cProbLog: Restricting the Possible Worlds of Probabilistic Logic Programs

Dimitar Shterionov, Gerda Janssens

KULeuven, Belgium,
e-mail: firstname.lastname@cs.kuleuven.be

Abstract. A program in the Probabilistic Logic Programming language ProbLog defines a distribution over possible worlds. Adding evidence (a set of ground probabilistic atoms with observed truth values) rules out some of the possible worlds. Generalizing the evidence atoms to First Order Logic constraints increases the expressive power of ProbLog. In this paper we introduce the first implementation of cProbLog – the extension of ProbLog with constraints. Our implementation transforms ProbLog programs with FOL constraints into ProbLog programs with evidence that specify the same possible worlds. We backup our design and implementation decisions with a series of examples.

1 Introduction

Statistical Relational Learning [5] (SRL) combines logic programming and machine learning with uncertainties. ProbLog [2, 7] is a powerful, general-purpose SRL system based on Prolog. The main inference task of ProbLog is to compute the marginal probabilities of a set of queries conditioned on some evidence[1] – the MARG task. ProbLog is a suitable formalism for a wide range of problems. In some cases, though, it lacks language expressivity and users need to write more complex programs to add additional knowledge. cProbLog [3] proposes to extend the expressive power of ProbLog by generalizing evidence atoms (that state which (atomic) observations are true or false) into First-Order Logic sentences. Each of these sentences expresses a *constraint* that has to hold.

The ProbLog program in Example 1 encodes a small road map among 4 cities. Each atom of the form $p_i :: road(a_i, b_i)$ is a probabilistic fact: p_i denotes the existence probability of the fact $road(a_i, b_i)$. It states that the road connecting cities a_i and b_i is available (eg. free of traffic jams) with probability p_i. Each probabilistic fact is either true or false in different interpretations of the program. An interpretation of the ProbLog program, called a *possible world*, contains all ground atoms with a specific truth value assignment. ProbLog defines a distribution over all possible worlds. A query is true in some of these possible worlds. Eg., `reach(c1, c4)` is true in 7 of the 16 possible worlds. Its (success) probability, computed over these possible worlds, is $P(reach(c1, c4)) = 0.7195$.

[1] When there is no evidence we refer to the computed probability as *success probability*.

Example 1.

```
0.7::road(c1, c2).    0.7::road(c2, c4).    0.5::road(c1, c3).    0.9::road(c3, c4).
reach(A, B):-road(A, B).
reach(A, B):-road(A, A1), reach(A1, B).
query(reach(c1, c4)).
```

In ProbLog additional knowledge such as *the road between city 3 and city 4 is not available*, is given as evidence: `evidence(road(c3, c4), false)`. This evidence makes all possible worlds where $road(c1, c3)$ is true invalid and the probability of the query is calculated with respect to the rest, in which $road(c3, c4)$ is false. In ProbLog, evidence can express only observations on ground atoms. That is why encoding evidence over more complex knowledge can become cumbersome. Eg., to state that *the road between city 2 and city 4 or the road between city 3 and city 4 is available* will require the following code to be added to the program of Example 1:

```
add_ev:- road(c2, c4).   add_ev:- road(c3, c4).   evidence(add_ev, true).
```

cProbLog allows one to encode complex additional knowledge as a single FOL sentence (or *constraint*): `constraint(road(c2, c4) or road(c3, c4)).`. The probability of the query is then computed with respect to the possible worlds which are valid according to the constraints. Constraints generalize evidence to FOL sentences. Furthermore, a cProbLog constraint can also contain universally and/or existentially quantified variables: $\exists X : road(X, c4)$. The possible values for every quantified variable need to be defined explicitly. To do so, we introduce in cProbLog the notion of *domains*.

In this paper we describe the first implementation of cProbLog. Our algorithm converts cProbLog constraints (FOL sentences) to ground ProbLog rules and evidence. To illustrate the requirements towards cProbLog and backup our design decisions we use a series of examples.

2 Background

2.1 ProbLog

ProbLog is a probabilistic logic formalism, based on Prolog. It combines techniques from logic programming and machine learning to reason with uncertain knowledge. A ProbLog program is a logic program where some facts are annotated with probabilities. A probabilistic fact has the form $p_i :: f_i$, where p_i denotes the existence probability of the fact f_i. It can be either true with probability p_i or false with probability $(1-p_i)$. Each probabilistic fact gives rise to two possible interpretations: one where the fact is true and another where the fact is false. A specific choice on the truth values of all probabilistic facts determines a unique interpretation, called a *possible world*. A ProbLog program defines a distribution over possible worlds according to the Distribution Semantics [10].

Let $\Omega = \{\omega_1, .., \omega_n\}$ be the set of possible worlds corresponding to a ProbLog program.Given that only probabilistic facts have probabilities, we view a possible world ω_i as the tuple $\omega_i = (\omega_i^+, \omega_i^-)$, where ω_i^+ is the set of true ground probabilistic atoms and ω_i^- the set of false ground probabilistic atoms. Each atom a_j in ω_i^+ is true with probability p_j; each atom a_j in ω_i^- is false with probability

$(1-p_j)$. The union $\omega_i^+ \cup \omega_i^-$ is the set of all possible ground probabilistic atoms of the ProbLog program with a specific truth value assignment corresponding to the possible world ω_i. The intersection $\omega_i^+ \cap \omega_i^-$ is the empty set.

A ProbLog program then defines a probability distribution over possible worlds as given in Equation 1, where with p_i we denote the probability of the atom a_i.

$$P(\omega_i) = \prod_{a_j \in \omega_i^+} p_j \prod_{a_j \in \omega_i^-} (1-p_j) \tag{1}$$

A query atom q is true in a subset of the possible worlds: $\Omega^q \subseteq \Omega$. Each $\omega_i^q \in \Omega^q$ has a corresponding probability as computed by Equation 1. The probability of q ($P(q)$) is the sum of the probabilities of all worlds in which q is true. Enumerating all possible worlds is almost always impossible. Thus, ProbLog uses a form of Weighted Model Counting for MARG inference [3, 4].

Additional knowledge about the problem can be encoded as *evidence*. Evidence is a set of pairs (a_i, α_i) with α_i the observed truth value of the ground atom a_i ($\alpha_i \in \{true, false\}$). We denote with $E = \{a_i\}$ the set of evidence atoms and with $e = \{\alpha_i\}$ – their corresponding truth values. Evidence restricts the set of possible worlds of a ProbLog program to the ones in which each $a_i = \alpha_i$ is valid. A query q can be true in some of these possible worlds. These are the ones in which the conjunction $q \wedge E = e$ is true: $\Omega^{q \wedge E=e} \subseteq \Omega^q \subseteq \Omega$. Using Bayes' rule ProbLog computes the marginal probability $P(q|E=e) = \frac{P(q \wedge E=e)}{P(E=e)}$.

Consider the ProbLog program of Example 2. It encodes a problem where some items need to be packed in a suitcase. Each item has a certain weight and a probability to be packed. The total weight of the suitcase needs to remain within a predetermined limit. The predicates `query/1` and `evidence/2` are ProbLog[2] built-ins to define queries and evidence. The query `inlimit(10)`, asks the probability of packing items with a total weight of less or equal than 10. Table 1 enumerates all possible worlds of the ProbLog program. Using Equation 1 for the possible worlds in which the query is true – $\{\omega_4, \omega_7, \omega_8, \omega_{11}, \omega_{12}, \omega_{14}, \omega_{15}, \omega_{16}\}$, we compute its probability: $P(inlimit(10)) = 0.9162$.

Example 2.

```
weight(skis,6).       weight(board, 8).       weight(boots,4).       weight(helmet,3).
0.16::pack(skis).     0.125::pack(board).     0.25::pack(boots).     0.33::pack(helmet).
inlimit(Limit) :- inlimit([skis,boots,board,helmet],Limit).
inlimit([],Limit) :- Limit>=0.
inlimit([I|R],Limit) :- pack(I), weight(I,W), L is Limit-W, inlimit(R,L).
inlimit([I|R],Limit) :- \+pack(I), inlimit(R,Limit).
query(inlimit(10)).
```

Example 3. For Example 2 an observation (evidence) like *boots are already packed* restricts the set of valid possible worlds to $\{\omega_4, \omega_{11}, \omega_{12}\}$ and gives the probability $P(inlimit(10)|pack(boots) = true) = \frac{0.2072}{0.25} = 0.8288$.

The requirement *if the skis are packed then also the boots need to be packed* can be expressed by the *additional* rules: `add_ev:- \+ pack(skis). add_ev:- pack(boots).` and the evidence *add_ev* is true. The conditional probability is then $P(inlimit(10)|add\_ev = true) = \frac{0.8112125}{0.88} = 0.9218$.

[2] When discussing the ProbLog system we refer to its most recent version ProbLog2

Possible World	pack: skis	boots	board	helmet	Query inlimit(10)	Probability of pack: skis	boots	board	helmet	$P(\omega)$
ω_1	T	T	T	T	F	0.16	0.25	0.125	0.33	0.002
ω_2	T	T	T	F	F	0.16	0.25	0.125	0.67	0.003
ω_3	T	T	F	T	F	0.16	0.25	0.875	0.33	0.012
ω_4	T	T	F	F	T	0.16	0.25	0.875	0.67	0.023
ω_5	T	F	T	T	F	0.16	0.75	0.125	0.33	0.005
ω_6	T	F	T	F	F	0.16	0.75	0.125	0.67	0.010
ω_7	T	F	F	T	T	0.16	0.75	0.875	0.33	0.035
ω_8	T	F	F	F	T	0.16	0.75	0.875	0.67	0.070
ω_9	F	T	T	T	F	0.84	0.25	0.125	0.33	0.009
ω_{10}	F	T	T	F	F	0.84	0.25	0.125	0.67	0.018
ω_{11}	F	T	F	T	T	0.84	0.25	0.875	0.33	0.061
ω_{12}	F	T	F	F	T	0.84	0.25	0.875	0.67	0.123
ω_{13}	F	F	T	T	F	0.84	0.75	0.125	0.33	0.026
ω_{14}	F	F	T	F	T	0.84	0.75	0.125	0.67	0.053
ω_{15}	F	F	F	T	T	0.84	0.75	0.875	0.33	0.182
ω_{16}	F	F	F	F	T	0.84	0.75	0.875	0.67	0.369

Table 1: Possible worlds and their probabilities of the packing program of Example 2.

2.2 cProbLog

By introducing FOL constraints cProbLog [3] increases the expressivity of the ProbLog language and allows users to encode complex evidence without directly transforming it to ProbLog. E.g., the second observation in Example 3 can be written as an FOL sentence: $pack(skis) \Rightarrow pack(boots)$.

Constraints are true in a subset of the possible worlds of a ProbLog program. Satisfying the constraints restricts the set of possible worlds of a ProbLog program. Thus, we categorize cProbLog constraints as *restrictive*. We write $\omega \models c$ to express that the constraint c is satisfied in a possible world ω. From Table 1 we see that the constraint $c = pack(skis) \Rightarrow pack(boots)$ is satisfied in $\Omega^C = \{\omega_1, .., \omega_4, \omega_9, .., \omega_{16}\}$ ($\omega_i^c \models c$, for $\omega_i^c \in \Omega^C$).

cProbLog was introduced in [3] where the authors define its semantics. In the current paper we present an in-depth analysis of the semantics, the design and the first implementation of cProbLog. Our approach is based on the relation between constraints and evidence. Our implementation is not strictly related to the rest of ProbLog inference pipeline and thus can easily be employed by other probabilistic logic programming languages.

3 Syntax and Semantics

3.1 Syntax

cProbLog extends the ProbLog language with FOL constraints. That is, the cProbLog language consists of (i) facts – probabilistic and non probabilistic, (ii) logic programming rules, (iii) *built-in*s and (iv) FOL constraints.

The FOL sentences of cProbLog constraints use standard logic operators: $\forall$, $\exists$, $\wedge$, $\vee$, $\neg$, $\Leftrightarrow$ and $\Rightarrow$, represented in cProbLog by `for_all`, `exists`, `and`, `or`, `not`, `iff` and `implies`. The quantifier delimiter is written as `:` and parentheses are used to indicate priority.

Example 4. In cProbLog the constraint of Example 3, *if the skis are packed then also the boots need to be packed* is written as: `pack(skis) implies pack(boots)`

Example 4 shows a cProbLog constraint with two ground probabilistic atoms. The generalization power of cProbLog, though, comes with the support of non ground constraints – FOL sentences with quantified variables. In a cProbLog constraint the range of values of every quantified variable needs to be explicitly defined. ProbLog is an untyped language hence we do not introduce type declarations for cProbLog, but rather an explicit enumeration of values, that is, a *domain*. The domain has to be specified in the constraint. There are two ways to express a domain: either as a set enumerating all possible values of the concrete variable or as a call to a predicate; this predicate serves as a generator for the values of the variable. In principle the domain definition can be a more complex (Prolog) expression as long as it generates the values for the variables.

Example 5. Assume it is required to have at least 2 items in the suitcase and one of them should be the boots. We can encode this constraint in two ways:
(i) `pack(boots) and exists X of item(X) : pack(X) and not X == boots.`
(ii) `pack(boots) and exists X in {skis, boots, board, helmet} : pack(X) and not X==boots.`
The first formulation requires that the ProbLog program defines a predicate `item/1` that, naturally, contains all items which can possibly be packed. It acts as value enumerator. The second one defines the domain of `X` as a set. We can simplify the constraint by omitting `boots` from the domain of `x` and remove `not X == boots.`: `pack(boots) and exists X in {skis, board, helmet} : pack(X)`

In cProbLog we use the `constraint/1` predicate to define a constraint. The last constraint of Example 5 can be stated in a cProbLog program as:
`constraint(pack(boots) and exists X in {skis, board, helmet} : pack(X)).`

3.2 Semantics

The semantics of cProbLog is based on the semantics of ProbLog (cf. Section 2.1). Consider a ProbLog program L and a set of constraints $C = \{c_1, ..c_n\}$ which, together, constitute a cProbLog program L^C. L has a set of possible worlds Ω. As stated in Section 2.2, constraints in cProbLog are restrictive. That is, the set of possible worlds of L^C (where the constraints are satisfied) is a subset of the possible worlds of L: $\Omega^C \subseteq \Omega$.

For $\Omega^C = \{\omega_k | \omega_k \in \Omega \text{ and } \omega_k \models C\}$, the distribution of the cProbLog program L^C over possible worlds is as defined in Equation 2.

$$P_C(\omega_i) = \begin{cases} \frac{P(\omega_i)}{\sum_{\omega_k \in \Omega^C} P(\omega_k)}, & \text{if } w_i \in \Omega^C \\ 0, & \text{if } \omega_i \in \Omega \setminus \Omega^C \end{cases}, \tag{2}$$

where $P(\omega_i)$ is as defined in Equation 1. The denominator which expresses a summation over the probabilities of all possible worlds which satisfy the constraints, is a normalization factor.

Table 2 shows the application of Equation 2 for each possible world of the program of Example 2 and the constraint of Example 4.

Possible World	pack: skis	boots	board	helmet	Query inlimit(10)	Constraint	$P(\omega)$	$P_C(\omega)$ $\omega \models C$
ω_1	T	T	T	T	F	T	0.002	0.0019
ω_2	T	T	T	F	F	T	0.003	0.0038
ω_3	T	T	F	T	F	T	0.012	0.0131
ω_4	T	T	F	F	T	T	0.023	**0.0266**
ω_5	T	F	T	T	F	F	/	0.0000
ω_6	T	F	T	F	F	F	/	0.0000
ω_7	T	F	F	T	T	F	/	0.0000
ω_8	T	F	F	F	T	F	/	0.0000
ω_9	F	T	T	T	F	T	0.009	0.0098
ω_{10}	F	T	T	F	F	T	0.018	0.0200
ω_{11}	F	T	F	T	T	T	0.061	**0.0689**
ω_{12}	F	T	F	F	T	T	0.123	**0.1399**
ω_{13}	F	F	T	T	F	T	0.026	0.0295
ω_{14}	F	F	T	F	T	T	0.053	**0.0600**
ω_{15}	F	F	F	T	T	T	0.182	**0.2067**
ω_{16}	F	F	F	F	T	T	0.369	**0.4197**
						Sum:	0.88	

Table 2: The possible worlds and the corresponding probabilities of the program of Example 2 given the constraint `pack(skis) implies pack(boots).`

For a ProbLog program the success probability of a query q is the sum of the probabilities of all possible worlds in which q is true. In cProbLog, satisfying a set of constraints C limits the set of possible worlds in which q is true to the ones where $q \wedge C$ is true: $\Omega^{q \wedge C} = \Omega^q \cap \Omega^C$. We define the conditional probability of a query (q) given a set of constraints (C) are satisfied as:

$$P(q|C) = \sum_{\omega_i \in \Omega^{q \wedge C}} P_C(\omega_i) = \frac{\sum_{\omega_i \in \Omega^{q \wedge C}} P(\omega_i)}{\sum_{\omega_i \in \Omega^C} P(\omega_i)} \tag{3}$$

Applying Equation 3 on Table 2, that is, summing up the numbers in bold results in: $P(inlimit(10)|$`pack(skis) implies pack(boots)`$) = 0.9218$.

Note the similarity between Equation 3 and the formula used for computing the conditional probability given evidence (cf. Section 2.1). cProbLog constraints are a generalization of evidence and in practice Equation 3 computes the probability over the possible worlds restricted by the constraints or evidence (Ω^C or $\Omega^{E=e}$) in which also a query q is true.

4 Inference with cProbLog

In Section 2 we showed that the same additional knowledge can be expressed either as a FOL constraint, or as an adequate ProbLog predicate (`add_ev/0`) that defines the "evidence" that has to be true (cf. Example 3). After adding the predicate `add_ev/0` to the initial ProbLog program and imposing the evidence $add\_ev = true$, ProbLog computes the conditional probability of the query given $add\_ev = true$. Since both add_ev and the constraint validate the same possible worlds, the computed probability is in practice the conditional probability of the query given that the constraint is satisfied. The transformation of constraints into ProbLog rules is the basis of our cProbLog implementation.

4.1 ProbLog Inference Pipeline

As stated in Section 2.1, explicitly enumerating possible worlds is not feasible. That is why the inference mechanism of ProbLog for the MARG task [4] employs several distinct tasks to transform the initial program to a weighted Boolean circuit and does a weighted model counting on this circuit. The weighted Boolean circuit, in practice, encodes implicitly the possible worlds and their probabilities. First, it grounds the initial ProbLog program L with respect to a set of query atoms Q and a set of evidence atoms $E = e$ yielding a ground program L_r relevant only to Q and E. An atom is relevant with respect to Q and E if it occurs in some proof of a goal $g \in Q \cup E$. A ground rule is relevant with respect to Q and E if its head is a relevant atom and its body only contains relevant atoms. ProbLog uses SLD resolution with the atoms in Q and E as its initial queries to determine L_r, that is, to collect all relevant ground atoms and all relevant ground rules. Furthermore, ProbLog uses tabling to avoid querying the same atom twice. Next L_r is converted into a CNF formula φ_r.

The conjunction φ_E of all atoms in E with observed value *true* and the negation of the atoms in E with value *false* states that the evidence should hold.

ProbLog then uses the formula $\varphi = \varphi_r \wedge \varphi_E$ to generate a Boolean circuit which allows to efficiently compute the conditional probabilities of the queries given the evidence [4].

4.2 cProbLog Inference Algorithm

Consider the simple case with no queries ($Q = \emptyset$) but only evidence (E), thus, L_r is relevant only to E. The models of φ satisfy the evidence. Consider a cProbLog constraint c that is ground and in CNF, which we write as φ_c. We can, by using the ProbLog approach, find the relevant grounding for each of the ground atoms of c and generate the corresponding CNF φ_r^c. The models of the conjunction $\varphi^c = \varphi_r^c \wedge \varphi_c$ are models of L which satisfy c. In the case that c and E contain exactly the same atoms and c enforces the same truth values as in E, $\varphi^c \Leftrightarrow \varphi$.

Instead of generating φ_r^c for the constraint c, we can also add the rule: `add_ev:-c.` and the evidence $add\_ev = true$. `add_ev` is true if and only if the constraint formula c is true, thus the CNF contains $add\_ev \Leftrightarrow \varphi_c$. ProbLog will then find the relevant instances of the rules for the ground atoms in c (as it does for evidence atoms) and converts them into $\varphi_r^{c'}$. Now we have $\varphi_r^{c'} \wedge (add\_ev \Leftrightarrow \varphi_c) \wedge (add\_ev = true)$ which is equivalent to $\varphi_r^c \wedge \varphi_c$.

Generally, $Q \neq \emptyset$, and L_r also contains the relevant groundings for the queries $q_i \in Q$ which will appear in φ_r as is needed for correct weighted model counting.

We cannot use general FO sentences with quantified variables as the body of a ProbLog rule. That is why, in order to use the *constraint-evidence* approach, for each constraint that is not in CNF and is not ground, we apply a set of transformations. They are given in Fig. 1. Our method first makes sure that the FOL constraint is in prenex normal form [6] (PNF). In PNF all quantifiers are moved at the left hand side and the formula (variables and conjunctives) is at the RHS. This form facilitates the application of our rewriting rules. Other

systems that handle FOL constraints (e.g. IDP [8]) push quantifiers inwards to delay their instantiation and detect failure as soon as possible. This is not the case for cProbLog: the constraints are satisfied during ProbLog grounding for which the complete instance set needs to be collected first. The rewriting rules of Fig. 1 exhaustively generate all possible combinations of the domain elements. The complexity of converting one constraint is then $O(n^m)$, where n is the size of the (largest) domain and m is the number of variables for a single constraint.

For a set of variables $x_1..x_n$ with domains D_{x_1} to D_{x_n} and a FOL sentence $F(x_1, .., x_n)$:

1. Convert to Prenex Normal Form:
$F(x_1, .., x_n) \overset{PNF}{\longrightarrow} F^{PNF}(x_1, .., x_n)$

2. Apply rewriting rules:

R1: $\exists x_i D_{x_i} : F^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i, .., x_n) \rightarrow F^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i/d_{x_i}, .., x_n)$
$\vee\ \exists x_i D_{x_i} \setminus \{d_{x_i}\} : F^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i, .., x_n)$

R2: $\forall x_i D_{x_i} : F^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i, .., x_n) \rightarrow F^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i/d_{x_i}, .., x_n)$
$\wedge\ \forall x_i D_{x_i} \setminus \{d_{x_i}\} : F^{PNF}(x_1/d_{x_1}, .., x_{i-1}/d_{x_{i-1}}, x_i, .., x_n)$

R3: $F^{PNF}(x_1/d_{x_1}, .., x_n/d_{x_n}) \rightarrow$ `add_ev :-` $F^{PNF}(x_1/d_{x_1}, .., x_n/d_{x_n})(\wedge/,, \vee/;)$.

Fig. 1: Transformation steps for constraint initialization.

Fig. 2 shows the application of our approach on a small probabilistic graph (similar to Example 1). Our rewriting rules in practice generate a ProbLog program from a cProbLog one. Once the constraints are converted to ProbLog clauses we add the clause `evidence(add_ev, true).`. ProbLog then can apply its inference mechanism to compute the probability of a query given the evidence that the constraints are satisfied. In the process it finds the relevant ground program with respect to the query (`reach(c1, c3)`) and the evidence (`add_ev`).

```
i_city(c1).    i_city(c2).
e_city(c2).    e_city(c3).
0.7::road(c1,c2).  0.8::road(c2,c3).
0.9::road(c1,c3).
reach(A,B):- road(A,B).
reach(A,B):- road(A,A1), reach(A1,B).
query(reach(c1,c3)).
constraint((for_all X of i_city(X),
  exists Y of e_city(Y)) : reach(X,Y)).
```

a. Code

1. Apply R2:
`exists Y of e_city(Y) : reach(c1,Y)`$\wedge$
`for_all X of i_city(X)`$_{X \neq 1}$`, exists Y of e_city(Y) : reach(X,Y)`
2. Apply R1:
`(reach(c1,c2)`$\vee$`exists Y of e_city(Y)`$_{Y \neq 2}$`: reach(c1,Y))`$\wedge$
`for_all X of i_city(X)`$_{X \neq 1}$`, exists Y of e_city(Y) : reach(X,Y)`
3. Apply R1:
`(reach(c1,c2)`$\vee$`reach(c1,c3))`$\wedge$
`for_all X of i_city(X)`$_{X \neq 1}$`, exists Y of e_city(Y) : reach(X,Y)`
4. Apply R2, R1, R1, R3:

```
add_ev :- (reach(c1,c2); reach(c1,c3)),
          (reach(c2,c2); reach(c2,c3)).
```

b. Conversion

Fig. 2: A small cProbLog program and the conversion of constraints to evidence.

5 Examples

This section aims at familiarizing the user with writing constraints in cProbLog and showing the equivalence with ProbLog evidence: where feasible we give the equivalent evidence atoms as it would have to be written in ProbLog.

5.1 Probabilistic Graph

The program presented in Fig. 3 encodes a probabilistic graph where each edge has a length of 1. The `path/3` predicate finds a path between two nodes, as well as its length. We use constraints to add knowledge about the path length(s). Eg., one can query for the probability of a path from `a` to `h` only considering paths longer than 5. The conditional probability $P(path(a, h)|C)$ is 0.014. Here,

```
0.6::edge(a,b). 0.7::edge(a,c). 0.55::edge(b,c). 0.36::edge(b,d). 0.45::edge(c,e). 0.7::edge(d,f).
0.8::edge(e,d). 0.25::edge(e,g). 0.25::edge(f,g). 0.3::edge(f,h). 0.4::edge(g,h).
path(A, B):- path(A, B, _).
path(A, B, 1):- edge(A, B).
path(A, B, L):- edge(A, B1), path(B1, B, L1), L is L1 + 1.
query(path(a, h)).          constraint(for_all L in {1,2,3,4,5} : not path(a, h, L)).
```

Fig. 3: The "length path" example program.

the domain of `L` is given as a set. For this example, instead of the constraint one can state the evidence: `evidence(path(a, h, 1), false). evidence(path(a, h, 2), false). evidence(path(a, h, 3), false). evidence(path(a, h, 4), false). evidence(path(a, h, 5), false).`

5.2 Burglary-earthquake-alarm Bayesian network

Fig. 4 illustrates a cProbLog program which encodes a Bayesian network [4]. The program defines two neighbors – John and Mary. Burglary and earthquake may trigger an alarm. If the alarm goes off, one of the people calls the owner of the house. The initial program states that either John or Mary or both may call. Using a constraint we encode that at most one of them calls. For this example we can easily write a predicate equivalent to our rule rewriting transformation but much more simple (see Fig. 4 b) and c)). The observed blow-up is due to the fact that variables are blindly instantiated with values from their domain, without evaluating any subgoals. The evaluation is (currently) left to ProbLog.

```
person(john).  person(mary).
0.1::burglary. 0.2::earthquake.
0.7::hears_alarm(X) :- person(X).
alarm:- burglary. alarm:- earthquake.
calls(X):- alarm, hears_alarm(X).
constraint((for_all X of person(X),
  for_all Y of person(Y)) :
  (calls(X) and calls(Y))
  implies X == Y).
query(burglary).  query(earthquake).
```

a) A cProbLog program.

```
add_ev:-
  ((\+ calls(mary); \+ calls(mary));
    mary==mary),
  (\+ calls(mary); \+ calls(john));
    mary==john)),
  ((\+ calls(john); \+ calls(mary));
    john==mary),
  (\+ calls(john); \+ calls(john));
    john==john)).
evidence(add_ev, true).
```

b) cProbLog transformation of the constraint.

```
ev1:-
   \+ calls(mary).
ev1:-
   \+ calls(john).
evidence(ev1, true).
```

c) User-defined clauses.

Fig. 4: An example program of a Bayesian network.

Despite this drawback, it can easily be noticed that with growth of the domain such manual encoding becomes infeasible. Moreover, the rules which cProbLog generates can be preprocessed and optimized.

5.3 Student exams

The example in Fig. 5 is about students and exams. There are four ways that a student passes an exam: by having studied enough; by luck; by cheating and by knowledge from previous experience. Passing an exam has certain probabilities, eg., a student passes an exam by luck with probability 0.4. One can query this program for the probability a student passes all exams. The first constraint is a ground constraint which expresses that one can pass "Machine Learning" or "Artificial Neural Networks (ANNs)" by luck but not both of them. The second constraint defines that 3 years of experience are not enough to pass any exam. For these constraints there is no straightforward alternative using evidence.

```
0.8::pass_exam(Exam, studied_enough). 0.4::pass_exam(Exam, luck). 0.7::pass_exam(Exam, cheating).
P::pass_exam(Exam, experience, Years):- P is Years/7.
exam(E):- member(E, [prolog, cog_sci, anns, comp_vis, m_learn]).
student(john, [prolog, cog_sci, anns, comp_vis, m_learn], 3).
succeed(Student):- student(Student, Exams, Experience), pass_all(Exams, Experience).
pass_all([], _).
pass_all([F|Rest], Exp):- pass_one(F, Exp), pass_all(Rest, Exp).
pass_one(Ex, _):-pass_exam(Ex, _).
pass_one(Ex, Years):-pass_exam(Ex, experience, Years).
constraint(pass_exam(m_learn, luck) implies not pass_exam(anns, luck)).
constraint((for_all X of exam(X), for_all Y in {0,1,2,3}) : not pass_exam(X, experience, Y)).
query(succeed(john)).
```

Fig. 5: The "student exams" example program.

6 Related Work

PCLP. A PCLP [9] (short for Probabilistic Constraint Logic Programming) theory $\mathcal{T}$ is defined by a set of constraints $\mathcal{C}_\mathcal{T}$, a set of random variables $\mathcal{V}_\mathcal{T}$ and a set of rules $\mathcal{R}_\mathcal{T}$. $V(t_1,..,t_n) \sim \{p_1 : c_1,..,p_m : c_m\}$ defines the random variable $V(t_1,..,t_n) \in \mathcal{V}_\mathcal{T}$ (with t_i a term), over the distribution[3] $\{p_1 : c_1,..,p_m : c_m\}$, where c_j is a constraint and p_j its probability. The constraints c_1 to c_m specify the possible values of the variable and the probability of their assignment. That is why we classify them as *generative* as opposed to the *restrictive* constraints of cProbLog, which rule out some of the possible worlds of a ProbLog program. A PCLP rule is valid only when the constraints in its body are satisfied.

With respect to the inference, the main difference among the two systems is that on the one hand, a PCLP theory $\mathcal{T}$ specifies a class of distributions in which the constraints are satisfied and the lower and upper bounds are computed from different subsets of the distributions within that class. On the other hand, cProbLog generalizes evidence by FOL constraints to allow computing conditional probabilities of queries under the conditions stated by the constraints.

Under certain conditions a PCLP theory can be mapped to a ProbLog program (with annotated disjunctions). However, there is no direct correspondence to a cProbLog program. In contrary, PCLP and cProbLog are complementary and may possibly coexist (a question which we aim to tackle in the future).

[3] The distribution can also be continuous eg., $X \sim \mathcal{N}(\mu, \sigma^2)$

CLP($\mathcal{BN}$). A CLP($\mathcal{BN}$)[1] program $L^{clp(BN)}$ is a set of clauses which contain in their bodies a (possibly empty) set of constraints. Each constraint has the form $\{V = Rv \text{ with } CPT\}$ and identifies the possible values of V with the random variable Rv. The conditional probability table CPT defines a probability distribution over the possible values of Rv. Each random variable is a Skolem term and appears in only one clause of $L^{clp(BN)}$. The constraints in CLP($\mathcal{BN}$), similarly to PCLP and in contrast to cProbLog, define the possible values of a variable (V) through the probability distribution of the random variable.

Each clause C_i, associated to a set of probability distributions defines a Bayesian network BN_i. For a given subset of clauses, the constraints create dependencies between the Bayes networks associated with these clauses to generate a large Bayesian network BN. A CLP($\mathcal{BN}$) program defines a unique joint probability distribution over the ground Skolem it contains as the Bayesian network BN. Computing the probability of a query given evidence in the framework of CLP($\mathcal{BN}$), is solving the Bayes' network generated from the conjunction of the networks corresponding to the query and to the evidence.

CHRiSM. A CHRiSM [11] program consists of a set of rewrite rules of the form `P ?? Hk \ Hr <=> G | B`, where `P` is a probabilistic expression, `Hk` is a conjunction of kept head constraints, `Hr` a conjunction of removed head constraints, `G` is a guard condition and `B` is the rule body. The rule body `B` is defined as a conjunction of CHRiSM constraints, Prolog goals and/or probabilistic disjunctions (an annotated disjunction or a CHRiSM-style disjunction).

A CHRiSM constraint `c(X1, .., Xn)` is a Prolog-like predicate.The initial multiset of constraints of a CHRiSM program is called a store $\mathcal{S}$. Applying exhaustively the rules of the program results in a new store. A rule can be applied if there is a matching substitution which unifies a subset of constraints from $\mathcal{S}$ to the head of that rule and also, the guard `G` is satisfied. Depending on the probability expression of the rule it can be either applied or ignored.

When a rule is applied all the constraints matching `Hr` are removed from $\mathcal{S}$ (the ones in `Hk` are kept), the goals in `B` are called and the constraints in `B` are added to the store. That is why we see CHRiSM constraints as *restrictive*. The goal is reached by a chain of rule applications, each specified by a *transition* probability. The probability of the final state (or the goal) is the product of the transition's probabilities. In (c)ProbLog the probability of a query considers all possible worlds defined by the random variables (the probabilistic atoms) of the program in which a query is true.

7 Conclusions and Future Work

In this paper we described the first implementation of cProbLog – a Constraint Probabilistic Logic Programming formalism which enriches ProbLog with FOL constraints. Our algorithm converts a constraint to a ground ProbLog rule and imposes evidence on it. With the existing inference mechanisms of ProbLog

the conditional probability of a query given this evidence is the same as the conditional probability of the query given that the constraints are satisfied.

Our approach is built on top of ProbLog's inference mechanism without changing it. It can be easily employed by other systems. A drawback of our current implementation is the rather naive grounding of formulas with quantifiers (cf. Fig. 4). Our current work aims at reducing the size of the grounding. Moreover, we investigate an approach which directly merges the CNF of the relevant ground program (with respect to a set of queries and evidence atoms) with the CNF of the constraints thus bypassing the conversion (from ground program to CNF) for the constraints. We investigate other fields of application of cProbLog, eg., using it to efficiently process annotated disjunctions [12] in ProbLog.

We also compared cProbLog to other Constraint Probabilistic Logic Programming formalisms from the perspective of type and usage of constraints.

References

1. Vítor Santos Costa and James Cussens. Clp(bn): Constraint logic programming for probabilistic knowledge. In *In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI03)*, pages 517–524. Morgan Kaufmann, 2003.
2. Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: a probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2468–2473. AAAI Press, 2007.
3. Daan Fierens, Guy Van den Broeck, Maurice Bruynooghe, and Luc De Raedt. Constraints for probabilistic logic programming. In *Proceedings of the NIPS Probabilistic Programming Workshop*, Dec 2012, 4 pages.
4. Daan Fierens, Guy Van Den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 2014.
5. Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.
6. Peter G Hinman. *Fundamental of mathematical logic*. Peters, Wellesley, MA, 2005.
7. Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11:235–262, 2011.
8. Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.
9. Steffen Michels, Arjen Hommersom, Peter J. F. Lucas, Marina Velikova, and Pieter W. M. Koopman. Inference for a new probabilistic constraint logic. In *IJCAI*, pages 2540–2546, 2013.
10. Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)*, pages 715–729. MIT Press, 1995.
11. Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. Chr(prism)-based probabilistic logic learning. *CoRR*, abs/1007.3858, 2010.
12. Joost Vennekens, Sofie Verbaeten, Maurice Bruynooghe, and Celestijnenlaan A. Logic programs with annotated disjunctions. In *In Proc. Int'l Conf. on Logic Programming*, pages 431–445. Springer, 2004.