A Top Down Interpreter for LPAD and CP-logic

Fabrizio Riguzzi

Dip. di Ingegneria – Università di Ferrara – Via Saragat, 1 – 44100 Ferrara, Italy.
fabrizio.riguzzi,@unife.it

Abstract. Logic Programs with Annotated Disjunctions and CP-logic are two different but related languages for expressing probabilistic information in logic programming. The paper presents a top down interpreter for computing the probability of a query from a program in one of these two languages. The algorithm is based on the one available for ProbLog. The performances of the algorithm are compared with those of a Bayesian reasoner and with those of the ProbLog interpreter. On programs that have a small grounding, the Bayesian reasoner is more scalable, but programs with a large grounding require the top down interpreter. The comparison with ProbLog shows that the added expressiveness effectively requires more computation resources.

1 Introduction

Logic Programs with Annotated Disjunctions (LPADs) [9] and CP-logic [8] are two recent formalisms combining logic and probability. They are interesting for the simplicity and clarity of their semantics that makes the reading of their programs very intuitive.

Even if the semantics of these two formalisms were defined in a different way, there exists a syntactic transformation that makes CP-logic programs equivalent to a large subset of LPADs, in particular to the most interesting subset of LPADs.

The LPADs and CP-logic semantics assigns a probability value to logic queries. In this paper, we consider the problem of computing this probability given a program and a query. In particular, we propose a top down interpreter that computes derivations for a query and then computes the probability of the query by using Boolean decision diagrams. The algorithm is based on the top down interpreter for ProbLog presented in [4]. This interpreter is highly optimized and answers queries from programs containing thousands of clauses. Due to the difference between ProbLog and LPADs, it was not possible to use all the optimizations.

Besides the interpreter, we consider an approach that exploits the possibility of translating an LPAD into a Bayesian network shown in [9]. The approach allows the use of Bayesian network reasoners on the problem.

In order to compare the algorithm with the Bayesian approach and with the ProbLog interpreter, we performed a number of experiments on a simple game of dice and on graphs of biological concepts. For the first problem the grounding of the program is small and the Bayesian reasoner is more scalable. For the second problem, the grounding is so large that the Bayesian approach could not be applied. As expected, the size of problems that were successfully solved is smaller than the one of ProbLog.

The paper is organized as follows. In Section 2 we present the syntax and semantics of ProbLog, LPADs and CP-logic, together with the approach for translating them into Bayesian networks. Section 3 describes the top down interpreter for ProbLog presented in [4]. Section 4 presents the top down interpreter for LPADs and CP-logic. In Section 5 we discuss the experiments performed and in Section 6 we conclude and present directions for future work.

2 Preliminaries

A ProbLog program [4] T is a set of clauses of the form

$$\alpha : h \leftarrow b_1, \ldots, b_n \tag{1}$$

where α is a real number between 0 and 1 and h and $b_1, \ldots, b_n$ are atoms.

The semantics of such programs is defined in terms of instances: an instance is a definite logic program obtained by selecting a subset of the clauses and removing the α. Its probability is given by the product of the α factor for all the clauses that are included in the instance and of $1 - \alpha$ for all the clauses not included. The probability $P^T_{PB}(Q)$ of a query Q according to program T is given by the sum of the probabilities of the instances that have the query as a consequence according to the least Herbrand model semantics.

A Logic Program with Annotated Disjunctions T [9] consists of a set of formulas of the form

$$h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \leftarrow b_1, \ldots b_m \tag{2}$$

In such a clause the h_i are logical atoms, the b_i are logical literals and the α_i are real numbers in the interval $[0, 1]$ such that $\sum_{i=1}^{n} \alpha_i = 1$.

The semantics of LPADs is given as well in terms of instances: an instance is a ground normal program obtained by selecting for each clause of the grounding of T one of the heads and by removing the α_i. The probability of the instance is given by the product of the α factors associated with the heads selected. The probability $P^T_{LP}(Q)$ of a formula Q according to program T is given by the sum of the probabilities of the instances that have the formula as a consequence according to the well founded [7] semantics.

A CP-logic program T [8] consists of a set of formulas of the form (2) where it is imposed that $\sum_{i=1}^{n} \alpha_i \leq 1$. The semantics of CP-logic was given in terms of probabilistic processes. However, it was shown in [8] that this semantics, when it is defined, is equivalent to the instance based semantics of the LPAD T' obtained from the CP-logic program T by replacing each clause of the form (2) with the clause

$$h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \vee none : 1 - \sum_{i=1}^{n} \alpha_i \leftarrow b_1, \ldots b_m \tag{3}$$

where *none* is a special atom that does not appear in the body of any clause.

It was shown in [9] that an LPAD can be translated into a Bayesian Logic Program (BLP) preserving the semantics. Since BLP encode Bayesian networks, this provides a way of translating an LPAD into a Bayesian network. This means that we can answer a query by using a Bayesian inference algorithm.

In order to convert an LPAD into a Bayesian network, its grounding must be generated. If the program contains function symbols, the number of different terms is infinite so the user has to provide a finite set of terms for instantiating the clauses, thus restricting the translation to the portion of the ground program of interest to the user. Moreover, the user has also to ensure that the grounded program is acyclic.

Even if the program does not contain function symbols, grounding each clause with every possible constants may generate a very large and cyclic network. Therefore, also in this case the intervention of the user is required.

3 The Top Down Interpreter for ProbLog

In [4] a proof procedure was given for computing the probability of a query Q from a ProbLog program T. The procedure involves the computation of all the possible SLD derivations for Q.

Consider a single derivation d for Q that uses the set of clauses $C_d = \{\alpha_1 : c_1, \ldots, \alpha_k : c_k\}$. Let us assign a Boolean random variable X_i to every clause c_i of T. X_i assumes value 1 if the clause c_i is selected and value 0 if the clause is not selected. The probability

$$P(X_1 = 1 \wedge \ldots \wedge X_k = 1)$$

is the sum of the probabilities of the instances containing these clauses, thus it is the probability of Q if it has only derivation d. Since each clause is independent from the other clauses, the probability above is given by $\prod_{i=1}^{k} \alpha_i$.

If Q has multiple derivations $pr(Q) = \{d_1, \ldots, d_l\}$, then its probability is given by

$$P(\bigvee_{d \in pr(Q)} \bigwedge_{\alpha_i : c_i \in C_d} X_i = 1)$$

Thus the problem of computing the probability of a query is reduced to the problem of computing the probability of a DNF formula. This problem is known to be NP-hard. In order to solve it, the authors of [4] use Binary Decision Diagrams (BDD) [2]. BDD represent a Boolean formula as a binary decision graph: one can compute the value of the function given an assignment of the variables by navigating the graph from the root to a leaf. The nodes of the graph are divided into levels and each level is associated with a Boolean variable. The next node is chosen on the basis of the value of the variable associated to that level: if the variable is 1 the high child is chosen, if the value is 0 the low child is chosen. The leaves are associated either with the value 1 or with the value 0: when we reach a leaf we return the value stored there. For example, a BDD for the Boolean function

$$X_{1,1} = 0 \vee X_{2,1} = 0 \wedge X_{2,2} = 1 \wedge X_{3,1} = 0 \vee X_{2,1} = 1 \wedge X_{2,2} = 0 \wedge X_{3,1} = 0 \quad (4)$$

is represented in Figure 1, where all the $X_{i,j}$ are Boolean variables, high children are reached by solid edges, low children by dashed edges and the leaves are represented by rectangular nodes.

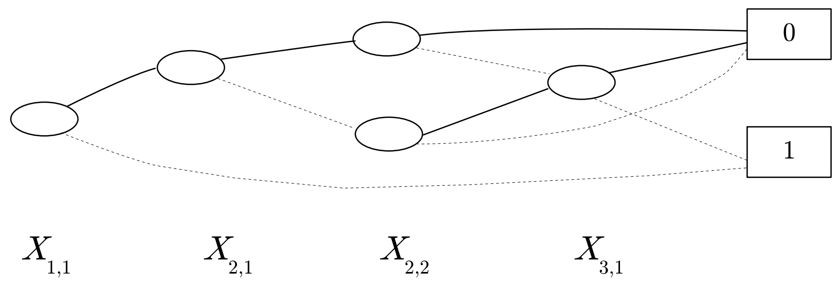


Fig. 1. BDD

A BDD is built by first building a full binary decision tree having 2^n nodes for level n and then simplifying it by merging isomorphic subgraphs until no further reduction is possible. Since the number of reductions depends on the order chosen for the variables, practical BDD tools use sophisticated heuristics for choosing a good order.

Given a BDD of a Boolean formula F, we can easily compute its probability because F can be represented as $F = (X = 1) \wedge F_1 \vee (X = 0) \wedge F_0$ where X is the variable associated to the root of the BDD, F_1 is the formula associated to the high child and F_0 is the formula associated to the low child. Since the two disjuncts are now mutually disjoint, the probability of F can be computed as $P(F) = P(X = 1) \cdot P(F_1) + P(X = 0) \cdot P(F_0)$. The probabilities $P(F_1)$ and $P(F_0)$ can then be computed recursively.

4 A Top Down Interpreter for LPAD and CP-logic

The notion of derivation presented above must be extended in three ways in order to compute the probability of an LPAD query. First, we must take into account the fact that clauses have more than one atom in the head, therefore each clause is not represented by a Boolean variable but by a multivalued variable with as many values as there are atoms in the head. Second, a variable is not associated with a clause but with a grounding of a clause, thus we have different variables for different groundings. Third, the body of LPAD clauses can contain negative literals: roughly speaking, a negative goal is treated by computing all the possible derivations for the goal, by selecting, for each derivation, a grounded clause and by including in the current derivation the clause with a head different from the one used for deriving the negative goal.

The interpreter we present is based on SLDNF and therefore is valid only for programs for which the Clark's completion semantics [3] and the well founded semantics coincide, as for acyclic programs [1].

In the following, we give an algorithmic definition of derivation. We adopt a mixed pseudo code: we use procedural features, such as assignments and functions, and declarative features, such as non-determinism, unification and coroutining (the predicate *dif* in particular).

A *derivation* from (G_1, C_1) to (G_n, C_n) in T of depth n is a sequence

$$(G_1, C_1), \ldots, (G_n, C_n)$$

such that each G_i is a goal of the form $\leftarrow l_1, \ldots, l_k$, C_i is a set of couples that stores the instantiated clauses and the heads used and (G_{i+1}, C_{i+1}) is obtained according to one of the following rules

- if l_1 is built over a built-in predicate, then l_i is executed, $G_{i+i} = \leftarrow l_2, \ldots, l_k$ and $C_{i+1} = C_i$
- if l_1 is a positive literal, then let $c = h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \leftarrow B$ be a fresh copy of a clause of T that resolves with G_i on l_1, let h_j be a head atom of c that resolves with l_1 and let θ be the mgu substitution of l_1 and h_j. For every couple $(c\delta, m) \in C_i$ such that $m \neq j$ and $c\delta$ unifies with $c\theta$, we impose the constraint $\mathit{dif}(c\delta, c\theta)$ so that further instantiations of $c\delta$ or $c\theta$ do not make the two clauses equal. Then $G_{i+1} = r$ where r is the resolvent of $h_j \leftarrow B$ with G_i on the literal l_1 and $C_{i+1} = C_i \cup \{(c\theta, j)\}$.
- if l_1 is a negative literal $\neg a_1$, then let $\mathcal{C}$ be the set of all the sets C such that there exists a derivation from $(\leftarrow a_1, \emptyset)$ to $(\leftarrow, C)$. Then $G_{i+1} = \leftarrow l_2, \ldots, l_k$ and $C_{i+1} =$Select$(\mathcal{C}, C_i)$, where Select is the function shown in Figure 2.

A derivation is *successful* if $G_n = \leftarrow$.

From the set $\mathcal{C}$ of the all the C such that there exists a derivation from $(\leftarrow Q, \emptyset)$ to $(\leftarrow, C)$ we can build the formula

$$F = \bigvee_{C \in \mathcal{C}} \bigwedge_{(c\theta, j) \in C} (X_{c\theta} = j)$$

where $X_{c\theta}$ is the multivalued variable associated to the clause $c\theta$. In order to deal with multivalued variables using BDD, an approach [6] consists in using a binary encoding: 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

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

Fig. 2. Function Select

```
function Select(
    inputs : 𝒞 : C sets for successful derivations of
            the negative goal,
        C_i : current set of used clauses
    returns : C_{i+1} : new set of used clauses)
C_{i+1} := C_i
for each C ∈ 𝒞
    select a (cθ, j) ∈ C// If the program is range restricted,
        // cθ is ground, see the discussion below
    for all δ such that (cδ, j) ∈ C_{i+1} and cδ unifies with cθ
        impose the constraint dif(cδ, cθ)
    perform one of the following operations
        1. select (cδ, m) ∈ C_{i+1} such that m ≠ j and cδ unifies with cθ,
            then C_{i+1} := C_{i+1} \ {(cδ, m)} ∪ {(cθ, m)}
        2. select (cδ, m) ∈ C_{i+1} such that m ≠ j and cδ unifies with cθ,
            then impose the constraint dif(cδ, cθ) and C_{i+1} := C_{i+1} ∪ {(cθ, m)}
        3. select m ≠ j such that ∄cδ (cδ, m) ∈ C_{i+1} with cδ that unifies with cθ,
            then C_{i+1} := C_{i+1} ∪ {(cθ, m)}.
return C_{i+1}
```

where $j_1 \dots j_q$ is the binary representation of j. Once we have transformed all multivalued equations into Boolean equations we can build the BDD.

In order to compute the probability of a multivalued formula represented by a BDD, we exploit the possibility offered by many BDD packages of specifying that the variables belonging to a certain set must be kept together and in the order given when building the diagram. Therefore, for every multivalued variable, we enclose in one such set all the binary variables associated to it.

Consider for example the program

$c_1 = a : 0.1. \quad c_2 = b : 0.3 \vee c : 0.6. \quad c_3 = a : 0.2 \leftarrow \neg b.$

This program has three successful derivations from $(\leftarrow a, \emptyset)$ to $(\leftarrow, C)$. Their C sets are

$C^1 = \{(c_1\emptyset, 0)\}$

$C^2 = \{(c_2\emptyset, 1), (c_3\emptyset, 0)\}$

$C^2 = \{(c_2\emptyset, 2), (c_3\emptyset, 0)\}$

These C sets produce the following formula with multivalued variables

$X_1 = 0 \vee X_2 = 1 \wedge X_3 = 0 \vee X_2 = 2 \wedge X_3 = 0$

where X_i corresponds to $c_i\emptyset$. The formula is then converted into formula (4) that produces the BDD of Figure 1.

The algorithm shown in Figure 3 computes the probability of a multivalued formula encoded by a BDD. It consists of two mutually recursive functions, Prob and ProbBool. The idea is that we call Prob in order to take into account a new multivalued variable and we call ProbBool to consider the individual binary variables. In particular, Prob(n) returns the probability of node n while the calls of ProbBool build a binary tree with a level for each bit of the multivalued variable, so that the last calls of ProbBool (the leaves) identify a single value and are called with a node whose binary variable belongs to the next multivalued variable. Then ProbBool calls Prob on the node to compute the probability of the subgraph and returns the product of the result and the probability associated to the value. The intermediate ProbBool calls sum up these partial results and return them to the parent Prob call. Note that ProbBool builds a full binary tree for a 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). As in [4], 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.

Note that for the algorithm to behave correctly the program must be range restricted, i.e., all the variables in the head of clauses must appear in the body. Consider for example the following program T

$c_1 = a(1) : 0.3 \leftarrow p(X).$
$c_1 = a(2) : 0.4 \leftarrow p(X).$
$c_3 = p(X) : 0.5.$

where the third clause (c_3) is not range restricted.

The only derivation from $(\leftarrow a(1), \emptyset)$ to $(\leftarrow, C)$ has the following C set

$C = \{((c_1\emptyset, 0), (c_3\emptyset, 0)\}$

and thus gives a probability of 0.15. The grounding T' of T is

$c_1 = a(1) : 0.3 \leftarrow p(1).$ $c_2 = a(1) : 0.3 \leftarrow p(2).$
$c_3 = a(2) : 0.4 \leftarrow p(1).$ $c_4 = a(2) : 0.4 \leftarrow p(2).$
$c_5 = p(1) : 0.5.$ $c_6 = p(2) : 0.5.$

thus there are two successful derivations of $a(1)$ whose C sets are

$C^1 = \{(c_1\emptyset, 0), (c_5\emptyset, 0)\}$ $C^2 = \{(c_2\emptyset, 0), (c_6\emptyset, 0)\}$

for a probability of 0.2775.

If the program is range restricted, every derivation from $(\leftarrow G, \emptyset)$ to $(\leftarrow, C)$ will contain in C couples $(j, c\theta)$ such that $c\theta$ is ground and thus the above problem does not appear.

However, the query can contain variables: from the program T', the algorithm for the query $a(X)$ would return probability 0.2775 for $X = 1$ and probability 0.36 for $X = 2$.

In [4] an algorithm was given for 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 we increment the value of d. At the end of each iteration we have a set of C sets of successful derivations *Successful* and a set of C sets for still open derivations *Open*. The true probability $P^T_{PB}(Q)$ of a query is such that

$$P(F_1) \leq P^T_{PB}(Q) \leq P(F_1 \vee F_2)$$

where F_1 (F_2) is the formula corresponding to *Successful* (*Open*) Thus we have an upper and a lower bound on $P^T_{PB}(Q)$.

The cycle terminates when $P(F_1 \vee F_2) - P(F_1) \leq \epsilon$, where ϵ is a used defined precision.

Fig. 3. Function Prob

```
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
P :=ProbBool(n, 0, 1, mVar)
return P

function ProbBool(
      inputs : n : BDD node,
            value : index of the value of the multivalued variable
            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 be the probability associated with value of index
            value of variable mVar
      return p_value×Prob(n)
else
      let b_n be the Boolean variable associated to n
      let b_p be 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 high and low children of n
            shift left 1 position the bits of value
            P :=ProbBool(h, value + 1, posBVar + 1, mVar)+
                  ProbBool(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 :=ProbBool(n, value + 1, posBVar + 1, mVar)+
                  ProbBool(n, value, posBVar + 1, mVar)
            return P
```

However, this approach cannot be used for LPADs. In fact, consider the following program

$c_1 = a : 0.1 \leftarrow p(X). \quad c_2 = p(1) : 0.9. \quad c_3 = p(2) : 0.9.$

If we have the query a and a depth bound $d = 1$, then at the end of the first iteration *Successful* is empty and *Open* contains the only set $\{(c_1\emptyset, 0)\}$. Thus $P(F_1) = 0$ and $P(F_1 \vee F_2) = 0.1$. However $P^T_{LP}(Q)$ is 0.1719 so $P(F_1 \vee F_2)$ is not an upper bound on $P(Q)$. In fact, there are two successful derivations of a, one has the C set $\{(c_1\{X/1\}, 0), (c_2\emptyset, 0)\}$ and the other has the C set $\{(c_1\{X/2\}, 0), (c_3\emptyset, 0)\}$. Thus the formula F is

$X_{c_1\{X/1\}} = 0 \wedge X_{c_2} = 0 \vee X_{c_1\{X/2\}} = 0 \wedge X_{c_3} = 0$

Since the two disjunct are not mutually exclusive, we can use the law for the probability of an or and obtain

$0.1 \cdot 0.9 + 0.1 \cdot 0.9 - 0.1 \cdot 0.9 \cdot 0.1 \cdot 0.9 = 0.18 - 0.0081 = 0.1719$

This problem depends on the fact that, while in ProbLog we consider non ground clauses, in LPAD we consider instantiated ones and a clause in a partial derivation may not be fully instantiated. When the derivation is continued, it may generated more than one derivation with different instantiation of the clause.

5 Experiments

We report here on two experiments performed in order to evaluate the performances of the top down interpreter: the first involves a game of dice and the second graphs of biological concepts. All the experiments were performed on a Linux machine with a 3.40 GHz Pentium D processor and 1 GB of RAM.

In the first experiment we consider two versions of a dice game proposed in [9]: the player throws a die a number of times and stops only when a certain number comes out. We want to predict the probability of a given outcome at a given time point.

The two versions differ only for the number of faces of the (idealized) die: the first version considers a two face die and the second version a three face die. The LPAD describing the first version is shown below:

$on(0, 1) : 1/2 \vee on(0, 2) : 1/2.$

$on(T, 1) : 1/2 \vee on(T, 2) : 1/2 \leftarrow T1 \ is \ T - 1, T1 >= 0, on(T1, 1).$

Atom $on(T, N)$ means that at time T we rolled a die and face N came out. The first rule states that at time 0 (the beginning of the game) we rolled a die and we got a 1 or a 2 with equal probability. The second rule states that at time T we roll a die if a die was rolled at the previous time point and we got a 1. If we roll a die, we get a 1 or a 2 with equal probability. Thus, we stop when we get a 2.

The LPAD describing the second version is similar to the one above and states that we stop throwing dice only when we get a 3.

For the top down interpreter we used an implementation of it in Yap Prolog[1] that uses CUDD[2] as the BDD manipulation package.

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

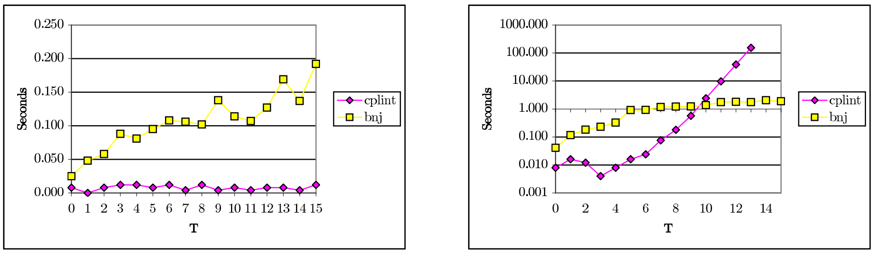
[2] `http://vlsi.colorado.edu/~fabio/`

For the Bayesian reasoner, we used the implementation of the junction tree inference algorithm [5] available in BNJ[3] version 2 release 7 2004.

The query $on(T, 1)$ was tried against both programs with T ranging from 0 to 15. The execution times of the top down interpreter (cplint) and of the Bayesian reasoner (bnj) are shown in Figures 4(a) and 4(b) for the two sided die and for the three sided die respectively.

When generating the ground program to be translated into a Bayesian network, only the constants relevant to the query were considered. So, for example, if the query was $on(3, 1)$, only constants 0, 1, 2 and 3 were considered for T and $T1$. For N, the constants 1 and 2 were considered for the first program and 1, 2 and 3 for the second program.

For the point not shown for cplint in Figure 4(b), the system started thrashing and the computation was interrupted after four hours.



(a) Two sided die. (b) Three sided die.

Fig. 4. Execution times for the die programs.

We consider now two programs with the same meaning as those above but that use negation. The one for the three sided die is

$on(0, 1) : 1/3 \vee on(0, 2) : 1/3 \vee on(0, 3) : 1/3.$

$on(T, 1) : 1/3 \vee on(T, 2) : 1/3 \vee on(T, 3) : 1/3 \leftarrow$

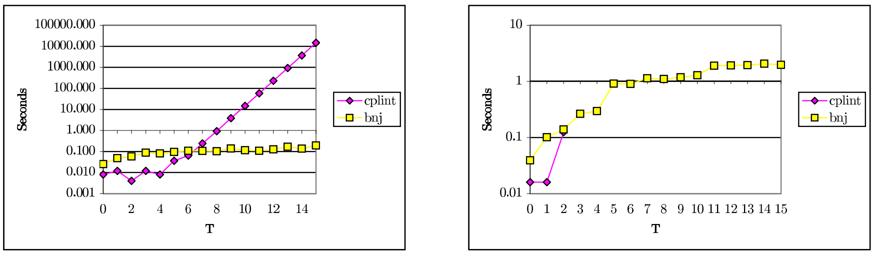
$\quad T1\ is\ T - 1, T1 >= 0, on(T1, N), \neg on(T1, 3).$

The computation times are shown in Figures 5(a) and 5(b) respectively under the same experimental settings discussed before.

The points not shown for cplint in Figure 5(b) are those for which Yap stopped returning an "out of database space" error.

The second experiment involves computing the probability of a path between two nodes in a graph. This experiment was chosen in order to compare the results with those [4] where the authors use the ProbLog interpreter for evaluating the probability of paths between nodes in a network of biological concepts. The dataset was kindly provided by the authors of [4] and is the same as the one used in the paper. The dataset consists of a number of subgraphs $G_1, G_2, \ldots G_n$

[3] `http://sourceforge.net/projects/bndev`



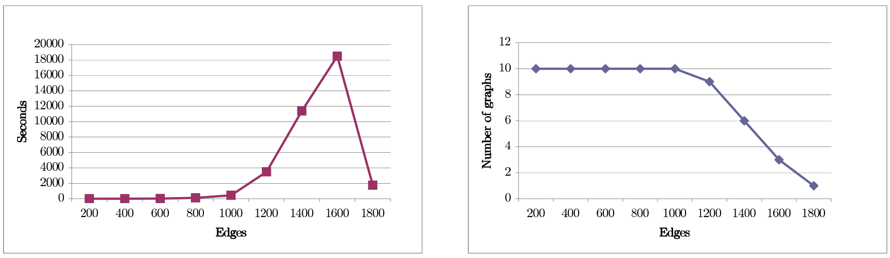
(a) Two sided die.　　　(b) Three sided die.

Fig. 5. Execution times for the die programs with negation.

extracted from a complete graph built around four Alzheimer genes. The complete graph contains 11530 edges and 5220 nodes. The subgraphs are obtained by subsampling, they have the sizes 200, 400, ..., 5000 edges and are such that $G_1 \subset G_2 \subset \ldots \subset G_n$. Subsampling was repeated 10 times.

The query $can\_reach(620, 983)$ was issued on every subgraph, where 620 and 683 are the identifiers of a couple of genes and *can_reach* is defined recursively with definite clauses in the usual way

The computation time for the probability of the query is shown in Figure 6(a) in seconds as a function of the number of edges. The time shown is the average computation time on the subgraphs on which the interpreter was successful. Figure 6(b) shows the number of graphs for which the computation succeeded: for the other graphs, the computer did not return an answer after 10 hours.



(a) Execution times.　　　(b) Number of successes.

Fig. 6. Biological graph experiments.

A comparison with bnj was not possible because the conversion program exhausted the available memory: the grounding of the definition for *can_reach* was too large.

These experiments show that, for small problems, Bayesian inference is more scalable. However, when problems with many constants are considered, using

Bayesian inference is not possible. Comparing cplint with the ProbLog interpreter of [4], we see that the added expressiveness of LPAD and CP-Logic has an impact on performances, since the ProbLog interpreter was able to answer the query for up to 4600 edges.

6 Conclusions

We have presented a top down interpreter for computing the probability of LPADs and CP-logic queries that is inspired to the one presented in [4].

We have experimentally compared the algorithm with a Bayesian inference algorithm and with the ProbLog interpreter.

In the future, we plan to extend the interpreter by considering also aggregates and the possibility of having the probabilities in the head depend on literals in the body.

References

1. Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *J. Log. Program.*, 19/20:9–71, 1994.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986.
3. K. L. Clark. Negation as failure. In *Logic and Databases*. Plenum Press, 1978.
4. L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2462–2467, 2007.
5. S. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, B, 50(2):157–224, 1988.
6. D. M. Miller and R. Drechsler. On the construction of multiple-valued decision diagrams. In *Proceedings 32nd IEEE International Symposium on Multiple-Valued Logic*, pages 245–253, 2002.
7. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
8. J. Vennekens, M. Denecker, and M. Bruynooghe. Representing causal information about a probabilistic process. In *10th European Conference on Logics in Artificial Intelligence, JELIA 2006*, LNAI. Springer, September 2006.
9. J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *The 20th International Conference on Logic Programming (ICLP 2004)*, 2004.