Probabilistic Logic Programming on the Web

Fabrizio Riguzzi[a], Elena Bellodi[b], Evelina Lamma[b], Riccardo Zese[b], Giuseppe Cota[b]

[a]*Dipartimento di Matematica e Informatica – University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy*
[b]*Dipartimento di Ingegneria – University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy*

Abstract

We present the web application "`cplint` on SWISH", that allows the user to write probabilistic logic programs and compute the probability of queries with just a web browser. The application is based on SWISH, a recently proposed web framework for logic programming. SWISH is based on various features and packages of SWI-Prolog, in particular its web server and its Pengine library, that allow to create remote Prolog engines and to pose queries to them. In order to develop the web application, we started from the PITA system which is included in `cplint`, a suite of programs for reasoning on Logic Programs with Annotated Disjunctions, by porting PITA to SWI-Prolog. Moreover, we modified the PITA library so that it can be executed in a multi-threading environment. Developing "`cplint` on SWISH" also required modification of the JavaScript SWISH code that creates and queries Pengines. "`cplint` on SWISH" includes a number of examples that cover a wide range of domains and provide interesting applications of Probabilistic Logic Programming (PLP). By providing a web interface to `cplint` we allow users to experiment with PLP without the need to install a system, a procedure which is often complex, error prone and limited mainly to the Linux platform. In this way, we aim to reach out to a wider audience and popularize PLP.

Keywords: Logic Programming, Probabilistic Logic Programming,

Email addresses: `fabrizio.riguzzi@unife.it` (Fabrizio Riguzzi), `elena.bellodi@unife.it` (Elena Bellodi), `evelina.lamma@unife.it` (Evelina Lamma), `riccardo.zese@unife.it` (Riccardo Zese), `giuseppe.cota@unife.it` (Giuseppe Cota)

Distribution Semantics, Logic Programs with Annotated Disjunctions, Web Applications

1. Introduction

Probabilistic Logic Programming (PLP) introduces probabilistic reasoning in logic programs in order to represent uncertain information. PLP is receiving an increasing attention due to its applications in particular in the Machine Learning field [1], where many domains are characterized by complex and uncertain relations among their entities.

Many PLP languages have adopted the distribution semantics [2], according to which a program defines a probability distribution over normal logic programs called worlds and the probability of a query is obtained from this distribution over programs. Examples of languages following the distribution semantics include Probabilistic Logic Programs [3], Independent Choice Logic [4], PRISM [5], pD [6], Logic Programs with Annotated Disjunctions (LPADs) [7], CP-logic [8] and ProbLog [9]. All these languages have the same expressive power as a theory in one language can be translated into another [10, 11].

Inference in PLP amounts to computing the probability of queries from a program. Solving this problem in a fast way is fundamental especially for Machine Learning applications, where a large number of queries have be to answered. Several algorithms have been proposed for this, such as PRISM [5], Ailog2 [12], ProbLog [13], SLGAD [14, 15] and PITA [16, 17]. Moreover, `cplint` (CP-logic interpreter) [18, 19] is a suite of programs for inference and learning of LPADs.

In this paper, we present the "`cplint` on SWISH" web application for performing inference on user-defined probabilistic logic programs. "`cplint` on SWISH" is available at `http://cplint.lamping.unife.it`. It is based on SWISH [20], a web framework for LP using features and packages of SWI-Prolog and its Pengines library. SWISH allows the user to write a logic program and ask a query over it. The query and the program are sent to the server using Javascript. The server then builds a Pengine (Prolog Engine) that evaluates the query and returns answers for it. Both the web server and the inference back-end are run entirely within SWI-Prolog.

Reasoning in "`cplint` on SWISH" is accomplished by PITA, which has ported to SWI-Prolog and made thread-safe. We also modified SWISH both

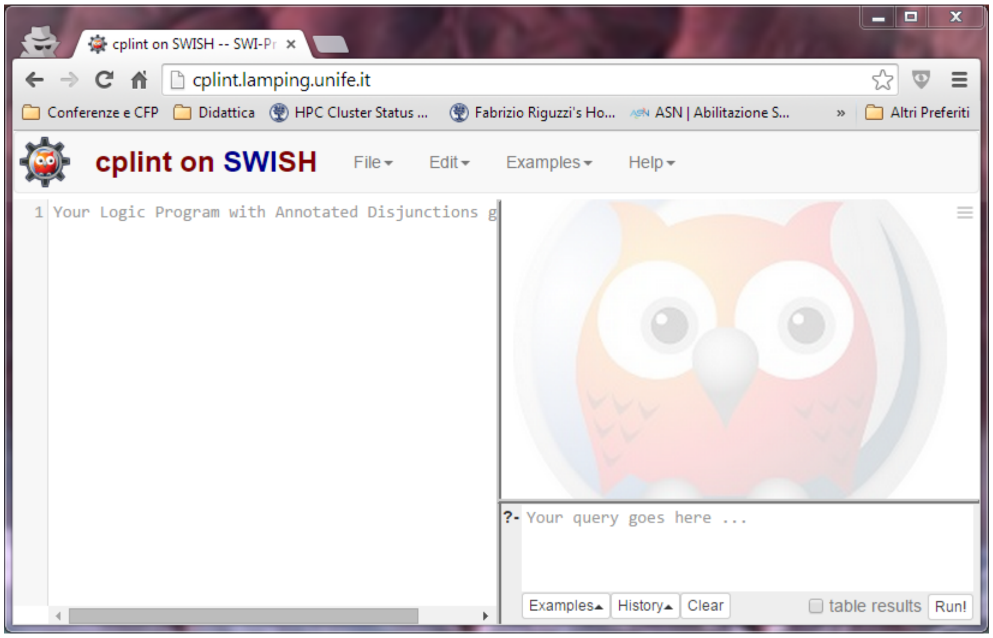


Figure 1: `cplint` interface on SWISH.

in its server and client parts. The interface of "`cplint` on SWISH" is shown in Figure 1 and offers several LPADs and query examples from different domains.

"`cplint` on SWISH" offers similar features as ProbLog2 [21], the current inference and learning engine for the PLP language ProbLog. The ProbLog2 web application allows the user to write the ProbLog program in a text area and press a button to run inference or learning. ProbLog2 exploits JavaScript for sending the program to the server in JSON format. The server is a PHP application that runs the system on the server machine and returns the result as a JSON object.

While ProbLog2 uses a mix of technologies, including PHP, Python 3 and the DSHARP compiler[1], "`cplint` on SWISH" uses a software stack completely running in the Prolog interpreter.

These web interfaces to PLP allow users to experiment with PLP without the need to install a system, a procedure which is often complex, error prone and limited mainly to the Linux platform. In this way, the aim is to reach

[1]`http://www.haz.ca/research/dsharp/`

out to a wider audience and popularize PLP, similarly to what is done for the functional probabilistic language Church [22], which is equipped with the `webchurch`[2] system for compiling Church programs into Javascript.

The paper is organized as follows. Section 2 provides an overview on the distribution semantics, Section 3 describes the PITA algorithm for reasoning over LPADs while Section 4 illustrates how PITA has been ported to SWI-Prolog. Section 5 describes the SWISH web platform and Section 6 the integration of `cplint` in it. Finally, Section 7 shows some examples available in the web application and Section 8 concludes the paper.

2. Probabilistic Logic Programming

The distribution semantics [2] is reaching a growing importance in PLP. In the distribution semantics, a probabilistic logic program defines a probability distribution over a set of normal logic programs (called *worlds*). The distribution is extended to a joint distribution over worlds and a ground query and the probability that the query is true is obtained from this distribution by marginalization.

The languages based on the distribution semantics differ in the way they define the distribution over logic programs. Each language allows probabilistic choices among atoms in clauses. Of these languages, LPADs and CP-Logic offer the most general syntax and we mainly concentrate on them here. They differ in that CP-Logic deems invalid some programs to which a causal meaning can not be attached but they are otherwise the same. Here we consider LPADs, which are sets of disjunctive clauses in which each atom in the head is annotated with a probability. We will briefly mention ProbLog and PRISM at the end of the section.

Formally a *Logic Program with Annotated Disjunctions* [7] consists of a finite set of annotated disjunctive clauses. An annotated disjunctive clause C_i is of the form

$$h_{i1} : \Pi_{i1}; \ldots ; h_{in_i} : \Pi_{in_i} \text{ :- } b_{i1}, \ldots , b_{im_i}.$$

In such a clause the semicolon stands for disjunction, $h_{i1}, \ldots h_{in_i}$ are logical atoms and $b_{i1}, \ldots , b_{im_i}$ are logical literals, $\Pi_{i1}, \ldots , \Pi_{in_i}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. Note that, if $n_i = 1$ and $\Pi_{i1} = 1$,

[2]`https://github.com/probmods/webchurch`

the clause corresponds to a non-disjunctive clause. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$, the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} \Pi_{ik}$. We denote by $ground(T)$ the grounding of an LPAD T, i.e., the results of replacing variables with constants in T.

Example 1. *Consider this LPAD from [23] that is inspired by the morphological characteristics of the Stromboli Italian island:*

$C_1 = eruption : 0.6; earthquake : 0.3$:- $sudden\_energy\_release, fault\_rupture(X).$
$C_2 = sudden\_energy\_release : 0.7.$
$C_3 = fault\_rupture(southwest\_northeast).$
$C_4 = fault\_rupture(east\_west).$

The Stromboli island is located at the intersection of two geological faults, one in the southwest-northeast direction, the other in the east-west direction, and contains one of the three volcanoes that are active in Italy. This program models the possibility that an eruption or an earthquake occurs at Stromboli. If there is a sudden energy release under the island and there is a fault rupture, then there can be an eruption of the volcano on the island with probability 0.6 or an earthquake in the area with probability 0.3. The energy release occurs with probability 0.7 and we are sure that ruptures occur in both faults.

We now present the distribution semantics for the case in which the program does not contain function symbols so that its Herbrand base is finite[3].

An *atomic choice* is a selection of the k-th atom for a grounding $C_i\theta_j$ of a probabilistic clause C_i and is represented by the triple (C_i, θ_j, k), where θ_j is a grounding substitution (a set of couples $Var/constant$ grounding C_i) and $k \in \{1, \ldots, n_i\}$. An atomic choice represents an equation of the form $X_{ij} = k$ where X_{ij} is a random variable associated with $C_i\theta_j$. A set of atomic choices κ is *consistent* if $(C_i, \theta_j, k) \in \kappa, (C_i, \theta_j, m) \in \kappa$ implies $k = m$, i.e., only one head is selected for a ground clause.

A *composite choice* κ is a consistent set of atomic choices. The probability of a composite choice κ is $P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik}$. A *selection* σ is a total composite choice (one atomic choice for every grounding of each probabilistic clause). Let us call S_T the set of all selections. A selection σ identifies a logic program w_σ called a *world*. The probability of w_σ is $P(w_\sigma) = P(\sigma) =$

[3]For the distribution semantics with function symbols see [2, 12, 17].

$\prod_{(C_i,\theta_j,k)\in\sigma}\Pi_{ik}$. Since the program does not contain function symbols, the set of worlds $W_T = \{w_1, \ldots, w_m\}$ is finite and $P(w)$ is a distribution over worlds: $\sum_{w\in W_T} P(w) = 1$.

The conditional probability of a query Q given a world w can be defined as: $P(Q|w) = 1$ if Q is true in w and 0 otherwise. We can obtain the probability of the query by marginalizing over the query:

$$P(Q) = \sum_{w} P(Q, w) = \sum_{w} P(Q|w)P(w) = \sum_{w\models Q} P(w) \tag{1}$$

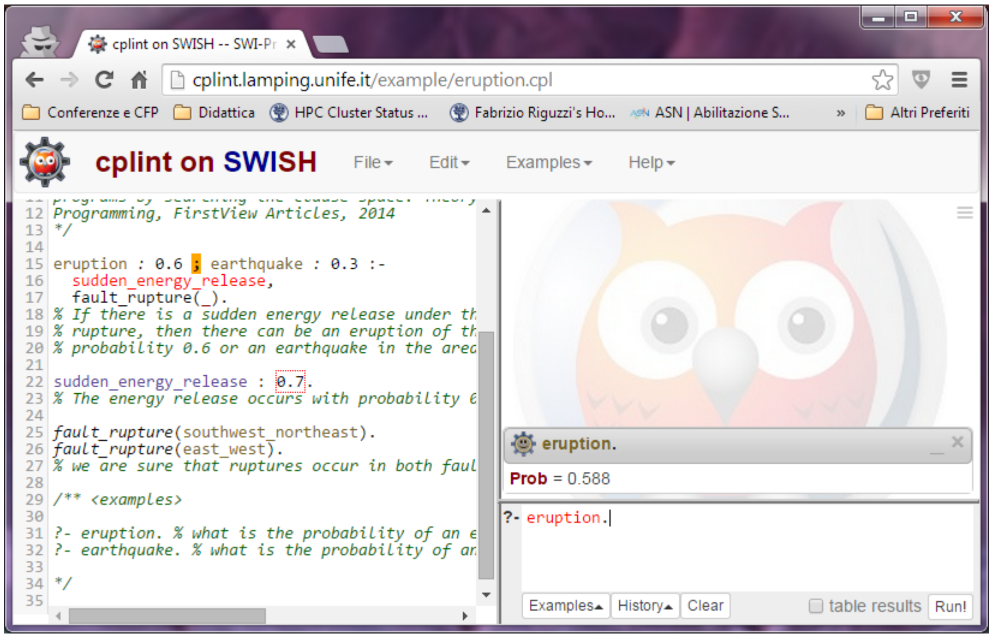
Example 2. *For the LPAD T of Example 1, clause C_1 has two groundings, $C_1\theta_1$ with $\theta_1 = \{X/southwest\_northeast\}$ and $C_1\theta_2$ with $\theta_2 = \{X/east\_west\}$, while clause C_2 has a single grounding $C_2\emptyset$. Since C_1 has three head atoms and C_2 two, T has $3 \times 3 \times 2$ worlds. The query eruption is true in 5 of them and its probability is $P(eruption) = 0.6 \cdot 0.6 \cdot 0.7 + 0.6 \cdot 0.3 \cdot 0.7 + 0.6 \cdot 0.1 \cdot 0.7 + 0.3 \cdot 0.6 \cdot 0.7 + 0.1 \cdot 0.6 \cdot 0.7 = 0.588$.*

Figure 2 shows the computation of the probability of eruption in "`cplint` on SWISH".

Inference in probabilistic logic programming is performed by finding a covering set of explanations for queries.

A composite choice κ *identifies* a set ω_κ that contains all the worlds associated with a selection that is a superset of κ: i.e., $\omega_\kappa = \{w_\sigma | \sigma \in S_T, \sigma \supseteq \kappa\}$. We define the set of worlds *identified* by a set of composite choices K as $\omega_K = \bigcup_{\kappa\in K} \omega_\kappa$. Given a ground atom Q, a composite choice κ is an *explanation* for Q if Q is true in every world of ω_κ. For example, the composite choice $\kappa_1 = \{(C_2, \emptyset, 1), (C_1, \{X/southwest\_northeast\}, 1)\}$ is an explanation for *eruption* in Example 1. A set of composite choices K is *covering* with respect to Q if every world w_σ in which Q is true is such that $w_\sigma \in \omega_K$. In Example 1, a covering set of explanations for *eruption* is $K = \{\kappa_1, \kappa_2\}$ where $\kappa_1 = \{(C_2, \emptyset, 1), (C_1, \{X/southwes\_northeast\}, 1)\}$ and $\kappa_2 = \{(C_2, \emptyset, 1), (C_1, \{X/east\_west\}, 1)\}$.

Given a covering set of explanations for a query, we can obtain a Boolean formula in Disjunctive Normal Form (DNF) where we replace each atomic choice of the form (C_i, θ_j, k) with the equation $X_{ij} = k$, we replace an explanation with the conjuction of the equations of its atomic choices and the set of explanations with the disjunction of the formulas for explanations. If we consider a world as the specification of a truth value for each equation $X_{ij} = k$, the formula evaluates to true exactly on the worlds where



```
programs by searching the clause space. Theory
Programming, FirstView Articles, 2014
*/

eruption : 0.6 ; earthquake : 0.3 :-
  sudden_energy_release,
  fault_rupture(_).
% If there is a sudden energy release under th
% rupture, then there can be an eruption of th
% probability 0.6 or an earthquake in the area

sudden_energy_release : 0.7.
% The energy release occurs with probability 0

fault_rupture(southwest_northeast).
fault_rupture(east_west).
% we are sure that ruptures occur in both faul

/** <examples>

?- eruption. % what is the probability of an e
?- earthquake. % what is the probability of ar

*/
```

```
eruption.
Prob = 0.588
```

```
?- eruption.
```

Figure 2: "`cplint` on SWISH" running on Example 1.

the query is true [12]. In Example 1, if we associate the variables X_{11} to $C_1\{X/southwest\_northeast\}$, X_{12} to $C_1\{X/east\_west\}$ and X_{21} to $C_2\emptyset$, the query is true if the following formula is true:

$$f(\mathbf{X}) = (X_{21} = 1 \wedge X_{11} = 1) \vee (X_{21} = 1 \wedge X_{12} = 1). \tag{2}$$

Since the disjuncts in the formula are not necessarily mutually exclusive, the probability of the query can not be computed by a summation as in Formula (1). The problem of computing the probability of a Boolean formula in DNF, known as *disjoint sum*, is #P-complete [24]. One of the most effective ways to date of solving the problem makes use of Decision Diagrams.

Since the random variables that are associated with atomic choices can assume multiple values, we need to use multi-valued Decision Diagrams (MDDs) [25]. An MDD represents a function $f(\mathbf{X})$ taking Boolean values on a set of multi-valued variables $\mathbf{X}$ by means of a rooted graph that has one level for each variable. Each node n has one child for each possible value of the multi-valued variable associated with n. The leaves store either 0 or 1. Since MDDs split paths on the basis of the values of a variable, the branches are mutually exclusive so a dynamic programming algorithm [9] can be applied for computing the probability. Figure 3(a) shows the MDD corresponding to Formula (2).

Most packages for the manipulation of decision diagrams are however restricted to work on Binary Decision Diagrams (BDD), i.e., decision diagrams where all the variables are Boolean. These packages offer Boolean operators between BDDs and apply simplification rules to the results of operations in order to reduce as much as possible the size of the BDD, obtaining a reduced BDD.

A node n in a BDD has two children: the 1-child and the 0-child. When drawing BDDs, rather than using edge labels, the 0-branch, the one going to the 0-child, is distinguished from the 1-branch by drawing it with a dashed line.

To work on MDDs with a BDD package we must represent multi-valued variables by means of binary variables. The following encoding, used in [26], gives good performances. For a multi-valued variable X_{ij}, corresponding to a ground clause $C_i\theta_j$, having n_i values, we use $n_i - 1$ Boolean variables $X_{ij1}, \ldots, X_{ijn_i-1}$ and we represent the equation $X_{ij} = k$ for $k = 1, \ldots n_i - 1$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, and the equation $X_{ij} = n_i$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijn_i-1}}$. The BDD corresponding to the MDD of Figure 3(a) is shown in Figure 3(b). BDDs obtained in this way can be used as well for computing the probability of queries by associating to each Boolean variable X_{ijk} a parameter π_{ik} that represents $P(X_{ijk} = 1)$. The parameters are obtained from those of multi-valued variables in this way: $\pi_{i1} = \Pi_{i1}, \ldots \pi_{ik} = \frac{\Pi_{ik}}{\prod_{j=1}^{k-1}(1-\pi_{ij})}, \ldots,$ up to $k = n_i - 1$.

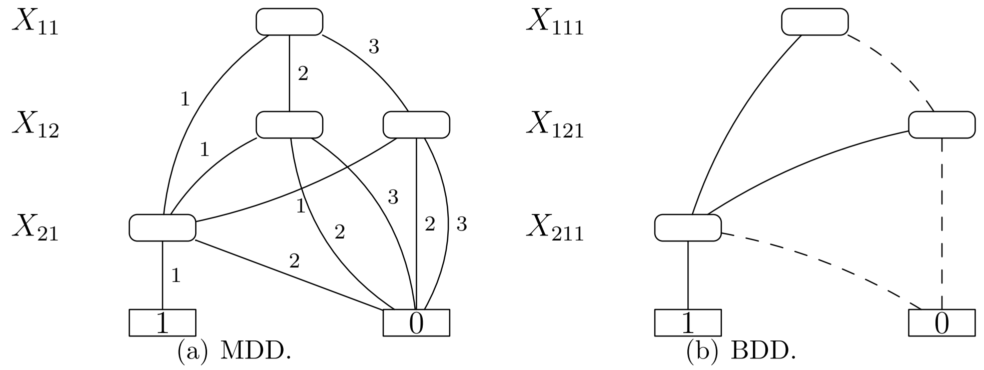


Figure 3: Decision diagrams for Example 1.

ProbLog [9] programs differs from LPADs as they allow probabilistic clauses only in the form of facts with two alternatives, one of which is implicit.

PRISM [5], similarly to ProbLog, allows probabilities only on facts but the alternatives can be more than two. In particular, PRISM offers the special predicate $msw(switch, value)$ that encodes a random switch (i.e., a random variable) and that can be used in the body of clauses to check that the random switch named *switch* takes the value named *value*. The possible values of each switch are defined by facts for the $values/2$ predicate, while the probability of each value is set by calling the predicate $set\_sw/2$.

3. The PITA System

PITA computes the probability of a query from a probabilistic program in the form of an LPAD by first transforming the LPAD into a normal program containing calls for manipulating BDDs. The idea is to add an extra argument to each subgoal to store a BDD encoding the explanations for the answers of the subgoal. The values of the subgoals' extra argument are combined using a set of general library functions:

- `init, end`: initialize and terminate the data structures for manipulating BDDs;
- `zero(-D), one(-D)`: return BDDs representing the Boolean constant 0 and 1;
- `and(+D1,+D2,-DO), or(+D1,+D2,-DO), not(+D1,-DO)`: Boolean operations between BDDs;
- `equality(+Var,+Value,-D)`: `D` is the BDD representing `Var=Value`, i.e. that the multi-valued random variable `Var` is assigned `Value`;
- `ret_prob(+D,-P)`: returns the probability of the BDD `D`.

These functions are implemented in C as an interface to the CUDD[4] library for manipulating BDDs. A BDD is represented in Prolog as an integer that is a pointer in memory to the root node of the BDD. Moreover, the predicate `get_var_n(+R,+S,+Probs,-Var)` is implemented in Prolog and returns the

[4]`http://vlsi.colorado.edu/~fabio/CUDD/`

multi-valued random variable associated with rule `R` with grounding substitution `S` and list of probabilities `Probs`.

The PITA transformation applies to atoms, literals and clauses. The transformation for an atom `h` and a variable `D`, `PITA(h,D)`, is `h` with the variable `D` added as the last argument. The transformation for a negative literal `b = \+ a` and a variable `D`, `PITA(b,D)`, is the Prolog conditional

```
(PITA(a,DN)->
  not(DN,D)
;
  one(D)
).
```

In other words, the data structure `DN` is negated if `a` has some explanations; otherwise the data structure for the constant function 1 is returned.

The disjunctive clause

```
Cr = h1:p1 ; ... ; hn:pn :- b1,...,bm.
```

where the parameters sum to 1, is transformed into the set of clauses `PITA(Cr)`:

```
PITA(Cr,i)=PITA(hi,D):- one(DD0),
  PITA(b1,D1),and(DD0,D1,DD1),....,
  PITA(bm,Dm),and(DDm-1,Dm,DDm),
  get_var_n(r,V,[p1,...,pn],Var),
  equality(Var,i,DD),and(DDm,DD,D).
```

for `i=1,...,n`, where `V` is a list containing all the variables appearing in `Cr`. If the parameters do not sum up to 1, then `n-1` rules are generated as the last head atom, `null`, does not influence the query since it does not appear in any body. In the case of empty bodies or non-disjunctive clauses (a single head with probability 1), the transformation can be optimized.

The PITA transformation applied to Example 1 yields

```
PITA(C1,1)=eruption(D) :-
  one(DD0),sudden_energy_release(D1),and(DD0,D1,DD1),
  fault_rupture(X,D2),and(DD1,D2,DD2),
  get_var_n(1,[X],[0.6,0.3,0.1],Var),
  equality(Var,1,DD),and(DD2,DD,D).
PITA(C1,2)=earthquake(D) :-
```

```
  one(DD0),sudden_energy_release(D1),and(DD0,D1,DD1),
  fault_rupture(X,D2),and(DD1,D2,DD2),
  get_var_n(1,[X],[0.6,0.3,0.1],Var),
  equality(Var,2,DD),and(DD2,DD,D).
PITA(C2,1)=sudden_energy_release(D) :- one(DD0),
  get_var_n(2,[],[0.7,0.3],Var),
  equality(Var,1,DD),and(DD0,DD,D).
PITA(C3,1)=fault_rupture(southwest_northeast,D) :- one(D).
PITA(C4,1)=fault_rupture(east_west,D) :- one(D).
```

Clause C_1 has three alternatives in the head but the last one is the `null` atom so only two clauses are generated. Clauses C_3 and C_4 are definite facts so their transformation is optimized as shown above.

PITA uses tabling, a logic programming technique that reduces computation time and ensures termination for a large class of programs [27]. The idea of tabling is simple: keep a store of the subgoals encountered in a derivation together with answers to these subgoals. If one of the subgoals is encountered again, its answers are retrieved from the store rather than recomputing them. Besides saving time, tabling ensures termination for programs without function symbols under the Well-Founded Semantics (WFS) [28].

PITA was originally developed for XSB Prolog[5]. It exploits its *answer subsumption* feature [27] that, when a new answer for a tabled subgoal is found, combines old answers with the new one according to a partial order or lattice. For example, if the lattice is on the second argument of a binary predicate `p`, answer subsumption may be specified by means of the declaration

```
:-table p(_, or/3 - zero/1).
```

where `zero/1` is the bottom element of the lattice and `or/3` is the join operation of the lattice. Thus if a table has an answer `p(a,d1)` and a new answer `p(a,d2)` is derived, the answer `p(a,d1)` is replaced by `p(a,d3)`, where `d3` is obtained by calling `or(d1,d2,d3)`.

In PITA, various predicates should be declared as tabled. For a predicate `p/n`, the declaration is

```
:-table p(_1,...,_n, or/3 - zero/1).
```

[5]`http://xsb.sourceforge.net/`

which indicates that answer subsumption is used to form the disjunction of BDDs in the last argument.

At a minimum, the predicate of the goal and all the predicates appearing in negative literals should be tabled with answer subsumption. If these predicates are not tabled with answer subsumption, PITA is not correct as it does not collect all answers for the subgoals of these predicates. It is usually useful to table every predicate whose answers have multiple explanations and are going to be reused often since in this way repeated computations are avoided.

4. PITA in `cplint`

`cplint` is a suite of programs for reasoning on LPADs. It includes various algorithms for inference and learning. It is available for Yap Prolog [29] and SWI-Prolog[6] [30]. In this section we describe how PITA was ported to SWI-Prolog, the basis of the "`cplint` on SWISH" web application.

Since SWI-Prolog has no tabling facilities, and thus also no answer subsumption, in order to port PITA to SWI-Prolog we used `bagof/3` to collect all the BDDs for the query and then compute the disjunction of all of them using `or/3`. So the disjunction of different explanations is not computed for each subgoal but only for the top level query. This does not pose any problem except for negated goals: for them, in fact, we can't collect a BDD encoding an explanation and negate it for each explanation but we need to collect all the explanations, disjoin them and negate the result.

Thus the transformation for a negative literal `b = \+ a` and a variable `D`, `PITA(b,D)`, is the Prolog conditional

```
(bagof(B,PITA(a,B),L)->
  or_list(L,DL),
  not(DL,D)
;
  one(D)
)
```

where `or_list(L,D)` returns in `D` the disjunction of all the BDDs in the list `L`.

[6]`http://www.swi-prolog.org/`

In order to compute the probability of an atom `Q`, the predicate `s(+Q,-P)` returns in `P` the probability of `Q`. `s/2` is implemented as:

```
s(Q,P) :-
  init,
  (bagof(D,PITA(Q,D),L) ->
    or_list(L,B)
  ;
    zero(B)
  ),
  ret_prob(B,P),
  end.
```

The query `Q` passed to `s/2` does not need to be ground. However, if it is not, it is grounded and the corresponding probability is returned. At the moment it is not possible to ask for other instantiations of the query. Moreover, the query `Q` must be an atom, it can't be a conjunction. We plan to remove these limitations in future versions.

`cplint` can be installed by the user with the goal `pack_install(cplint)` at the SWI-Prolog prompt. After this call, `pita` can be loaded with the command `use_module(library(pita))`.

5. SWISH

SWISH[7] is a web application that allows the user to write Prolog programs and ask queries through the browser. The SWISH page is divided into three panes, one with a program editor (on the left), one with a query editor (on the bottom right) and one that shows the query results (on the top right). When the user hits return after writing a query, a *runner* is created that collects the text in the program editor (if any) and the query and sends these to the server, which creates a Pengine (Prolog Engine). The Pengine compiles the program into a temporary private module. The Pengine assesses whether executing the query can compromise the system. If this fails, an error is displayed. If the query is considered safe, it executes the query and communicates with the runner about the results using JSON messages.

[7] `http://swish.swi-prolog.org/`

SWISH is based on SWI-Prolog and uses its Pengines library [31], which allows to create Prolog engines from an ordinary Prolog thread, from another Pengine, or from JavaScript running in a web client.

A Pengine is composed of a Prolog thread, a dynamic clause database (private to the Pengine), a message queue for incoming requests, and a message queue for outgoing responses.

Pengines follow a master/slave architecture in which the master creates a Pengine on the slave and posts a query to it. The conversations between the master and the slave follows a communication protocol called the Prolog Transport Protocol (PLTP) that is layered on top of HTTP.

We now show an example from [31]: we use `pengine_create/1` to create a slave Pengine in a remote Pengine server.

```
:- use_module(library(pengines)).
main :-
    pengine_create([
        server('http://pengines.org'),
        src_text("
            q(X) :- p(X).
            p(a). p(b). p(c).
        ")
    ]),
    pengine_event_loop(handle, []).

handle(create(ID, _)) :-
    pengine_ask(ID, q(X), []).
handle(success(ID, [X], false)) :-
    writeln(X).
handle(success(ID, [X], true)) :-
    writeln(X),
    pengine_next(ID, []).
```

The option `src_text` is used to send the program to be queried in textual form to the Pengine. `pengine_event_loop/2` is used to start an event loop that listens for event terms and calls `handle/1` on them. If the event term is `create(ID,_)`, it means that the Pengine with id `ID` has been created and the event handler uses `pengine_ask/3` to ask a query. Predicate `pengine_ask/3` is deterministic, the results of the query will be returned in the form of event terms.

If the event term is of the form `success(ID, Query, More)`, `ID` is the Pengine's id that succeeded in solving the query, `Query` holds an instantiation of the query and `More` is either `true` or `false`, indicating whether we can expect the Pengine to be able to return more solutions or not, in case we call `pengine_next/2`. If `More` is true, `handle/1` calls `pengine_next/2` to get the following solution. Thus running `main/0` will write the terms `q(a)`, `q(b)` and `q(c)` to standard output.

Code sent to Pengines is executed in a "sandboxed" environment that ensures that only predicates that do not have side effects such as accessing the file system, loading foreign extensions, defining other predicates outside the sandbox environment, etc., are called. Goals' safety is validated using a call to `safe_goal/1` of `library(sandbox)` prior to execution.

SWI-Prolog also offers a JavaScript library `pengine.js` that allows the creation of Pengine JavaScript objects. These, in turn, create Pengine objects on the server that can be queried from JavaScript.

The SWISH web server is implemented by the SWI-Prolog HTTP package, a series of libraries for serving data on HTTP [32].

SWISH exploits TogetherJS[8] in order to make the development of the code collaborative. TogetherJS is an open source JavaScript library built and hosted by Mozilla. This library permits a real time interaction between users and offers different built-in features:

Audio and Text Chat The collaborators can chat by talking or texting to each other.

User Focus The collaborators see each other's mouse cursors and clicks.

Co-browsing The collaborators can follow each other to different pages on the same domain.

Real time content sync The content is synchronized between all the collaborators.

It is possible to start collaborating on SWISH by clicking the item "File" in the menu bar and then clicking on "Collaborate..". The TogetherJS dock will appear and you can invite another user by sharing the generated link.

[8]https://togetherjs.com/

6. "cplint on SWISH"

In order to implement "cplint on SWISH", we had to modify the foreign language C library of PITA in order to allow different threads to use it at the same time. In fact, the library makes use of static global variables that hold data structures for the CUDD manager and the association between the random variables and CUDD variables. If two different threads use the library, there would be a conflict on these variables. Therefore, we added an extra argument Environment to all the library predicates. This argument holds the data structures and allows the computation of the probability by different threads.

So init(-Env), when called, returns a pointer to a data structure storing the environment that must be given as input to all the other predicates:

- zero(+Env,-D), one(+Env,-D)
- and(+Env,+D1,+D2,-DO), or(+Env,+D1,+D2,-DO), not(+Env,+D1, -DO)
- equality(+Env,+Var,+Value,-D)
- ret_prob(+Env,+D,-P)
- end(+Env)

As a consequence, the PITA transformation function must take as input the variable Env that stores the environment.

To allow Pengines to execute the pita library predicates, they are declared as safe by the code:

```
:- multifile sandbox:safe_primitive/1.

sandbox:safe_primitive(pita:init).
sandbox:safe_primitive(pita:ret_prob(_,_)).
...
```

in the pita module file.

The PITA library was also modified with respect to the application of the transformation. While PITA uses a predicate load/1 that loads the program file and applies the transformation to it, we decided to use term expansion through the predicate term_expansion/2, a de-facto standard in Prolog for source-to-source transformations. When compiling a module, SWI-Prolog will consider each term `T` in the program one by one and apply `term_expansion(T,NewT)`, then it will compile `NewT` instead of `T`. So if the user provides clauses for the `term_expansion/2` predicate, the system will compile a modified version of the input.

After loading `pita` with `use_module(library(pita))`, the PITA predicate `set/2` must be used to set the PITA flag `compiling` to `on`. All the clauses for `term_expansion/2` check this flag before performing the transformation. If it is not set to `on`, the transformation is not applied. After setting `compiling` to `on`, consulting a file containing an LPAD performs its translation into Prolog and loads the result in memory.

Similarly, if a file containing an LPAD includes the directives `:-use_module (library(pita)).` and `:-set(compiling,on).` at the beginning, when it is consulted from the top level it is transformed and loaded in memory.

"`cplint` on SWISH" has the interface shown in Figure 1. It allows the user to write an LPAD in the left pane and write a query in the bottom right pane. When the user presses enter at the end of the query or presses the Run! button, a Pengine is created with the program. This is done by the `runner.js` JavaScript file that creates a new Pengine object. The creation of the object was modified by adding to the program source some directives for loading the `pita` library, for disabling the check for discontiguous clauses and for enabling compilation. This is done by the following snippet of `runner.js`:

```
data.prolog = new Pengine({
  ...
  src: ":-use_module(library(pita)).
    :-style_check(-discontiguous).
    :-set(compiling,on). "
    + query.source,
  ...
  oncreate: handleCreate,
  ...
  });
```

that stores a new Pengine object in the runner's `data.prolog` attribute. `query.source` holds the program text.

The `handleCreate` function is performed at the creation of the Pengine and was modified to allow the computation of the probability. The query given by the user is sent to the Pengine with the `ask` method in a transformed form: the query atom without the full stop is inserted into a call to `s/2` in this way:

```
function handleCreate() {
  var elem = this.pengine.options.runner;
  var data = elem.data('prologRunner');
  this.pengine.ask("s("+termNoFullStop(data.query.query)+",Prob)");
  elem.prologRunner('setState', "running");
}
```

Here `data.query.query` is a string containing the query. The top right pane will then show the value of `Prob` variable, together with the other variables' values possibly present in the query. Figure 2 shows Example 1 code together with the query `eruption` and its result.

7. Examples

"`cplint` on SWISH" contains a number of examples, accessible from the "Example" menu. The complete list of available examples is in the Appendix. Below we describe two examples that, to our knowledge, have not been encoded yet in LPADs.

The first, the *Monty Hall puzzle* [33], models the TV game show hosted by Monty Hall in which a player has to choose which of three closed doors to open. Behind one door there's a prize while behind the other two there is nothing. Once the player has selected the door, Monty Hall opens one of the remaining closed doors which does not contains the prize, and then he asks the player if he would like to change his door with the other closed door or not. The problem of this game is to determine whether the player should switch. The prize is behind one of the three doors with equal probability:

```
prize(1):1/3; prize(2):1/3; prize(3):1/3.
```

The player has selected door 1

```
selected(1).
```

The following clauses model the choice of which door Monty Hall will open after the player's decision:

```
open_door(A):0.5; open_door(B):0.5 :-
  member(A,[1,2,3]),
  member(B,[1,2,3]),
  A<B,
  \+ prize(A),
  \+ prize(B),
  \+ selected(A),
  \+ selected(B).

open_door(A) :-
  member(A,[1,2,3]),
  \+ prize(A),
  \+ selected(A),
  member(B,[1,2,3]),
  prize(B),
  \+ selected(B).
```

In case the player keeps its choice, he wins if he has selected a door with the prize behind:

```
win_keep :-
  selected(A),
  prize(A).
```

In case the player switches, he wins if the prize is behind a door that he has not selected and that Monty Hall has not opened:

```
win_switch :-
  member(A,[1,2,3]),
  \+ selected(A),
  prize(A),
  \+ open_door(A).
```

Querying `win_keep` and `win_switch` we obtain 1/3 and 2/3 respectively, so the player should switch.

The second example is the *three-prisoners puzzle* following [34]:

> Of three prisoners a, b, and c, two are to be executed, but a does not know which. Thus, a thinks that the probability that i will

be executed is 2/3 for $i \in \{a, b, c\}$. He says to the jailer, “Since either b or c is certainly going to be executed, you will give me no information about my own chances if you give me the name of one man, either b or c, who is going to be executed.” But then, no matter what the jailer says, naive conditioning leads a to believe that his chance of execution went down from 2/3 to 1/2.

Each prisoner is safe with probability 1/3:

```
safe(a):1/3; safe(b):1/3; safe(c):1/3.
```

If a is safe, the jailer tells that one of the other prisoners will be executed uniformly at random:

```
tell_executed(b):1/2; tell_executed(c):1/2 :-
  safe(a).
```

Otherwise, he tells that the only unsafe prisoner will be executed:

```
tell_executed(A) :-
  member(A,[b,c]),
  member(B,[b,c]),
  A\=B,
  safe(B).
```

The jailer speaks if he tells that somebody will be executed:

```
tell:-
  tell_executed(_).
```

a is safe after the jailer utterance if he is safe and the jailer speaks:

```
safe_after_tell :-
  safe(a),
  tell.
```

By computing the probability of `safe(a)` and `safe_after_tell` we get the same probability of 1/3. We can see this also by considering conditional probabilities: the probability of `safe(a)` given the jailer utterance `tell` is

$$P(\texttt{safe(a)}|\texttt{tell}) = \frac{P(\texttt{safe(a)}, \texttt{tell})}{P(\texttt{tell})} = \frac{P(\texttt{safe\_after\_tell})}{\texttt{P(tell)}} = \frac{1/3}{1} = 1/3$$

because the probability of `tell` is 1.

8. Conclusions

Web-based systems are, today, the way to reach out to a wider audience. In order to popularize Probabilistic Logic Programming, we have implemented the web application "`cplint` on SWISH" that allows the user to easily write a PLP program and compute the probability of queries with just a web browser. "`cplint` on SWISH" already includes a number of examples that cover a wide range of domains and provide interesting applications of PLP. "`cplint` on SWISH" has been implemented by exploiting the features of the system SWISH for Prolog programming and querying on the Web, and by porting the PITA system for inference on LPAD from its original XSB implementation to SWI-Prolog.

In the future, we plan to allow the user to pose conjunctive and conditional queries. Moreover, we plan to extend "`cplint` on SWISH" with other inference procedures and with learning algorithms.

References

[1] L. De Raedt, P. Frasconi, K. Kersting, S. Muggleton (Eds.), Probabilistic Inductive Logic Programming - Theory and Applications, Vol. 4911 of LNCS, Springer, Berlin, 2008.

[2] T. Sato, A statistical learning method for logic programs with distribution semantics, in: Proceedings of ICLP, MIT Press, Cambridge, MA, Tokyo, 1995, pp. 715–729.

[3] E. Dantsin, Probabilistic logic programs and their semantics, in: Logic Programming: Proceedings of the First and Second Russian Conference on Logic Programming, Vol. 592 of LNCS, Springer, Berlin, Irkutsk and St. Petersburg, Russia, 14-18 and 11-16 September, 1991, pp. 152–164.

[4] D. Poole, The independent choice logic for modelling multiple agents under uncertainty, Artif. Intell. 94 (1-2) (1997) 7–56.

[5] T. Sato, Y. Kameya, PRISM: A language for symbolic-statistical modeling, in: Proceedings of IJCAI, Morgan Kaufmann, Burlington, MA, Nagoya, Japan, 1997, pp. 1330–1339.

[6] N. Fuhr, Probabilistic Datalog: Implementing logical information retrieval for advanced applications, J. Am. Soc. Inf. Sci. 51 (2) (2000) 95–110.

[7] J. Vennekens, S. Verbaeten, M. Bruynooghe, Logic programs with annotated disjunctions, in: Proceedings of ICLP, Vol. 3131 of LNCS, Springer, Berlin, Saint-Malo, France, 2004, pp. 195–209.

[8] J. Vennekens, M. Denecker, M. Bruynooghe, CP-logic: A language of causal probabilistic events and its relation to logic programming, Theory Pract. Log. Program. 9 (3) (2009) 245–308.

[9] L. De Raedt, A. Kimmig, H. Toivonen, ProbLog: A probabilistic Prolog and its application in link discovery., in: Proceedings of IJCAI, IJCAI/AAAI, Palo Alto, CA, Hyderabad, India, 2007, pp. 2462–2467.

[10] J. Vennekens, S. Verbaeten, Logic programs with annotated disjunctions, Tech. Rep. CW386, K. U. Leuven, Leuven, Belgium (2003).

[11] L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, J. Vennekens, Towards digesting the alphabet-soup of statistical relational learning, in: Proceedings of the NIPS Workshop on Probabilistic Programming, Whistler, Canada, 2008, pp. 1–3.
URL `http://probabilistic-programming.org/wiki/NIPS*2008_Workshop/Schedule#talk-deraedt`

[12] D. Poole, Abducing through negation as failure: stable models within the Independent Choice Logic, J. Log. Program. 44 (1-3) (2000) 5–35.

[13] A. Kimmig, B. Demoen, L. D. Raedt, V. S. Costa, R. Rocha, On the implementation of the probabilistic logic programming language ProbLog, Theory Pract. Log. Program. 11 (2-3) (2011) 235–262.

[14] F. Riguzzi, Inference with logic programs with annotated disjunctions under the well founded semantics, in: Proceedings of ICLP, Vol. 5366 of LNCS, Springer, Berlin, Udine, Italy, 2008, pp. 667–771. `doi:10.1007/978-3-540-89982-2\string_54`.

[15] F. Riguzzi, SLGAD resolution for inference on Logic Programs with Annotated Disjunctions, Fundam. Inform. 102 (3-4) (2010) 429–466. `doi:10.3233/FI-2010-392`.

[16] F. Riguzzi, T. Swift, Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions, in: Technical Communications of ICLP, Vol. 7 of LIPIcs, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Edinburgh, Scotland, 2010, pp. 162–171. `doi:10.4230/LIPIcs.ICLP.2010.162`.

[17] F. Riguzzi, T. Swift, Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics, Theory Pract. Log. Program. 13 (Special Issue 02 - 25th Annual GULP Conference) (2013) 279–302. `doi:10.1017/S1471068411000664`.

[18] F. Riguzzi, A top down interpreter for LPAD and CP-logic, in: Proceedings of AI*IA, Vol. 4733 of LNAI, Springer, Berlin, Rome, Italy, 2007, pp. 109–120. `doi:10.1007/978-3-540-74782-6\string_11`.

[19] F. Riguzzi, Extended semantics and inference for the Independent Choice Logic, Log. J. IGPL 17 (6) (2009) 589–629. `doi:10.1093/jigpal/jzp025`.

[20] Swish, `http://pengines.swi-prolog.org/apps/swish/index.html`.

[21] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt, Inference and learning in probabilistic logic programs using weighted boolean formulas, Theory and Practice of Logic Programming 15 (2015) 358–401. `doi:10.1017/S1471068414000076`.
URL `http://arxiv.org/abs/1304.6810`

[22] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, J. B. Tenenbaum, Church: a language for generative models, in: D. A. McAllester, P. Myllymäki (Eds.), UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008, AUAI Press, 2008, pp. 220–229.
URL `http://uai.sis.pitt.edu/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24`

[23] F. Riguzzi, N. Di Mauro, Applying the information bottleneck to statistical relational learning, Mach. Learn. 86 (1) (2012) 89–114. `doi:10.1007/s10994-011-5247-6`.

[24] L. G. Valiant, The complexity of enumeration and reliability problems, SIAM J. Comp. 8 (3) (1979) 410–421.

[25] A. Thayse, M. Davio, J. P. Deschamps, Optimization of multivalued decision algorithms, in: Proceedings of MVL, IEEE Computer Society Press, Los Alamitos, CA,, Rosemont, IL, 1978, pp. 171–178.

[26] T. Sang, P. Beame, H. A. Kautz, Performing bayesian inference by weighted model counting, in: Proceedings of AAAI, AAAI Press / The MIT Press, Palo Alto, CA, Pittsburgh, PA, 2005, pp. 475–482.

[27] T. Swift, D. S. Warren, XSB: extending prolog with tabled logic programming, TPLP 12 (1-2) (2012) 157–187. doi:10.1017/S1471068411000500.

[28] A. Van Gelder, K. A. Ross, J. S. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (3) (1991) 620–650.

[29] Yap Prolog, www.dcc.fc.up.pt/~vsc/Yap/.

[30] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theory and Practice of Logic Programming 12 (1-2) (2012) 67–96.

[31] T. Lager, J. Wielemaker, Pengines: Web logic programming made easy, TPLP 14 (4-5) (2014) 539–552. doi:10.1017/S1471068414000192.

[32] J. Wielemaker, Z. Huang, L. van der Meij, SWI-Prolog and the web, TPLP 8 (3) (2008) 363–392. doi:10.1017/S1471068407003237.

[33] C. Baral, M. Gelfond, J. N. Rushton, Probabilistic reasoning with answer sets, Theory and Practice of Logic Programming 9 (1) (2009) 57–144.

[34] P. Grünwald, J. Y. Halpern, Updating probabilities, J. Artif. Intell. Res. (JAIR) 19 (2003) 243–278. doi:10.1613/jair.1164.

[35] J. Vennekens, S. Verbaeten, M. Bruynooghe, Logic programs with annotated disjunctions, in: J. P. Delgrande, T. Schaub (Eds.), 10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6-8, 2004, Proceedings, 2004, pp. 409–415.

[36] E. Bellodi, F. Riguzzi, Expectation Maximization over binary decision diagrams for probabilistic logic programs, Intell. Data Anal. 17 (2) (2013) 343–363.

[37] E. Bellodi, F. Riguzzi, Structure learning of probabilistic logic programs by searching the clause space, Theory and Practice of Logic Programming 15 (2) (2015) 169–212. `doi:10.1017/S1471068413000689`.
URL `http://journals.cambridge.org/repo_A91VE4W3`

[38] H. Blockeel, Probabilistic logical models for mendel's experiments: An exercise, in: Inductive Logic Programming (ILP 2004), Work in Progress Track, 2004, pp. 1–5.

[39] W. Meert, J. Struyf, H. Blockeel, CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods, in: Proceedings of ILP, Vol. 5989 of LNCS, Springer, Berlin, Leuven, Belgium, 2009, pp. 96–109.

[40] H. Christiansen, J. P. Gallagher, Non-discriminating arguments and their uses, in: Proceedings of ICLP, Vol. 5649 of LNCS, Springer, Berlin, Pasadena, CA, 2009, pp. 55–69.

[41] T. Sato, K. Kubota, Viterbi training in prism, Theory and Practice of Logic Programming 15 (2) (2014) 147–168. `doi:10.1017/S1471068413000677`.

[42] P. Singla, P. Domingos, Discriminative training of Markov logic networks, in: Proceedings of AAAI/IAAI, AAAI Press/The MIT Press, Palo Alto, CA, Pittsburgh, PA, 2005, pp. 868–873.

[43] F. Riguzzi, T. Swift, The PITA system: Tabling and answer subsumption for reasoning under uncertainty, Theory Pract. Log. Program. 11 ((ICLP Special Issue)4–5) (2011) 433–449.

Appendix A. List of Examples

- Coin [35]: models the throw of a coin with uncertain fairness.
- Dice [35]: models repeated throws of a dice until the face six comes out.

- Epidemic [36]: models the development of an epidemic or a pandemic.
- Earthquake [23]: models the occurrence of an earthquake depending on its possible causes.
- Eruption [37]: Example 1.
- Mendel inheritance of the color of pea plants [38].
- Mendel inheritance of human bloodtype [39].
- Path [9]: models the probability of the connection between two nodes in a graph.
- Bayesian network [35]: encodes a simple alarm Bayesian network.
- Hidden Markov Model [40].
- Probabilistic Context Free Grammar [41]: models a parser for a PCFG grammar.
- UWCSE: link prediction [39].
- Cora: entity resolution [42].
- Monty Hall puzzle [33].
- Three-prisoners puzzle [34].
- Trigger [8]: shows the use of noisy or for modeling a russian roulette with two guns.
- Light [8]: shows the use of negation.
- Coins [35]: similar to the Coin example but with two coins.
- Three sided dice [43]: similar to the Dice example where the dice have three faces.
- Mendel inheritance of color of pea plants - larger family [38]: as the Mendel example with a larger number of individuals.