Extended Semantics and Inference for the Independent Choice Logic

Fabrizio Riguzzi, *ENDIF, Università di Ferrara, Via Saragat, 1, 44100 Ferrara, Italy.*
E-mail: fabrizio.riguzzi@unife.it

Abstract

The Independent Choice Logic (ICL), proposed by Poole, is a language for expressing probabilistic information in logic programming that adopts a distribution semantics: an ICL theory defines a distribution over a set of normal logic programs. The probability of a query is then given by the sum of the probabilities of the programs where the query is true.

The ICL semantics requires the theory to be acyclic. This is a strong limitation that rules out many interesting programs. In this paper we present an extension of the ICL semantics that allows theories to be modularly acyclic.

Inference with ICL can be performed with the Ailog2 system that computes explanations to queries and then makes them mutually incompatible by means of an iterative algorithm.

We propose the system PICL (for Probabilistic inference with ICL) that computes the explanations to queries by means of a modification of SLDNF-resolution and then makes the explanations mutually incompatible by means of Binary Decision Diagrams.

PICL and Ailog2 are compared on problems that involve computing the probability of a connection between two nodes in biological graphs and in social networks. Moreover, they are also applied to three games of dice.

The problems considered are easily expressible in P-log, a probabilistic language based on Answer Set Programming. Therefore, the Plog system was also applied to the programs.

PICL was able to handle larger problems than Ailog2 and Plog. Moreover, it was the fastest of the three algorithms except for one case of one of dice games.

Keywords: Probabilistic Logic Programming, Independent Choice Logic, Modularly acyclic programs, SLDNF-Resolution.

1 Introduction

Combining logic and probability has been studied in philosophy and artificial intelligence since the 50's [4, 10, 33, 13]. In recent times, the work on this topic has received a renewed attention due to the advances in Statistical Relational Learning [12] and Probabilistic Inductive Logic Programming [24]. Various languages have been proposed that combine relational and statistical aspects, such as Probabilistic Logic Programs [7], Independent Choice Logic [22], PRISM [31], pD [9], Bayesian Logic Programs [14], Logic Programs with Annotated Disjunctions [38], ProbLog [8] and P-log [5], to name a few.

The Independent Choice Logic (ICL) is one of the earliest languages for representing probabilistic information in logic programming: it first appeared as a language called Probabilistic Horn Abduction (PHA) [21] and then was successively developed in [22]. ICL defines probability distributions over sets of mutually exclusive facts called alternatives. A set of normal logic program is then obtained by combining an acyclic program with a selection from the grounded sets of alternatives. A distribution on the programs is defined on the basis of those specified over the alternatives. The semantics of ICL is an instance of the distribution semantics [30]: a theory defines a probability distribution over normal logic programs and the probability of a query is given by the sum of the probabilities of the programs where the query is true. Other languages that adopt a distribution semantics include Probabilistic Logic Programs [7], PRISM [31], pD [9], Logic Programs with Annotated Disjunctions [38] and ProbLog [8].

P-log [5] differs slightly from these languages because it defines a probability distribution over the stable models of the logical part of the program rather than over programs. P-log is equipped with the system Plog for reasoning that exploits an Answer Set Programming system for computing the stable models.

The paper presents two contributions. The first is an extension of the semantics of ICL for allowing modularly acyclic programs. The second contribution of the paper is a novel inference algorithm for ICL that can be applied to this larger class. The first approach for performing inference with ICL was proposed in [20]: the algorithm computes the probability of a query for PHA/ICL by finding all the explanations (i.e., set of hypotheses) for a goal and then summing up the probabilities of individual explanations. This algorithm requires the explanations to be mutually incompatible, which is guaranteed if the clauses for the same atom have mutually exclusive bodies. This restriction was lifted in [23] by making the explanations mutually exclusive using an iterative algorithm that "splits" them. The resulting algorithm is implemented in the Ailog2 system.

The algorithm proposed in this paper, called PICL for Probabilistic inference with ICL (pronounced like "pickle"), uses a modification of SLDNF-resolution for computing explanations that is sound and complete for modularly acyclic theories. The algorithm is very similar to the one actually adopted by Ailog2 that is able to deal with modularly acyclic theories as well. PICL differs from Ailog2 because explanations are made disjoint by using Binary Decision Diagrams, analogously to what is done for ProbLog programs in [8].

The resulting algorithm was applied to problems of computing the probability of paths in biological and social networks, and to games of dice. The considered problems are easily expressible in P-log so we also applied the Plog system to the datasets.

The programs describing biological and social networks are modularly acyclic but not acyclic. We tested marginal queries for all problems and conditional queries for social networks. PICL had the best performances, answering queries faster and solving more complex queries than Ailog2 and Plog, except for one case of a game of dice in which Plog was the fastest.

The paper is organized as follows. In Section 2 we present some preliminary notions on logic programming. Section 3 describes the syntax and semantics of ICL together with the existing inference algorithm. Section 4 presents the definition of a semantics for modularly acyclic ICL theories together with an algorithm for finding a covering set of explanations for queries. In Section 5 we illustrate the algorithm for making explanations incompatible by using Binary Decision Diagrams. Section 7 describes related works. In Section 8 we discuss the experiments performed for evaluating the algorithms and Section 9 concludes the paper and presents directions for future work.

2 Logic Programming Preliminaries

A *first order alphabet* Σ is a set of predicate symbols and function symbols (or functors) together with their arity. A *term* is either a variable or a functor applied to a tuple of terms of length equal to the arity of the functor. If the functor has arity 0 it is called a *constant*. An *atom* is a predicate symbol applied to a tuple of terms of length equal to the arity of the predicate. A *literal* is either an atom a or its negation $\neg a$. In the latter case it is called a *negative literal*. In logic programming, predicate and function symbols are indicated with alphanumeric strings starting with a lowercase character while variables are indicated with alphanumeric strings starting with an uppercase character.

A *normal logic program* T is a finite set of of formulas of the form

$$h \leftarrow b_1, \ldots, b_n$$

called *clauses* where h is an atom and $b_1, \ldots, b_n$ are literals. h is called the *head* of the clause and $b_1, \ldots, b_m$ is called the *body*. If the body is empty the clause is called a *fact*. The formula $\leftarrow b_1, \ldots, b_n$ is called a *goal*. A conjunction of literals $b_1 \wedge \ldots \wedge b_m$ (also indicated with $b_1, \ldots, b_m$) is called a *query*.

A term, atom, literal, goal, query or clause is *ground* if it does not contain variables. A *substitution* θ is an assignment of variables to terms: $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$. The *application of a substitution to a term, atom, literal, goal, query or clause* C, indicated with $C\theta$, is the replacement of the variables appearing in C and in θ with the terms specified in θ. $C\theta$ is called an *instance* of C.

The *Herbrand universe* $H_U(T)$ is the set of all the ground terms that can be built with function symbols appearing in T. The *Herbrand base* $H_B(T)$ of a program T is the set of all the ground atoms that can be built with predicates appearing in T and terms from $H_U(T)$. A program that does not contain function symbols of arity greater than 0 is called *functor-free*. If T is functor-free, then $H_B(T)$ is finite, otherwise is countably infinite. A *grounding* of a clause C is obtained by replacing all the variables of C with ground terms from $H_U(T)$. The grounding $g(T)$ of a program T is the program obtained by replacing each clause with the set of all of its groundings. If the program is functor-free or is ground, $g(T)$ is finite, otherwise it is countably infinite. A *Herbrand interpretation* (or just *interpretation*) is a set of ground atoms, i.e., a subset of $H_B(T)$. An *interpretation* I *for a set of predicates* S is a subset of $H_B(T)$ that contains only atoms whose predicate is in S.

Various semantics have been proposed for normal logic programs. In this paper we consider Clark's completion [6], stable models [11] and the well-founded semantics [36]. These semantics coincide for modularly acyclic programs, defined below, and assign a program a single two-valued model, i.e., a Herbrand interpretation. In the following M_T will be used to indicate such a model for a program T

SLDNF-resolution [6] is an inference rule for normal logic programs. In order to compute answers to a goal G with SLDNF-resolution, we build a *SLDNF-forest* $F(G)$ for G. An SLDNF-forest is a set of SLDNF-trees[1]. An *SLDNF-tree* $T(G)$ for a goal G and selection rule R is a tree in which each node n is associated to a goal $G(n)$. In $T(G)$, the root is associated to G. The forest $F(G)$ contains at least one tree $T(G)$. In a tree $T(G)$, if a positive literal is selected by R in the goal $G(n)$ of a node n, then

[1]The notions of SLDNF-forest and SLDNF-tree are defined, albeit with slightly different names, in [2].

n has one child c_C for each clause C that resolves with $G(n)$ on the selected literal. The goal $G(c_C)$ is the resolvent of $G(n)$ with C. The nodes for which the goal is empty are marked as successful and represent answers to the goal at the root. The nodes for which no resolution is possible and whose goal is not empty are marked as failed. If a negative literal $\neg a$ is selected by R in the goal $G(n)$ of a node n, the forest $F(G)$ for the goal at the root will contain an SLDNF-forest $F(\leftarrow a)$ as a subset. If all the leaves of the tree $T(\leftarrow a)$ in $F(\leftarrow a)$ are marked as failed, then n has a single child c such that $G(c)$ is $G(n)$ with $\neg a$ deleted. Again, if $G(c) = \leftarrow$, then c is marked as successful. If the tree $T(\leftarrow a)$ in $F(\leftarrow a)$ is finite and there is a leaf marked as successful, then n is a leaf marked as failed. An SLDNF-forest $F(G)$ is finite if it is a finite set of finite trees. A path in an SLDNF-tree $T(G)$ starting from the root G is called an *SLDNF-derivation of the goal G from the program T*. If the last node in the path is an empty goal we say that the SLDNF-derivation is *successful.* If a negative literal that is not ground is selected in a node we say that the goal G *flounders.*

A program is *range-restricted* if all the variables appearing in the head of each clause appear also in positive literals in the body. If a normal program is range-restricted, every successful SLDNF-derivation for G completely grounds G [18].

We report here the definition of acyclic normal logic programs and of bounded literals and queries.

Definition 2.1 (Acyclic programs [1])

- A *level mapping* for a program T is a function $|\ | : H_B(T) \rightarrow N$ from ground atoms to natural numbers. For $a \in H_B(T)$, $|a|$ is the *level* of a. If $l = \neg a$ where $a \in H_B(T)$, we define $|l| = |a|$.
- A program T is called *acyclic with respect to a level mapping* if for every ground instance $a \leftarrow B$ of a clause of T, the level of a is greater then the level of each literal in B.
- A program T is called *acyclic* if there exists some level mapping such that T is acyclic with respect to it.

Definition 2.2 (Boundedness [1])

- A literal is *bounded with respect to a level mapping* if the set of levels of its ground instantiations is finite.
- A query (goal) is *bounded with respect to a level mapping* if all of its literals are.
- A query (goal) is *bounded with respect to an acyclic program T* if is bounded with respect to the level mapping for which T is acyclic.

Acyclic programs have the important property that the main semantics for normal logic programs, namely the well-founded semantics, stable models and Clark's completion, coincide. Moreover, SLDNF-resolution is a correct and complete procedure for answering queries in them.

Theorem 2.3
If T is an acyclic normal logic program then

1. The well-founded model M_T of T is two-valued and coincides with the only stable model and with the unique Herbrand model of the Clark's completion of the program.
2. For all ground atoms a that do not flounder, $a \in M_T$ iff there exists a successful SLDNF-derivation for the goal $\leftarrow a$ from T
3. The SLDNF-forest for a goal $\leftarrow Q$ where Q is a query bounded with respect to T is finite.

Proof. An acyclic program is locally stratified, therefore by Theorem 6.1 in [36] the well-founded model coincides with the perfect model and is thus two-valued. By Corollary 5.6 in [36] the well-founded model is equal to the only stable model of the program. Moreover, by Theorem 4.4 in [1] the perfect model coincides with the unique Herbrand model of the Clark's completion.

Item 2 is item iv of Theorem 4.4 in [1]. Item 3 is Theorem 4.1 of [1]. ■

However, acyclicity is a quite strong requirement that rules out many interesting programs, as the next example shows.

Example 2.4

Consider the program T_1 that defines the predicate *path*/2 such that $path(x, y)$ is true if there is a path from x to y in a directed graph. Such a program contains the clauses

$$\begin{array}{rcl} path(X,Y) & \leftarrow & edge(X,Y). \\ path(X,Y) & \leftarrow & edge(X,Z), path(Z,Y). \end{array}$$

plus a set E_1 of ground facts for the *edge*/2 relation that represent the edges between nodes of the graph. *path*/2 defines the transitive closure of *edge*/2.

Suppose E_1 contains the only fact

$$edge(a,b).$$

This program is not acyclic because it contains the ground rule

$$path(a,a) \quad \leftarrow \quad edge(a,a), path(a,a).$$

that imposes the contradictory constraint $|path(a,a)| > |path(a,a)|$

So even the simple program above is not acyclic. We would like to find a class of programs that is larger than the one of acyclic programs but still retains its important properties. One such class is that of modularly acyclic programs [29] that we define below.

A predicate p *directly depends* on a predicate q if q appears in the body of a rule that has p in the head. The relation "*depends*" is the transitive closure of the relation "directly depends". A predicate p is *recursive* on a predicate q if $p = q$ or p depends on q and q depends on p.

Recursiveness is an equivalence relation between predicates: it partitions the set of predicates of a program T into equivalence classes $Q_1, \ldots, Q_D$. For each equivalence class Q_i, consider the set V_i containing all the clauses of T that have a predicate of Q_i in the head. The sets $V_1, \ldots, V_D$ form a partition of T and are called *components* of

T. We write $V_i \sqsubset V_j$ if there is a predicate of Q_j that directly depends on a predicate of Q_i. We denote with $\lhd$ the transitive closure of $\sqsubset$. The elements of $S_i = \bigcup_{V_j \lhd V_i} Q_j$ are called the *predicates used by* V_i.

The elements of $S_i = \bigcup_{V_j \sqsubset V_i} Q_j$ are called the *predicates used by* V_i. We denote with $\lhd$ the transitive closure of $\sqsubset$.

Definition 2.5 (Reduction of a Component)
Let V_i be a component of a program T and let S_i be the set of predicates used by V_i. Consider an interpretation I for S_i.

The *reduction of* V_i *modulo* I, denoted with $R_I(V_i)$, is obtained in the following way:

- ground in all possible ways the rules of V_i obtaining $g(V_i)$;
- delete from $g(V_i)$ all the rules having a literal in the body whose predicate is in S_i, but which is false in I;
- delete from the bodies of the remaining rules all the literals having predicates in S_i (which are true);
- set $R_I(V_i)$ to the set of remaining ground rules.

Definition 2.6 (Modular Acyclicity)
A normal program T is *modularly acyclic* if for every component V_i of T

- the well founded model M_i of the union of all the components V_j such that $V_j \lhd V_i$ is total, and
- the reduction of V_i modulo M_i is acyclic

Example 2.7
Consider the program T_1 of Example 2.4. The program has two components: V_1 contains the fact for $edge/2$ and V_2 contains the clauses for $path/2$, with $V_1 \sqsubset V_2$. The well founded model M_2 of V_1 is total and is equal to $\{edge(a,b)\}$.

The reduction of V_2 modulo M_2 is

$$\begin{array}{rcl} path(a,b). & & \\ path(a,a) & \leftarrow & path(b,a). \\ path(a,b) & \leftarrow & path(b,b). \end{array}$$

which is acyclic with respect to the following level mapping:
$|path(b,a)| = 0$
$|path(a,a)| = |path(b,b)| = 1$
$|path(a,b)| = 2$
Consider now a general definition for $edge/2$. If the graph that $edge/2$ represents is acyclic, then we can sort the nodes according to a topological sort, i.e., a total order that is consistent with the partial order induced by the graph. Given a node x and a topological sort s, let $s(x)$ be the position of x in the order s.

The reduction of V_2 modulo M_2 ($R_{M_2}(V_2)$) is acyclic because it is acyclic with respect to the level mapping that assigns the number $s(y) - s(x) + N - 1$ to $path(x,y)$ where N is the total number of nodes (in the example above $N = 2$).

Now consider the program T_2 for computing paths in a graph:

$$path(X,Y) \quad \leftarrow \quad path(X,Y,[X],Z).$$

$$\begin{aligned} path(X,Y,V,[Y|V]) &\leftarrow edge(X,Y). \\ path(X,Y,V0,V1) &\leftarrow edge(X,Z), append(V0,\_S,V1), \neg member(Z,V0), \\ &\quad\; path(Z,Y,[Z|V0],V1). \end{aligned}$$

plus a set E_2 of ground facts for the $edge/2$ relation, the definition D_m of $member/2$ and the definition D_a of $append/3$. $path/2$ encodes the transitive closure of the $edge/2$ relation in a way that prevents SLDNF-resolution from going into a loop by keeping track of the nodes already visited.

This program contains five components: V_1 is equal to E_2, V_2 is equal to D_m, V_3 is equal to D_a, V_4 contains the definition of $path/4$ and V_5 the definition of $path/2$. The following relations hold: $V_1 \sqsubset V_4$, $V_2 \sqsubset V_4$, $V_3 \sqsubset V_4$ and $V_4 \sqsubset V_5$.

A total well-founded model exists for $V_{1,2,3} = V_1 \cup V_2 \cup V_3$. Let M_4 be the model for $V_{1,2,3}$. Consider the level mapping that assigns the follow numbers:

- l to $path(x,y,v1,v2)$ where l is the difference between the length of list $v2$ and the length of list $v1$ in the case where $v1$ is a prefix of $v2$.
- 0 to $path(x,y,v1,v2)$ if $v1$ is not a prefix of $v2$.

$R_{M_4}(V_4)$ is acyclic with respect to the above level mapping so T_3 is modularly acyclic.

Modularly acyclic programs benefit from the following important properties:

Theorem 2.8
if T is a modularly acyclic normal program then

1. The well-founded model M_T of T is two-valued and coincides with the only stable model and with the unique Herbrand model of Clark's completion of the program.

2. For all ground atoms a that do not flounder, $a \in M_T$ iff there exists a successful SLDNF-derivation for the goal $\leftarrow a$ from T

Proof. Theorem 2.1 in [29] states that T has a two valued well-founded model that coincides with the unique Herbrand model of the Clark's completion of T. By Corollary 5.6 in [36] the well-founded model is equal to the only stable model of the program.
Item 2 is item 5 in Theorem 2.1 of [29]. ∎

The proof of Theorem 2.1 in [29] is done by induction on the components. In the proof of item 5 it is assumed that the literals in the body are ordered from left to right by ascending component order, that the literals are selected from left to right and that the goal does not flounder under this ordering.

We now define the notion of boundedness of modularly acyclic programs.

Definition 2.9 (Boundedness for Modularly Acyclic Programs)

- A literal l whose predicate belongs to component V is *bounded with respect to a modularly acyclic program* T if it is bounded with respect to $R_M(V)$, the reduction of V modulo the model M of the components that precede V.
- A query (goal) is *bounded with respect to a modularly acyclic program* T if all of its literals are bounded with respect to T.

- A modularly acyclic program T is *bounded* if, for every component V of T, for every clause C of V, for every literal l in the body of C that belongs to a predicate of a component $V' \lhd V$, l is bounded with respect to T.

THEOREM 2.10
if T is a bounded modularly acyclic normal program and Q is a query bounded with respect to T, then the SLDNF-forest $F(\leftarrow Q)$ is finite.

PROOF. As for Theorem 2.8 we suppose that the literals in the body are ordered from left to right by ascending component order, that the literals are selected from left to right and that the goal does not flounder under this ordering.

The proof is done by induction on the structure of the formula and on the components. For the base case, consider a query composed of a single literal l that belongs to the lowest component. T may contain only facts matching with l so the forest $F(\leftarrow l)$ is finite.

In the recursive case, consider a conjunction of queries $L_1 \wedge L_2$. For the inductive hypothesis, the forests $F(\leftarrow L_1)$ and $F(\leftarrow L_2)$ are finite. The forest $F(\leftarrow L_1, L_2)$ can be obtained from $F(\leftarrow L_1)$ and $F(\leftarrow L_2)$ and is finite.

Consider a literal l belonging to component V and suppose that the theorem holds for all the literals belonging to lower components and for all conjunctions of such literals. For every clause of V that can be resolved with the atom of l, the SLDNF-forest that can be built for the portion of the body relative to preceding components is finite for the inductive hypothesis, since that portion of the body is bounded. Since T is modularly acyclic, the reduction $R_M(V)$ of V modulo the model M of the union of the preceding components is acyclic. Since l is bounded with respect to $R_M(V)$, the SLDNF-forest F' for the goal $\leftarrow l$ in $R_M(V)$ is finite. From F' we can build a finite forest $F(\leftarrow l)$ for the soundness and completeness of SLDNF-resolution. ∎

Note that the boundedness of the program is essential for the theorem to hold. In fact, consider the program T_3:

$$\begin{array}{rcl} p(0). & & \\ p(s(X)) & \leftarrow & p(X). \\ q & \leftarrow & p(X). \end{array}$$

T_3 contains two components, one defining $p/1$ and the other defining $q/0$. The program is modularly acyclic but is not bounded because the literal $p(X)$ in the body of the third rule is not bounded. In fact, the SLDNF-forest for the goal $\leftarrow q$ is infinite.

3 Independent Choice Logic

3.1 Syntax and Semantics

An *Independent Choice Logic theory* [22] is a triple $T = (F, C, P_0)$ where

- F is a normal logic program,
- C is a finite set of *alternatives* $\{\chi_1, \ldots, \chi_m\}$ where χ_i is a finite set of atoms $\{a_{i,1}, \ldots, a_{i,n_i}\}$ called *hypotheses*,

- P_0 is a probability distribution over the atoms of each alternative, i.e., $P_0(i,j)$ is the probability of $a_{i,j}$ and $\sum_{j=1}^{n_i} P_0(i,j) \leqslant 1$.

Alternatives can be expressed by means of declarations of the form

$$disjoint([a_{i,1} : P_0(i,1), \ldots, a_{i,n_i} : P_0(i,n_i)]).$$

An alternative $\chi_i = \{a_{i,1}, \ldots, a_{i,n_i}\}$ with non-ground atoms is a compact way of representing a set of alternatives, one for each grounding $\chi_i\theta = \{a_{i,1}\theta, \ldots, a_{i,n_i}\theta\}$ of χ_i, that share the same distribution over the atoms.

If $\sum_{j=1}^{n_i} P_0(i,j) < 1$, an implicit atom *null* is added to alternative χ_i and is assigned index $n'_i = n_i + 1$. Therefore, the alternative has one more atom $a_{i,n'_i} = null$. Its probability is $P_0(i,n'_i) = 1 - \sum_{j=1}^{n_i} P_0(i,j)$. *null* does not appear in the body of any clause of F.

A triple (χ_i, θ, j), where χ_i is an alternative, θ is a substitution that grounds χ_i and $j \in \{1, \ldots, n'_i\}$ is called an *atomic choice*: it represents the choice of a single atom from the grounding $\chi_i\theta$ of an alternative χ_i. A set of atomic choices κ is *consistent* if no two different atoms are selected from the same grounding of the same alternative, i.e., if $(\chi_i, \theta, j) \in \kappa, (\chi_i, \theta, k) \in \kappa \Rightarrow j = k$. A *composite choice* κ is a consistent set of atomic choices. A *selection* σ is a composite choice that contains exactly one atom from every grounding of every alternative. Let $\mathcal{S}_T$ be the set of all possible selections for theory T.

A *possible world* w_σ for a selection σ is the union of all the atoms chosen by the selection σ together with the program F, i.e.,

$$w_\sigma = F \cup \bigcup_{(\chi_i,\theta,j)\in\sigma} \{a_{i,j}\theta\}$$

Let $\mathcal{W}_T$ be the set of all possible worlds. In [22] F is required to be acyclic, as a consequence each possible world is an acyclic normal logic program. Thus the stable model semantics, the well-founded semantics and the unique Herbrand model of Clark's completion of a possible world coincide. In the following we will write $w_\sigma \models \phi$ to mean that the closed formula ϕ is true in the unique model of the program according to the three semantics.

Note that, if the theory is functor-free or is ground, then the set of selections (and of possible worlds) is finite. Otherwise such a set is infinite and it is not countable, as can be shown by using Cantor's diagonal argument.

If a program is functor-free, we can assign a probability to each possible world:

$$P(w_\sigma) = \prod_{(\chi_i,\theta,j)\in\sigma} P_0(i,j)$$

The product is over all the atomic choices in σ therefore it contains exactly one factor for every grounding of every alternative. Note that the product may contain repeated factors in the case in which the same atom j is selected from different groundings of the same alternative. It is easy to see that P satisfies the axioms of probability.

ICL assigns a probability to a closed formula ϕ by summing up the probabilities of possible worlds where the formula is true:

$$P(\phi) = \sum_{\sigma\in\mathcal{S}_T, w_\sigma\models\phi} P(w_\sigma). \tag{3.1}$$

Example 3.1
Consider the dependency of a person's sneezing from him having the flu or hay fever:

$$
\begin{aligned}
&strong\_sneezing(X) \leftarrow flu(X), flu\_strong\_sneezing(X).\\
&strong\_sneezing(X) \leftarrow hay\_fever(X), hay\_fever\_strong\_sneezing(X).\\
&moderate\_sneezing(X) \leftarrow flu(X), flu\_moderate\_sneezing(X).\\
&moderate\_sneezing(X) \leftarrow hay\_fever(X), hay\_fever\_moderate\_sneezing(X).\\
&flu(david).\\
&hay\_fever(david).\\
&disjoint([flu\_strong\_sneezing(X) : 0.3, flu\_moderate\_sneezing(X) : 0.5]).\\
&disjoint([hay\_fever\_strong\_sneezing(X) : 0.2,\\
&hay\_fever\_moderate\_sneezing(X) : 0.6]).
\end{aligned}
$$

This program has two alternatives:

$$
\begin{aligned}
\chi_1 &= \{flu\_strong\_sneezing(X), flu\_moderate\_sneezing(X), null\}\\
\chi_2 &= \{hay\_fever\_strong\_sneezing(X), hay\_fever\_moderate\_sneezing(X), null\}
\end{aligned}
$$

and it models the fact that sneezing can be caused by flu or hay fever. Flu causes strong sneezing, moderate sneezing or no sneezing with probabilities 0.3, 0.5 and $1 - 0.3 - 0.5 = 0.2$ respectively; hay fever causes strong sneezing, moderate sneezing or no sneezing with probabilities 0.2, 0.6 and $1 - 0.2 - 0.6 = 0.2$ respectively.

This program has 3^2 possible selections/possible worlds. One selection is

$$\sigma = \{(\chi_1, \{X/david\}, 1), (\chi_2, \{X/david\}, 2)\}$$

corresponding to the possible world

$$
\begin{aligned}
&strong\_sneezing(X) \leftarrow flu(X), flu\_strong\_sneezing(X).\\
&strong\_sneezing(X) \leftarrow hay\_fever(X), hay\_fever\_strong\_sneezing(X).\\
&moderate\_sneezing(X) \leftarrow flu(X), flu\_moderate\_sneezing(X).\\
&moderate\_sneezing(X) \leftarrow hay\_fever(X), hay\_fever\_moderate\_sneezing(X).\\
&flu(david).\\
&hay\_fever(david).\\
&flu\_strong\_sneezing(david).\\
&hay\_fever\_moderate\_sneezing(david).
\end{aligned}
$$

whose probability is $0.3 \cdot 0.6 = 0.18$. *strong_sneezing*(*david*) is true in 5 worlds. Its probability is

$$0.3 \cdot 0.2 + 0.3 \cdot 0.6 + 0.3 \cdot 0.2 + 0.5 \cdot 0.2 + 0.2 \cdot 0.2 = 0.44$$

.

If the program is not functor-free and is not ground, the semantics of ICL given above is not well defined. In fact, each selection is infinite and the probability of a single possible world is 0, being an infinite product of numbers all smaller than 1.

In order to assign a semantics to non-functor-free ICL, we must resort to the axiomatization of probability theory given in [15] that is based on measure theory.

Kolmogorov defined probability functions (or measures) as real-valued functions over an algebra Ω of subsets of a set $\mathcal{W}$ called the *sample space*. The set Ω is an algebra of $\mathcal{W}$ iff

1. $\mathcal{W} \in \Omega$
2. Ω is closed under complementation, i.e., $\omega \in \Omega \rightarrow (\mathcal{W} \setminus \omega) \in \Omega$
3. Ω is closed under finite union, i.e., $\omega_1 \in \Omega, \omega_2 \in \Omega \rightarrow (\omega_1 \cup \omega_2) \in \Omega$

The elements of Ω are called *measurable sets*. Not every subset of $\mathcal{W}$ need be present in Ω.

Given a sample space $\mathcal{W}$ and an algebra Ω of subsets of $\mathcal{W}$, a probability measure is a function $\mu : \Omega \rightarrow R$ that satisfies the following axioms

1. $\mu(\omega) \geqslant 0$ for all $\omega \in \Omega$
2. $\mu(W) = 1$
3. $\omega_1 \cap \omega_2 = \emptyset \rightarrow \mu(\omega_1 \cup \omega_2) = \mu(\omega_1) + \mu(\omega_2)$ for all $\omega_1 \in \Omega, \omega_2 \in \Omega$

These are the finite additivity version of *Kolmogorov probability axioms*. In the more general case, Ω is required to be a σ-algebra, i.e., it must be closed under countable unions rather than finite unions, and axiom 3 is required to hold for countable unions. Since in ICL we do not have infinite unions, the finite additivity version of the axioms is sufficient.

In ICL, the sample space is the set of possible worlds $\mathcal{W}_T$ and the algebra of subsets Ω_T is defined by means of sets of composite choices. A composite choice κ identifies a set of possible worlds ω_κ that contains all the worlds relative to a selection that is a superset of κ, i.e.,

$$\omega_\kappa = \{w_\sigma | \sigma \in \mathcal{S}_T, \sigma \supseteq \kappa\}$$

Similarly we can define the set of possible worlds associated to a set of composite choices K:

$$\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$$

Ω_T is then defined as the set of sets of possible worlds identified by finite sets of finite composite choices:

$$\Omega_T = \{\omega_K | K \text{ is a finite set of finite composite choices}\}$$

It is easy to see that Ω_T is an algebra over $\mathcal{W}_T$.

We now have to define the measure μ over the set Ω_T so that the axioms are respected. First of all note that, if K_1 is a finite set of finite composite choices, there may be other finite sets K_2 of finite composite choices such that $\omega_{K_1} = \omega_{k_2}$. For the case of Example 3.1, the set

$$K_1 = \{\kappa_1, \kappa_2\}$$

where

$$\begin{aligned}\kappa_1 &= \{(\chi_1, \{X/david\}, 1)\} && (3.2)\\ \kappa_2 &= \{(\chi_2, \{X/david\}, 1)\} && (3.3)\end{aligned}$$

describes the same set of possible worlds as

$$K_2 = \{\kappa'_1, \kappa'_2, \kappa'_3\} \tag{3.4}$$

where

$$\begin{aligned}\kappa'_1 &= \{(\chi_1, \{X/david\}, 1), (\chi_2, \{X/david\}, 2)\}\\ \kappa'_1 &= \{(\chi_1, \{X/david\}, 1), (\chi_2, \{X/david\}, 3)\} && (3.5)\\ \kappa'_1 &= \{(\chi_2, \{X/david\}, 1)\}\end{aligned}$$

K_1 and K_2, however, have an important difference: in K_2 $\omega_{\kappa'_1}$, $\omega_{\kappa'_2}$ and $\omega_{\kappa'_3}$ are disjoint, while in K_1, ω_{κ_1} and ω_{κ_2} are not disjoint. Finite set of finite composite choices whose composite choices produce disjoint sets of possible worlds are called *mutually incompatible*.

Formally, two composite choices κ_1 and κ_2 are *incompatible* if their union is inconsistent, i.e., if there exists an alternative χ_i and a substitution θ grounding χ_i such that $(\chi_i, \theta, j) \in \kappa_1, (\chi_i, \theta, k) \in \kappa_2$ and $j \neq k$. It is clear that if κ_1 and κ_2 are incompatible then $\omega_{\kappa_1} \cap \omega_{\kappa_2} = \emptyset$.

A set K of composite choices is *mutually incompatible* if for all $\kappa_1 \in K, \kappa_2 \in K, \kappa_1 \neq \kappa_2 \Rightarrow \kappa_1$ and κ_2 are incompatible. The set of composite choices K_2 above is mutually incompatible for the theory of Example 3.1 while K_1 is not.

Mutually incompatible finite set of finite composite choices have the following important properties.

LEMMA 3.2
Given a finite set K of finite composite choices, there exists a finite set K' of finite composite choices that is mutually incompatible and such that $\omega_K = \omega_{K'}$.

PROOF. Section 4.5 of [23] presents a terminating algorithm for obtaining K' from K
. ■

LEMMA 3.3 (Lemma A.8 in [21], Lemma 5.1 in [23])
If K and K' are both mutually incompatible sets of composite choices such that $\omega_K = \omega_{K'}$, then $\sum_{\kappa \in K} \prod_{(\chi_i,\theta,j)\in\kappa} P_0(i,j) = \sum_{\kappa \in K'} \prod_{(\chi_i,\theta,j)\in\kappa} P_0(i,j)$

The measure $\mu : \Omega_T \to [0,1]$ is defined in [23] as

$$\mu(\omega) = \sum_{\kappa \in K} \prod_{(\chi_i,\theta,j)} P_0(i,j)$$

where K is a mutually incompatible finite set of finite composite choices such that $\omega_K = \omega$. By Lemma 3.3 we have that it does not matter which K is chosen, so μ is truly a function. By Lemma 3.2, given a finite set K of finite composite choices, we can always find an equivalent mutually incompatible set, so the measure μ is well defined.

Lemma 5.2 in [23] showed that the probability measure thus defined satisfies the finite additivity version of Kolmogorov axioms reported above.

We now have to use the measure so defined to assign a probability to queries. Given a closed formula ϕ, a composite choice κ is an *explanation for* ϕ if ϕ is true in every world of ω_κ. In Example 3.1, the composite choice κ_1 shown above is an explanation for $strong\_sneezing(david)$.

A set K of composite choices is *covering with respect to* ϕ if every world in which ϕ is true belongs to ω_K. The sets K_1 and K_2 are both covering for for $strong\_sneezing(david)$, while $K_3 = \{\kappa_1\}$ is not.

Definition 3.4
The probability of a ground formula ϕ is given by the measure according to μ of the set of worlds where ϕ is true, i.e.,

$$P(\phi) = \mu(\{w|w \in \mathcal{W}_T \wedge w \models \phi\}$$

Theorem 3.6 in the next section shows that, if F is acyclic and ϕ is a ground query, there is a finite set K of finite explanations of ϕ such that K is covering. Therefore $\{w|w \in \mathcal{W}_T, w \models \phi\} \in \Omega_T$ and $P(\phi)$ is well defined. We thus have that (see Proposition 5.3 from [22]):

$$P(\phi) = \sum_{\kappa \in K} \prod_{(\chi_i,\theta,j)\in\kappa} P_0(i,j) \tag{3.6}$$

where K is a mutually incompatible finite set of finite explanations for ϕ that is covering with respect to ϕ. In the case of Example 3.1, K_2 is a covering set of explanations for $strong\_sneezing(david)$ that is mutually incompatible, so

$$P(strong\_sneezing(david)) = 0.3 \cdot 0.6 + 0.3 \cdot 0.2 + 0.2 = 0.44$$

If $\mathcal{W}_T$ is finite, then $\Omega_T = 2^{\mathcal{W}_T}$ and $P(\phi)$ given by equation 3.1 coincides with $P(\phi)$ given by equation 3.6. In fact, in this case, each possible world is associated to a finite composite choice. Moreover, the composite choices associated to two different possible worlds are mutually incompatible. Thus the set of worlds where ϕ is true corresponds to a mutually incompatible finite set of finite composite choices.

3.2 ICL Inference Algorithms

The algorithm proposed in [23] for computing the probability of a ground query Q from an ICL theory consists of two phases:

1. finding a finite set K of finite explanations for Q that is covering with respect to Q,
2. computing $P(Q)$ from K.

In order to describe phase 1, let us now give some definitions. If K is a set of composite choices, then a *complement* of K is a set K' of composite choices such that for all worlds $w \in \mathcal{W}_T$, $w \in \omega_{K'}$ iff $w \notin \omega_K$.

If K is a set of composite choices, then composite choice κ' is a *dual* of K if $\forall \kappa \in K$ $\kappa' \cup \kappa$ is inconsistent. A dual is *minimal* if no proper subset is also a dual. Let

```
**function** $duals(K)($
    **inputs :** $K$: set of composite choices
    **returns :** the set of duals to $K)$
Suppose $K = \{\kappa_1, \ldots, \kappa_n\}$.
Let $D_0 = \{\{\}\}$; // $D_i$ is the set of duals of $\{\kappa_1, \ldots, \kappa_i\}$
for $i = 1$ to $n$ do
    $D_i = \{d \cup \{(\chi, \theta, k)\} | d \in D_{i-1}, (\chi, \theta, j) \in \kappa_i, k \neq j, consistent(d \cup \{(\chi, \theta, k)\})\}$
endfor
    return $mins(D)$
```

FIG. 1. Function $duals(K)$.

$duals(K)$ be the set of all minimal duals of K. $duals(K)$ can be computed with the algorithm shown in Figure 1 [23], where the function *mins* is defined as follows:

$$mins(S) = \{\kappa \in S : \forall \kappa' \in S, \kappa' \not\subset \kappa\}$$

and the function $consistent(\kappa)$ returns *true* if κ is consistent.

The function $duals(K)$ returns a set of composite choices in which each element κ' is obtained in the following way: an atomic choice (χ_i, θ, j) is picked from each composite choice $\kappa \in K$, a k is selected from $\{1, \ldots, n'_i\}$ such that $k \neq j$ and (χ_i, θ, k) is added to κ'. By repeating this process in all possible ways we get $duals(K)$.

Duals and complements have the following relationship:

LEMMA 3.5 (Lemma 4.8 of [23])
If K is a set of composite choices, $duals(K)$ is a complement of K.

In fact, each element of $duals(K)$ is incompatible with each element of K and every composite choice that is incompatible with each element of K is in $duals(K)$.

We now present the algorithm for computing explanations. We first define the following notion: if K_1 and K_2 are sets of composite choices, the *conjunction of* K_1 *and* K_2 is defined as the set of composite choices:

$$K_1 \otimes K_2 = \{\kappa_1 \cup \kappa_2 : \kappa_1 \in K_1, \kappa_2 \in K_2, consistent(\kappa_1 \cup \kappa_2)\}$$

If Q is a ground query, its explanations are computed with the function $expl(Q)$ defined recursively as follows[2]:

$$expl(Q) = \begin{cases} mins(expl(A) \otimes expl(B)) & \text{if } Q = A \wedge B \\ duals(expl(A)) & \text{if } Q = \neg A \\ \{\{(\chi_i, \theta, j)\}\} & \text{if } Q = a_{i,j}\theta \\ \{\} & \text{if } Q \in g(F) \\ mins(\cup_i expl(B_i)) & \text{if } Q \notin \mathcal{C}, Q \leftarrow B_i \in g(F) \end{cases}$$

where $\mathcal{C}$ is the union of the groundings of each alternative and where it is assumed that an atom that appears in the grounding of an alternative does not appear in other

[2][23] considered also the case of disjunction appearing in the query and in the body of rules. We do not consider this case for simplicity.

groundings of alternatives nor in the head of ground rules. If the theory is acyclic $expl(Q)$ is well defined.

Note that the use of functions *mins* when computing duals and $expl(Q)$ is not necessary: all the properties of duals and explanations holds also if the minimization is not performed.

Theorem 3.6 (Theorem 4.24 in [23])
If Q is a ground query, Q is true in world w_σ iff there is some $\kappa \in expl(Q)$ such that $\kappa \subseteq \sigma$. Moreover, $expl(Q)$ is a finite set of finite sets.

This theorem states that $expl(Q)$ is a finite set of finite explanations that is covering with respect to Q. The theorem was proved for the case of an acyclic F [23] by induction on the levels of the program, and, for each level, by induction on the structure of the formula.

[23] proposed two approaches for performing phase 2, i.e., for computing the probability of a ground query Q given $expl(Q)$:

- applying the inclusion-exclusion formula that computes the probability of a disjunction of N formulas, or
- making the explanations mutually incompatible by "splitting" them

The inclusion-exclusion formula is given by:

$$P(Q) = \sum_{i=1}^{N}(-1)^{i-1}\left(\sum_{1\leqslant j_1<\ldots<j_i\leqslant N} P(\kappa_{j_1}\wedge\ldots\wedge\kappa_{j_i})\right) \tag{3.7}$$

where $expl(Q) = \{\kappa_1, \ldots, \kappa_N\}$. This approach requires computing the probability of every possible conjunction of explanations in $expl(Q)$, thus making it infeasible for all but the smallest programs.

The splitting algorithm is illustrated in Figure 2, where, if $\chi_i\theta$ is the grounding of an alternative and κ is a consistent choice such that $\nexists j : (\chi_i, \theta, j) \in \kappa$, the *split* of κ on $\chi_i\theta$ is defined as the set of composite choices

$$\{\kappa \cup \{(\chi_i, \theta, 1)\}, \ldots, \kappa \cup \{(\chi_i, \theta, n'_i)\}\}$$

The function $disjoint(K)$ repeatedly picks couples (κ_1, κ_2) from K and, if they are not mutually incompatible, it selects an atomic choice (χ_i, θ, j) that is κ_1 but not in κ_2 and it splits κ_2 on $\chi_i\theta$. The resulting K_2 will have $n'_i - 1$ composite choices which are incompatible with κ_1 and one which is still compatible. By repeating the process we obtain that, for each couple (κ_1, κ_2) we pick, either κ_1 is incompatible with κ_2 or one is contained in the other and we can safely remove it.

4 Modularly Acyclic ICL

In this section we define the class of modularly acyclic ICL theories and we provide a semantics for it.

Definition 4.1 (ICL Modular Acyclicity)
An ICL theory T is *modularly acyclic* if every possible world of $\mathcal{W}_T$ is modularly acyclic.

```
function $disjoint(K)$(
    inputs : $K$: set of composite choices
    returns : mutually incompatible set of composite choices equivalent to $K$
repeat
    if $\kappa_1, \kappa_2 \in K$ and $\kappa_1 \subset \kappa_2$
    then $K := K \setminus \{\kappa_2\}$
    else if $\exists \kappa_1, \kappa_2 \in K$, such that $consistent(\kappa_1 \cup \kappa_2)$ then
        choose $(\chi_i, \theta, j) \in \kappa_1 \setminus \kappa_2$
        let $K_2$ be the split of $\kappa_2$ on $\chi_i\theta$
        $K := K \setminus \{\kappa_2\} \cup K_2$
    else exit and return $K$
forever
```

Fig. 2. Function *disjoint*(K).

Thus, the well-founded model M_σ of each possible world w of T is two valued.

Definition 4.2 (Boundedness for Modularly Acyclic ICL Theories)

- A modularly acyclic ICL theory T is *bounded* if every possible world $w_\sigma \in \mathcal{W}_T$ of T is bounded.
- A literal is *bounded with respect to a modularly acyclic ICL theory* T if it is bounded with respect to every possible world $w \in \mathcal{W}_T$ of T.
- A query (goal) is *bounded with respect to a modularly acyclic ICL theory* T if all of its literals are.

In order to define a semantics for modularly acyclic ICL theories we need to prove that bounded queries have a finite set of finite explanations that are covering.

Theorem 3.6 states that $expl(Q)$ is the set of covering explanations for Q for acyclic ICL theories. The proof employed in [23] is based on induction on the acyclicity level of the program and, for each level, on induction on the structure of formulas. The assumption of acyclicity is essential for the proof to hold and is equivalent to computing explanations by applying the function $expl(Q)$.

If the ICL theory is modularly acyclic, the proof does not hold anymore because the structure of formulas interacts with the levels of the program.

Example 4.3
Consider the following modularly acyclic ICL theory

$$\begin{aligned}
&p \leftarrow a, q.\\
&q \leftarrow b, p.\\
&disjoint([a : 0.3, b : 0.7]).
\end{aligned}$$

Suppose the query is p. p belongs to different acyclicity levels in different possible worlds. Thus we can not perform induction on the levels and, for each level, on the structure of the formula.

In fact, by using the definition of $expl(Q)$, the formula $expl(a) \otimes expl(q)$ must be computed. Since the explanations for q are computed without taking into account the explanations for a, this leads to an unbounded recursion.

Therefore, we propose a different algorithm for computing the covering set of explanations. The algorithm is called *SLDNFICL-resolution* and is an extension of SLDNF-resolution for ICL theories.

In order to compute explanations for a goal G with SLDNFICL-resolution, we build a *SLDNFICL-forest* $F(G)$ for G. An SLDNFICL-forest is a set of SLDNFICL-trees. An *SLDNFICL-tree* $T(G)$ for a goal G is a tree in which each node n is associated to a goal $G(n)$ and to a composite choice $\kappa(n)$. Moreover, the leaves of $T(G)$ can be marked as *successful* or *failed*. In $T(G)$, the root is associated to the goal G and to the empty composite choice. The forest $F(G)$ contains at least one tree $T(G)$. Given a node n of a tree $T(G)$ that is not marked and a selection rule R, its children are obtained in one of the following ways

1. If $G(n) = \leftarrow$ then n is a leaf marked as *successful*.
2. If $l_s = R(G(n))$ is a positive literal then
   (a) n has one child c_C for each clause $C = h \leftarrow B$ of F such that C resolves with $G(n)$ on l_s. Moreover, $G(c_C) = S$ where S is the resolvent of C with $G(n)$ on the literal l_s and $\kappa(c_C) = \kappa(n)$.
   (b) n has one child $c_{\chi_i,\theta,j}$ for each alternative $\chi_i = \{a_{i,1}, \ldots, a_{i,n'_i}\}$ and for each j such that $a_{i,j}$ unifies with l_s with substitution θ and such that $consistent(\kappa(n) \cup \{(\chi_i, \theta, j)\}) = true$. Moreover, $G(c_{\chi_i,\theta,j}) = (G(n) \setminus \{l_s\})\theta$ and $\kappa(c_{\chi_i,\theta,j}) = \kappa(n) \cup \{(\chi_i, \theta, j)\}$.

   If there are no clauses satisfying the condition in 2a and there are no alternatives satisfying the conditions in 2b, then n is a leaf marked as *failed*.
3. If $l_s = R(G(n))$ is a ground negative literal $\neg a$, then the SLDNFICL-forest $F(\leftarrow G)$ will contain the SLDNFICL-forest $F(\leftarrow a)$ as a subset. Let K^a be the set of all the composite choices for successful leaves of $T(\leftarrow a)$. For each element κ of $duals(K^a)$ (computed with the algorithm in Figure 1) such that $consistent(\kappa(n) \cup \kappa) = true$, n has one child c_κ with $G(c_\kappa) = G(n) \setminus \{l_s\}$ and $\kappa(c_\kappa) = \kappa(n) \cup \kappa$. If there are no elements of $duals(K^a)$ satisfying the condition above then n is a leaf marled as *failed*.

An SLDNFICL-forest $F(G)$ is finite if it is a finite set of finite trees. A path in an SLDNFICL-tree $T(G)$ starting from the root G is called an *SLDNFICL-derivation of the goal G from the program T* and can be represented as a sequence of couples $(G, \emptyset), (G_1, \kappa_1) \ldots (G_n, \kappa_n)$ where the first element of the couples is a goal and the second is a composite choice. If the last node has an empty goal we say that the SLDNFICL-derivation is *successful*. Each step of an SLDNFICL-derivation is called an *SLDNFICL-resolution step*. If a negative literal that is not ground is selected in a node we say that the goal G *flounders*.

Given a goal $G = \leftarrow Q$, we call $L(G)$ the set of successful leaves of $T(G)$ and we call $expl_{ICL}(Q)$ the set $\{\kappa(l) | l \in L(G)\}$

In the following we assume that R selects the leftmost literal, that the literals in bodies respect the ascending order of components and that the goal does not flounder under this ordering.

For SLDNFICL-resolution to work correctly, it is necessary that, when an atomic choice (χ_i, θ, j) is added to the composite choice $\kappa(n)$, the set of alternatives $\chi_i\theta$ is ground. In fact, the function *consistent* requires its input to be a composite choice, which would not be true if θ does not ground χ_i.

Various conditions can be imposed to obtain this requirement. We pose three conditions. First we partition the set of predicates of a program in two disjoint sets: the set of predicates that appear in the head of rules and the set of predicates that appear in alternatives. A literal built on this latter set is called a *choice literal*. Therefore a positive literal selected in the body of a rule can unify only with the head of a clause or with an atom of an alternative but not both and cases 2a and 2b are mutually exclusive. The second condition is that choice literals are ground when they are selected while the third condition is that all the atoms in each alternative, excluding *null*, must have exactly the same variables.

It is clear that these conditions ensure that the alternative $\chi_i\theta$ is ground when the atomic choice (χ_i, θ, j) is added to $\kappa(n)$.

To ensure the groundness of the selected choice literals, we can consider *choice-safe* theories in which every variable of a choice literal in the body of a clause appears in a positive literal in the body that is not a choice literal and that belongs to a component preceding the one to which the clause belongs. If we have a choice-safe theory in which F is range-restricted and if we move all the choice literals to the right of all the literals belonging to preceding components and to the left of the literals belonging to the current component, then the choice literals will be ground when they are selected. In fact, every variable appearing in a choice literal will be assigned to a ground term by the preceding literals.

If T is choice-safe and F is range-restricted, $\kappa(n)$ will always be ground. In the following we will consider only choice-safe theories with a range-restricted F.

Example 4.4
Consider the program

$a \leftarrow b.$
$a \leftarrow \neg c, e.$
$\chi_1 = \{b, null\}$
$\chi_2 = \{c, d, null\}$
$\chi_3 = \{e, null\}.$

The forest for the goal $\leftarrow a$ is shown in Figure 3, where each edge is annotated with the atomic choices that are added by the resolution step to the composite choice.

Thus there are three successful SLDNFICL-derivations for $\leftarrow a$. Their final composite choices are

$\kappa_1 = \{(\chi_1, \emptyset, 1)\}$
$\kappa_2 = \{(\chi_2, \emptyset, 2), (\chi_3, \emptyset, 1)\}$
$\kappa_3 = \{(\chi_2, \emptyset, 3), (\chi_3, \emptyset, 1)\}$

Example 4.5
Consider the following example:

$a(X) \leftarrow b(X), c(X), \neg d.$
$b(1).$
$d \leftarrow e.$
$d \leftarrow f, \neg g.$

$$\begin{tikzcd}
& & \leftarrow a \arrow[dl] \arrow[dr] & & & \leftarrow c \arrow[d, "{(\chi_2,\emptyset,1)}"'] \\
& \leftarrow b \arrow[dl, "{(\chi_1,\emptyset,1)}"'] & & \leftarrow \neg c, e \arrow[dl, "{(\chi_2,\emptyset,2)}"'] \arrow[dr, "{(\chi_2,\emptyset,3)}"] & & \leftarrow \\
\leftarrow & & \leftarrow e \arrow[d, "{(\chi_3,\emptyset,1)}"'] & & \leftarrow e \arrow[d, "{(\chi_3,\emptyset,1)}"] & \\
& & \leftarrow & & \leftarrow &
\end{tikzcd}$$

FIG. 3. Forest for Example 4.4

$$\begin{tikzcd}
& \leftarrow a(1) \arrow[d] & & & \leftarrow d \arrow[dl] \arrow[dr] & & & \leftarrow g \arrow[d, "{(\chi_4,\emptyset,1)}"'] \\
& \leftarrow b(1), c(1), \neg d \arrow[d] & & \leftarrow e \arrow[d, "{(\chi_2,\emptyset,1)}"'] & & \leftarrow f \neg g \arrow[d, "{(\chi_3,\emptyset,1)}"] & & \leftarrow \\
& \leftarrow c(1), \neg d \arrow[d, "{(\chi_1,\{X/1\},1)}"'] & & \leftarrow & & \leftarrow \neg g \arrow[dl, "{(\chi_4,\emptyset,2)}"'] \arrow[dr, "{(\chi_4,\emptyset,3)}"] & & \\
& \leftarrow \neg d \arrow[dl, "{\substack{(\chi_2,\emptyset,2)\\(\chi_3,\emptyset,2)}}"'] \arrow[dr, "{\substack{(\chi_2,\emptyset,2)\\(\chi_4,\emptyset,1)}}"] & & & \leftarrow & & \leftarrow & \\
\leftarrow & & \leftarrow & & & & &
\end{tikzcd}$$

FIG. 4. Forest for Example 4.5

$\chi_1 = \{c(X), null\}$
$\chi_2 = \{e, null\}$
$\chi_3 = \{f, null\}$
$\chi_4 = \{g, h, null\}$
The forest for the goal $\leftarrow a(1)$ is shown in Figure 4 where each edge is annotated with the atomic choices added by the resolution step to the composite choice.

In the following we assume that the goal $G = \leftarrow Q$ is ground. The case in which G is not ground can be treated similarly by isolating the portion of $F(G)$ relative to the same instantiation of G.

LEMMA 4.6
Let T be a bounded modularly acyclic ICL theory and Q a ground query bounded with respect to T. Then $F(\leftarrow Q)$ is finite.

PROOF. The proof is by structural induction on the set of forests $\mathcal{F}$, where the partial order is the subset relation.

In the base case the forest $F(\leftarrow Q)$ contains a single tree $T(\leftarrow Q)$. If $T(\leftarrow Q)$ is infinite, there exists a possible world in which the SLDNF-tree for $\leftarrow Q$ is infinite but then the possible world would not be bounded modularly acyclic or Q would not be bounded against the hypothesis.

In the recursive case, each forest F' such that $F' \subset F(\leftarrow Q)$ is finite. Then $F(\leftarrow Q)$ is also finite by contradiction as for the base case. ■

We can now prove the soundness and completeness of SLDNFICL-resolution.

Theorem 4.7
Let T be a bounded modularly acyclic ICL theory and Q be a ground query bounded with respect to T. Then, Q is true in world w_σ iff there exist a $\kappa \in expl_{ICL}(Q)$ such that $\kappa \subseteq \sigma$.

Proof. We prove the theorem by structural induction on the set of forests $\mathcal{F}$.

In the base case, the forest $F(\leftarrow Q)$ contains a single tree $T(\leftarrow Q)$. No negative literal is selected in the nodes. Consider a successful leaf $l \in L(Q)$: $\kappa(l)$ is consistent by construction and each step of the SLDNFICL-derivation $(\leftarrow Q, \emptyset) \ldots (\leftarrow, \kappa(l))$ is an SLDNF-resolution step for $\leftarrow Q$ in each world w_σ such that $w_\sigma \in \omega_{\kappa(l)}$, so $w_\sigma \models Q$

Now let w_σ be a world where Q is derivable by SLDNF. For each step of a successful SLDNF-derivation of Q in w_σ, there is a corresponding SLDNFICL-resolution step: if a clause C from F is used in the SLDNF step, then rule 2a of SLDNFICL can be applied to go from a node to a child and if an atomic choice $a_{i,j}\theta$ is used in the SLDNF step, then rule 2b of SLDNFICL can be applied. Thus, if we visit $T(\leftarrow Q)$ starting from the root and choosing at each step the child that corresponds to a step of the SLDNF-derivation of $\leftarrow Q$ from w_σ, the leaf l that is reached in this way is such that $\kappa(l) \subseteq \sigma$.

Now consider a forest $F(\leftarrow Q)$ made up of more than one tree. Suppose that the theorem holds for every $F(\leftarrow O)$ with $F(\leftarrow O) \subset F(\leftarrow Q)$. For each negative selected literal $\neg a$ in a node n in $T(\leftarrow Q)$ there is a forest $F(\leftarrow a)$ in $F(\leftarrow Q)$. *duals* ensures that for each $\kappa \in expl_{ICL}(a)$ and for each child c of n, there exists a triple $(\chi, \theta, j) \in \kappa$ such that $(\chi, \theta, k) \in \kappa(c)$ and $k \neq j$. Since $expl_{ICL}(a)$ is covering for the inductive hypothesis, then a will not be derivable by SLDNF in every possible world of $\omega_{\kappa(c)}$.

Let w_σ be a world of T such that Q is derivable by SLDNF in it. We can identify at least one leaf l such that $\kappa(l) \subseteq \sigma$ by starting from the root of $T(\leftarrow Q)$ and by visiting the tree choosing at each step the child that corresponds to the clause or atomic choice used in an SLDNF-resolution step of a successful derivation for $\leftarrow Q$ in w_σ. If no negative literal is selected, we are in the same situation as in the base case. If a negative literal $\neg a$ is selected in a node n, the tree $T(\leftarrow Q)$ will contain a child of n for each possible dual of the set $expl_{ICL}(a)$. By the inductive hypothesis, $expl_{ICL}(a)$ is a set of explanations that is covering with respect to a. Let $c(n, a)$ be the set of children c of n such that $\kappa(c) \subseteq \sigma$. By Lemma 3.5, $c(n, a)$ is not empty. Therefore by continuing the visit of $T(\leftarrow Q)$ from the nodes of $c(n, a)$ we will eventually get to a leaf l such that $\sigma \supseteq \kappa(l)$ ■

Thus, $expl_{ICL}(Q)$ is a finite set of finite composite choices that are explanations for Q and $expl_{ICL}(Q)$ is covering with respect to Q. Therefore, Definition 3.4 is well defined and equation 3.6 holds.

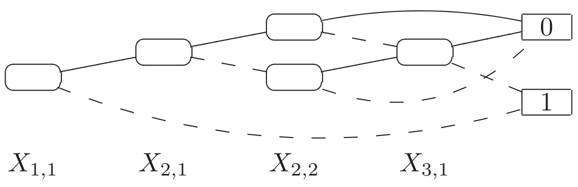


FIG. 5. BDD representing function 5.1

5 Making Explanation Incompatible with Decision Diagrams

In [8] the authors proposed to use Binary Decision Diagrams (BDDs) [3] for making the explanations for queries in the ProbLog language mutually incompatible. A BDD represents a Boolean function $f(\mathbf{X})$ on a set of Boolean variables $\mathbf{X}$ by means of a rooted graph that has one level for each variable. Each node has two children, one corresponding to value 0 (0-child) of the variable associated to the level of the node, and one corresponding to value 1 (1-child) of the variable. The leaves store either 0 or 1. Given values for all the variables, a BDD allows to compute the value of the function by traversing the graph starting from the root and returning the value associated to the leaf that is reached.

BDD software packages take as input a function $f(\mathbf{X})$ and incrementally build the diagram so that isomorphic portions of it are merged, possibly changing the order of variables if useful. This often allows the diagram to have a number of nodes much smaller than exponential in the number of variables that a naive representation of the function would require.

For example, consider the set of binary variables $\mathbf{X} = \{X_{1,1}, X_{2,1}, X_{2,2}, X_{3,1}\}$. The function $f(\mathbf{X})$ defined by

$$f(\mathbf{X}) = \overline{X_{1,1}} \vee (\overline{X_{2,1}} \wedge X_{2,2} \wedge \overline{X_{3,1}}) \vee (X_{2,1} \wedge \overline{X_{2,2}} \wedge \overline{X_{3,1}}) \tag{5.1}$$

is represented by the BDD of Figure 5, where solid lines connect a node to its 1-child and dashed lines to its 0-child.

In Problog each atomic choice is associated to a binary variable and a BDD is built for the function that returns 1 if the assignment of the variables corresponds to an explanation for the query. Thus the variables of $\mathbf{X}$ are independent random variables of which we know the individual marginal distributions $\{P_0(x)|X \in \mathbf{X}\}$. In this case we can compute the probability that $f(\mathbf{X}) = 1$ with the function *prob_binary* shown in Figure 6.

Differently from [8], in ICL the atomic choices may have cardinality larger than two so we must use a generalization of the approach. In ICL, we associate to each grounding $\chi_i\theta$ of each alternative χ_i a multivalued variable $X_{i,\theta}$ whose domain is the set $\{1, \ldots, n'_i\}$. The Boolean formula

$$f(\mathbf{X}) = \bigwedge_{\kappa \in expl_{ICL}(Q)} \bigvee_{(\chi_i,\theta,j)\in\kappa} (X_{i,\theta} = j) \tag{5.2}$$

returns value 1 iff the assignment of the multivalued variables corresponds to an explanation for Q.

```
function prob_binary(
    inputs : $n$: BDD node
    returns : $P$: probability)
if $n$ is a leaf
    return the value associate to $n$
else
    let $X(n)$ be the variable associated to the level of $n$
    let $c_0, c_1$ be the children of $n$
    return $P_0(X(n)=0) \cdot \mathit{prob\_binary}(c_0) + P_0(X(n)=1) \cdot \mathit{prob\_binary}(c_1)$
return $P$
```

Fig. 6. Function *prob_binary*.

One possibility for representing equation 5.2 is to use Multivalued Decision Diagram [35] that generalize BDDs for variables with more than two values. Another possibility is to represent multivalued variables with binary encoding [17]: if multivalued variable X_i can assume p different values, we use $q = \lceil \log_2 p \rceil$ binary variables $X_{i,1}, \ldots, X_{i,q}$ where $X_{i,1}$ is the most significant bit. The equation $X_i = j$ can be represented with binary variables in the following way

$$X_{i,1} = j_1 \wedge \ldots \wedge X_{i,q} = j_q$$

where $j_1 \ldots j_q$ are bits and $j_1 \ldots j_q$ is the binary representation of $j-1$. Once we have transformed all multivalued equations into Boolean equations we can build the BDD.

In our implementation, we decided to use the latter approach because highly efficient packages are available for processing BDD.

Example 5.1
Consider the program of Example 4.4. Let us call X_1, X_2 and X_3 the variables associated to the alternatives $\chi_1\emptyset, \chi_2\emptyset$ and $\chi_3\emptyset$ respectively. The domains of X_1, X_2 and X_3 are $\{1,2\}$, $\{1,2,3\}$ and $\{1,2\}$ respectively. The query a is true in a world w iff the following function

$$f_1(\mathbf{X}) = (X_1 = 1) \vee (X_2 = 2) \wedge (X_3 = 1) \vee (X_2 = 3) \wedge (X_3 = 1) \tag{5.3}$$

returns value 1. In order to represent this function with binary variables, we use one variable $(X_{1,1})$ for X_1, two $(X_{2,1}, X_{2,2})$ for X_2 and one $(X_{3,1})$ for X_3. Equation 5.3 can be represented with binary variables as follows

$$f_2(\mathbf{X}) = (X_{1,1} = 0) \vee [(X_{2,1} = 0) \wedge (X_{2,2} = 1) \wedge (X_{3,1} = 0)] \vee [(X_{2,1} = 1) \wedge (X_{2,2} = 0) \wedge (X_{3,1} = 0)] \tag{5.4}$$

which is equivalent to Equation 5.1 and yields the BDD of Figure 5.

The algorithm shown in Figure 7 computes the probability of a formula on multivalued variables encoded by a BDD. It consists of two mutually recursive functions, *prob* and *prob_bool*. *prob* is called in order to take into account a new multivalued variable and *prob_bool* is called to consider the individual binary variables. In particular, *prob*(n) returns the probability of node n while the calls of *prob_bool* build a binary tree with a level for each bit of the multivalued variable X, so that the leaf calls of *prob_bool* identify a single value x of X and are called with a node whose binary variable belongs to the next multivalued variable. These call then *prob* to compute the probability of the subgraph and return the product of the result by the probability associated to value x. The intermediate *prob_bool* calls sum up these partial results and return them to the parent *prob* call. Note that *prob_bool* builds a full binary tree for a multi-valued variable even if there is not a node for every binary variable (for example, because the result is not influenced by the value of one bit).

In order for this algorithm to be correct, it is necessary that the binary variables representing the same multivalued variable are kept together and in the order from the most significant to the least significant bit when building the diagram. This is achieved by employing the feature offered by BDD packages of specifying groups of variables that must be kept together and in a given order. For every multivalued variable, we enclose in one such group all the binary variables associated to it.

As in [8], *prob* is optimized by storing, for each computed node, the value of its probability, so that if the node is visited again the probability can be retrieved.

We are now ready to present the system PICL for computing the probability of a query Q from an ICL theory: PICL (shown in Figure 8) finds the set $expl_{ICL}(Q)$ of the all the explanations for Q, builds a BDD from $expl_{ICL}(Q)$ and then computes the probability of Q from the BDD.

If the conditional probability of a query Q given another query E must be computed, rather then computing $P(Q \wedge E)$ and $P(E)$ separately, an optimization can be done: first all successful SLDNFICL-derivations for E are identified and then all successful SLDNFICL-derivations for Q starting from an explanation for E are found, as shown in Figure 9.

6 Implementation

The explanations for a query are found by means of a Prolog meta-interpreter that has two extra arguments for storing the current explanation in input and output. The meta-interpreter explores the SLDNFICL-forest depth-first, accumulating the atomic choices taken in the current input explanation. When an empty goal is reached, the current input explanation is returned as output. In order to find all the explanations for a goal, the meta-interpreter is invoked within a `findall/3` call.

Once all the explanations have been found, they are given as input to an external predicate *compute_prob* that contains the calls to *build_BDD* and *prob*. *compute_prob* is defined by means of C code and takes care of the BDD manipulation by calling the library CUDD[3]. *compute_prob* cycles over the explanations, over the individual atomic choices of explanations and over each binary variable representing atomic choices. The Boolean operations on BDD are translated into CUDD API calls.

PICL was implemented using the Yap Prolog[4] and gcc compilers. PICL source code is part of the `cplint` reasoning suite that is available in the development version of Yap Prolog. The user manual can be found at `http://www.ing.unife.it/software/cplint/`.

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

[4] `http://www.ncc.up.pt/~vsc/Yap/`

```
function prob(
    inputs : $n$ : BDD node,
    returns : $P$ : probability of the formula)
if $n$ is the 1-terminal then return 1
if $n$ is the 0-terminal then return 0
let $mVar$ be the multivalued variable
    corresponding to the Boolean variable associated to $n$
$mVar$ is represented by the following structure
{   int $nBit$, // number of bits for the variable
    int $bVar[nBit]$, // vector of binary variables
    int $nVal$, // number of values
    float $Pr[nVal]$ // vector of probabilities
 }
$P := prob\_bool(n, 0, 1, mVar)$
return $P$

function prob_bool(
    inputs : $n$ : BDD node,
        $value$ : index of the value of the multivalued variable in the $mVar.Pr$
            vector
        $posBVar$ : position of the Boolean variable, 1 most significant
        $mVar$ : multivalued variable
    returns : $P$ : probability of the formula)
if $posBVar = mVar.nBit + 1$ then // the last bit has been reached
    let $p_{value} = mVar.Pr[value]$
    // $p_{value}$ is the probability associated with value of index $value$ of
        variable $mVar$
    return $p_{value} \times prob(n)$
else
    let $b_n$ be the Boolean variable associated to $n$
    let $b_p = mVar.bVar[posBVar]$
    //$b_p$ is the Boolean variable in position $posBVar$ of $mVar$
    if $b_n = b_p$
        // variable $b_p$ is present in the BDD
        let $h$ and $l$ be the 1- and 0-children of $n$
        shift left 1 position the bits of $value$
        $P := prob\_bool(h, value + 1, posBVar + 1, mVar)+$
            $prob\_bool(l, value, posBVar + 1, mVar)$
        return $P$
    else
        // variable $b_p$ is absent from the BDD
        shift left 1 position the bits of $value$
        $P := prob\_bool(n, value + 1, posBVar + 1, mVar)+$
            $prob\_bool(n, value, posBVar + 1, mVar)$
        return $P$
```

FIG. 7. Functions *prob* and *prob_bool*.

```
function PICL(
    inputs : $Q$: ground query
    returns : $P(Q)$: probability of the query)
compute $expl_{ICL}(Q)$
$BDD := build\_BDD(expl_{ICL}(Q))$
let $n$ be the root node of $BDD$
return $prob(n)$

function $build\_BDD$(
    inputs : $K$: set of composite choices
    returns : $BDD$: BDD)
let $BDD = false$
for all $\kappa \in K$
    $BDDterm = true$
    for all $(\chi, \theta, j) \in \kappa$
        let $X_k$ be the multivalued variable corresponding to $\chi\theta$
        let $j_1, \ldots, j_q$ be the binary representation of $j - 1$
        $BDDterm := BDDterm \wedge X_{k,1} = j_1 \wedge \ldots \wedge X_{k,q} = j_q$
    $BDD := BDD \vee BDDterm$
return $BDD$
```

Fig. 8. The PICL algorithm.

```
function PICL_COND(
    inputs : $Q$ : ground query
        $E$ : ground evidence
    returns : $P(Q|E)$ : probability of $Q$ given $E$)
compute $expl_{ICL}(E)$
find all the sets $K_{QE}$ obtained in the following way
    select a $\kappa_E$ from $expl_{ICL}(E)$
    find a successful derivation from $(\leftarrow Q, \kappa_E)$ to $(\leftarrow, \kappa_{QE})$
let $K_{QE}$ be the set of explanation for such derivations
$BDD_E := build\_BDD(expl_{ICL}(E))$
$BDD_{QE} := build\_BDD(K_{QE})$
let $n_E$ and $n_{QE}$ be the root nodes of $BDD_E$ and $BDD_{QE}$
$P(E) := prob(n_E)$
$P(QE) := prob(n_{QE})$
if $P(E) \neq 0$ then
    return $\frac{P(QE)}{P(E)}$
else
    return $undefined$
```

Fig. 9. The PICL_COND algorithm.

Ailog2[5] performs inference on ICL and is compared with PICL in the experiments of Section 8. This is possible because Ailog2 is also able to handle correctly modularly acyclic programs. In fact, it uses a meta-interpretation approach very similar to the one of PICL: explanations are computed depth-first and, when an and $L_1 \wedge L_2$ has be considered, L_1 is resolved first and L_2 later. The explanation for L_1 is used as the starting explanation in the derivation of L_2, thus avoiding the infinite recursion problem show in Example 4.3.

Once all the explanations have been found, Ailog2 makes them mutually incompatible by means of the iterative algorithm shown in Figure 2.

7 Discussion and Related Works

[7] introduced the distribution semantics for probabilistic logic programs. The paper discusses a language that is essentially ICL without function symbols. The paper also proposes an algorithm for performing inference that first computes explanations, as we do, and then computes the probability of the goal by applying the inclusion--exclusion formula. As already noted for ICL, the inclusion-exclusion formula works only for very small programs because it requires the computation of the probability of every possible conjunction of explanations.

pD [9] presents a Datalog language very similar to ICL, in which probability distributions are defined over the rules. The inference algorithm it proposes computes all the explanations for a goal and then uses the inclusion-exclusion formula for computing the probability of the disjunction of the explanations. As already noted for ICL and [7], this approach is infeasible in practice.

PRISM [32] is a language that follows the distribution semantics and assigns probabilities to ground facts. The algorithm proposed for inference requires the bodies of rules for the same ground atom to be mutually exclusive, i.e., no couple of bodies can be true in the same world, thus making it inapplicable to the graph problems considered in Section 8.

In Logic Programs with Annotated Disjunctions (LPADs) [38] the clauses can be disjunctive with atoms in the head annotated with a probability. A functor-free LPAD defines a probability distribution over the normal programs obtained by selecting one atom in the head of the grounding of each disjunctive clause. For acyclic functor-free programs, LPADs has been shown to have a close relationship with ICL [37]: an LPAD can be converted into an ICL theory and vice versa in a way that preserves the semantics. A semantics for LPADs with function symbols has not been defined yet.

P-log [5] is a recent formalism for introducing probability in Answer Set Programming (ASP). P-log has a rich syntax that allows to express a variety of stochastic and non-monotonic information. The semantics of a P-log program T is given in terms of a translation of it into an Answer Set program $\pi(T)$ whose stable models are the possible worlds of T. A probability is then assigned to each stable model and the probability of a query is given, as for other distribution semantics languages, by the sum of the probabilities of the possible worlds where the query is true. P-log differs from the languages mentioned before because the possible worlds are generated not only because of stochastic choices but also because of disjunctions and unstratified

[5] `http://www.cs.ubc.ca/~poole/aibook/code/ailog/ailog2.html`

negations appearing in the logical part of a P-log program. As a consequence, the distribution obtained by multiplying all the probability factors of choices that are true in a stable model is not normalized. In order to get a probability distribution over possible worlds, the unnormalized probability of each stable model must be divided by the sum of the unnormalized probabilities of each possible world.

The syntax of probabilistic assertions and the possibility of using disjunction and non-stratified negation makes P-log able to express very subtle nuances. Moreover, special attention is devoted to the representation of the effects of actions according to the most recent accounts of causation [19]. However, function symbols are generally not dealt with by the semantics since it is required that the number of possible worlds is finite.

The Plog system[6] computes the probability of a query by translating the P-log program into $\pi(T)$, by grounding it, by running an ASP system and by elaborating the output to return the desired value. The need to compute all stable models of a program may be problematic in domains described by a large ground program.

[8] first proposed the use of BDDs for programs with a distribution semantics. ProbLog and ICL differ significantly in their semantics: in ProbLog, the instances are obtained by selecting clauses directly from the theory rather than from the grounding of the theory. In order to apply the ProbLog inference algorithm to ICL, it would be necessary to first ground the ICL theory, which is infeasible for non-trivial theories.

The inference algorithm of ProbLog is also capable of computing the probability of the query in an approximate way, returning an upper and a lower bound of the probability. This involves the use of iterative deepening: the SLD-tree is built only up to a given depth d and at each iteration the value of d is incremented. At the end of each iteration, there is a set of explanations for successful derivations *Successful* and a set of explanations for still open derivations *Open*. The true probability $P(Q)$ of a query is such that

$$P(F_1) \leqslant P(Q) \leqslant P(F_1 \vee F_2)$$

where F_1 (F_2) is the formula corresponding to *Successful* (*Open*) The cycle terminates when $P(F_1 \vee F_2) - P(F_1) \leqslant \epsilon$, where ϵ is a user defined precision. Following a similar approach, we plan to develop an approximate inference algorithm for ICL in the future.

Ailog2 [23] implements an approximate algorithm that uses a depth bound on the derivations and a probability threshold on the explanations: each derivation is cut if the depth bound is overcome or when the probability of the current explanation falls below the threshold. The probabilities of all the explanations for cut derivations are then summed up and considered as an error to the probability of the uncut explanations computed by making them disjoint. An upper bound is obtained by summing the probability of uncut explanations with the error. This method is very efficient but it usually produces a loose upper bound that is often greater than 1.

[28] proposed an algorithm for performing inference with LPADs where a modification of SLDNF-resolution is used for computing explanations in combination with BDDs. PICL in this paper draws inspiration from [28] and applies the ideas presented there to ICL. In the future we plan to extend the results in this paper also to LPADs, by defining a semantics for LPADs with function symbols.

[6] `http://www.cs.ttu.edu/~wezhu/`

In [26, 27, 25] we present an algorithm for performing inference on non-modularly acyclic functor-free LPADs. The algorithm, called SLGAD resolution, is based on SLG resolution for normal programs under the well-founded semantics and uses tabling to avoid redundant computations and to avoid infinite loops. This approach can be extended in two ways: on one side, for reasoning on non-modularly acyclic ICL theories, on the other side, for reasoning on non functor-free LPADs.

8 Experiments

In this section we compare the performances of PICL with those of Ailog2 and of Plog on a number of inference problems. Both PICL and Ailog2 were ran under Yap Prolog. Ailog2 was ported to Yap starting from the source code available on the web. For Plog we used the freely available implementation. All the experiments were performed on Linux machines with an Intel Core 2 Duo E6550 (2333 MHz) processor and 4 GB of RAM.

We consider problems of computing the probability of paths in biological networks (Section 8.1) and social networks (Section 8.2). Moreover, we consider the encoding of three games of dice (Section 8.3).

8.1 Biological Networks

The biological networks proposed by [34] contain nodes that represent biological entities such as genes, proteins, tissues, organisms, biological processes, and molecular functions. An edge between two nodes indicates a relationship between the entities and is annotated with a probability that expresses the strength of the association. These networks can be used to discover indirect relationships between two concepts, such as the probability that a gene is involved in a disease.

We consider the network used in [8] to evaluate the performance of the ProbLog interpreter. It is built around four Alzheimer genes and contains 11530 edges and 5220 nodes.

The probability of an indirect association was computed with the following ICL theory

```
path(X,Y):-path(X,Y,[X],Z).
path(X,Y,V,[Y|V]):- arc(X,Y).
path(X,Y,V0,V1):- X\==Y, arc(X,Z), append(V0,_,V1),
     \+ member(Z,V0), path(Z,Y,[Z|V0],V1).
arc(X,Y):- edge(X,Y).
arc(X,Y):- edge(Y,X).
```

plus a disjoint assertion of the form

```
disjoint([edge(a,b):p]).
```

for each edge in the graph between nodes `a` and `b` and with a probability of `p`. This program is syntactically very similar to the one used in [8] and it has also the same meaning.

We used the method and the datasets of [8]: we query the probability that the two genes HGNC_620 and HGNC_983 are related from subnetworks $G_1, G_2, \ldots G_n$

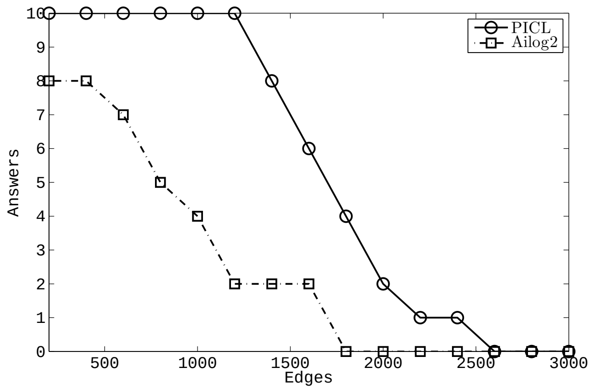


FIG. 10. Number of successes for the biological graph experiments.

extracted from the complete network by subsampling. They contain 200, 400, ..., 5000 edges respectively and are such that $G_1 \subset G_2 \subset \ldots \subset G_n$. Subsampling was repeated 10 times and each G_1 is such that there exists at least one path between the two genes.

We run all the algorithms on the graphs in sequence starting from G_1 and we stopped after one day or at the first graph for which the program ended for lack of memory. Ailog2 was run with a configuration that avoids approximation. This is achieved by setting the depth bound to 10^8 and the probability threshold to 10^{-20000} .

Figure 10 shows the number of graphs for which the computation succeeded. PICL is able to solve the highest number of problems for all graph sizes. The maximum size for which PICL succeeds for all samples is 1200 edges, while the maximum size for which it succeeds on at least one graph is 2400. The ProbLog inference algorithm succeeded on graphs with sizes from 1400 to 4600 edges, but it uses an approximate inference algorithm, see Section 7, while PICL is exact.

Figures 11 shows the average CPU time in seconds as a function of the number of edges. The average is computed over all the samples on which each algorithm was successful. In this figure and in all the following ones the time on the Y axis is in logarithmic scale. PICL solved each problem faster than Ailog2. This is more clearly showed by Figure 12 in which the CPU time is averaged only over the graphs over which both PICL and Ailog2 succeed: Ailog2 is close to PICL in Figure 11 for the sizes 1200 and 1400 because PICL solves more problems, as is shown in Figures 10 and 12.

Figure 13 shows the CPU time for PICL separated into the time spent building explanations and the time spent elaborating BDDs: in all cases the first dominates the latter, thus suggesting that, for the networks on which there is success, building the BDD is relatively easy.

Plog was also applied to this dataset. The following program was used

```
bool={t,f}.
node={a,b,c,...}.
edge: node,node -> bool.
```

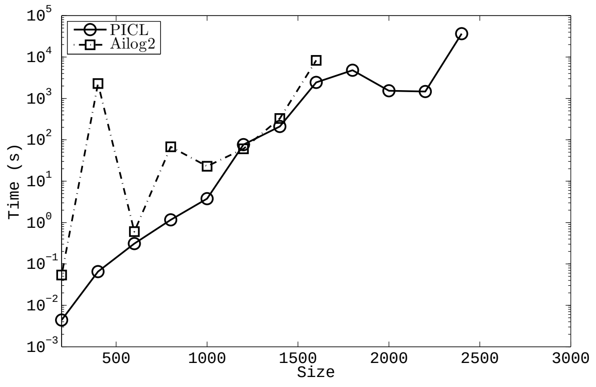


Fig. 11: CPU times for the biological graph experiments, averaged over every sample on which the computation succeeded.

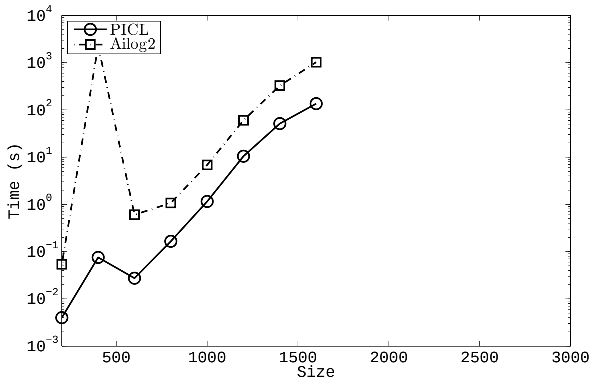


Fig. 12: CPU times for the biological graph experiments, averaged over every sample on which PICL and Ailog2 succeeded.

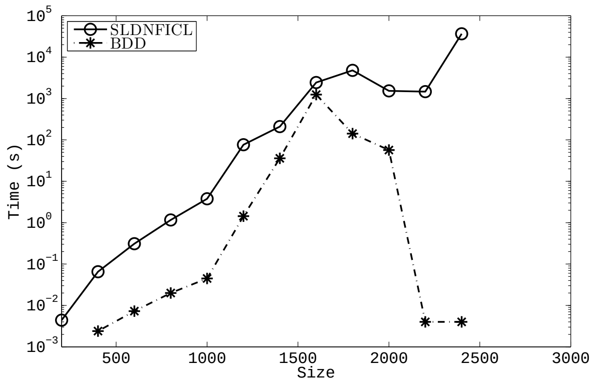


Fig. 13: CPU times for the biological graph experiments for resolution and BDD manipulation

```
4  #domain node(X),node(Y),node(Z).
5  path(X,Y):- arc(X,Y).
6  path(X,Y):- arc(X,Z), path(Z,Y).
7  arc(X,Y):-  edge(Y,X,t).
8  arc(X,Y):-  edge(X,Y,t).
9  [r(a,b)] random(edge(a,b)).
10 [r(a,b)] pr(edge(a,b,t))=5150/10000.
...
```

Line 1 states that `bool` is a type with two constants `t` and `f`. Line 2 defines the `node` type with as many constants as there are nodes in the network (here indicated with `a`, `b`, `c`,...). Line 3 states that `edge` is a function from a couple of nodes to a Boolean. It can also be seen as a three-argument predicate, where the last argument is the value of the function. Line 4 assigns the `node` domain to every variable with names `X`, `Y` or `Z` that appear in the program. This is necessary for generating the correct grounding. Lines 5, 6, 7 and 8 define the `path/2` relation: note that there is no need to keep track of the current path since the models of the program are computed bottom-up and so there is no danger of going into a loop.

Lines 9 and 10 contain the probabilistic part of the program. Line 9 states that `edge/2` applied to the couple of nodes `(a,b)` is a function that takes a random value from its range. The range was defined in line 3. Line 10 defines the probability of `edge(a,b)` of taking the value `t` to be 5150/10000 or 0.515. There will be a couple of declarations of this type for every disjoint assertion `disjoint([edge(a,b):p])`.

Plog was asked to compute the probability of a path between the two genes over the G_1 networks in the ten samples. After 24 hours of computation time the elaboration was stopped. In no sample Plog was able to return an answer for the query. In five samples, the elaboration was stopped before the computation of the stable models was complete. In the other samples, all the stable models were computed but resulted in such a high number of models that the following post-processing phase did not complete: the average size of the text file holding the stable models was about 5 GB. This is due to the high number of constants in the domain that yield a very large ground program.

8.2 Social Networks

We consider a kind of link-based classification problem [16] in social networks in which we want to predict the classification of a node given the labels of a set of connected nodes. Such a problem can be described by an ICL theory containing the clause

```
class(Y,C):-path(X,Y),class(X,C).
```

plus (possibly probabilistic) facts for `class/2` for the labeled nodes and a definition of path. The definition of path we consider in this experiment differs from the one given in Section 8.1 because we want to have uncertainty also on the spreading of the label from a node to another: even if an edge exists between two nodes, the propagation of the label is not certain. Thus, the definition of path we use is

```
path(X,Y):- path(X,Y,[X],Z).
path(X,Y,V,[Y|V]):- edge(X,Y), spread(X,Y,V).
path(X,Y,V0,V1):- edge(X,Z), append(V0,_,V1),
     \+ member(Z,V0),path(Z,Y,[Z|V0],V1), spread(X,Y,Z,V0,V1).
```

plus the disjoint assertions

```
disjoint([spread(X,Y,V):0.8]).
disjoint([spread(X,Y,Z,V0,V1):0.8]).
```

In this way we assign a probability of 0.8 to the fact that a label spreads from a node to a neighboring node. Finally, we have disjoint assertions of the form

```
disjoint([edge(a,b):0.8]).
```

for each edge in the network between nodes `a` and `b` to model an edge with probabilistic existence.

We considered the Citeseer, Cora and WebKB social networks[7]. Each node in Citeseer and Cora is a research paper and the edges represent citations. Each node in WebKB is a web page from a computer science department of a US University and the edges represent hypertext links. WebKB is divided into four independent subnetworks: Cornell, Texas, Washington and Wisconsin.

The number of edges in the networks is: 4732 for Citeseer, 5429 for Cora, 304 for Cornell, 329 for Texas, 446 for Washington and 530 for Wisconsin.

On these datasets we tested marginal queries, in order to verify if the results obtained on the biological networks are confirmed. Moreover, we tested also conditional queries.

For both marginal and conditional queries, we considered all the couples of nodes that have from 1 to 100 different paths between them. Then, for each number of paths, 100 couples are sampled. If, for a number of paths, there are less than 100 couples, all of them are considered. The marginal query for a couple (a, b) is $path(a, b)$.

For conditional queries, we selected only the couples of nodes whose longest path is longer than 4. For each couple (a, b), we picked one couple (c, d) where c and d are nodes selected randomly from the first and the second half respectively of the longest path between a and b. Then, supposing NP is the number of paths between a and b, we selected other $\lfloor (NP - 1) * 0.7 \rfloor$ couples (c, d), with c and d from the first and the second half respectively of a randomly chosen path between a and b. Let E be the set of the atoms $path(c, d)$ for all such (c, d) couples. The query we consider is $path(a, b)$ given E.

All the algorithms were executed on the marginal and conditional queries in order of increasing number of paths between the nodes in the query. The runs were interrupted after one day or at the first query for which the program ended for lack of memory.

The time in seconds required by each query was averaged over the total number of queries with the same number of paths. The plots of time against the number of paths for marginal queries are shown in figures 14, 15, 16, 17, 18 and 19 for the Citeseer, Cora, Cornell, Texas, Washington and Wisconsin networks respectively. The corresponding figures for conditional queries are 20, 21, 22, 23, 24 and 25. An algorithm missing from a legend means that it succeeded in 0 problems.

Plog was applied with the following program

[7] Available at `http://www.cs.umd.edu/~sen/lbc-proj/LBC.html`

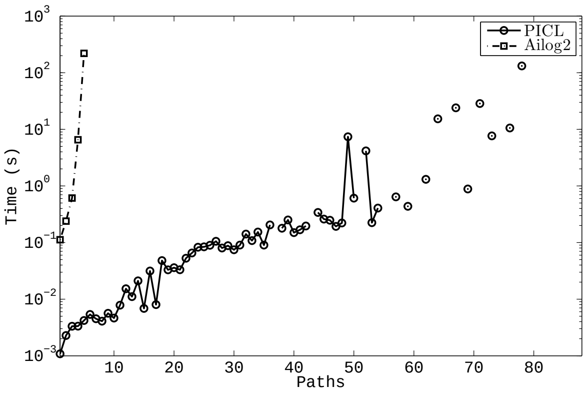


FIG. 14. Average CPU time for marginal queries in the Citeseer network.

```
bool={t,f}.
node={a,b,c,...}.
edge: node,node -> bool.
spread1: node,node -> bool.
spread2: node,node,node -> bool.
#domain node(X).
#domain node(Y).
#domain node(Z).
path(X,Y):- edge(X,Y), spread1(X,Y,t).
path(X,Y):- edge(X,Z), path(Z,Y), spread2(X,Y,Z,t).
[r1(X,Y)] random(spread1(X,Y)).
[r1(X,Y)] pr(spread1(X,Y,t))=8/10.
[r2(X,Y,Z)] random(spread2(X,Y,Z)).
[r2(X,Y,Y)] pr(spread2(X,Y,Z,t))=8/10.
[r(a,b)] random(edge(a,b)).
[r(a,b)] pr(edge(a,b,t))=5150/10000.
...
```

in which `spread1` and `spread2` are two random functions that model the spreading of labels.

We did not succeed in getting an answer for any of the networks and of the queries: for Citeseer and Cora Plog returned the message “Error in input”, while for the WebKB networks we got a segmentation fault. This is probably due to the size of the problems which is even higher than that of the biological network programs.

The results on marginal queries confirm those of the biological network and show an even larger difference between PICL and Ailog2. The latter shows a dramatic increase of the computation time for queries regarding nodes with about five paths between them. The experiments on the conditional queries show that PICL still has the best performances.

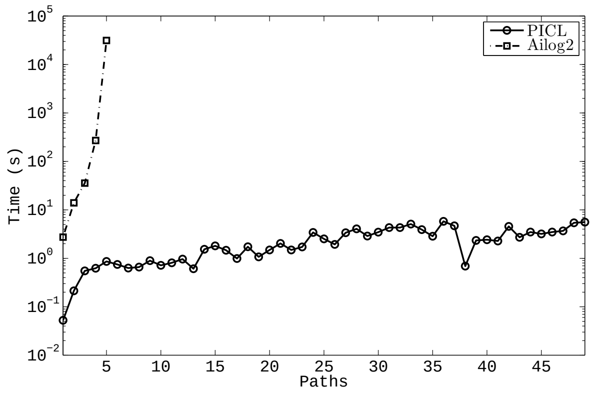


FIG. 15. Average CPU time for marginal queries in the Cora network.

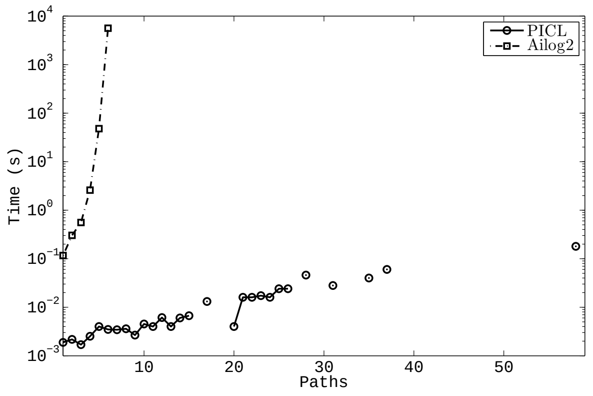


FIG. 16. Average CPU time for marginal queries in the Cornell network.

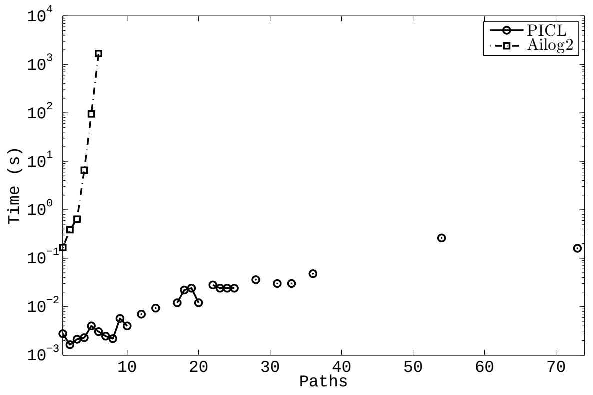
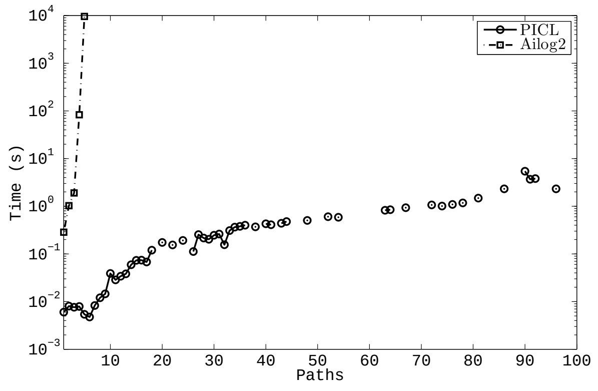


FIG. 17. Average CPU time for marginal queries in the Texas network.



S

FIG. 18. Average CPU time for marginal queries in the Washington network.

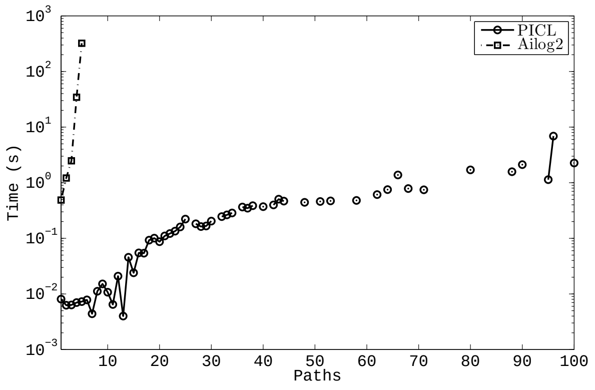


FIG. 19. Average CPU time for marginal queries in the Wisconsin network.

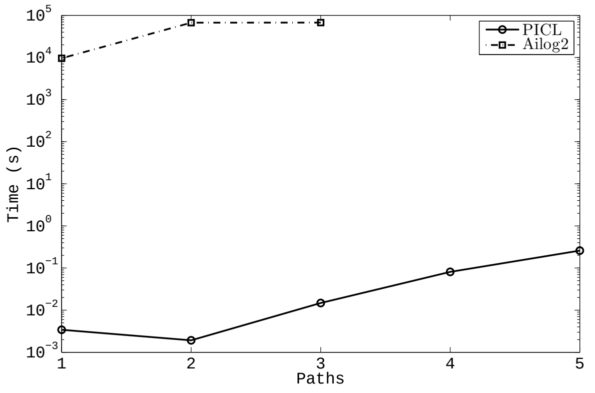


FIG. 20. Average CPU time for conditional queries in the Citeseer network.

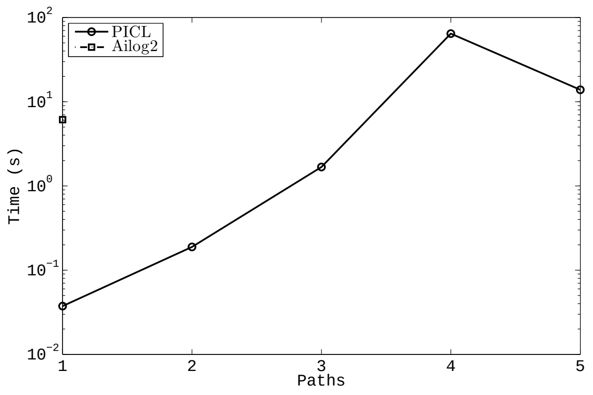


FIG. 21. Average CPU time for conditional queries in the Cora network.

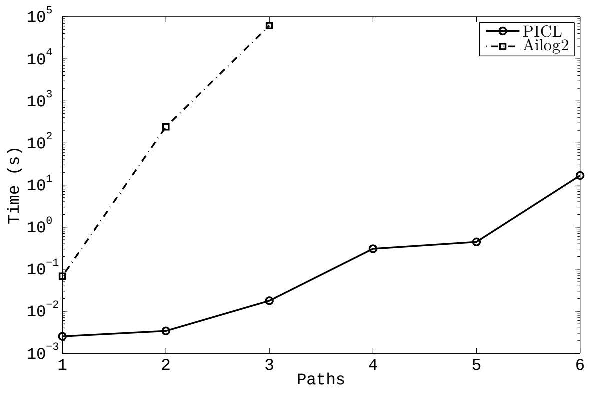


FIG. 22. Average CPU time for conditional queries in the Cornell network.

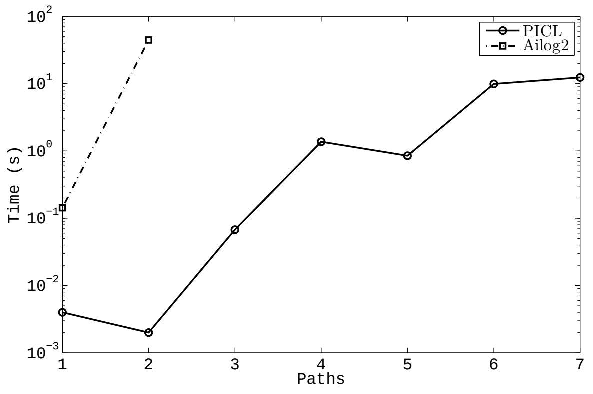


FIG. 23. Average CPU time for conditional queries in the Texas network.

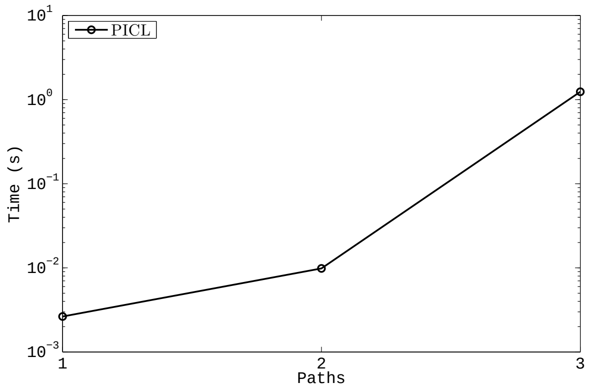


FIG. 24. Average CPU time for conditional queries in the Washington network.

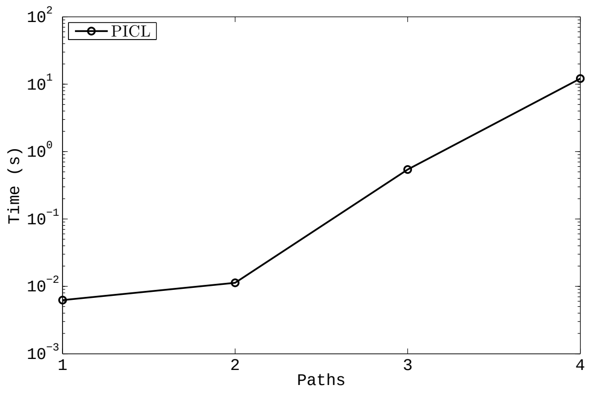


FIG. 25. Average CPU time for conditional queries in the Wisconsin network.

8.3 Games of dice

PICL, Ailog2 and Plog were also tested on programs encoding games of dice similar to the one presented in [38]. The player of the games repeatedly throws a die and stops only when a certain subset of the faces comes up. Time is indicated with a non-negative integer and the game starts at time 0. We want to compute the probability that a certain face is obtained at a certain time point.

The predicate `on(T,F)` indicates that the die was thrown at time `T` and face `F` was obtained. The predicate `face(T,F)` represents a random choice among the faces of the die at time `T`. For a die with three faces, the problem can be encoded with the following program (`die1`)

```
on(0,F):-face(0,F).
on(T,F):-T>0, face(T,F), T1 is T-1, on(T1,F1),\+ on(T1,3) .
disjoint([face(N,1):0.333333,face(N,2):0.333333,face(N,3):0.333333]). .
```

This program states that, at time 0, one of the three faces is obtained with equal probability. At time `T`, a face is obtained with equal probability if, at the previous time

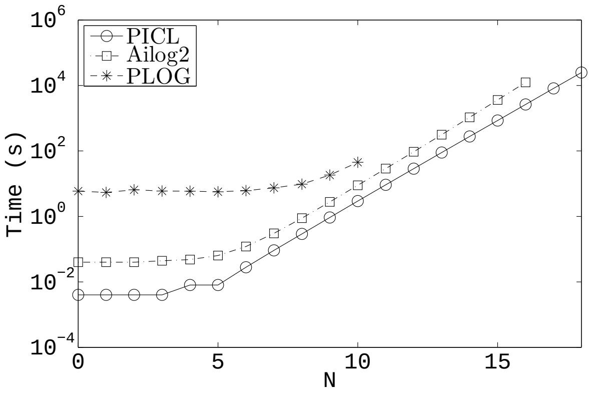


FIG. 26. Execution times for a three sided die

point, the die was thrown (`on(T1,F)`) and face 3 was not obtained (`\+ on(T1,3)`).

The Plog program that encodes the above problem is

```
1 score={1..3}.
2 time={0..2}.
3 #domain time(T),score(F),score(F1).
4 face: time -> score.
5 [r(T)] random(face(T)).
6 on(0,F):- face(0,F).
7 on(T,F):- gt(T,0), face(T,F), on(minus(T,1),F1), not on(minus(T,1),3).
```

The indication of the time domain in line 2 is used in order to generate the grounding: the statement in line 3 asserts that `T` is a variable with domain `time`, so the grounder will replace `T` with all the possible values of `time`. Line 4 defines `face` as a function from time to score, while line 5 expresses that `face` is random. Since no indication is given regarding the probability distribution of the values in the range of `face`, the distribution is assumed to be uniform.

We query the probability of `on(T,1)` for increasing values of `T`. For PICL and Ailog2 we used the same program for each value of `T`, while for Plog we considered programs with a value of the upper bound of `time` equal to `T`. This was done in order to restrict as much as possible the size of the ground program that is generated by Plog. The execution times of PICL, Ailog2 and Plog for increasing values of `T` are shown in Figure 26. We considered also two other games in which a four-sided die is used: in the first the player stops as soon as he gets 3 or 4 (`die2`), in the second he stops as soon as he gets 4 (`die3`). The execution times for answering the query `on(T,1)` are shown in Figures 27 and 28 respectively. The points absent from the figures correspond to instances for which the algorithm failed.

The figures indicate that PICL is faster than Ailog2 and Plog and is able to solve more complex problems. The only exception to this is `die3` in which Plog is as fast as PICL for `T`=6 and faster for `T`=7. However, for `T`=8 and 9 Plog exhausted the 4Gb available memory and was killed. These results shows that Plog is comparable to PICL when the query is true in most of the possible worlds and also that Plog

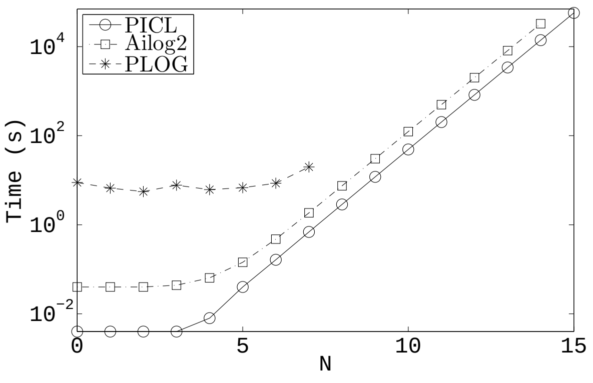


FIG. 27. Execution times for a four-sided die, 3 and 4 to stop

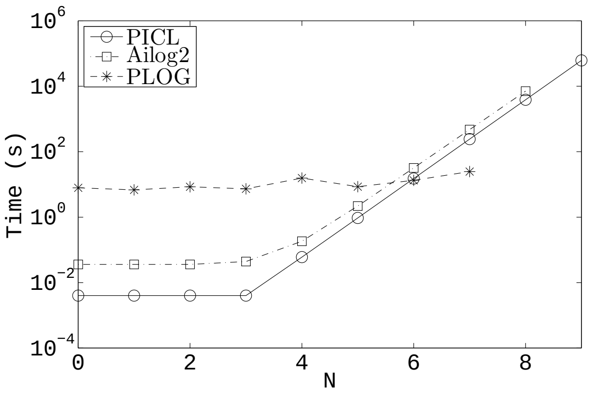


FIG. 28. Execution times for a four-sided die, 3 to stop

requires more memory.

9 Conclusions

We have proposed an extension of the ICL semantics that allows the theories to be modularly acyclic rather than simply acyclic, thus significantly extending the range of problems that can be encoded. The semantics is based on the notion of SLDNFICL--resolution, an extension of SLDNF-resolution that allows to compute explanations for ICL queries in the extended semantics

Moreover, we have presented the system PICL that computes the probability of queries by first computing explanations using SLDNFICL-resolution and then by making them mutually incompatible by using Binary Decision Diagrams.

We have experimentally compared PICL with Ailog2 and Plog on a graph of biological concepts, on social networks and on games of dice. The results show that PICL consistently outperforms Ailog2 and Plog, often by a large margin, both in the case of marginal and conditional queries. The only exception is one case of one dice game in which Plog is the fastest but however fails to solve the two largest problems for which PICL succeeds.

In the future, we plan to develop approximate versions of PICL along the lines of the ProbLog interpreter as discussed in Section 7. Moreover, we plan to consider also non-modularly acyclic programs using the techniques developed in [26, 27, 25]. for LPADs.

References

[1] K. R. Apt and M. Bezem. Acyclic programs. *New Gener. Comput.*, 9(3/4):335–364, 1991.

[2] Krzysztof R. Apt and Kees Doets. A new definition of SLDNF-resolution. *J. Log. Program.*, 18(2):177–190, 1994.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Comput.*, 35(8):677–691, 1986.

[4] R. Carnap. *Logical Foundations of Probability*. University of Chicago Press, 1950.

[5] C.Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. *The. Pra. Log. Program.*, 9(1):57–144, 2009.

[6] K. L. Clark. Negation as failure. In *Logic and Databases*. Plenum Press, 1978.

[7] Evgeny Dantsin. Probabilistic logic programs and their semantics. In *Russian Conference on Logic Programming*, volume 592 of *LNCS*, pages 152–164. Springer, 1991.

[8] L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*, pages 2462–2467, 2007.

[9] Norbert Fuhr. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *J. Am. Soc. Inf. Sci.*, 51(2):95–110, 2000.

[10] H. Gaifman. Concerning measures in first order calculi. *Israel J. Math.*, 2:118, 1964.

[11] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[12] L. Getoor and B. Taskar, editors. *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

[13] Joseph Y. Halpern. An analysis of first-order logics of probability. *Artif. Intell.*, 46(3):311–350, 1990.

[14] K. Kersting and L. De Raedt. Towards combining inductive logic programming and Bayesian networks. In *Inductive Logic Programming*, volume 2157 of *LNAI*. Springer, 2001.

[15] A. N. Kolmogorov. *Foundations of the Theory of Probability*. Chelsea Publishing Company, New York, 1950.

[16] Qing Lu and Lise Getoor. Link-based classification. In *International Conference on Machine Learning*, pages 496–503. AAAI Press, 2003.

[17] D. M. Miller and R. Drechsler. On the construction of multiple-valued decision diagrams. In *IEEE International Symposium on Multiple-Valued Logic*, pages 245–253. IEEE Computer Society Press, 2002.

[18] Stephen Muggleton. Learning stochastic logic programs. *Electron. Trans. Artif. Intell.*, 4(B):141–153, 2000.

[19] J. Pearl. *Causality*. Cambridge University Press, 2000.

[20] David Poole. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Gener. Comput.*, 11(3):377–400, 1993.

[21] David Poole. Probabilistic horn abduction and bayesian networks. *Artif. Intell.*, 64(1):81–129, 1993.

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

[23] David Poole. Abducing through negation as failure: stable models within the independent choice logic. *J. Log. Program.*, 44(1-3):5–35, 2000.

[24] Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors. *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *LNCS*. Springer, 2008.

[25] F. Riguzzi. SLGAD resolution for inference on Logic Programs with Annotated Disjunctions. *Journal of Algorithms in Logic, Informatics and Cognition.* , in press.

[26] F. Riguzzi. Inference with logic programs with annotated disjunctions under the well founded semantics. In *International Conference on Logic Programming*, number 5366 in LNCS, pages 667–771. Springer, 2008.

[27] F. Riguzzi. The SLGAD procedure for inference on Logic Programs with Annotated Disjunctions. In Marco Gavanelli and Toni Mancini, editors, *RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, number 451 in CEUR Workshop Proceedings. Sun SITE Central Europe, 2009.

[28] Fabrizio Riguzzi. A top down interpreter for LPAD and CP-logic. In *Congress of the Italian Association for Artificial Intelligence*, number 4733 in LNAI, pages 109–120. Springer, 2007.

[29] Kenneth A. Ross. Modular acyclicity and tail recursion in logic programs. In *Principles of Database Systems*, pages 92–101, New York, NY, USA, 1991. ACM.

[30] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*, pages 715–729, 1995.

[31] Taisuke Sato and Yoshitaka Kameya. Prism: A language for symbolic-statistical modeling. In *International Joint Conference on Artificial Intelligence*, pages 1330–1339, 1997.

[32] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. Artif. Intell. Res.*, 15:391–454, 2001.

[33] D. Scott and P. Krauss. Assigning probabilities to logical formulas. In J. Hintikka and P. Suppes, editors, *Aspects of Inductive Logic*. North-Holland, 1966.

[34] Petteri Sevon, Lauri Eronen, Petteri Hintsanen, Kimmo Kulovesi, and Hannu Toivonen. Link discovery in graphs derived from biological databases. In *International Workshop on Data Integration in the Life Sciences*, volume 4075 of *LNCS*, pages 35–49. Springer, 2006.

[35] A. Thayse, M. Davio, and J. P. Deschamps. Optimization of multivalued decision algorithms. In *International Symposium on Multiple-Valued Logic*, pages 171–178, Los Alamitos, CA, USA, 1978. IEEE Computer Society Press.

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

[37] J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, K. U. Leuven, 2003.

[38] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, volume 3131 of *LNCS*, pages 195–209. Springer, 2004.