], Ngo and Haddawy's

Probabilistic Logic Programs [5], Muggleton and Cussens' Stochastic Logic Programs [6], and Kersting and De Raedt's Bayesian Logic Programs [7] have shown that such a goal is indeed attainable.

The purpose of probabilistic first order languages is to propose a concise encoding of probability distributions for unobserved variables. Note that manipulating and reasoning on unknown values is a well-known problem in first-order representations. As an example, First-Order Logic is often used to express existential properties, such as:

$$\forall x \exists y, Make(x) \rightarrow OwnsCar(y, x)$$

A natural interpretation of this formula is that every make of car has at least one owner. In other words, for every make $x$ there is an individual $y$ that owns a car of this make. Notice that the formula does not state who the owner(s) may be, just that one exists. In some cases, e.g., for inference purposes, it would be useful to refer to the individual, even if we do not know its actual name. A process called *skolemization* can replace the original formula by a formula without existential quantifiers:

$$\forall x, Make(x) \rightarrow OwnsCar(y, s(x))$$

where $y = s(x)$ and $s(x)$ is called a Skolem function: we know the function describes an individual for each $x$, but we do not know which individual.

Skolem functions have an interesting analogy in probabilistic relational models (PRMs) [3]. PRMs express probability distributions of a field in the database by considering related fields, thus encoding a Bayesian network that represents the joint probability distribution over all the fields in a relational database. The Bayes network constructed by PRMs can then be used to infer probabilities about *missing values* in the database. We know that the field must take one value, we know that the value will depend on related fields, and we know the values for at least some of these related fields. As for Skolem functions, PRMs refer to fields that are unknown function of other fields. But, in contrast with First Order Logic, PRMs do allow us to estimate probabilities for the different outcomes of the function: they allow us to represent partial information on Skolem functions.

Can we take this process a step further and use a Bayesian network to represent the joint probability distribution over terms constructed from the Skolem functors in a logic program? We extend the language of logic programs to make this possible. Our extension is based on the idea of defining a language of Skolem functions where we can express properties of these functions. Because Skolem functions benefit from a special interpretation, we use Constraint Logic Programming (CLP), so we call the extended language CLP($\mathcal{BN}$). We show that any PRM can be represented as a CLP($\mathcal{BN}$) program.

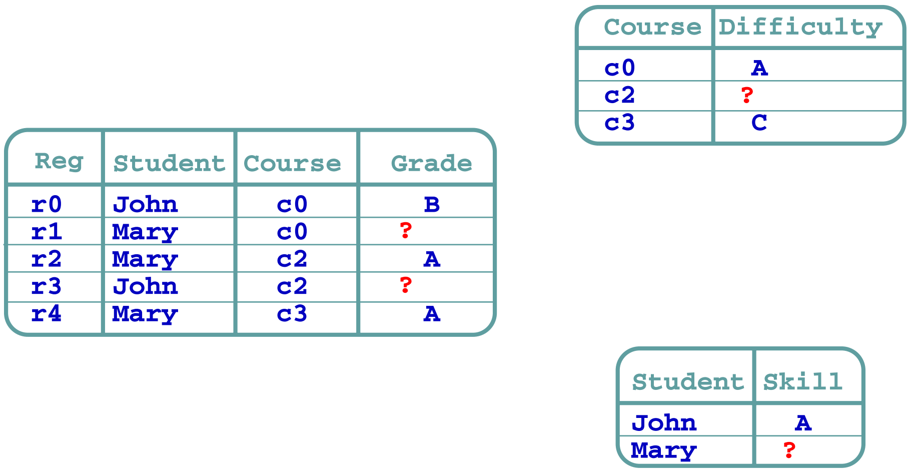
Our work in CLP($\mathcal{BN}$) has been motivated by our interest in multi-relational data mining, and more specifically in inductive logic programming (ILP). Because CLP($\mathcal{BN}$) programs are a kind of logic program, we can use existing ILP systems to learn them, with only simple modifications to the ILP systems. Induction of clauses can be seen as model generation, and parameter fitting can be seen as generating the CPTs for the constraint of a clause. We show that the ILP system ALEPH [8] is able to learn CLP($\mathcal{BN}$) programs.

Next, we present the design of CLP($\mathcal{BN}$) through examples. We then discuss the foundations of CLP($\mathcal{BN}$), including detailed syntax, proof theory (or operational semantics), and model-theoretic semantics. We next discuss important features of CLP($\mathcal{BN}$), namely its ability to support aggregation and recursion. Finally, we present the results of experiments in learning CLP($\mathcal{BN}$) programs using ILP. Lastly, we relate CLP($\mathcal{BN}$) with PRMs and with other related work.

## 2 CLP($\mathcal{BN}$) Goes to School

We introduce CLP($\mathcal{BN}$) through a simplified version of the school database originally used to explain Probabilistic Relational Models [3] (PRMs). We chose this example because it stems from a familiar background and because it clearly illustrates how CLP($\mathcal{BN}$) relates to prior work on PRMs. Figure 2 shows a simplified fragment of the school database. The schema consists of three relations: *students*, *courses*, and *grades*. For each student, we have a primary key, `Student`, and its `Skill`. To simplify, the value for skill is the expected final grade of the student: an `A` student would thus be a top student. For each course, `Course` is the primary key and `Difficulty` gives the course's difficulty: an `A` difficulty course would be a course where we would expect even the average student to do very well. Lastly, the *Registration* records actual student participation in a course. This table's key is a registration key. Both `Student` and `Course` are foreign keys giving student and course data. The last field in the table gives the `grade` for that registration.

Figure 1 shows an example database with these 3 tables. Notice that some non-key data is missing in the database. For example, we do not know what was `mary`'s grade on `c0`, maybe because the grade has not been input yet. Also, we



| Course | Difficulty |
|---|---|
| c0 | A |
| c2 | ? |
| c3 | C |

| Reg | Student | Course | Grade |
|---|---|---|---|
| r0 | John | c0 | B |
| r1 | Mary | c0 | ? |
| r2 | Mary | c2 | A |
| r3 | John | c2 | ? |
| r4 | Mary | c3 | A |

| Student | Skill |
|---|---|
| John | A |
| Mary | ? |

**Fig. 1.** A Simplified School Database with Tables on Students, Courses and Grades

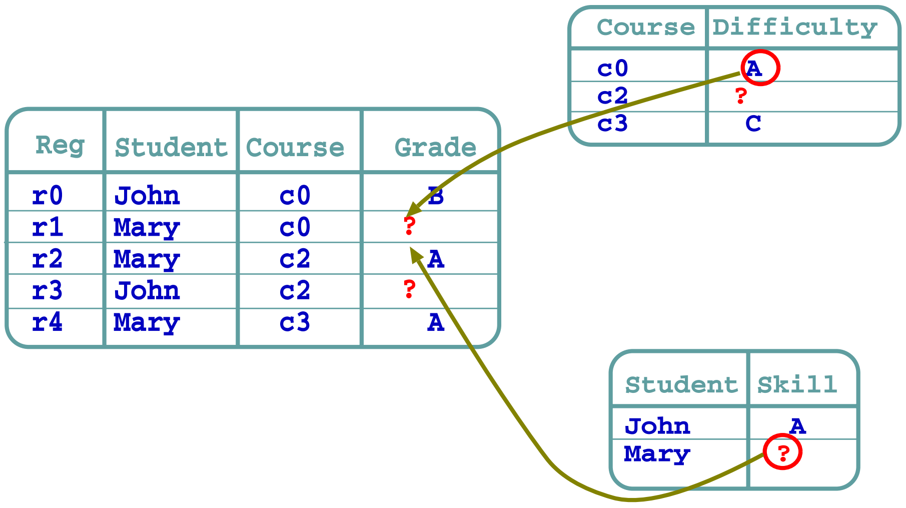


**Fig. 2.** Direct Dependency Between Random Variables in Simplified School Database

do have `john`'s skill, but we could not obtain data on `mary`. This is a common problem in databases: often, the database only contains partial data for some items. A fundamental observation in the PRM design is that such missing data can be represented as *random variables*. The idea is that columns for which there is missing data should be seen as sets of random variables. If a specific value is known, we say that the database includes *evidence* on this item. For example, we may say that the `Skill` attribute in the Student table is a set of random variables, and that we have evidence for the skill variable corresponding to key `john`. Other sets of random variables are for `Difficulty` and `Grade`.

An immediate step in the PRMs is that we can estimate probabilities on the missing values through considering other items in the database. Such items may be on the same relation, or may also be found at a different relations. In our database example, for instance, it would make sense to assume that better skilled students would have better grades, e.g., an `A` level student would have an higher probability of achieving an `A`. Moreover, it makes sense to assume that a grade will also depend on the course's difficulty: the easier the course, the better the grades. We can go one step further and argue that given course and student information, a grade should be *conditionally independent* on the other elements on the database. This reasoning suggests that all the random variables in our PRM-extended database should form a Bayesian network.

Figure 2 shows a fragment of the Bayesian network induced by this rule. At first sight, our expectations for `mary`'s grade on course `c0` depend on data we have on `mary` and course `c0`.

CLP($\mathcal{BN}$) is grounded on the idea that such beliefs can easily be expressed in Logic. Namely, the previous example can be expressed in a variety of ways, but one approach (using Prolog Notation) would be as follows:

```
grade(r1,Grade) :-
    skill(mary,Mskill),
    difficulty(c0,C0Difficulty),
    Grade = ‘G’(MSkill,C0Difficulty).
```

Grade is as an attribute of registration r1. We know that its actual value will depend on mary's skill and course c0's difficulty. The clause says exactly that: Grade is a random variable that can also be described as an unknown function, G(), of r1, mary's skill, Mskill, and c0's difficulty, C0Difficulty. Such unknown functions are often used in Logic, where they are usually called *Skolem functions*: thus, in our formalism we shall say that a random variable is given by a Skolem function of its arguments.

Note that we do have some expectations on G(). Such data can be expressed in a variety of ways, say through the fact:

```
random_variable(‘G’(_,_),[‘A’,‘B’,‘C’,‘D’],[0.4,0.3,0.2,0.1]).
```

that states that the random variable has domain A,B,C,D and a discrete conditional probability table that we represent as a Prolog list with a number of floating point values between 0 and 1.

Of course, one of the main advantages of the PRMs (and of using first-order representations), is that we can generalize. In this case, we could write a single rule for Grade by lifting the constants in our grade clause and making the individual registration an argument to the Skolem function. We will need to access the foreign keys, giving the following clause:

```
grade(Registration,Grade) :-
    registration_student(Registration, Student),
    registration_course(Registration, Course),
    difficulty(Course,Difficulty),
    skill(Student,Skill),
    Grade = ‘S’(Registration,Skill,Difficulty).

random_variable(‘S’(_,_,_),[‘A’,‘B’,‘C’,‘D’],[0.4,0.3,0.2,0.1]).
```

Next, we need rules for difficulty and skill. In our model, we do not have helpful dependencies for Skill and Difficulty, so the two columns should be given from priors. We thus just write:

```
skill(Student,‘S1’(Student)).
difficulty(Course,‘S2’(Course)).

random_variable(‘S1’(_),[‘A’,‘B’,‘C’,‘D’],[0.25,0.25,0.25,0.25]).
random_variable(‘S2’(_),[‘A’,‘B’,‘C’,‘D’],[0.25,0.25,0.25,0.25]).
```

At this point we have a small nice little logic program that fully explains the database. We believe this representation is very attractive (indeed, a similar approach was proposed independently by Blockeel [9]), but it does have one major limitation: it hides the difference between doing inference in first order logic and in Bayesian network, as we discuss next.

*Evidence.* We have observed that `mary`'s grade on `c0` depends on two factors: course `c0`'s difficulty, and `mary`'s skill. In practice, the actual database does have some extra information that can help in refining the probabilities for this grade.

First, we have actual data on an item. Consider `c0`'s difficulty: we actually know that `c0` is an easy course. We thus have two sources of information about `c0`'s difficulty: we know that it is a random function, `'S1'(c0)`; but we also know that it takes the value `A`. Logically, this evidence can be seen as setting up the equation `S1(c0) = 'A'`. Unfortunately, unification cannot be easily redefined in Prolog.

One simple solution would be to add evidence through an extra fact:

```
evidence('S1'(c0),'A').
```

We assume that somehow this evidence is going to be used when we actually run the program. This solution is not entirely satisfactory, as we now have two separate sources of data on skill: the `skill/2` relation and some facts for `evidence/2`.

Evidence plays indeed a very important role in Bayesian networks. Imagine we want to know the probability distribution for `mary`'s grade on course `c0`. We have more accurate probabilities knowing `c0` is an easy course. And, even though we do not have actual evidence on `mary`'s skill, `Mskill`, we can achieve a better estimate for its probability distribution if we consider evidence relevant to `Mskill`. Namely, `mary` has attended two other courses, `c2` and `c3`, and that she had good grades on both. In fact, course `c3` should be quite a good indicator, as we know grades tend to be bad (a `C`). We do not know the difficulty of course `c2`, but we can again make an estimate by investigating evidence on the students that attended this course. Following all sources of evidence in a network can be quite complex [10], even for such simple examples. In this case, the result is the network shown in Figure 3. Bayesian networks have developed a number of both exact and approximate methods to estimate the probabilities for `Grade` given all the evidence we have on this graph [1].

Evaluating all the relevant evidence is a complex process: first, we need to track down all relevant sources of evidence, through algorithms such as knowledge based model construction [11]. Next, we need to perform probabilistic inference on this graph and marginalize the probabilities on the query variables. To do so would require an extra program, which would have to process both the original query and every source of evidence.

*Constraints.* The previous approach suggests that a Prolog only approach can be used to represent all the properties of Bayesian networks, but that it does expose the user to the mechanisms used by the Bayesian network to accept and propagate evidence. The user would have the burden of knowing which random variables have evidence, and she would be responsible to call a procedure for probabilistic inference.

Such difficulties suggest that we may want to work at an higher abstraction level. Constraint Logic Programming is an important framework that was designed in order to allow specific *interpretations* on some predicates of interest. These interpretations can then be used to implement specialized algorithms over

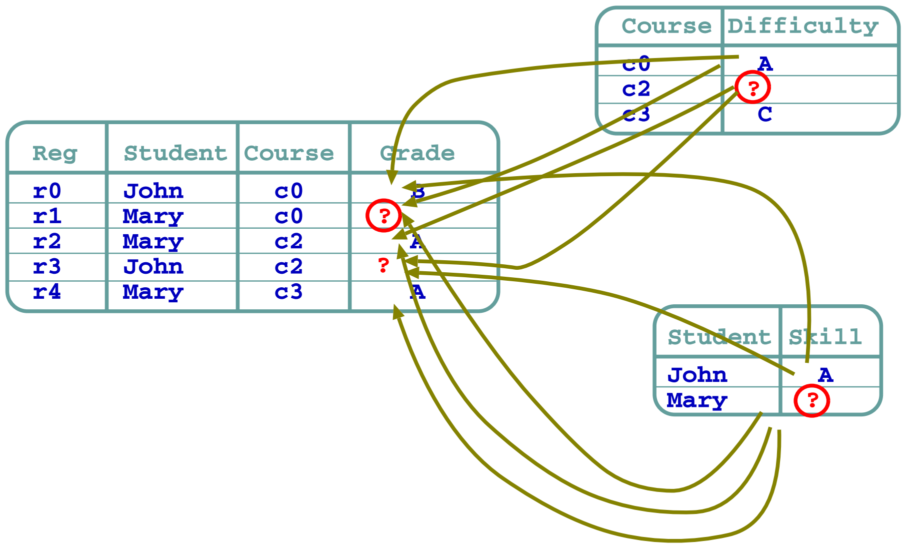


**Fig. 3.** Full Bayesian Network Induced by Random Variables in the Network

the standard Prolog inference. For example, CLP($\mathcal{R}$) defines constraints over reals: it redefines equality and arithmetic operations to create constraints, or *equations*, that are manipulated by a separate *constraint solver*. The constraint solver maintains a *Store* of all active arithmetic constraints, and calls specialized algorithms to process the equations into some canonical form.

We believe the constraint framework is a natural answer to the problems mentioned above. First, through constraints we can extend *equality* to support evidence. More precisely we can redefine the equality:

```
{ S1(c0) = 'A' }
```

to state that random variable `S1(c0)` has received evidence `A`. In constraint programming, we are not forced to bind `S1(c0)` to `A`. Instead, we can add `S1(c0) = 'A'` to a store for later evaluation.

The second advantage of using constraint programming is that it is natural to see a Bayesian Network as a store: both constraints stores and Bayesian networks are graphs; in fact, it is well known that there is a strong connection between both [12]. It is natural to see the last step of probabilistic inference as constraint solving. And it is natural to see marginalization as projection.

Moreover, because constraint stores are opaque to the actual inference process, it is possible to have a *global constraint store* that accommodates all evidence so far. In other words, any time we add evidence to the database we can add this evidence to a global constraint store. Probabilistic inference will then be called having this global Bayesian network as a backdrop.

We have presented the key ideas of CLP($\mathcal{BN}$). Next, we discuss the CLP($\mathcal{BN}$) language in some more detail.

## 3 The CLP($\mathcal{BN}$) Language

A CLP($\mathcal{BN}$) program is a constraint logic program that can encode Bayesian constraints. Thus, CLP($\mathcal{BN}$) programs are sets of Prolog clauses, where some clauses may contain $\mathcal{BN}$ constraints. $\mathcal{BN}$ constraints are of the form {`X = F with P`}, where $X$ must be a Prolog variable, $F$ a term, and $P$ a probability distribution.

As an example, consider the following clause:

```
skill(S,Skill) :-
 {Skill = skill(S) with p([ ‘A’, ‘B’, ‘C’, ‘D’],
                          [0.25,0.25,0.25,0.25],[])}.
```

This clause sets a constraint on `Skill`. The constraint declares that `Skill` should be constrained to the term `skill(S)`, an unknown function, or *Skolem function*, of `S`. Throughout the paper we refer to this term that uniquely identifies a random variable as the *Skolem term*. The constraint declares some further information on `skill(S)` through the *with* construct. In this case, the right hand side of `with` declares that *skill*($S$) is a discrete random variable with 4 possible values and a prior distribution:

1. `skill(S)` has domain `A`, `B`, `C` and `D`;
2. it has an uniform probability distribution over those values;
3. and that `skill(S)` has no parent nodes.

The right-hand-side of the `with` is a Prolog term. Thus, the same constraint could be written as:

```
skill(S,Skill) :-
 cpt(skill(S), CPT),
 {Skill = skill(S) with CPT }.

cpt(skill(_), p([ ‘A’, ‘B’, ‘C’, ‘D’],
                [0.25,0.25,0.25,0.25],[])).
```

One advantage of this approach is that it makes it straightforward to represent different CPTs for different students with a single constraint. Imagine we have extra information on student’s academic story: in this case, we could expect senior students to have better skills than first-year students.

```
skill(S,Skill) :-
 cpt(skill(S), CPT),
 {Skill = skill(S) with CPT }.

cpt(skill(S), p([‘A’,‘B’,‘C’,‘D’],[PA, PB, PC, PD],[])) :-
  skill_table(S, PA, PB, PC, PD).
skill_table(S, 0.25, 0.25, 0.25, 0.25) :-
  freshman(S).
skill_table(S, 0.35, 0.30, 0.20, 0.15) :-
  sophomore(S).
skill_table(S, 0.38, 0.35, 0.17, 0.10) :-
  junior(S).
skill_table(S, 0.40, 0.45, 0.15,0.00) :-
  senior(S).
```

In general, all CLP($\mathcal{BN}$) objects are first class objects. They can be specified as compile-time constants, but they can also be computed through arbitrary logic programs. And they can be fully specified before or *after* setting up the constraint, so

```
skill(S,Skill) :-
 {Skill = skill(S) with CPT },
 cpt(skill(S), CPT).
```

is a legal program, and so is:

```
skill(S,CPT,Skill) :-
 {Skill = skill(S) with CPT }.
```

*Conditional Probabilities.* Let us next consider an example of a conditional probability distribution (CPT). We may remember from Figure 2 that a registration's grade depends on the course's difficulty and on the student's intelligence. This is encoded in the following clause:

```
grade(Registration, Grade) :-
  registration_student(Registration, Student),
  registration_course(Registration, Course),
  difficulty(Course,Dif),
  intelligence(Student,Skill),
  grade_table(TABLE),
  {
     Grade = grade(Course, Dif, Skill) with
       p([‘A’,‘B’,‘C’,‘D’],TABLE,[Dif,Skill])
  }.
```

The constraint says that *Grade* is a Skolem function of *Reg*, *Dif*, and *Skill*. We know that *Grade* must be unique for each *Reg*, and we know that the probability distribution for the possible values of *Grade* depend on the random variables *Dif* and *Skill*. These variables are thus the parents in *Grades*'s CPT, i.e., the third argument in the `with` term. The actual table must include $4^3$ cases: we save some room in the text by assuming it was given by an auxiliary predicate `grade_table/1`.

Figure 4 shows an alternative, pictorial, representation for a CLP($\mathcal{BN}$) clause in this example. The representation clearly shows the clause as having two components: the logical component sets all variables of interest, and the Bayesian constraint connects them in a sub-graph.

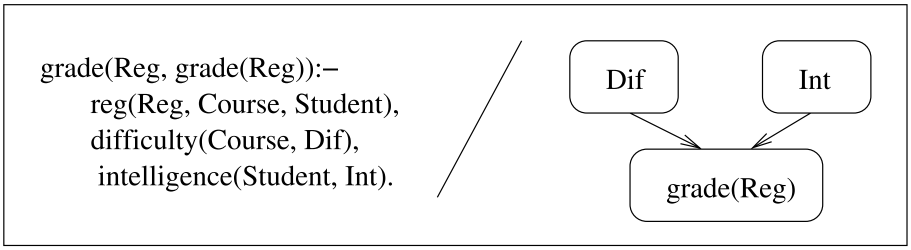


**Fig. 4.** Pictorial representation of a grade clause

## 3.1 Execution

The evaluation of a CLP($\mathcal{BN}$) program results in a network of constraints. In the previous example, the evaluation of

```
?- grade(r1,Grade).
```

will set up a constraint network with `grade(r2)` depending on `dif(course)` and `int(student)`. CLP($\mathcal{BN}$) will output the marginal probability distribution on `grade(r2)`.

Table 1 shows in some detail the actual execution steps for this query in the absence of prior evidence. The binding store and the query store grow along as we execute a query on grade `r1`. The leftmost column shows the step number, the middle column shows new bindings, and the rightmost column shows new constraints. We represent each binding as an equality, and we represent a constraint as the corresponding Skolem term. For space considerations, we abbreviate names of constants and variables, and we do not write the full CPTs, only the main functor and arguments for each Skolem term.

**Table 1.** A Query on Grade

| *Step* | *Bindings* | *Skolem Terms* |
|---|---|---|
| 0 | $\{R = \texttt{r1}\}$ | $\{\}$ |
| 1 | $\cup\{S = \texttt{mary}\}$ | |
| 2 | $\cup\{C = \texttt{c0}\}$ | |
| 3 | | $\cup\{D(\texttt{c0})\}$ |
| 4 | | $\cup\{S(\texttt{mary})\}$ |
| 5 | | $\cup\{G(r1, D(\texttt{c0}), I(\texttt{mary}))\}$ |

Each step in the computation introduces new bindings or $\mathcal{BN}$ constraints. In step 1 the call to `registration_student/2` obtains a student, `mary`. In step 2 the call to `registration_course/2` obtains a course, `c0`. The course's difficulty is obtained from `c0` in step 3. Step 4 gives `mary`'s skill. We then link the two variables together to obtain the CPT for *Grade*.

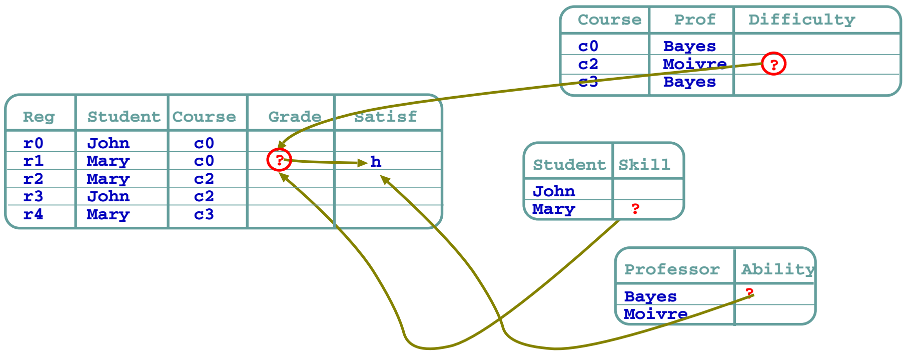


**Fig. 5.** School Database Extended to Include Satisfaction and Professor Data

*Execution: Conditioning on Evidence.* One major application of Bayesian network systems is conditioning on evidence. To give this next example, we will add some extra information to the database, as shown in Fig 5. First, we assume that now we have some information on the professors that actually taught the course. We shall need an extra table for professors, and an extra column on courses saying which professor teaches each course. Second, we are interested in how happy students were in our courses. Thus, we add an extra field for courses saying how happy, or *Satisfied*, the student was.

Satisfaction is a random variable. We do not always know it. We do know that it depends on grade and that it even with bad grades, students will be happy to attend courses taught by very good professors. Notice that in order to obtain a professor's ability, we need to navigate in the database: we find the course associated with the registration, the professor who taught the course, and finally the professor's ability. The corresponding program is shown next:

```
satisfaction(Reg, Sat) :-
   registration_course(Reg, Course),
   professor(Course, Prof),
   ability(Prof, Abi),
   grade(Reg, Grade),
   sat_table(Abi, Grade, Table),
   { Sat = satisfaction(Reg) with Table }.
```

Next, imagine that `mary` was the only student who actually gave her satisfaction, and that she was highly satisfied with registration `r1`. Given this extra evidence on satisfaction for `r1`, can we compute the new marginal for grade?

We need to be able to query for *Grade*, given that `r2`'s satisfaction is bound to $h$. In CLP($\mathcal{BN}$) the random variable for satisfaction can be obtained by asking a query,and evidence can be introduced by unifying the answer to the query. The full query would thus be:

```
?- grade(r1,X), satisfaction(r1,h).
```

**Table 2.** A Query on Grade and Satisfaction

| *Step* | *Bindings* | *Skolem Terms* |
|---|---|---|
| 0 | $\{R = \texttt{r1}\}$ | $\{\}$ |
| 1 | $\cup\{S = \texttt{mary}\}$ | |
| 2 | $\cup\{C = \texttt{c0}\}$ | |
| 3 | | $\cup\{D(\texttt{c0})\}$ |
| 4 | | $\cup\{S(\texttt{mary})\}$ |
| 5 | | $\cup\{G(r1, D(\texttt{c0}), S(\texttt{mary}))\}$ |
| 6 | $\{C' = \texttt{c0}\}$ | |
| 7 | $\cup\{P' = \texttt{Bayes}\}$ | |
| 8 | | $\cup\{A(\texttt{Bayes})\}$ |
| $9-13$ | | |
| 14 | | $\cup\{S(\texttt{r1}, A(\texttt{Bayes}), G(\texttt{r1}, \ldots))\}$ |
| 15 | | $\cup\{S(\texttt{r1}, \ldots) = \texttt{h}\}$ |

Table 2 shows how the execution steps update the stores in this case.

The first five steps repeat the computation for `grade`. Step 5 and 6 find the professor for the course. Step 8 finds its ability. Next, we recompute *Grade*. The computation will in fact be redundant, as the Skolem term for *Grade* was already in the store. The final step introduces evidence. Unification in CLP($\mathcal{BN}$) implements evidence through updating the original constraint in the store. The full store is then marginalized against the query variables by the constraint solver.

*Evidence in Store.* Imagine again we ask $grade(r1, X)$ but now given the database shown in Figure 1. The actual query should now be:

```
?- grade(r1,X),
   grade(r0,‘B’), grade(r2,‘A’), grade(r4, ‘A’),
   difficulty(c0, ‘A’), difficulty(c3, ‘C’),
   skill(john, ‘A’).
```

Writing such long queries is cumbersome, to say the least. It may be unclear which evidence is relevant, whereas giving all the evidence in a database may be extremely expensive computationally.

**Table 3.** A Query on Grade

| *Step* | *Bindings* | *Skolem Terms* |
|---|---|---|
| 0 | $\{R = \texttt{r1}\}$ | $\{G(\texttt{r0}, D(\texttt{c0}), S(\texttt{john})), D(\texttt{c0}), S(\texttt{john}), \ldots\}$ |
| 1 | $\cup\{S = \texttt{mary}\}$ | |
| 2 | $\cup\{C = \texttt{c0}\}$ | |
| 3 | | $\cup\{D(\texttt{c0})\}$ |
| 4 | | $\cup\{S(\texttt{mary})\}$ |
| 5 | | $\cup\{G(\texttt{r1}, D(\texttt{c0}), S(\texttt{mary}))\}$ |

CLP($\mathcal{BN}$) allows the user to declare evidence in the program. This is simply performed by stating evidence as a fact for the predicate. Currently, we use the construct {} to inform CLP($\mathcal{BN}$) that a fact introduces evidence:

```
grade(r0, 'B') :- {}.
grade(r2, 'A') :- {}.
grade(r4, 'A') :- {}.
```

This *global evidence* is processed at compile-time, by running the evidence data as goals and adding the resulting constraints to the *Global Store*. Execution of `grade(1,X)` would thus be as shown in Table 3.

## 4 Foundations

We next present the basic ideas of CLP($\mathcal{BN}$) more formally. For brevity, this section necessarily assumes prior knowledge of first-order logic, model theory, and resolution.

First, we remark that CLP($\mathcal{BN}$) programs are constraint logic programs, and thus inherit the well-known properties of logic programs. We further interpret a CLP($\mathcal{BN}$) program as defining a set of probability distributions over the models of the underlying logic program. Any Skolem function $sk$ of variables $X_1, ..., X_n$, has an associated CPT specifying a probability distribution over the possible denotations of $sk(X_1, ..., X_n)$ given the values, or bindings, of $X_1$, ..., $X_n$. The CPTs associated with a clause may be thought of as a Bayes net fragment, where each node is labeled by either a variable or a term built from a Skolem function. Figure 4 illustrates this view using a clause that relates a registration's grade to the course's difficulty and to the student's intelligence.

### 4.1 Detailed Syntax

The alphabet of CLP($\mathcal{BN}$) is the alphabet of logic programs. We shall take a set of functors and call these functors *Skolem functors*; *Skolem constants* are simply Skolem functors of arity 0. A Skolem term is a term whose primary functor is a Skolem functor. We assume that Skolem terms have been introduced into the program during a Skolemization process to replace the existentially-quantified variables in the program. It follows from the Skolemization process that any Skolem functor $sk$ appears in only one Skolem term, which appears in only one clause, though that Skolem term may have multiple occurrences in that one clause. Where the Skolem functor $sk$ has arity $n$, its Skolem term has the form $sk(W_1, ..., W_n)$, where $W_1, ..., W_n$ are variables that also appear outside of any Skolem term in the same clause.

A CLP($\mathcal{BN}$) program in canonical form is a set of *clauses* of the form $H \leftarrow A/B$. We call $H$ the head of the clause. $H$ is a literal and $A$ is a (possibly empty) conjunction of literals. Together they form the logical portion of the clause, $C$. The probabilistic portion, $B$, is a (possibly empty) conjunction of atoms of the form: $\{V = Sk$ `with` $CPT\}$. We shall name these atoms *constraints*. Within a constraint, we refer to $Sk$ as the Skolem term and $CPT$ as the conditional probability table. We focus on discrete variables in this paper. In this case, $CPT$ may be an unbound variable or a term or the form $\mathbf{p}(D, T, P)$. We refer to $D$ as the domain, $T$ as the table, and $P$ as the parent nodes.

A CLP($\mathcal{BN}$) constraint $B_i$ is well-formed if and only if:

1. All variables in $B_i$ appear in $C$;
2. $Sk's$ functor is unique in the program; and,
3. There is at least one substitution $\sigma$ such that $CPT\sigma = \mathbf{p}(D\sigma, T\sigma, P\sigma)$, and **(a)** $D\sigma$ is a ground list, all members of the list are different, and no sub-term of a term in the list is a Skolem term; **(b)** $P\sigma$ is a ground list, all members of the list are different, and all members of the list are Skolem terms; and **(c)** $T\sigma$ is a ground list, all members of $T\sigma$ are numbers $p$ such that $0 \leq p \leq 1$, and the size of $T\sigma$ is a multiple of the size of $D\sigma$.

If the probabilistic portion of a clause is empty, we also call the clause a *Prolog clause*. According to this definition, every Prolog program is a CLP($\mathcal{BN}$) program.

## 4.2 Operational Semantics

A query for CLP($\mathcal{BN}$) is an ordinary Prolog query, which is a conjunction of positive literals. In logic programming, a query is answered by one or more proofs constructed through resolution. At each resolution step, terms from two different clauses may be unified. If both of the terms being unified also participate in CPTs, or Bayes net constraints, then the corresponding nodes in the Bayes net

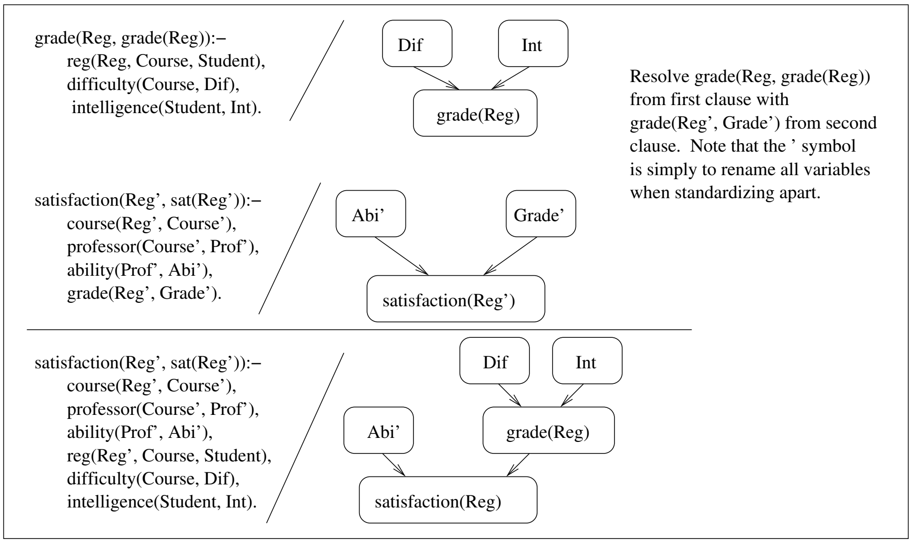

grade(Reg, grade(Reg)):–
reg(Reg, Course, Student),
difficulty(Course, Dif),
intelligence(Student, Int).
Dif
Int
grade(Reg)
Resolve grade(Reg, grade(Reg)) from first clause with grade(Reg', Grade') from second clause. Note that the ' symbol is simply to rename all variables when standardizing apart.
satisfaction(Reg', sat(Reg')):–
course(Reg', Course'),
professor(Course', Prof'),
ability(Prof', Abi'),
grade(Reg', Grade').
Abi'
Grade'
satisfaction(Reg')
satisfaction(Reg', sat(Reg')):–
course(Reg', Course'),
professor(Course', Prof'),
ability(Prof', Abi'),
reg(Reg', Course, Student),
difficulty(Course, Dif),
intelligence(Student, Int).
Dif
Int
Abi'
grade(Reg)
satisfaction(Reg)

**Fig. 6.** Resolution

constraints must be *unified* as illustrated in Figure 6. In this way we construct a large Bayes net consisting of all the smaller Bayes nets that have been unified during resolution.

A cycle may arise in the Bayes Net if we introduce a constraint such that $Y$ is a parent of $X$, and $X$ is an ancestor of $Y$. In this case, when unifying $Y$ to an argument of the CPT constraint for $X$, $X$ would be a sub-term of the CPT constraint for $Y$ which causes unification failure. To detect this failure it is necessary to do a proper unification test using the 'occur-check', something standard Prolog does not do (for efficiency reasons).

To be rigorous in our definition of the distribution defined by a Bayes net constraint, let $C_i/B_i$, $1 \leq i \leq n$, be the clauses participating in the proof, where $C_i$ is the ordinary logical portion of the clause and $B_i$ is the attached Bayes net, in which each node is labeled by a term. Let $\theta$ be the answer substitution, that is, the composition of the most general unifiers used in the proof. Note that during resolution a clause may be used more than once but its variables always are renamed, or standardized apart from variables used earlier. We take each such renamed clause used in the proof to be a distinct member of $\{C_i/B_i | 1 \leq i \leq n\}$. We define the application of a substitution $\theta$ to a Bayes net as follows. For each node in the Bayes net, we apply $\theta$ to the label of that node to get a new label. If some possible values for that node (according to its CPT) are not instances of that new label, then we marginalize away those values from the CPT.

## 4.3 Model-Theoretic Semantics

A CLP($\mathcal{BN}$) program denotes a set of probability distributions over models. We begin by defining the probability distribution over ground Skolem terms that is specified by the probabilistic portion of a CLP($\mathcal{BN}$) program. We then specify the probability distribution over models, consistent with this probability distribution over ground Skolem terms, that the full CLP($\mathcal{BN}$) program denotes.

A CLP($\mathcal{BN}$) program $P$ defines a unique joint probability distribution over ground Skolem terms as follows. Consider each ground Skolem term to be a random variable whose domain is a finite set of non-Skolem constants.[1] We now specify a Bayes net $\mathcal{BN}$ whose variables are these ground Skolem terms. Each ground Skolem term $s$ is an instance of exactly one Skolem term $t$ in the program $P$. To see this recall that, from the definition of Skolemization, any Skolem functor appears in only one term in the program $P$, and this one term appears in only one clause of $P$, though it may appear multiple times in that clause. Also from the definition of Skolemization, $t$ has the form $sk(W_1, ..., W_m)$, where $sk$ is a Skolem functor and $W_1, ..., W_m$ are distinct variables. Because $s$ is a ground instance of $t$, $s = t\sigma$ for some substitution $\sigma$ that grounds $t$. Because $t = sk(W_1, ..., W_n)$ appears in only one clause, $t$ has exactly one associated (generalized) CPT, $T$, conditional on the Skolem terms in $W_1, ..., W_n$. Let the

[1] This can be extended to a finite subset of the set of ground terms not containing Skolem symbols (functors or constants). We restrict ourselves to constants here merely to simplify the presentation.

parents of $s$ in $\mathcal{BN}$ be the Skolem terms in $W_1\sigma, ..., W_m\sigma$, and let the CPT be $T\sigma$. Note that for any node in $\mathcal{BN}$ its parents are sub-terms of that node. It follows that the graph structure is acyclic and hence that $\mathcal{BN}$ is a properly defined Bayes net, though possibly infinite. Therefore $\mathcal{BN}$ uniquely defines a joint distribution over ground Skolem terms; we take this to be the distribution over ground Skolem terms defined by the program $P$.

The meaning of an ordinary logic program typically is taken to be its least Herbrand model. Recall that the individuals in a Herbrand model are themselves ground terms, and every ground term denotes itself. Because we wish to consider cases where ground Skolem terms denote (non-Skolem) constants, we instead consider Herbrand quotient models [13]. In a Herbrand quotient model, the individuals are equivalence classes of ground terms, and any ground term denotes the equivalence class to which it belongs. Then two ground terms are equal according to the model if and only if they are in the same equivalence class. We take the set of minimal Herbrand quotient models for $P$ to be those derived as follows.[2] Take the least Herbrand model of the logical portion of $P$, and for each non-Skolem constant, merge zero or more ground Skolem terms into an equivalence class with that constant. This equivalence class is a new individual, replacing the merged ground terms, and it participates in exactly the relations that at least one of its members participated in, in the same manner. It follows that each resulting model also is a model of $P$. The set of models that can be constructed in this way is the set $S$ of minimal Herbrand quotient models of $P$. Let $D$ be any probability distribution over $S$ that is consistent with the distribution over ground Skolem terms defined by $P$. By consistent, we mean that for any ground Skolem term $t$ and any constant $c$, the probability that $t = c$ according to the distribution defined by $P$ is exactly the sum of the probabilities according to $D$ of the models in which $t = c$. At least one such distribution $D$ exists, since $S$ contains one model for each possible combination of equivalences. We take such $\langle D, S\rangle$ pairs to be the models of $P$.

## 4.4 Agreement Between Operational and Model-Theoretic Semantics

Following ordinary logic programming terminology, the negation of a query is called the "goal," and is a clause in which every literal is negated. Given a program and a goal, the CLP($\mathcal{BN}$) operational semantics will yield a derivation of the empty clause if and only if every model $\langle D, S\rangle$ of the CLP($\mathcal{BN}$) program falsifies the goal and hence satisfies the query for some substitution to the variables in the query. This follows from the soundness and refutation-completeness of SLD-resolution. But in contrast to ordinary Prolog, the proof will be accompanied by a Bayes net whose nodes are labeled by Skolem terms appearing in the query or proof. The following theorem states that the answer to any query of

[2] For brevity, we simply define these minimal Herbrand quotient models directly. Alternatively, we can define an ordering based on homomorphisms between models and prove that what we are calling the minimal models are indeed minimal with respect to this ordering.

this attached Bayes net will agree with the answer that would be obtained from the distribution $D$, or in other words, from the distribution over ground Skolem terms defined by the program $P$. Therefore the operational and model-theoretic semantics of CLP($\mathcal{BN}$) agree in a precise manner.

**Theorem 1.** *For any CLP($\mathcal{BN}$) program $P$, any derivation from that program, any grounding of the attached Bayes net, and any query to this ground Bayes net,*[3] *the answer to the query is the same as if it were asked of the joint distribution over ground Skolem terms defined by $P$.*

*Proof.* Assume there exists some program $P$, some derivation from $P$ and associated ground Bayes net $B$, and some query $Pr(q|E)$ such that the answer from $B$ is not the same as the answer from the full Bayes net $\mathcal{BN}$ defined by $P$. For every node in $B$ the parents and CPTs are the same as for that same node in $\mathcal{BN}$. Therefore there must be some path through which evidence flows to $q$ in $\mathcal{BN}$, such that evidence cannot flow through that path to $q$ in $B$. But by Lemma 1, below, this is not possible.

**Lemma 1.** *Let $B$ be any grounding of any Bayes net returned with any derivation from a CLP($\mathcal{BN}$) program $P$. For every query to $B$, the paths through which evidence can flow are the same in $B$ and in the full Bayes net $\mathcal{BN}$ defined by $P$.*

*Proof.* Suppose there exists a path through which evidence can flow in $\mathcal{BN}$ but not in $B$. Consider the shortest such path; call the query node $q$ and call the evidence node $e$. The path must reach $q$ through either a parent of $q$ or a child of $q$ in $\mathcal{BN}$. Consider both cases. *Case 1*: the path goes through a parent $p$ of $q$ in $\mathcal{BN}$. Note that $p$ is a parent of $q$ in $B$ as well. Whether evidence flows through $p$ in a linear or diverging connection in $\mathcal{BN}$, $p$ cannot itself have evidence—otherwise, evidence could not flow through $p$ in $\mathcal{BN}$. Then the path from $e$ to $p$ is a shorter path through which evidence flows in $\mathcal{BN}$ but not $B$, contradicting our assumption of the shortest path. *Case 2*: the path from $e$ to $q$ flows through some child $c$ of $q$ in $\mathcal{BN}$. Evidence must flow through $c$ in either a linear or converging connection. If a linear connection, then $c$ must not have evidence; otherwise, evidence could not flow through $c$ to $q$ in a linear connection. Then the path from $e$ to $c$ is a shorter path through which evidence flows in $\mathcal{BN}$ but not $B$, again contradicting our assumption of the shortest path. Therefore, evidence must flow through $c$ in a converging connection in $\mathcal{BN}$. Hence either $c$ or one of its descendants in $\mathcal{BN}$ must have evidence; call this additional evidence node $n$. Since $n$ has evidence in the query, it must appear in $B$. Therefore its parents appear in $B$, and their parents, up to $q$. Because evidence can reach $c$ from $e$ in $B$ (otherwise, we contradict our shortest path assumption again), and a descendant of $c$ in $B$ (possibly $c$ itself) has evidence, evidence can flow through $c$ to $q$ in $B$.

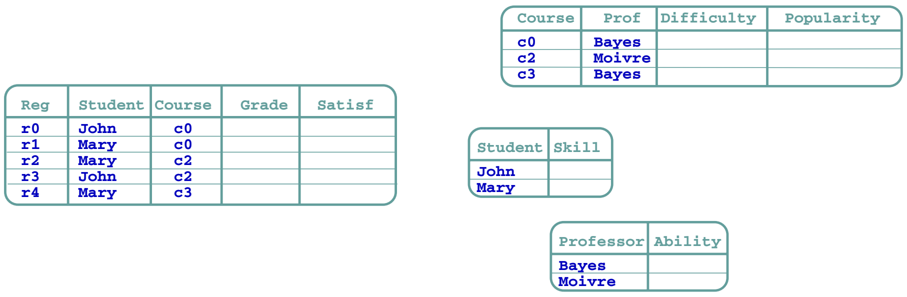
[3] For simplicity of presentation, we assume queries of the form $Pr(q|E)$ where $q$ is one variable in the Bayes net and the evidence $E$ specifies the values of zero or more other variables in the Bayes net.

## 5 Non-determinism and Aggregates

One important property of relational databases is that they allow users to query for properties of sets of elements, or *aggregates*. Aggregates are also particularly important in the application of Probabilistic Relational Models, as they allow one to state that the value of a random variable depends on a set of elements that share some properties [14].

To clarify this concept, imagine that we run a private school, and we want to find out which courses are most attractive. To do so, we would want one extra attribute on the *Courses* table giving how popular the course is, as shown in Figure 7. Ideally, one would ask students who have attended the course and average the results. On the other hand, if we cannot obtain a representative sample, we can try to estimate popularity from the average of student satisfaction for that course.

Towards this goal, an extension to the Bayesian network is shown in Figure 8. We need an operator to aggregate on the set of satisfactions for a course, and then we can estimate the field's value from the aggregate.



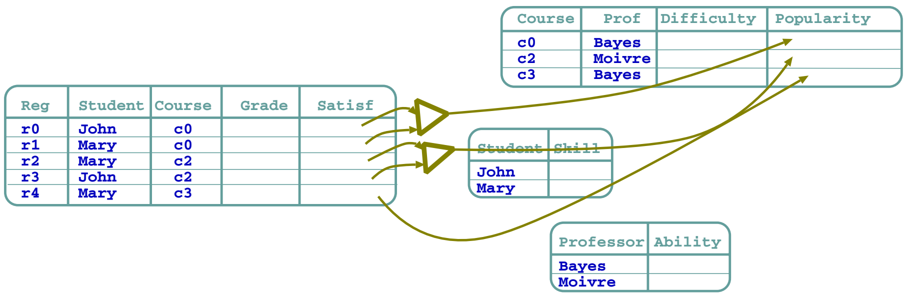
| Course | Prof | Difficulty | Popularity |
|---|---|---|---|
| c0 | Bayes | | |
| c2 | Moivre | | |
| c3 | Bayes | | |

| Reg | Student | Course | Grade | Satisf |
|---|---|---|---|---|
| r0 | John | c0 | | |
| r1 | Mary | c0 | | |
| r2 | Mary | c2 | | |
| r3 | John | c2 | | |
| r4 | Mary | c3 | | |

| Student | Skill |
|---|---|
| John | |
| Mary | |

| Professor | Ability |
|---|---|
| Bayes | |
| Moivre | |

**Fig. 7.** School Database Extended to Include a Field on Course Popularity



| Course | Prof | Difficulty | Popularity |
|---|---|---|---|
| c0 | Bayes | | |
| c2 | Moivre | | |
| c3 | Bayes | | |

| Reg | Student | Course | Grade | Satisf |
|---|---|---|---|---|
| r0 | John | c0 | | |
| r1 | Mary | c0 | | |
| r2 | Mary | c2 | | |
| r3 | John | c2 | | |
| r4 | Mary | c3 | | |

| Student | Skill |
|---|---|
| John | |
| Mary | |

| Professor | Ability |
|---|---|
| Bayes | |
| Moivre | |

**Fig. 8.** School Database: Popularity is a random variable, and depends on the average of student satisfaction

CLP($\mathcal{BN}$) can deal with such aggregates in a straightforward way, by taking advantage of the second order features in Prolog, as shown in the next clause:

```
rating(C, Rat) :-
  setof(S,R^(registration(R,C),
           satisfaction(R,S)), Sats),
  average(Sats, Average),
  rating_table(Table),
  { Rat = rating(C) with prob([a,b,c,d],Table,[Average])
```

The call to `setof` obtains the satisfactions of all students registered in the course. The procedure `average/3` generates a the conditional probability of their average as a new random variable, *Average*. The course's rating, *Rat*, is assumed to be highly dependent on *Average*.

## 5.1 Building Aggregates

Aggregates are deterministic functions. Given $n$ discrete random variables that range over $k$ possible values, the aggregate value will take one well defined value. Hence, the probability of that value will be 1, and 0 for the remaining $k-1$ values. Writing the CPTs for aggregates should therefore be quite straightforward.

Unfortunately, aggregates create two problems. First, CPTs are most often represented as tables, where the size of the table grows exponentially with the number of dimensions. As the number of nodes $n$ in the aggregate grows, table size grows exponentially. The current implementation of CLP($\mathcal{BN}$) uses *divorcing* [1]. The idea is to introduce hidden nodes, also called *mediating* variables, between a node and its parents, so that the total number of parents for every node can be guaranteed to never exceed a small number. The current system implements an aggregate node through a binary tree of mediating variables.

Figure 9 shows a fragment of an example network for an artificially generated School database with 4096 students. The query node is the gray node below. The gray node above is a evidence node for `course_rating`. The node is an aggregate of 68 `student_satisfaction` nodes, hence building the full table would require $3 * 3^{68}$ entries. Figure 9 shows the hierarchy of mediating nodes constructed by CLP($\mathcal{BN}$): note that the value of each node is deterministic on the ancestor nodes.

Figure 9 clearly shows the second problem we need to address in making CLP($\mathcal{BN}$) effective for real data-bases. The Bayes network shown here was created to answer a query on `course_difficulty`, shown as the gray node below. Given the original query, the algorithm searches for related evidence (shown as the other gray nodes). The knowledge-based model-construction algorithm searches parents recursively, eventually finding a number of nodes with evidence. Next, it needs to consider the Markov Blanket for these nodes, thus leading to searching other nodes. In this case, eventually almost every random variable in the database was included in the actual Bayes net (even though most of the nodes will have little relevancy to the original query).

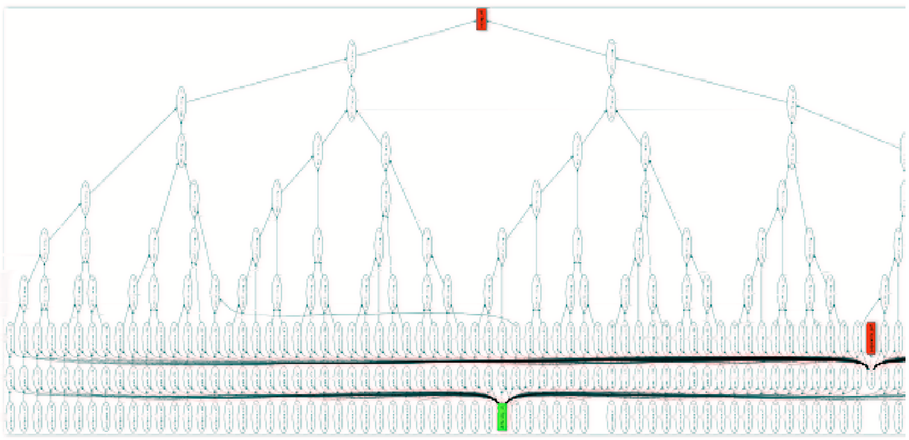


**Fig. 9.** A Bayesian Network Generated for an Ability Query. The Artificial network includes 4096 students, 128 professors, and 256 courses.

We observed that it is often the case that some nodes in a database are highly connected and take a central position in the graph. If evidence reaches these central nodes probabilistic inference will end up involving most of the network.

1. Exact inference is very expensive, and may not be possible at all.
2. Standard approximate inference such as Gibbs sampling may not always converge as often these networks include deterministic operations, such as `average` in current the example.

Processing such large networks effectively [15,16,17,18] and choosing the best strategy for different networks is one of the major challenges in the development of CLP($\mathcal{BN}$).

## 6 Recursion and Sequences

Recursion in Logic provides an elegant framework for modeling sequences of events, such as Markov Models. Next we discuss how the main ideas of CLP($\mathcal{BN}$) can be used to represent Hidden Markov Models (HMMs) [19], which are used for a number of applications ranging from Signal Processing, to Natural Language Processing, to Bioinformatics, and Dynamic Bayes Networks (DBNs). This was inspired by prior work on combining the advantages of multi-relational approaches with HMMs and DBNs: evaluation and learning of HMMs is part of PRISM [20,21], Dynamic Probabilistic Relational Models combine PRMs and DBNs [22], Logical HMMs have been used to model protein structure

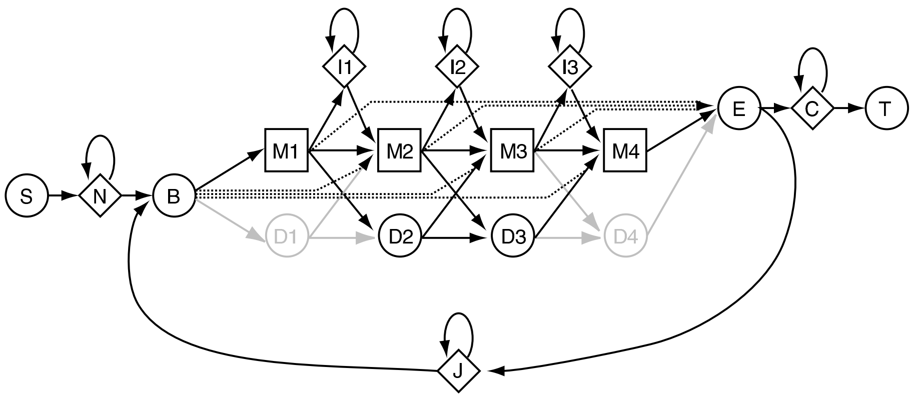


**Fig. 10.** Plan7 (From the HMMer Manual)

data [23,24]. More recently, non-directed models such as LogCRFs have also been proposed toward this goal [25].

Next, we discuss how to model HMMs and DBNs in CLP($\mathcal{BN}$). We present our experience in modeling profile-HMMs (pHMMs), an HMM structure widely used in Bioinformatics for homology detection between a sequence and a family of sequences. We chose pHMMs because they are extremely important in practice, and because they are not a trivial application. We focus on HMMer, an open-source tool that implements the Plan7 model, and which is one of the most widely used tools [26]. HMMer was used to build the well-known Pfam protein database [27].

HMMer is based on the Plan7 model, shown in Figure 10. The model describes a number of related sequences that share the same *profile*: a number of columns, each one corresponding to a well-preserved amino-acid. The example shows a relatively small profile, we can have profiles with hundreds of columns. A *match* state corresponds to an amino-acid in the sequence being a good match to the amino-acids found at the same position in the profile. *Insert* and *delete* states correspond to gaps: inserts are new material inserted in the sequence, and deletes removed material. There also three other character emitting-states: `N`, `E`, and `J`. The `N` states corresponds to material preceding the match, the `E` states to material after the match, and the `J` states allow several matches on the same sequence.

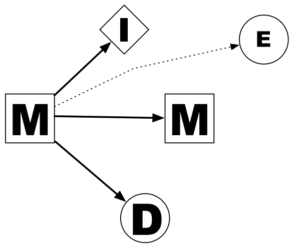


**Fig. 11.** M-State in Plan7

We model a pHMM by writing rules for each type of state. First, each state has two indexes: $I$ refers to the current position in the sequence, and $J$ to the column we are at. Second, whereas in a standard Bayesian Network we write how a variable depends on its parents, in an HMM we think in terms of *transitions* between *states*. As an example, consider the case of a typical `M` state, shown in Figure 11.

If we are at `M` state we can next move to an `I` state, (meaning a match is followed by a gap), to a `D` state, meaning the sequence will skip the next match state, or to the next `M` state. The model also makes it possible to jump directly to the end of the match. The CLP($\mathcal{BN}$) clause is as follows:

```
m_state(I,J,M) :-
   I1 is I+1,
   J1 is J+1,
   i_state(I1,J,NI),
   m_state(I1,J1,M1),
   d_state(I1,J1,ND),
   e_state(I1,NE),
   m_i_prob(J,MIP),
   m_m_prob(J,MMP),
   m_d_prob(J,MDP),
   m_e_prob(J,MEP),
   { M = m(I,J) with p([0,1],trans([MIP,MMP,MDP,MEP]),
                                   [ NI, M1, ND, NE])) },
   emitting_state(m, I, J, M).
```

The $M$ variable refers to the random variable for the current state. The rule is not very complex:

1. We can move from $M(I,J)$ to $I(I+1,J)$, $M(I+I,J+1)$, $D(I+1,J+1)$, or $E(I+1)$;
2. The transition probabilities at column $I$ are $P_{M\rightarrow I} = MIP$, $P_{M\rightarrow M} = MMP$ $P_{M\rightarrow D} = MDP$, $P_{M\rightarrow E} = MEP$, such that

$$MIP + MMP + MDP + MEP = 1$$

3. $M$ is a binary random variable with the given transition probabilities;
4. *trans* indicates we are setting up a constraint with transition probabilities; such constraints need specialized solvers, such as Viterbi or forward propagation.
5. `emitting_state/3`: if the state emits a symbol, access evidence for sequence element $I$.

*Implementation.* One can observe that HMMs are highly-recursive programs, and executing in the standard Prolog way would result in calling the same goal repeatedly over and over again. This problem can be addressed by *tabling* calls so that only the first one is actually executed, and repeated calls just need to lookup a data-base [28]. Tabled execution of these programs has the same complexity as standard dynamic programming algorithms. To the best of our knowledge, PRISM was the first language to use tabling for this task [20]. The CLP($\mathcal{BN}$) implementation originally relied on YAP's tabling mechanism [29]. Unfortunately, the YAP implementation of tabling is optimized for efficient evaluation of non-deterministic goals; we have achieved better performance through a simple program transformation.

Given this tabling mechanism, implementing algorithms such as Viterbi is just a simple walk over the constraint store.

*Experiments.* We tried this model with a number of different examples. The most interesting example was the *Globin* example from the standard HMMer distribution. The example matches a Plan7 model of the Globin family of proteins against an actual globin from Artemia. The Globin model has 159 columns, and the protein has 1452 amino-acids. The run generates 692 k states (random variables) and is about two orders of magnitude slower than the highly optimized `C`-code in HMMer. HMMer uses a much more compact and specialized representation than CLP($\mathcal{BN}$). Also, CLP($\mathcal{BN}$) actually creates the complete graph; in contrast, HMMer only needs to work with a column at a time. On the other hand, CLP($\mathcal{BN}$) has some important advantages: it provides a very clear model of the HMM, and it relatively straightforward to experiment and learn different structures.

## 7 Learning with CLP(BN)

We have performed some experiments on learning with CLP($\mathcal{BN}$). In both cases the goal is learn a model of a database as a CLP($\mathcal{BN}$) program.

The learning builds upon work performed for learning in Bayesian networks and in Inductive Logic Programming. We leverage on the Aleph ILP system. Essentially, we use Aleph to generate clauses which are then rewritten as CLP($\mathcal{BN}$) clauses. The rewriting process is straightforward for deterministic goals. If non-deterministic goal are allowed, we aggregate over the non-deterministic goals. We assume full data in these experiments, hence the parameters can be learned by maximum likelihood estimation. Next, we score the network with this new clause. Note that the score is used to control search in Aleph.

### 7.1 The School Database

We have so far used the school database as a way to explain some basic concepts in CLP($\mathcal{BN}$), relating them to PRMs. The school database also provides a good example of how to learn CLP($\mathcal{BN}$) programs.

First, we use an interpreter to generate a sample from the CLP($\mathcal{BN}$) program. The smallest database has 16 professors, 32 courses, 256 students and 882 registrations; the numbers roughly double in each successively larger database. We have no missing data. Can we, given this sample, relearn the original CLP($\mathcal{BN}$) program?

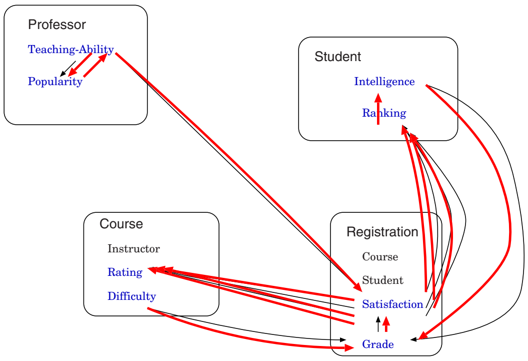


**Fig. 12.** Pictorial representation of the CLP($\mathcal{BN}$) clauses learned from the largest schools database, before removal of cycles

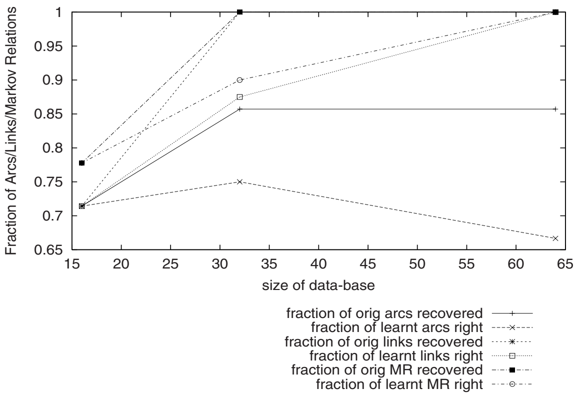


**Fig. 13.** Graph of results of CLP($\mathcal{BN}$)-learning on the three sizes of schools databases. Links are arcs with direction ignored. A Markov relation (MR) holds between two nodes if one is in the Markov blanket of the other.

From the ILP point of view, this is an instance of multi-predicate learning. To simplify the problem we assume each predicate would be defined by a single clause. We use the Bayesian Information Criterion (BIC) score to compare alternative clauses for the same predicate. Because ALEPH learns clauses independently, cycles may appear in the resulting CLP($\mathcal{BN}$) program. We therefore augment ALEPH with a post-processing algorithm that simplifies clauses until no cycles remain; the algorithm is greedy, choosing at each step the simplification that will least affect the BIC score of the entire program.

The following is one of the learned CLP($\mathcal{BN}$) clauses; to conserve space, we do not show the full conditional probability table.

```
registration_grade(A,B) :-
   registration(A,C,D), course(C,E),
   course_difficulty(C,F), student_intelligence(D,G),
   {F = registration_grade(A,F,G) with
        p([‘A’,‘B’,‘C’,‘D’],...,[F,G] }.
```

Figure 12 illustrates, as a PRM-style graph, the full set of clauses learned for the largest of the databases before simplification; this would be the best network according to BIC, if not for the cycles. Figure 13 plots various natural measures of the match between the learned program *after cycles have been removed* and the original program, as the size of the database increases. By the time we get to the largest of the databases, the only measures of match that do not have a perfect score are those that deal with the directions of arcs.

## 7.2 EachMovie

Next, we experiment our learning algorithm on the EachMovie data-set. This data-set includes three tables: there is data on 1628 movies, including movie type, store-info, and a link to the IMDB database. There is data on 72000 people who voted on the movies. Input was voluntary, and may include age, gender and ZIP code. From ZIP code it is possible to estimate geographical location and to get a good approximation of average income. Lastly, there are 2.8 million votes. Votes can be organized by class and range from 0 to 5. Our task is to predict how every non-key column in the database depends on the other non-key fields. That is we try to predict individual voting patterns, movie popularity, and people information. Given that there is a large amount of data, we use log-likelihood to score the network.

The data-set introduces a number of challenges. Firstly, there is missing data, especially in the `people` table. Following Domingos, we cannot assume that the individuals who refused to give their ZIP address or their age follow the same distribution as the ones who do [30]. Instead, we introduce an *unknown* evidence value, which says the individual refused to provide the information.

Aggregates are fundamental in these models because we often want to predict characteristics of groups of entities. In the School work we build aggregates *dynamically* during clause-construction by aggregating over non-deterministic goals. but doing so is just too expensive for this larger database. In this work, we use pre-computed aggregates:

– For each person, we compute how many votes and average score.
– For each movie, we compute how many people voted on this movie and average score.

A first result on the full data-set is shown in Figure 14. As for the school database, predicates are defined by a single clause. Learning proceeded greedily in this experiment: we first learn the predicate that best improves global log-likelihood. Next, we use this predicate plus the database to learn the other predicates. The process repeats until every predicate has been learned.

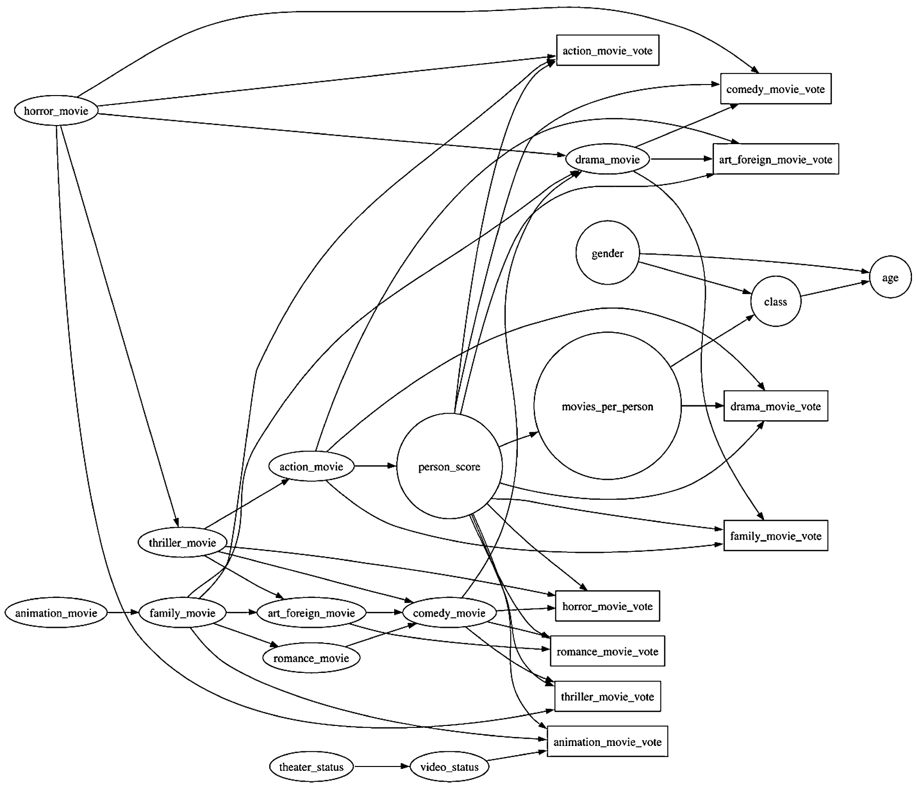


**Fig. 14.** EachMovie

Figure 14 was generated using the dot program. To the left it shows connections between movies of the different types (e.g., being an animation movie affects whether you are a family movie). The center/left of the network is about people. The system inferred that the average person score affects the number of movies seen per individual, and this in turn affects class. Last, the network includes voting patterns for movies. As an example, votes on family movies seem to depend on whether it is an action movie also, on whether it is also a drama, and on the person's average vote.

## 8 Relationship to PRMs

Clearly from the preceding discussion the CLP($\mathcal{BN}$) representation owes an intellectual debt to PRMs. As the reader might suspect at this point, any PRM can be represented as a CLP($\mathcal{BN}$) program. In this section we present an algorithm to convert any PRM into a CLP($\mathcal{BN}$) program. But before that, we address the natural question, "given that we already have PRMs, of what possible utility is the CLP($\mathcal{BN}$) representation?"

First, there has been much work on incorporating probabilities into first-order logic (see Section 9). Hence while there is great interest in the relationship between PRMs and these various probabilistic logics [31,32,33,34], this relationship is difficult to characterize. Approaches such as CLP($\mathcal{BN}$) and BLPs are closely related to PRMs, and can help us to better understand the relationship between PRMs and various probabilistic logics. Second, because CLP($\mathcal{BN}$)s are an extension of logic programming, they permit recursion and the use of function symbols, e.g., to construct data structures such as lists or trees. This expressivity may be useful for a variety of probabilistic applications and is not available in PRMs. Of course we must note that the uses of recursion and recursive data structures are not unlimited. CLP($\mathcal{BN}$)s disallow resolution steps that introduce a cycle into a Bayes net constraint. Third, and most importantly from the authors' viewpoint, the CLP($\mathcal{BN}$) representation is amenable to learning using techniques from inductive logic programming (ILP). Hence CLP($\mathcal{BN}$)s provide a way of studying the incorporation of probabilistic methods into ILP, and they may well give insight into novel learning algorithms for PRMs. The methods of learning in PRMs [3] are based upon Bayes net structure learning algorithms and hence are very different from ILP algorithms. The CLP($\mathcal{BN}$) representation provides a bridge through which useful ideas from ILP might be transferred to PRMs.

The remainder of this section presents an algorithm to convert any PRM into a CLP($\mathcal{BN}$) program. Because of space limits, we necessarily assume the reader already is familiar with the terminology of PRMs.

We begin by representing the skeleton of the PRM, i.e., the database itself with (possibly) missing values. For each relational table $R$ of $n$ fields, one field of which is the key, we define $n-1$ binary predicates $r_2, ..., r_n$. Without loss of generality, we assume the first field is the key. For each tuple or record $\langle t_1, ..., t_n\rangle$ our CLP($\mathcal{BN}$) program will contain the fact $r_i(t_1, t_i)$ for all $2 \leq i \leq n$. If $t_i$ is a missing value in the database, then the corresponding fact in the CLP($\mathcal{BN}$) program is $r_i(t_1, skr_i(t_1))$, where $skr_i$ is a Skolem function symbol. It remains to represent the Bayes net structure over this skeleton and the CPTs for this structure.

For each field in the database, we construct a clause that represents the parents and the CPT for that field within the PRM. The head (consequent) of the clause has the form $r_i(Key, Field)$, where the field is the $i^{th}$ field of relational table $R$, and $Key$ and $Field$ are variables. The body of the clause is constructed in three stages, discussed in the following three paragraphs: the relational stage, the aggregation stage, and the CPT stage.

The relational stage involves generating a translation into logic of each slot-chain leading to a parent of the given field within the PRM. Recall that each step in a slot chain takes us from the key field of a relational table $R$ to another field, $i$, in that table, or vice-versa. Each such step is translated simply to the literal $r_i(X, Y)$, where $X$ is a variable that represents the key of $R$ and $Y$ is a variable that represents field $i$ of $R$, regardless of directionality. If the next step in the slot chain uses field $i$ of table $R$, then we re-use the variable $Y$; if the next step instead uses the key of table $R$ then we instead re-use variable $X$. Suppose field $i$ is the foreign key of another table $S$, and the slot chain next takes us to field $j$ of $S$. Then the slot chain is translated as $r_i(X, Y), s_j(Y, Z)$. We can use the same translation to move from field $j$ of $S$ to the key of $R$, although we would re-order the literals for efficiency. For example, suppose we are given a student key *StudentKey* and want to follow the slot chain through registration and course to find the teaching abilities of the student's professor(s). Assuming that the course key is the second field in the registration table and the student key is the third field, while the professor key is the second field of the course table, and ability is the second field of the professor table, the translation is as below. Note that we use the first literal to take us from *StudentKey* to *RegKey*, while we use the second literal to take us from *RegKey* to *CourseKey*.

$registration_3(RegKey, StudentKey)$,
$registration_2(RegKey, CourseKey)$,
$course_2(CourseKey, ProfKey)$,
$professor_2(ProfKey, Ability)$

In the preceding example, the variable *Ability* may take several different bindings. If this variable is a parent of a field, then the PRM will specify an aggregation function over this variable, such as *mean*. Any such aggregation function can be encoded in a CLP($\mathcal{BN}$) program by a predicate definition, as in ordinary logic programming, i.e. in Prolog. We can collect all bindings for *Ability* into a list using the Prolog built-in function *findall* or *bagof*, and then aggregate this list using the appropriate aggregation function such as *mean*. For the preceding example, we would use the following pair of literals to bind the variable $X$ to the mean of the abilities of the student's professors.

```
findall(Ability,  ( registration2(RegKey, CourseKey),
                    course2(CourseKey, ProfKey),
                    professor2(ProfKey, Ability),   L ),
mean(L, X)
```

At this point, we have constructed a clause body that will compute binding for all the variables that correspond to parents of the field in question. It remains only to add a literal that encodes the CPT for this field given these parents.

## 9 Other Related Work

The key idea in CLP($\mathcal{BN}$)s is that they provide joint probability distributions over the variables in the answer to a query, i.e., in a single proof. Hence it is not necessary to reconcile various probabilities obtained from different clauses or through different proofs. We combine information using aggregation (see Section 5), and the predicates for aggregation are part of the CLP($\mathcal{BN}$) program. This contrasts with the approach taken in both [35] and [7] where a combining rule is added on top of the logical representation.

CLP($\mathcal{BN}$) implements *Knowledge-based model construction (KBMC)* in that it uses logic "as a basis for generating Bayesian networks tailored to particular problem instances" [11]. However, in contrast to many KBMC approaches

[11,36], a probability in a CLP($\mathcal{BN}$) program does not give the probability that some first-order rule is *true*. Instead it is a (soft) constraint on possible instantiations of a variable in a rule. This also distinguishes it from the work in [4,37]. In these approaches instead of ground atomic formulas (*atoms*) being true or false as in normal logic programming semantics, they are true with a certain probability. In PRISM programs [4] a *basic distribution* gives the probabilities for the `msw` ground atoms mentioned in the PRISM program; this is then extended to define probabilities for all atoms which can be derived using rules in the program. In contrast, in a Bayesian logic program (BLP) [7] the distribution associated with a ground atom is unrestricted; it need not be always be over the two values $\{\text{true}, \text{false}\}$. In this respect BLPs are closer to CLP($\mathcal{BN}$) than, say, PRISM programs. The central difference is that BLPs represent random variables with ground atoms—in CLP($\mathcal{BN}$) they are represented by (Bayesian) variables.

In Angelopoulos's probabilistic finite domain *Pfd* model [38] hard constraints between variables and probability distributions over the same variables are kept deliberately separate, thereby allowing a normal CLP constraint solver to find variable instantiations permitted by the hard constraints. However, in addition to normal CLP, each such instantiation is returned with its probability. The main difference to our work is that we do not put hard constraints on Bayesian variables. Also CLP($\mathcal{BN}$) exploits conditional independence to permit efficient inference, whereas currently computation within *Pfd* is exponential in the number of variables involved.

## 10 Conclusions and Future Work

We have presented CLP($\mathcal{BN}$), a novel approach to integrating probabilistic information in logic programs. Our approach is based on the key idea that constraints can be used to represent information on undefined variables. Logical inference is used to define a Bayesian network that can be processed by a Bayesian solver. CLP($\mathcal{BN}$)s are closely related to PRMs, but they permit recursion, the use of functor symbols, and the representation is amenable to learning using techniques from inductive logic programming. Our first implementation of CLP($\mathcal{BN}$) system used Yap as the underlying Prolog system and the Kevin Murphy's Bayesian Network Toolbox as the Bayesian solver [39]. This allowed flexibility in choosing different engines. The newer versions include specialized solvers written in Prolog. The solvers implement variable elimination, Gibbs sampling, and Viterbi. We have successfully experimented the system with both database style and recursive programs.

The main focus of our future work will be in learning with CLP($\mathcal{BN}$) programs. Namely, we are now working with CLP($\mathcal{BN}$) on inducing regulatory networks [40,41]. We are also looking forward at integrating CLP($\mathcal{BN}$) with some of recent work in generating statistical classifiers [42,43,44,45,46]. Last, it would be interesting to study whether the ideas of CLP($\mathcal{BN}$) also apply to undirected models [47]. We are also considering directions to improve CLP(($\mathcal{BN}$). Regarding implementation, most effort will focus on tabling [28,29] that avoids repeated invocation of the same literal and can be quite useful in improving performance of logic programs, namely for database applications. As we have seen, CLP($\mathcal{BN}$) will directly benefit from this work. At present a CLP($\mathcal{BN}$) program generates a query-specific BN, and then standard BN algorithms (e.g. junction tree propagation) are used to compute the desired probabilities. Given the well-known connections between constraint processing and probabilistic computations as given by Dechter [12] it would be interesting to bring the probabilistic computations inside CLP($\mathcal{BN}$).

In common with many logical-probabilistic models [7,36,4,37], CLP($\mathcal{BN}$) exploits its logical framework to quantify over random variables, thereby facilitating the definition of large and complex BNs. An alternative approach, not explicitly based on first-order logic, is the BUGS language [48]. BUGS programs permit Bayesian statistical inference by defining large BNs with one (instantiated) node for each data point. It would be interesting to see if CLP($\mathcal{BN}$) could also be used for such statistical inference, particularly since CLP($\mathcal{BN}$), unlike BUGS, allows recursion.

## Acknowledgments

This work was supported by the NSF Colleague project and by the DARPA EELD program. We gratefully acknowledge discussions with Jesse Davis, Inês Dutra, Peter Haddawy, Kristian Kersting, and Irene Ong. Maleeha Qazi participated in the development of learning algorithms for CLP($\mathcal{BN}$) and obtained the learning results for the Artificial School Database. V. Santos Costa was partially supported by CNPq and Fundação para a Ciência e Tecnologia.

## References

1. Jensen, F.V.: Bayesian Networks and Decision Graphs. Springer, Heidelberg (2001)
2. Russel, S., Norvig, P.: Artificial intelligence (1996)
3. Getoor, L., Friedman, N., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Dzeroski, S., Lavrac, N. (eds.) Relational Data Mining, pp. 307–335. Springer, Berlin (2001)
4. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. Journal of Artificial Intelligence Research 15, 391–454 (2001)
5. Ngo, L., Haddawy, P.: Probabilistic logic programming and bayesian networks. In: Algorithms, Concurrency and Knowledge, pp. 286–300. Springer, Heidelberg (1995)
6. Muggleton, S.: Stochastic logic programs. In: De Raedt, L. (ed.) Advances in Inductive Logic Programming. Frontiers in Artificial Intelligence and Applications, vol. 32, pp. 254–264. IOS Press, Amsterdam (1996)
7. Kersting, K., De Raedt, L.: Bayesian logic programs. Technical Report 151, Institute for Computer Science, University of Freiburg, Germany (2001)
8. Srinivasan, A.: The Aleph Manual (2001)
9. Blockeel, H.: Prolog for Bayesian networks: A meta-interpreter approach. In: Proceedings of the 2nd International Workshop on Multi-Relational Data Mining (MRDM-2003), pp. 1–13 (2003), `http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=40881`

10. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers Inc., San Francisco (1988)
11. Koller, D., Pfeffer, A.: Learning probabilities for noisy first-order rules. In: IJCAI 1997, Nagoya, Japan (1997)
12. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence 113, 41–85 (1999)
13. Page, C.D.: Anti-unification in constraint logics. PhD thesis, University of Illinois at Urbana-Champaign, UIUCDCS-R-93-1820 (1993)
14. Getoor, L., Friedman, N., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Relational Data Mining, pp. 307–335. Springer, Heidelberg (2001)
15. Jaakkola, T., Jordan, M.I.: Variational probabilistic inference and the QMR-DT network. Journal of Artificial Intelligence Research 10, 291–322 (1999)
16. Choi, A., Darwiche, A.: A variational approach for approximating Bayesian networks by edge deletion. In: Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI), pp. 80–89 (2006)
17. Poon, H., Domingos, P.: Sound and efficient inference with probabilistic and deterministic dependencies. In: Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, Boston, Massachusetts, USA, July 16-20, 2006, AAAI Press, Menlo Park (2006)
18. Chavira, M., Darwiche, A.: Compiling Bayesian networks using variable elimination. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI), pp. 2443–2449 (2007)
19. Rabiner, L.R.: A tutorial on hidden Markov models and selected apllications in speech recognition. In: Waibel, A., Lee, K.-F. (eds.) Readings in Speech Recognition, pp. 267–296. Kaufmann, San Mateo (1990)
20. Sato, T., Kameya, Y., Zhou, N.-F.: Generative modeling with failure in PRISM. In: IJCAI 2005, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005, pp. 847–852 (2005)
21. Sato, T., Kameya, Y.: New Advances in Logic-Based Probabilistic Modeling. In: Probabilistic Inductive Logic Programming, Springer, Heidelberg (2007)
22. Sanghai, S., Domingos, P., Weld, D.S.: Dynamic probabilistic relational models. In: IJCAI 2003, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003, pp. 992–1002 (2003)
23. Kersting, K., Raiko, T., Kramer, S., De Raedt, L.: Towards Discovering Structural Signatures of Protein Folds based on Logical Hidden Markov Models. In: Proceedings of the Pacific Symposium on Biocomputing (PSB 2003), Kauai, Hawaii, pp. 3–7 (2003)
24. Kersting, K., de Raedt, L., Raiko, T.: Logical Hidden Markov Models. Journal of Artificial Intelligence Research 25, 425–456 (2006)
25. Kersting, K., de Raedt, L., Gutmann, B., Karwath, A., Landwehr, N.: Relational Sequence Learning. In: Probabilistic Inductive Logic Programming, Springer, Heidelberg (2007)
26. Eddy, S.: Profile hidden Markov models. Bioinformatics 14, 755–763 (1998)
27. Bateman, A., Coin, L., Durbin, R., Finn, R., Hollich, V., Griffiths, S., Khanna, A., Marshall, M., Moxon, S., Sonnhammer, E., Studholme, D., Yeats, C., Eddy, S.: The pfam protein families database. Nucleic Acids Research 32, 138–141 (2004)
28. Ramakrishnan, I.V., et al.: Efficient Tabling Mechanisms for Logic Programs. In: 12th ICLP, Tokyo, Japan, pp. 687–711 (1995)

29. Rocha, R., Silva, F., Santos Costa, V.: On Applying Or-Parallelism and Tabling to Logic Programs. Theory and Practice of Logic Programming Systems 5(1–2), 161–205 (2005)
30. Domingos, P.: Personal communication (December 2002)
31. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: Ben-David, S., Case, J., Maruoka, A. (eds.) ALT 2004. LNCS (LNAI), vol. 3244, pp. 19–36. Springer, Heidelberg (2004)
32. Muggleton, S.: Learning structure and parameters of stochastic logic programs. In: Matwin, S., Sammut, C. (eds.) ILP 2002. LNCS (LNAI), vol. 2583, pp. 198–206. Springer, Heidelberg (2003)
33. Cussens, J.: Logic-based Formalisms for Statistical Relational Learning. In: Introduction to Statistical Relational Learning, MIT Press, Cambridge (2007)
34. de Raedt, L., Kersting, K.: Introduction. In: Probabilistic Inductive Logic Programming, Springer, Heidelberg (2007)
35. Clark, K.L., McCabe, F.G.: PROLOG: A language for implementing expert systems. Machine Intelligence 10, 455–470 (1982)
36. Haddawy, P.: An overview of some recent developments in Bayesian problem solving techniques. AI Magazine (1999)
37. Poole, D.: Probabilistic Horn abduction and Bayesian networks. Artificial Intelligence 64, 81–129 (1993)
38. Angelopoulos, N.: Probabilistic Finite Domains. PhD thesis, Dept of CS, City University, London (2001)
39. Murphy, K.P.: The Bayes Net Toolbox for Matlab. Computing Science and Statistics (2001)
40. Ong, I.M., Glasner, J.D., Page, D.: Modelling regulatory pathways in e. coli from time series expression profiles. In: Proceedings of the Tenth International Conference on Intelligent Systems for Molecular Biology, Edmondon, Alberta, Canada, August 3-7, 2002, pp. 241–248 (2002)
41. Ong, I.M., Topper, S.E., Page, D., Santos Costa, V.: Inferring regulatory networks from time series expression data and relational data via inductive logic programming. In: Proceedings of the Sixteenth International Conference on Inductive Logic Programming, Santiago de Compostela, Spain (2007)
42. Davis, J., Burnside, E.S., Dutra, I., Page, D., Ramakrishnan, R., Santos Costa, V., Shavlik, J.W.: View learning for statistical relational learning: With an application to mammography. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI 2005, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005, pp. 677–683. Professional Book Center (2005)
43. Landwehr, N., Kersting, K., De Raedt, L.: nFOIL: Integrating naïve Bayes and FOIL. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, Pittsburgh, Pennsylvania, USA, July 9-13, 2005, pp. 795–800 (2005)
44. Davis, J., Burnside, E.S., de Castro Dutra, I., Page, D., Santos Costa, V.: An integrated approach to learning bayesian networks of rules. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 84–95. Springer, Heidelberg (2005)
45. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI) (2007)

46. Davis, J., Ong, I., Struyf, J., Burnside, E., Page, D., Santos Costa, V.: Change of Representation for Statistical Relational Learning. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI) (2007)
47. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62, 107–136 (2006)
48. Spiegelhalter, D., Thomas, A., Best, N., Gilks, W.: BUGS 0.5 Bayesian inference using Gibbs Sampling Manual. MRC Biostatistics Unit, Cambridge (1996)