arXiv:2008.01394v3 [cs.AI] 1 Sep 2020

MAP Inference for Probabilistic Logic Programming

ELENA BELLODI[1], MARCO ALBERTI[2], FABRIZIO RIGUZZI[2], RICCARDO ZESE[1]
[1] *Dipartimento di Ingegneria – Università di Ferrara*
[2] *Dipartimento di Matematica e Informatica – Università di Ferrara*
Via Saragat 1, 44122, Ferrara, Italy
(*e-mail:* `firstname.surname@unife.it`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

In Probabilistic Logic Programming (PLP) the most commonly studied inference task is to compute the marginal probability of a query given a program. In this paper, we consider two other important tasks in the PLP setting: the Maximum-A-Posteriori (MAP) inference task, which determines the most likely values for a subset of the random variables given evidence on other variables, and the Most Probable Explanation (MPE) task, the instance of MAP where the query variables are the complement of the evidence variables. We present a novel algorithm, included in the PITA reasoner, which tackles these tasks by representing each problem as a Binary Decision Diagram and applying a dynamic programming procedure on it. We compare our algorithm with the version of ProbLog that admits annotated disjunctions and can perform MAP and MPE inference. Experiments on several synthetic datasets show that PITA outperforms ProbLog in many cases. This paper is *under consideration* for acceptance in Theory and Practice of Logic Programming.

1 Introduction

Probabilistic Logic Programming (PLP) (De Raedt et al. 2008; Riguzzi 2018) has emerged as one of the most prominent approaches for modeling complex domains containing many uncertain relationships among their entities. In this field, many languages are equipped with the distribution semantics (Sato 1995). Examples of such languages are Independent Choice Logic (Poole 1997), PRISM (Sato 1995), Logic Programs with Annotated Disjunctions (LPADs) (Vennekens et al. 2004a) and ProbLog (De Raedt et al. 2007). All these languages have the same expressive power, as a theory in one language can be translated into each of the others (De Raedt et al. 2008). LPADs offer a general syntax as the constructs of all the other languages can be directly encoded in this language. Under the distribution semantics, an LPAD defines a probability distribution over a set of normal logic programs called worlds, by associating to each disjunctive clause a random variable, whose value determines the selection of one of the atoms in the head.

The inference task that has received most attention from the PLP community is computing the marginal probability of a ground query atom q given evidence e on a subset of the other atoms, $P(q|e)$. In the absence of e, this is also known as the success probability of a query $P(q)$, defined as the sum of the probabilities of all the worlds that entail q.

Other important inference tasks are the *maximum a posteriori* (MAP) and the *most probable explanation* (MPE) tasks. In general terms, given a joint probability distribution over a set of random variables, values for a subset of the variables (evidence), and another

disjoint subset of the variables (query), the MAP problem consists of finding the most probable values for the query variables given the evidence. The MPE problem is the MAP problem where the set of query variables is the complement of the set of evidence variables. In PLP, the MPE problem can be expressed as taking the truth of some atoms as evidence, and finding the world of an LPAD that has the highest probability among those that entail the evidence. Solving the MAP problem, given evidence and a subset of the random variables, consists of finding the assignment to those variables that maximizes the probability of the assignment given the evidence, i.e., the sum of the probabilities of the worlds compatible with the assignment and the evidence.

The PITA algorithm (for "Probabilistic Inference with Tabling and Answer subsumption") (Riguzzi and Swift 2010; Riguzzi and Swift 2011; Riguzzi and Swift 2013) takes as input an LPAD and computes the probability of success of a query by building Binary Decision Diagrams (BDDs) for every subgoal encountered during the derivation of the query. In this paper, we present and evaluate experimentally an extension of PITA to perform the MPE and MAP tasks. We compare PITA to the version of ProbLog presented by Shterionov et al. (2015), which supports Annotated Disjunctions in the head of clauses (such as LPADs), allowing to perform the MPE (and MAP) task as well. ProbLog answers MPE queries by converting each annotated disjunction into a set of probabilistic facts with appropriate probability values and a Prolog rule for each of its head atoms having mutually exclusive bodies, then it generates the grounding of the resulting program. Then, the program is converted into a Conjunctive Normal Form (CNF) Boolean formula and knoweldge compilation is applied. As done by Shterionov et al. (2015) the CNF formula is compiled into a d-DNNF instead of a BDD. d-DNNF are more succinct than BDDs, which means that, given a formula, its d-DNNF version is smaller than its BDD version. However, software packages for the manipulation of BDDs are highly optimized and the experiments show that the use of BDDs is sometimes advantageous. For answering MAP queries ProbLog uses a different strategy resorting to Decision Theoretic ProbLog (DTProbLog) (Van den Broeck et al. 2010) that exploits Algebraic Decision Diagrams.

We ran experiments on several synthetic datasets; the results show that PITA performs better than ProbLog on the MAP and MPE tasks in many cases.

The paper is structured as follows: in Section 2 we summarize the necessary background notions, in Section 3 we define the MAP and MPE problems for LPADs, in Section 4 we present their implementation in PITA, in Section 5 we assess the scalability of our system and compare it with the same techniques implemented in ProbLog (Shterionov et al. 2015), and in Section 6 we conclude the article.

2 Background

2.1 Logic Programs with Annotated Disjunctions

LPADs (Vennekens et al. 2004b) consist of a finite set of annotated disjunctive clauses r_i of the form $h_{i1} : \Pi_{i1}; \ldots ; h_{in_i} : \Pi_{in_i} \leftarrow b_{i1}, \ldots , b_{im_i}$, where $b_{i1}, \ldots , b_{im_i}$ are logical literals that form the *body* of r_i, denoted by $body(r_i)$, while $h_{i1}, \ldots h_{in_i}$ are logical atoms and $\{\Pi_{i1}, \ldots , \Pi_{in_i}\}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. If $n_i = 1$ and $\Pi_{i1} = 1$ the clause is a non-disjunctive and non-probabilistic clause.

If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$, the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} \Pi_{ik}$. $ground(\mathcal{P})$ denotes the grounding of an LPAD $\mathcal{P}$. We do not allow function symbols, so $ground(\mathcal{P})$ is finite.

Definition 1 (*Variable associated to a clause's grounding*)
To each grounding substitution θ_j of each clause r_i, a discrete random variable X_{ij} is associated, whose range is $0, \ldots, n_i$ and whose probability distribution is given by

$$P(X_{ij} = k) = \begin{cases} \Pi_{ik} & \text{if } 1 \leq k \leq n_i \\ 1 - \sum_{k=1}^{n_i} \Pi_{ik} & \text{if } k = 0 \end{cases}$$

$X_{ij} = k$ means that the k-th head atom, or the *null* atom if $k = 0$, is chosen for grounding θ_j of clause r_i.

We now present the distribution semantics for the case in which the program does not contain function symbols so that its Herbrand base is finite[1].

An *atomic choice* is an equation $X_{ij} = k$. A set of atomic choices κ is *consistent* if $X_{ij} = k \in \kappa, X_{ij} = m \in \kappa$ implies $k = m$, i.e., only one head is selected for a ground clause. A *composite choice* κ is a consistent set of atomic choices. The probability of a composite choice κ is $P(\kappa) = \prod_{X_{ij}=k\in\kappa} P(X_{ij} = k)$. A *selection* σ is a total composite choice (one atomic choice for every grounding of each probabilistic clause). Let us call S_T the set of all selections. A selection σ identifies a normal logic program w_σ called a *world*. The probability of w_σ is $P(w_\sigma) = P(\sigma)$. Since the program does not contain function symbols, the set of worlds $W_T = \{w_1, \ldots, w_m\}$ is finite and $P(w)$ is a distribution over worlds: $\sum_{w\in W_T} P(w) = 1$. The conditional probability of a query Q given a world w can be defined as: $P(Q|w) = 1$ if Q is true in w and 0 otherwise. We can obtain the probability of the query by marginalizing over the query:

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

Example 1
Given the LPAD

```
red(b1):0.6; green(b1):0.3; blue(b1):0.1 :- pick(b1).
pick(b1):0.6; no_pick(b1):0.4.
ev:- \+ blue(b1).
```

the query `ev` is true in five worlds so its probability is $P(\texttt{ev}) = 0.6 \cdot 0.6 + 0.6 \cdot 0.3 + 0.4 \cdot 0.6 + 0.4 \cdot 0.3 + 0.4 \cdot 0.1 = 0.94$.

A composite choice κ *identifies* a set ω_κ that contains all the worlds associated with a selection that is a superset of κ: i.e., $\omega_\kappa = \{w_\sigma | \sigma \in S_T, \sigma \supseteq \kappa\}$. We define the set of worlds *identified* by a set of composite choices K as $\omega_K = \bigcup_{\kappa\in K} \omega_\kappa$. Given a ground literal Q, a composite choice κ is an *explanation* for Q if Q is true in every world of ω_κ. A set of composite choices K is *covering* with respect to Q if every world w_σ in which Q is true is such that $w_\sigma \in \omega_K$. Given a covering set of explanations for a query, we can obtain a

[1] For the distribution semantics with function symbols see (Sato 1995; Poole 2000; Riguzzi and Swift 2013; Riguzzi 2016).

Boolean formula $f(\mathbf{X})$ in Disjunctive Normal Form (DNF) where: (1) each atomic choice yields an equation $X_{ij} = k$, (2) we replace an explanation with the conjunction of the equations of its atomic choices and the set of explanations with the disjunction of the formulas for all explanations. If we consider a world as the specification of a truth value for each equation $X_{ij} = k$, the formula evaluates to true exactly on the worlds where the query is true (Poole 2000). Since the disjuncts in the formula are not necessarily mutually exclusive, the probability of the query can not be computed by a summation as in Formula (1). The problem of computing the probability of a Boolean formula in DNF, known as *disjoint sum*, is #P-complete (Valiant 1979). One of the most effective ways of solving the problem makes use of Decision Diagrams.

2.2 Binary Decision Diagrams

We can apply *knowledge compilation* (Darwiche and Marquis 2002) to the Boolean formula $f(\mathbf{X})$ in order to translate it into a "target language" that allows the computation of its probability in polynomial time. We can use Decision Diagrams (DD) as a target language. A DD has one level for each variable and two leaves, one associated with the 1 Boolean function and the other with the 0 Boolean function. Each variable node has as many children as its values. A DD can be used to compute the value of a Boolean function given the values of the variables by starting at the root and following the path according to the variable values until a leaf is reached. The label of the leaf is the value of the Boolean function. Most packages for the manipulation of DDs are however restricted to work on Binary Decision Diagrams (BDD), i.e., decision diagrams where all the variables are Boolean. These packages offer Boolean operators among BDDs and apply simplification rules to the results of operations in order to reduce as much as possible the size of the diagram, producing a reduced BDD.

A node n in a BDD has two children: the 1-child and the 0-child. To work with a BDD package we must represent multi-valued variables by means of binary variables. We use the following encoding, called *order encoding*: for a multi-valued variable X_{ij}, corresponding to a ground clause $C_i\theta_j$, having n_i values, we use $n_i - 1$ Boolean variables $X_{ij1}, \ldots, X_{ijn_i-1}$ and we represent the equation $X_{ij} = k$ for $k = 1, \ldots n_i - 1$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, and the equation $X_{ij} = n_i$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijn_i}}$. Note that $\lceil \log_2 n_i \rceil$ binary variables would be sufficient to represent an n_i-valued variable, but the encoding that we use allows for faster BDD processing. A parameter π_{ik} is associated with each Boolean variable X_{ijk}. The parameters are obtained from those of multi-valued variables in this way: $\pi_{i1} = \Pi_{i1}$, …, $\pi_{ik} = \frac{\Pi_{ik}}{\prod_{j=1}^{k-1}(1-\pi_{ij})}$, up to $k = n_i - 1$. In order to manage BDD we exploit the CUDD (Colorado University Decision Diagram)[2] library, a library written in C that provides functions to manipulate different types of Decision Diagrams. In CUDD, BDD nodes are described by two fields: *pointer*, a pointer to the node, and *comp*, a Boolean indicating whether the node is complemented. In fact three types of edges are admitted: an edge to a 1-child, an edge to a 0-child and a complemented edge to a 0-child, meaning that the function encoded by the child must be complemented. Moreover, the root node can

[2] `https://github.com/ivmai/cudd`

be complemented. For these types of BDD, only the 1 leaf is needed. Once a BDD for a query has been built, it is possible to compute the probability of the query using a dynamic programming algorithm (Raedt et al. 2007), which is shown in Algorithm 1.

Algorithm 1 Function Prob: computation of the probability of a BDD.

```
1: function PROB(node)
2:     if node is a terminal then
3:         return 1
4:     else
5:         if TableProb(node.pointer) ≠ null then
6:             return TableProb(node)
7:         else
8:             p0 ←PROB(child_0(node))
9:             p1 ←PROB(child_1(node))
10:            if child_0(node).comp then
11:                p0 ← (1 − p0)
12:            end if
13:            Let π be the probability of being true of var(node)
14:            Res ← p1 · π + p0 · (1 − π)
15:            Add node.pointer → Res to TableProb
16:            return Res
17:        end if
18:    end if
19: end function
```

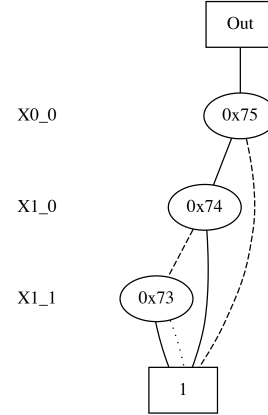


Fig. 1. BDD for Example 1.

The BDD for the query `ev` from Example 1 is shown in Figure 1, where edges going to the 1-child are solid, edges going to the 0-child are dashed and complemented edges going to the 0-child are dotted. Variables X1_0 and X1_1 encode the first rule and variable X0_0 the second rule. Node labels are just identifiers.

3 MAP and MPE Inference for LPADs Programs

Definition 2 (*MAP Problem*)
Given an LPAD $\mathcal{P}$, a conjunction of ground atoms e, the *evidence*, and a set of random variables $\mathbf{X}$ (query random variables), associated to some ground rules of $\mathcal{P}$, the MAP problem is to find an assignment $\mathbf{x}$ of values to $\mathbf{X}$ such that $P(\mathbf{x}|e)$ is maximized, i.e., solve

$$\arg\max_{\mathbf{x}} P(\mathbf{x}|e)$$

The MPE problem is a MAP problem where $\mathbf{X}$ includes all the random variables associated with all ground clauses of $\mathcal{P}$.

In the following, we indicate the query random variables in the program by prepending the functor `map_query` to the rules.

Shterionov et al. (2015) showed that the encoding presented in Section 2.2 using $n_i - 1$ Boolean variables for a clause with n_i heads does not work, as configurations of the variables exist that do not correspond to any value for the rule random variable. The problem is that the order encoding is redundant and a value for the random variable associated with a rule may be encoded by multiple tuples of values of the Boolean variables besides the intended one. One of those unintended encodings may get chosen because it has a higher probability but this does not reflect on the correct choice of the multivalued variable. Shterionov et al. (2015) proposed a different encoding, where n_i Boolean variables

X_{ijk} for a clause with n_i heads are used and constraints are imposed, namely that one and only one X_{ijk} must be true. This is achieved by building the constraint formula

$$(\bigvee_{k=1}^{n_i} X_{ijk}) \wedge \bigwedge_{k=1}^{n_i} \bigwedge_{m=k+1}^{n_i} (\neg X_{ijk} \vee \neg X_{ijm})$$

for each multi-valued variable X_{ij}, translating it into a BDD and conjoining it with the BDD built for the query.

Example 2
Given the program of Example 1

```
map_query red(b1):0.6; green(b1):0.3; blue(b1):0.1 :- pick(b1).
map_query pick(b1):0.6; no_pick(b1):0.4.
ev:- \+ blue(b1).
```

where all the random variables are query, evidence `ev` has the MPE assignment $\mathbf{x}$:

```
[rule(1, pick(b1), [pick(b1):0.6, no_pick(b1):0.4], true),
rule(0, red(b1), [red(b1):0.6, green(b1):0.3, blue(b1):0.1], pick(b1))],
```

where predicate `rule/4` specifies clause number (zero-based), selected head, clause head, clause body, in that order. For this assignment, $P(\mathbf{x}|ev) = 0.36$, meaning that the most probable explanation $\mathbf{x}$ has a probability of 0.36. The corresponding BDD is shown in Figure 2, where variables X0_k are associated with the second clause and X1_k with the first clause.

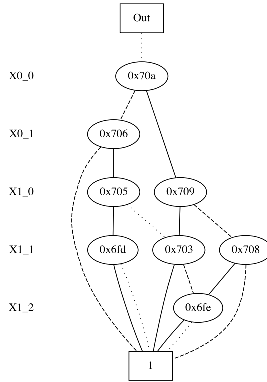


Fig. 2. BDD for the MPE problem of Example 2.

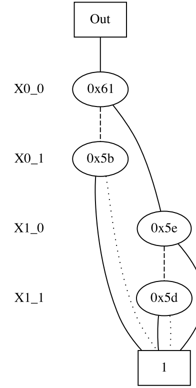


Fig. 3. BDD for the MAP problem of Example 3.

Example 3
Given the program

```
red(b1):0.6; green(b1):0.3; blue(b1):0.1 :- pick(b1).
map_query pick(b1):0.6; no_pick(b1):0.4.
ev:- \+ blue(b1).
```

The evidence `ev` has the MAP assignment:

```
[rule(1, pick(b1), [pick(b1):0.6, no_pick(b1):0.4], true)].
```

For this assignment, $P(\mathbf{x}|ev) = 0.54$. The corresponding BDD is shown in Figure 3, where variables X0_k are associated to the second rule and X1_k to the first rule.

Example 4
Consider the following LPAD:

```
map_query disease:0.05.
map_query malfunction:0.05.
positive :- malfunction.
map_query positive:0.999 :- disease.
map_query positive:0.0001 :- \+(malfunction), \+(disease).
```

The LPAD models the diagnosis of a disease by means of a lab test. The disease probability is 0.05, and, in case of disease, the test result will be positive with probability 0.999. However, there is a 5% chance of an equipment malfunction; in this case, the test will always be positive. Additionally, even in absence of disease or malfunction, the test result will be positive with probability 0.0001. The LPAD has 16 worlds, each corresponding to selecting, or not, the head of each annotated disjunctive clause.
Let us suppose for the test result to be positive: is the patient ill? Given evidence `ev = positive`, the MPE assignment is

```
[rule(1, '', [malfunction:0.05, '' :0.95], true),
rule(0, disease, [disease:0.05, '' :0.95], true),
rule(2, positive, [positive:0.999, '' :0.001], disease),
rule(3, '', [positive:0.0001, '' :0.9999], (\+malfunction,\+disease))]
```

where `''` indicates the *null* head. The most probable world is the one where an actual disease caused the positive result, and its probability is $P(\mathbf{x}|e) = 0.04702$.

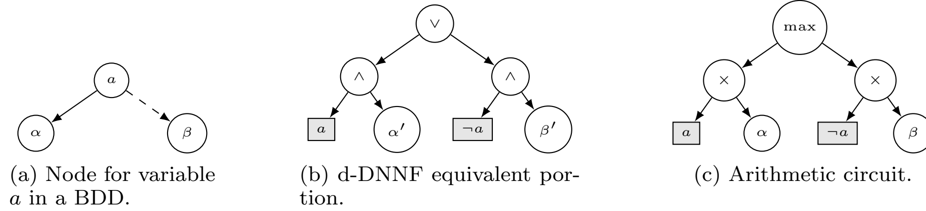
Likewise, if we perform a MAP inference taking only the choice of the first clause as query variable, the result is `[rule(0, disease, [disease:0.05, '' : 0.95], true)]`, so the patient is ill. However, if we take the choices for the first two clauses as query variables, i.e., if we look for the most likely combination of `disease` and `malfunction` given `positive`, the MAP task produces

```
[rule(1, malfunction, [malfunction:0.05, '' : 0.95], true),
rule(0, '', [disease:0.05, '' : 0.95], true)]
```

meaning that the patient is not ill and the positive test is explained by an equipment malfunction. This examples shows that the value assigned to a query variable in a MAP task can be affected by the presence of other variables in the set of query variables; in particular, MPE and MAP inference over $\mathbf{X}$ may assign different values to the same variable given the same evidence.

4 Integration of MAP and MPE Inference into the PITA System

PITA (Probabilistic Inference with Tabling and Answer subsumption) (Riguzzi and Swift 2010; Riguzzi and Swift 2013) computes the probability of a query from a probabilistic program in the form of an LPAD by first transforming the LPAD into a normal program containing calls for manipulating BDDs. The idea is to add an extra argument to



(a) Node for variable a in a BDD. (b) d-DNNF equivalent portion. (c) Arithmetic circuit.

Fig. 4. Translation from BDDs to d-DNNF.

each subgoal to store a BDD encoding the explanations for the answers of the subgoal. The values of the subgoals' extra argument are combined using a set of general library functions:

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

These functions are implemented in C as an interface to the CUDD library for manipulating BDDs. A BDD is represented in Prolog as an integer that is a pointer in memory to its root node.

Let us first consider the MPE task. PITA solves it using the dynamic programming algorithm proposed by Darwiche (2004, Section 12.3.2) for computing MPE over d-DNNFs, which define a propositional language that generalizes BDDs. In fact, a BDD can be seen as a d-DNNF by using the translation shown in Figure 4: a BDD node (Figure 4a) for variable a with children α and β is translated into the d-DNNF portion shown in Figure 4b, where α' and β' are the translations of the BDD α and β respectively. The algorithm proposed by Darwiche (2004) computes the probability of the MPE by replacing $\wedge$-nodes with product nodes and $\vee$-nodes with max-nodes: the result is an arithmetic circuit (Figure 4c) that, when evaluated bottom-up, gives the probability of the MPE and can be used to identify the MPE assignment. The equivalent algorithm operating on BDDs - Function MAPInt in Algorithm 2 - modifies Algorithm 1 and returns both a probability and a set of assignments to random variables. At each node, instead of computing $Res \leftarrow p1 \cdot \pi + p0 \cdot (1 - \pi)$ as in Algorithm 1 line 14, it returns the assignment of the children having the maximum probability. This is computed in lines 39-43 in Algorithm 2. In MPE there are no non-query variables, so the test in line 24 succeeds only for the BDD leaf. MAPInt in practice computes the probability of paths from the root to the 1 leaf and returns the probability and the assignment corresponding to the most probable path. In a MAP task, i.e., when we have non-query variables, function MAPInt cannot be used because when a node for a non-query variable is reached, it must be summed out instead of maximized out, and maximization and summation operations are not commutative. However, if its children are nodes for query variables, which of the two assignments for the children should be propagated towards the root? If query variables are mixed with non-query variables in the BDD variable ordering, function MAPInt

does not work. In case that the non-query variables appear last in the ordering, when MAPInt reaches a node for a non-query variable, it can sum out all non-query variables using function Prob from Algorithm 1. This assigns a probability to the node that can be used by MAPInt to identify the most probable path from the root. So PITA solves MAP by reordering variables in the BDD, putting first the query variables.

With CUDD we can either create BDDs from scratch with a given variable order or modify BDDs according to a new variable order. Changing the position of a variable is made by successive swapping of adjacent variables (Somenzi 2001): the swap can be performed in a time proportional to the number of nodes associated with the two swapped variables. Changing the order of two adjacent variables does not affect the other levels of the BDD, so changes can be applied directly to the current BDD saving memory. To further reduce the cost of the swapping, the CUDD library keeps in memory an interaction matrix specifying which variables directly interact with others. This matrix is updated only when a new variable is inserted into the BDD, is symmetric and can be stored by using a single bit for each pair, making it very small. Moreover, the cost of building it is negligible compared to the cost of manipulating the BDD without checking it. Jiang et al. empirically demonstrated that changing the order of variables by means of sequential swapping is usually much more time efficient than rebuilding the BDD following a fixed variable order (Jiang et al. 2017).

PITA differs from ProbLog in both tasks. For MPE inference, ProbLog applies the algorithm of (Darwiche 2014) to d-DNNF. For MAP, ProbLog uses DTProbLog, an algorithm for maximizing an utility function by making decisions. In DTProbLog utility values are assigned to some ground literals, some ground atoms are probabilistic and some are decision. The aim is to find an assignment to decision variables that maximizes utility, given by the sum of the utility for the literals that are made true by the decisions. DTProbLog uses Algebraic Decision Diagrams (ADDs) as a target compilation language. ADDs are BDDs where leaves are associated with real numbers instead of Boolean values. ADDs built by DTProbLog contain only decision variables, probabilistic variables are compiled away. We differ from DTProbLog because we do not compile away non-query variables but we simply rearrange the BDD. As shown by the experiments, this is sometimes advantageous.

5 Experimental Results

Experiments aim at analyzing how PITA scales when doing MAP and MPE inference w.r.t. the data size, and at comparing their performance with the same tasks performed by ProbLog2.1 (Fierens et al. 2015) in terms of inference time.

Experiments were performed on GNU/Linux machines with Intel Xeon E5-2697 v4 (Broadwell) at 2.30 GHz and 128 GB of RAM available and were set to a maximum execution time of 24h. Four artificially generated datasets were used: *growing head (gh)*, *growing negated body (gnb)*, *blood* (Shterionov et al. 2015), and *probabilistic graphs*. *Growing head* is a set of 15 programs with annotated disjunctions with an increasing number of head atoms; *growing negated body* is a set of 50 programs with an increasing number of negated body atoms; *blood* is a set of 100 programs regarding the inheritance of blood type with an increasing number of ancestors (mother+father for each person); *probabilistic graphs* is a set of $N \times M$ programs, where $N = \{50, 100, 150, 200, 250, 300, 400, 450, 500\}$ is the

Algorithm 2 Function MAP: computation of the maximum a posterior state of a set of query variables and of its probability

```
1:  function MAP(root)
2:      for all query variables var do
3:          AtLeastOne ← BDD_Zero
4:          AtMostOne ← BDD_One
5:          for i ← 1 to values(var) do
6:              AtLeastOne ← BDD_Or(AtLeastOne, bVar(var, i))
7:              for j ← i + 1 to values(var) do
8:                  NotBoth ← BDD_Not(BDD_And(bVar(var, i), bVar(var, j)))
9:                  AtMostOne ← BDD_And(AtMostOne, NotBoth)
10:             end for
11:         end for
12:         const ← BDD_And(AtLeastOne, AtMostOne)
13:         root ← BDD_And(root, const)
14:     end for
15:     Reorder BDD root so that variables associated to query variables come first in the order
16:     Let root' be the new root
17:     TableMAP ← ∅
18:     TableProb ← ∅
19:     (Prob, MAP) ←MAPInt(root, false)        ▷ MAPBV: map assignment for Boolean random variables
20:     return (Prob, MAP)
21: end function
22: function MAPInt(node, comp)    ▷ Internal function implementing the dynamic programming algorithm
23:     comp ← node.comp ⊕ comp
24:     if var(node) is not associated to a query var then
25:         p ←Prob(node)                                                          ▷ Algorithm 1
26:         if comp then
27:             return (1 − p, [])
28:         else
29:             return (p, [])
30:         end if
31:     else
32:         if TableMAP(node.pointer) ≠ null then
33:             return TableMAP(node.pointer)
34:         else
35:             (p0, MAP0) ←MAPInt(child_0(node), comp)
36:             (p1, MAP1) ←MAPInt(child_1(node), comp)
37:             Let π be the probability of being true of the variable at level level
38:             p1 ← p1 · π
39:             if p1 > p0 then
40:                 Res ← (p1, [var(node) = 1|MAP1])
41:             else
42:                 Res ← (p0, MAP0)
43:             end if
44:             Add (node.pointer) → Res to TableMAP
45:             return Res
46:         end if
47:     end if
48: end function
```

number of nodes of the graphs and $M = 10$ is the number of different probabilistic edge configurations for each graph size. The graphs have been randomly generated according to the Barabási-Albert model (Barabasi and Albert 1999) with parameters $m_o = m = 2$. These benchmarks can be found at `http://ml.unife.it/material/`. In the following, results are commented separately for MAP and MPE inference.

5.1 MPE Results

For these experiments we ran PITA and ProbLog 2.1 on all datasets, except for *blood* on which only PITA could be applied due to Problog2.1 execution timing out. ProbLog2.1 was run with the command `problog-cli.py mpe` *program.pl*. This system requires to specify evidence in *program.pl* with the `evidence/1` fact. For *gh* and *gnb*, evidence corresponds to `a0`, for *blood* to `bloodtype(p,a)`, for *probabilistic graphs* to `path(0,N-1)`

(e.g. `path(0,49)` when $N = 50$). Inference times are compared in Figures 5, 6, 7, 8; for *probabilistic graphs* the average time over the 10 configurations for each N was computed. PITA outperforms ProbLog on *gh* and *blood*, where the latter times out starting from program size 13 or from the beginning, respectively; on *gnb* and *probabilistic graphs* the systems are comparable for small program sizes, then PITA is slower. This shows that, in some cases, BDDs are competitive with d-DNNF thanks to the use of highly optimized packages.

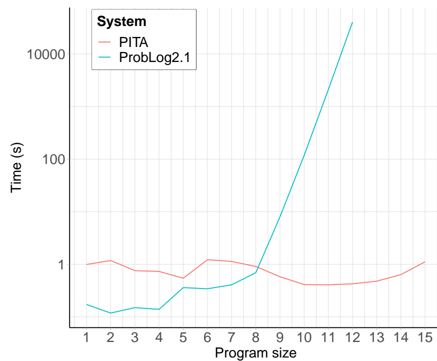


Fig. 5. MPE results on the *growing head* dataset (log scale on Y axis).

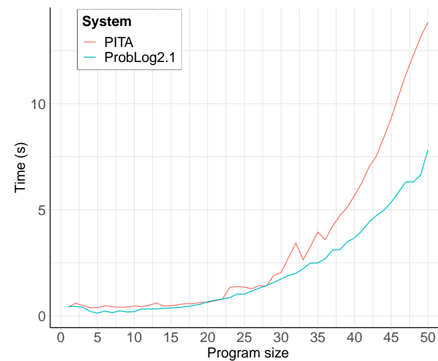


Fig. 6. MPE results on the *growing negated body* dataset.

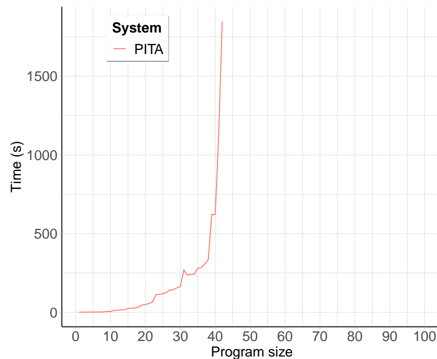


Fig. 7. MPE results on the *blood* dataset.

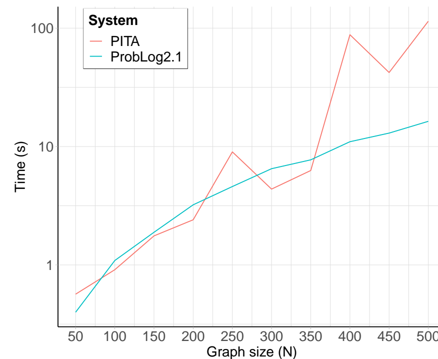


Fig. 8. MPE results on the *probabilistic graphs* dataset (log scale on Y axis).

5.2 MAP Results

For these experiments we ran PITA and ProbLog2.1 with the command `problog-cli.py map` *program.pl*. As MAP assignments of ground atoms must be explicitly queried, PITA requires the specification of the keyword `map_query` in front of the desired clauses. Analogously, ProbLog2.1 uses the keyword `query` that however can only be applied to probabilistic facts; so, for the datasets containing clauses with multiple probabilistic heads, a syntactical transformation was applied before specifying the `query` ground atoms.

For *gh* we used the program of size 11, containing 19 probabilistic clauses, and queried the 10%, 20%,...,90% of them. For *gnb* we used the program of size 10, containing 46 probabilistic clauses, and for *blood* the program of size 1, having 31 probabilistic clauses. For *probabilistic graphs*, for each of the 10 edge configurations for each graph size N, we queried 20%, 50% and 80% of the clauses: the 50-node graphs contain 96 probabilistic `edge` facts, the 100-node graphs contain 196 `edge` facts, until the 500-node graphs which contain 996 `edge` facts. We could not use the maximum size LPADs for *gh*, *gnb* and *blood* due to memory errors or time-outs ($> 24h$), hence we chose a program size for which we could get results in a reasonable time. Evidence is the one specified in Section 5.1. Inference times are compared in Table 1 for *gh, gnb, blood* and in Table 2 for *probabilistic graphs* with $N = 50$; for $N \geq 100$ only PITA gave a result (almost always $< 1min$, maximum time $= 10min$ with $N = 500$), while ProbLog2.1 always gave memory error or an error from the program. As expected, MAP inference takes more time, especially on *gh* and *blood*; PITA performs better than ProbLog on all datasets except *blood*, indicating that BDD reordering is advantageous with respect to the use of ADDs.

6 Conclusions

In this paper, we presented an algorithm to solve the Maximum-A-Posteriori (MAP) and the Most-Probable-Explanation (MPE) problems on Logic Programs with Annotated Disjunctions. We integrated the algorithm into the PITA solver, which is available as a SWI-Prolog package and in the cplint on SWISH web application (Alberti et al. 2016; Alberti et al. 2017) at `http://cplint.eu`. We experimentally compared the algorithm with the ProbLog version (2.1) that supports annotated disjunctions and can perform the MAP and MPE tasks. The results on several synthetic datasets show that PITA performs better than ProbLog in many cases. From our experimentation, we can conclude that since d-DNNF are theoretically better than BDD, one should first try ProbLog. In case the running time is high, however, using BDDs with PITA is an option to be considered because we demonstrated that in some cases the performance may be better.

In the future we plan to investigate the algorithm for finding Viterbi proofs (Shterionov et al. 2015), i.e., partial truth value assignments (or partial possible worlds) such that for all full assignments extending the proof, the query holds.

Acknowledgments

This work was partly supported by the “National Group of Computing Science (GNCS-INDAM)”.

References

Alberti, M., Bellodi, E., Cota, G., Riguzzi, F., and Zese, R. 2017. `cplint` on SWISH: Probabilistic logical inference with a web browser. *Intell. Artif. 11,* 1, 47–64.

Alberti, M., Cota, G., Riguzzi, F., and Zese, R. 2016. Probabilistic logical inference on the web. In *AI*IA 2016: Advances in Artificial Intelligence, 21st Congress of the Italian Association for Artificial Intelligence, Pisa*, G. Adorni, S. Cagnoni, M. Gori, and M. Maratea, Eds. Lecture Notes in Computer Science, vol. 10037. Springer International Publishing, 351–363.

Table 1. MAP inference time (s) comparison on the *growing head, growing negated body, blood* datasets. In bold the best results. "t-o" means time-out (>24hours),"me" means memory error.

	Growing head			*Growing negated body*			*Blood*	
	ProbLog2.1	PITA		ProbLog2.1	PITA		ProbLog2.1	PITA
10%	402.547	**1.802**	10%	**0.332**	0.486	10%	**1.105**	1.778
20%	860.220	**0.547**	20%	0.825	**0.544**	20%	**8.663**	3576.321
30%	394.450	**0.711**	30%	18.429	**0.559**	30%	**836.331**	t-o
40%	2267.646	**0.913**	40%	477.893	**0.648**	40%	**79957.043**	t-o
50%	2436.738	**0.949**	50%	30687.162	**0.797**	50%	me	me
60%	6420.507	**2.315**	60%	t-o	**1.161**	60%	me	me
70%	t-o	**10.805**	70%	me	**1.510**	70%	me	me
80%	me	**119.071**	80%	me	**1.388**	80%	me	me
90%	me	**2520.562**	90%	me	**0.918**	90%	me	me

Table 2. MAP inference time (s) comparison on the *probabilistic graphs* dataset with $N = 50$. In bold the best results. "t-o" means time-out (>24hours),"me" means memory error, "pe" means program error. "PL" stands for ProbLog2.1.

	Graph 1		*Graph 2*		*Graph 3*		*Graph 4*		*Graph 5*	
	PL	PITA	PL	PITA	PL	PITA	PL	PITA	PL	PITA
20%	pe	**0.567**	pe	**1.907**	3789.323	**2.075**	4997.582	**1.294**	pe	**1.721**
50%	me	**0.598**	me	**0.702**	me	**0.608**	me	**0.584**	me	**0.628**
80%	me	**0.619**	me	**0.686**	me	**0.623**	me	**0.598**	me	**0.632**

	Graph 6		*Graph 7*		*Graph 8*		*Graph 9*		*Graph 10*	
	PL	PITA	PL	PITA	PL	PITA	PL	PITA	PL	PITA
20%	4596.1	**2.393**	2812.411	**0.951**	3187.299	**1.713**	pe	**1.712**	2955.692	**1.733**
50%	me	**0.637**	me	**9.174**	me	**0.617**	me	**0.661**	me	**0.588**
80%	me	**0.547**	me	**1.140**	me	**3.066**	me	**0.559**	me	**0.598**

Barabasi, A.-L. and Albert, R. 1999. Emergence of scaling in random networks. *Science 286,* 5439, 509–512.

Darwiche, A. 2004. New advances in compiling CNF into decomposable negation normal form. In *16th European Conference on Artificial Intelligence (ECAI 20014)*, R. L. de Mántaras and L. Saitta, Eds. IOS Press, 328–332.

Darwiche, A. and Marquis, P. 2002. A knowledge compilation map. *J. Artif. Intell. Res. 17*, 229–264.

De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., Kimmig, A., Landwehr, N., Mantadelis, T., Meert, W., Rocha, R., Santos Costa, V., Thon, I., and Vennekens, J. 2008. Towards digesting the alphabet-soup of statistical relational learning. In *NIPS 2008 Workshop on Probabilistic Programming.*

De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S., Eds. 2008. *Probabilistic Inductive Logic Programming.* LNCS, vol. 4911. Springer.

De Raedt, L., Kimmig, A., and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, M. M. Veloso, Ed. Vol. 7. AAAI Press/IJCAI, 2462–2467.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D. S., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theor. Pract. Log. Prog. 15,* 3, 358–401.

Jiang, C., Babar, J., Ciardo, G., Miner, A. S., and Smith, B. 2017. Variable reordering in binary decision diagrams. In *26th International Workshop on Logic and Synthesis*. 1–8.

Poole, D. 1997. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artif. Intell. 94*, 7–56.

Poole, D. 2000. Abducing through negation as failure: Stable models within the independent choice logic. *J. Logic Program. 44,* 1-3, 5–35.

Raedt, L. D., Kimmig, A., and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, M. M. Veloso, Ed. 2462–2467.

Riguzzi, F. 2016. The distribution semantics for normal programs with function symbols. *Int. J. Approx. Reason. 77*, 1–19.

Riguzzi, F. 2018. *Foundations of Probabilistic Logic Programming.* River Publishers, Gistrup,Denmark.

Riguzzi, F. and Swift, T. 2010. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010)*. LIPIcs, vol. 7. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 162–171.

Riguzzi, F. and Swift, T. 2011. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theor. Pract. Log. Prog. 11,* 4–5, 433–449.

Riguzzi, F. and Swift, T. 2013. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theor. Pract. Log. Prog. 13,* 2, 279–302.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, L. Sterling, Ed. MIT Press, 715–729.

Shterionov, D. S., Renkens, J., Vlasselaer, J., Kimmig, A., Meert, W., and Janssens, G. 2015. The most probable explanation for probabilistic logic programs with annotated disjunctions. In *24th International Conference on Inductive Logic Programming (ILP 2014)*, J. Davis and J. Ramon, Eds. Lecture Notes in Computer Science, vol. 9046. Springer, Berlin, Heidelberg, 139–153.

Somenzi, F. 2001. Efficient manipulation of decision diagrams. *Int. J. Softw. Tools Technol. Transf. 3,* 2, 171–181.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput. 8,* 3, 410–421.

Van den Broeck, G., Thon, I., van Otterlo, M., and De Raedt, L. 2010. DTProbLog: A decision-theoretic probabilistic Prolog. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, M. Fox and D. Poole, Eds. AAAI Press, 1217–1222.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. 2004a. Logic programs with annotated disjunctions. In *24th International Conference on Logic Programming (ICLP 2004)*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3131. Springer, 431–445.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. 2004b. Logic programs with annotated disjunctions. In *24th International Conference on Logic Programming (ICLP 2004)*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3131. Springer, Berlin, 195–209.