Using Iterative Deepening for Probabilistic Logic Inference

Theofrastos Mantadelis and Ricardo Rocha

CRACS & INESC TEC & Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
{theo.mantadelis;ricroc}@dcc.fc.up.pt

Abstract. We present a novel approach that uses an iterative deepening algorithm in order to perform probabilistic logic inference for ProbLog, a probabilistic extension of Prolog. The most used inference method for ProbLog is exact inference combined with tabling. Tabled exact inference first collects a set of SLG derivations which contain the probabilistic structure of the ProbLog program including the cycles. At a second step, inference requires handling these cycles in order to create a non-cyclic Boolean representation of the probabilistic information. Finally, the Boolean representation is compiled to a data structure where inference can be performed in linear time. Previous work has illustrated that there are two limiting factors for ProbLog's exact inference. The first factor is the target compilation language and the second factor is the handling of the cycles. In this paper, we address the second factor by presenting an iterative deepening algorithm which handles cycles and produces solutions to problems that previously ProbLog was not able to solve. Our experimental results show that our iterative deepening approach gets approximate bounded values in almost all cases and in most cases we are able to get the exact result for the same or one lower scaling factor.

Keywords: Probabilistic Logic Programming, Inference Engine, Cycle Handling, Iterative Deepening, ProbLog.

1 Introduction

ProbLog [8] is a probabilistic framework that extends Prolog with probabilistic facts. ProbLog's most fundamental task is the efficient computation of a query's success probability and, for that, ProbLog employs several inference methods and uses several different state-of-the-art technologies. The most used inference method for ProbLog is *exact inference*. ProbLog is also able to compute conditional probabilities, solve multiple queries and compute the probabilities of answers. State-of-the-art Probabilistic Logic Programming (PLP) systems, such as ProbLog, often use a three step inference mechanism: (i) SLD/SLG logic resolution; (ii) Boolean formula preprocessing; and (iii) knowledge compilation.

Inference in PLP systems impose several challenges which still have not been fully addressed. Currently, an important limitation is the efficiency of knowledge compilation of highly connected graphs. At the Boolean formula preprocessing step, big cyclic graph based problems are also almost intractable. Motivated by the need of providing a solution for these problems, several approximation methods have been proposed. One of the most prominent and used for ProbLog is *program sampling* [8]. Program sampling is able to compute a result for many queries that would be intractable for exact inference, but program sampling is usually much more time consuming than exact inference when the problem is tractable, making it often an unusable inference method. Initial work in ProbLog [8], proposed an approach based in the k-best derivations. This approach works for the calculation of lower bound probabilities with a small k. The early stopped derivations which are used to compute the upper bound probability become intractable even for a small k. The scaling of k-best derivations approach was proven in most cases worst than tabled exact inference, thus making it unusable.

SkILL [4], a Stochastic Inductive Logic Learner which produces First Order Logic theories from probabilistic annotated data, uses MetaProbLog[1] as its inference engine to analyse the probabilistic data. In particular, SkILL uses MetaProbLog's exact inference to compute the success probability of induced theories. When exact inference for a theory is intractable, SkILL then computes the probability of that theory by using MetaProbLog's program sampling inference. Whenever SkILL resolves to program sampling, the time overhead is significant. Motivated by the above observations and SkILL's usage of MetaProbLog, we have identified the need to be able to compute an approximation for intractable queries in speeds comparable to exact inference.

To address the mentioned problems, we propose a new inference method based on *iterative deepening search*. The underlying idea is to perform the Boolean formula preprocessing step in a bounded fashion producing two Boolean formulae: *one more specific* and *one more general* than the exact Boolean formula. Afterwards, we compute the probability of the two bounded formulae as lower and upper bounded probabilities. Finally, after completing an iteration, we can increase the bound and compute the next iteration until we either reach an exact probability, a desirable bound interval, a maximum bound, or time out. Our approach thus incrementally computes the Boolean formula preprocessing step and as a result generates and compiles subformulae that incrementally grow/shrink towards the exact formula creating a lower/upper probability bound, respectively. In this way, we are able to compute good approximations in a very fast way even for the hardest problems.

The main contributions of this paper are:

1. The application of iterative deepening to handle cycles in probabilistic logic programs in order to compute lower and upper bounds.

[1] MetaProbLog is an implementation of the ProbLog semantics and can be found at: `www.dcc.fc.up.pt/metaproblog`. Other implementations are ProbLog1 and ProbLog2 and can be found at: `dtai.cs.kuleuven.be/problog`.

2. The full integration and compatibility of the new algorithm with all existing optimizations and system features in MetaProbLog, such as: variable compaction [12], general (stratified) negation, multiple queries and evidence.
3. An experimental evaluation of iterative deepening using three key datasets against exact inference and program sampling inference. The iterative deepening algorithm clearly over performed the other inference methods in two datasets and equally performed with exact inference at the third dataset.

The rest of the paper is structured as follows. First, we briefly introduce ProbLog and the distribution semantics in Section 2.1. We then present AND-OR graphs, which are a fundamental step for our method, in Section 2.2. The detailed description of our algorithm is given in Section 3. Section 4 contains the experimental evaluation. Finally, future work and conclusions are presented in Section 6.

2 ProbLog

We start by giving a brief introduction to ProbLog which follows the *distribution semantics* [16], and by defining the success probability of logic programs. Then, we describe the exact inference method and how the collective proofs are represented as AND-OR graphs.

2.1 ProbLog and the Distribution Semantics

ProbLog programs use the syntax of Prolog and extend it with probabilistic facts [8]. A ProbLog program T consists of a set of facts annotated with probabilities $p_i :: pf_i$ (called *probabilistic facts*) together with a set of standard definite clauses or definite clauses that also contain positive and/or negative probabilistic facts in their body $h : -b_1, \ldots, b_n$ (called *background knowledge* (BK)). A probabilistic fact pf_i is true with probability p_i. These facts correspond to random variables, which are assumed to be mutually independent. We define $L_T = \{p_1 :: pf_1, \ldots, p_n :: pf_n\}$ as the set of all probabilistic facts in a ProbLog program. Formally, a ProbLog program is of the form $T = L_T \cup BK$. Finally, as syntactic sugar, ProbLog implementations allow probabilistic heads to definite clauses.

We define as possible world $L_T = L_{true} \cup L_{false}$ *and* $L_{true} \cap L_{false} = \emptyset$, where L_{true} and L_{false} are the sets containing all probabilistic facts of the ProbLog program T that are set to true and false, respectively. It is clear that a ProbLog program T has a number of possible worlds exponential to the number of probabilistic facts (2^N where N is the number of probabilistic facts).

The probability of a possible world (P_{world}) equals to the product of the probability of all probabilistic facts in L_{true} and 1 - probability of all probabilistic facts in L_{false}, i.e.,

$$P_{world} = \prod_{p_i :: pf_i \in L_{true}} p_i \cdot \prod_{p_j :: pf_j \in L_{false}} (1 - p_j),$$

where $L_{true} \cup L_{false} = L_T$ and $L_{true} \cap L_{false} = \emptyset$. The sum of the probabilities of all possible worlds equals to:

$$\sum_{w_i \in Worlds} P_{w_i} = 1.0.$$

The most fundamental task of ProbLog is to calculate the success probability of a query. In ProbLog, inquiring the success probability of a query means asking for the probability that a randomly selected possible world satisfies that query. Such worlds contain the probabilistic facts needed to satisfy the query, but can also contain more probabilistic facts. The success probability $P_s(q|T)$ of a query q is the summation of the probabilities of all possible worlds for which there exists a substitution θ such that $q\theta$ is entailed by T, i.e., $P_s(q|T) = \sum_{w_i \in Worlds} P(q|w_i) \cdot P_{w_i}$, where $P(q|w_i) = 1.0$ if there exists a substitution θ such that $w_i \cup BK \models q\theta$ and $P(q|w_i) = 0.0$ otherwise. The equation states that we are able to calculate the success probability of a query by summing the probabilities of all worlds that satisfy the query.

The naive approach of enumerating all possible worlds and then summing the ones that satisfy the query quickly becomes computationally intractable. For that reason ProbLog uses different strategies to calculate the success probability of a query. The most used inference method of ProbLog is the exact inference method with general (stratified) negation and tabling support. ProbLog complies to the closed world assumption and for that reason the ProbLog's general negation mechanism is limited to stratified programs [10]. Exact inference is a three step inference approach:

1. **SLG resolution** is used to prove the query and collect the proofs that compactly represent the possible worlds where the query succeeds. For the purpose of this paper, we will use SLG resolution for ProbLog programs as presented in [10].
2. **Boolean formula preprocessing** takes the compact representation of the possible worlds in order to perform cycle handling [10] and optimize it as a Boolean formula [11].
3. **Knowledge compilation** is used to compile the collected Boolean formula to Reduced Ordered Binary Decision Diagrams (ROBDDs) [1], or to smooth decomposable Deterministic Negated Normal Form (sd-DNNF) [7], or to Sentient Decision Diagrams (SDDs) [6].

2.2 AND-OR Graphs

We represent the collected proofs as an AND-OR graph. An AND-OR graph is a directed graph composed by AND and OR nodes. An AND node indicates that all child nodes must be true, while an OR node indicates that at least one of the child nodes must be true. The SLG derivations of a query q with respect to a logic program can be represented as an AND-OR graph. To solve a query q, the different clauses $(c_{i \in 1..m} : -l_{i,1}, ..., l_{i,n}.)$ of the predicate q are processed as follows. For each different clause c_i all literals $l_{i,j}$ in the body are grouped as

children of an AND node. The different AND nodes are then grouped as children of an OR node labeled with q.

An AND-OR graph of a query has the following characteristics: (i) cycles that appear in the logic program also appear in the AND-OR graph; (ii) for each subgoal g there is only one OR node; (iii) an OR-node has multiple parents if the subgoal is repeated and goals proven as facts are represented by special OR nodes without children, called terminal nodes; and (iv) the edge from a child node to a parent node states that the parent depends on the child node.

Formally, an AND-OR graph for a query q is a directed graph $G = (V_{and}, V_{or}, V_{term}, E)$ with V_{and} a set of AND nodes, V_{or} a set of labeled OR nodes, $V_{term} \subset V_{or}$ a set of terminal nodes, $V_{nonterm} = V_{or} \setminus V_{term}$ and $E \subseteq R$ a set of directed edges, where $R = (V_{and} \times V_{or}) \cup (V_{nonterm} \times V_{and}) \cup (V_{nonterm} \times V_{or})$ and the OR node with label q as root.

In order to compile the collected proofs, ProbLog must first process the AND-OR graph and produce a Boolean formula that does not contain cyclic references but, often, converting a cyclic AND-OR graph to a non cyclic one is a hard task [10]. Furthermore, compiling an AND-OR graph to any of the knowledge compilation approaches has complexity exponential to the tree width of the AND-OR graph [7]. In this paper, we propose a new method to iteratively compute the Boolean formula to two Boolean formulae, one *more specific* and one *more general*. In that way, we are able to compute lower and upper bounds with lower complexity than computing the exact probability.

Figure 1 presents the probabilistic graph for the following ProbLog program, which will be used as our running example.

```
0.5::e(a,b).    0.4::e(a,c).    0.6::e(a,f).
0.2::e(b,a).    0.8::e(b,c).    0.7::e(b,f).
0.9::e(c,a).    0.1::e(c,b).    0.3::e(c,f).

p(X, Y) :- e(X, Y).
p(X, Y) :- e(X, Z), Z \= Y, p(Z, Y).
```

In order to prove the query $p(a, f)$, SLG resolution collects the AND-OR graph presented in Fig. 2. The query defines the entry point of the AND-OR graph which we annotate by shading the node gray. With rhombus we annotate the AND nodes; with ellipses we annotate the OR nodes (notice that all OR nodes are labeled with a logical goal); and with rectangles we annotate the leaf nodes which are the probabilistic facts. The AND-OR graph represents not only the relevant information used to proven the query by SLG resolution but also any cycle found in the proving. For example, observe that the OR nodes $\{p(a, f); p(b, f); p(c, f)\}$ all have paths (by following the directed edges) through AND nodes that would return to the initial OR node, thus creating the cycles.

When computing the exact probability of query q, one requires to handle the cycles that are introduced in the AND-OR graph. ProbLog uses the algorithm presented at [10], which treats positive cycles [3] as failures. The algorithm is implicitly transforming the AND-OR graph to a larger one (in the worst case

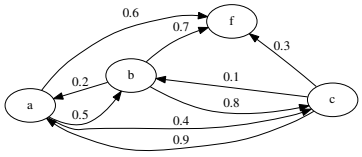

Fig. 1: An example probabilistic graph.

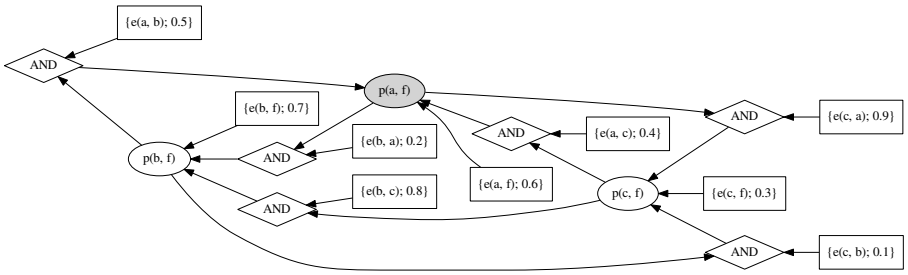

Fig. 2: The AND-OR graph collected by SLG resolution for query $p(a, f)$ for the probabilistic graph of Fig. 1.

exponentially larger). Figure 3 presents the transformed graph where the cycles have been fully handled; we annotate the detected cycles with double octagons. The reader can notice that some nodes, such as $\{p(b, f); p(c, f)\}$, appear in different paths and that others, such as $\{p(a, f) = cycle, p(b, f) = cycle\}$, are characterized as cycles when they appear twice in the same path. The cycle handling algorithm, uses a set of logical rules and memoization that permits the re-usage of computations. This re-usage allows a significant reduction of the work and size of the expansion. Still, the cycle handling remains an exponentially hard task and often generates a very large AND-OR graph where the knowledge compilation step often fails.

3 Iterative Deepening Cycle Handling

The proposed approach does not modify at all the first step (SLG resolution); it introduces a new cycle handling algorithm at the second step (Boolean formula preprocessing) which is fully compatible with all existing optimizations; and it modifies the third step by calling it multiple times in order to compute the probabilities of different bounds.

The underlying idea of our approach is similar to the iterative deepening depth first search (DFS) algorithm but, instead of searching for a specific node, we are interested in traversing the whole graph structure (or as much of it as possible) and transform it to a cyclic free graph.

Algorithm 1 presents the generalized AND-OR graph to ROBDD definitions approach with iterative deepening modifications, which includes the relevant parts of the original cycle handling algorithm together with the extensions re-

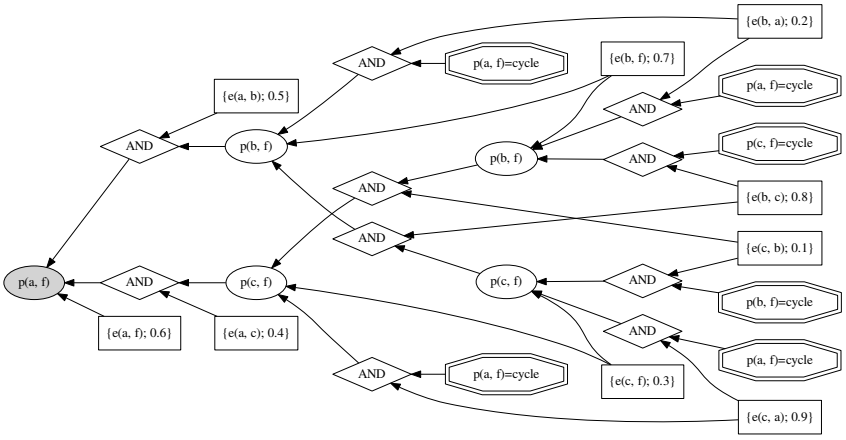

Fig. 3: The cyclic free AND-OR graph that ProbLog produces before ROBDD compilation.

quired to implement an iterative deepening strategy. Optimizations and the handling for general negation has been omitted for simplification. The extensions are noted as underlined text. Our inference method calls Alg. 1 for a user-defined number of iterations and, at each iteration, the bound is increased by a user-defined step.

Algorithm 1 recursively traverses the AND-OR graph structure calling at each recursion level the auxiliary procedure PREPROCESSING_METHOD() (line 2 in Alg. 1), which is responsible of handling an AND-OR graph that contains a single OR node. ProbLog supports several different preprocessing methods. In this work, we used the Recursive Node Merging preprocessing method [11]. The preprocessing procedure is responsible for writing the AND-OR graph as a depth breadth trie. MetaProbLog uses depth breadth tries in order to perform optimizations on the writing of Boolean formulae. For the purposes of this paper, the reader can assume that a depth breadth trie is a simple representation of the Boolean formula. For more details on the preprocessing methods we direct the reader to [11]. We note that these optimizations are independent from this work but our implementation is fully compatible and fully supports them.

Whenever an AND-OR subgraph T_{nested} is found, the algorithm needs to choose the appropriate of four different conditions. First, the algorithm verifies if T_{nested} introduces a positive cycle [3] and handles it as a failure [10] (lines 5-6 in Alg. 1). The second condition occurs when the T_{nested} has been processed earlier and the results can be reused (lines 7-10 in Alg. 1). If neither the first nor second condition apply, the algorithm checks whether T_{nested} was encountered within the introduced bound of the iteration. If T_{nested} is within the bound (lines 11-15 in Alg. 1), then the algorithm will recursively try to compute the newly found reference. Otherwise, if it is out of the bound (lines 16-20 in Alg. 1), then the algorithm will assume either false or true depending on whether it is a lower or upper iteration. After replacing the AND-OR subgraph, the algorithm

Algorithm 1 The generalized AND-OR graph to ROBDD definitions approach with iterative deepening

input The AND-OR graph T, the depth breadth trie DBT where the generated ROBDD definitions are stored, the ancestor AND-OR graphs A_T of the AND-OR graph T, the reference L_{begin} to the next free ROBDD definition, the current depth ($Depth$) in the AND-OR graph, the available bound ($Bound$) of this iteration and a Boolean starting value for $Assumed$.

output Updates DBT to contain the ROBDD definitions generated for T and returns the representative reference L_{end} and the Boolean variable $\underline{Assumed}$ that indicates whether an assumption was taken.

call as $(L_{end}, \underline{Assumed}) :=$ PROCESS$_{\text{AND-OR}}(T, DBT, \emptyset, L_{begin}, 1, \underline{Bound}, \underline{false})$.

1: **function** PROCESS$_{\text{AND-OR}}(T, DBT, A_T, L_{begin}, \underline{Depth}, \underline{Bound}, \underline{Assumed})$
2: $(L_{end}, T_{nested}) :=$ PREPROCESSING_METHOD(T, DBT, L_{begin})
3: **if** $T_{nested} \neq$ **null then** {T *contains a sub AND-OR graph* T_{nested}}
4: $A_{T_{nested}} := A_T \cup \{T\}$
5: **if** $T_{nested} \in A_{T_{nested}}$ **then** {T_{nested} *introduces a cycle*}
6: Replace the occurrence of T_{nested} in AND-OR graph T with **false**.
7: **else if** IS_MEMOIZED$(T_{nested}, A_{T_{nested}}, \underline{Bound - Depth})$ **then**
8: $(L_{T_{nested}}, \underline{Assumed_{T_{nested}}}) :=$ GET_MEMOIZED_RESULT$(T_{nested}, A_{T_{nested}}, \underline{Bound - Depth})$
9: Replace the occurrence of T_{nested} in AND-OR graph T with $L_{T_{nested}}$.
10: $\underline{Assumed := Assumed \cup Assumed_{T_{nested}}}$
11: **else if** $\underline{Depth < Bound}$ **then** {T_{nested} *is not a cycle, neither is memoized and current depth is still in bound*}
12: $(L_{T_{nested}}, \underline{Assumed_{T_{nested}}}) :=$ PROCESS$_{\text{AND-OR}}(T_{nested}, DBT, A_{T_{nested}}, L_{end}+1, \underline{Depth+1}, \underline{Bound}, \underline{Assumed})$
13: Replace the occurrence of T_{nested} in AND-OR graph T with $L_{T_{nested}}$.
14: $L_{end} := L_{T_{nested}}$
15: $\underline{Assumed := Assumed \cup Assumed_{T_{nested}}}$
16: **else** {*Current depth is out of bound*}
17: $\underline{\text{Assume } L_{T_{nested}} \text{ is false for lower inference and true of upper inference}}$
18: $\underline{\text{Replace the occurrence of } T_{nested} \text{ in AND-OR graph } T \text{ with } L_{T_{nested}}.}$
19: $\underline{L_{end} := L_{T_{nested}}}$
20: $\underline{Assumed := true}$
21: **return** PROCESS$_{\text{AND-OR}}(T, DBT, A_T, L_{end}+1, \underline{Depth+1}, \underline{Bound}, \underline{Assumed})$
22: **else** {T *is fully processed*}
23: MEMOIZE$(T, A_T, \underline{Bound - Depth}, L_{end}, \underline{Assumed})$
24: **return** $(L_{end}, \underline{Assumed})$

continues recursively by increasing by one the used depth (line 21 in Alg. 1). Finally, when an AND-OR graph is fully processed (contains a single node), the result is memoized for reuse and the result is returned (lines 22-24 in Alg. 1).

For cyclic handling, we use a memoization technique that compares the subsets of the ancestors of AND-OR graphs [10]. This technique allows us to discover cycles and to widen our re-usage compared with the normal DFS strategy. When the algorithm memorizes a computed AND-OR graph, it keeps track of the ancestors in list A_T (called the *ancestor list*) of the AND-OR graph in order to identify the possibly introduced cycles. With iterative deepening, the algorithm also requires to memoize the number of recursions remaining ($Bound - Depth$) and whether an assumption was taken ($Assumed$) for computing the AND-OR graph (line 23 in Alg. 1). When the algorithm checks whether a memoized result can be reused, in addition of checking the ancestor list, it also needs to check whether the number of used recursions of the stored AND-OR graph is equal or greater than the currently remaining recursions. In case the memorized recursions are less than the currently remaining recursions, it means that the memorized AND-OR graph reference contains less probabilistic information than what the current iteration is able to compute and thus the memorized result is not reused. This way we can allow iterations with different bounds to use previously computed results.

For example, assume that the AND-OR graph $t(1)$ with the ancestor list $A_{t(1)}$ at the lower iteration with $Bound = 5$ in the recursion with $Depth = 3$ has been computed as the ROBBD defintion $L_{t(1)}$ with no assumptions. The algorithm then memoizes the term: $and - or\_graph(t(1), A_{t(1)}, 2, L_{t(1)}, false)$ and can reuse this computations of $t(1)$ in any lower iteration with ancestor list $A^{new}_{t(1)}$ as long $Bound - Depth \leq 2$ and $A_{t(1)} \subseteq A^{new}_{t(1)}$.

When the current depth equals the bound of the current iteration, any occurrence of a sub graph is assumed to be *false* for lower bounded iterations and *true* for upper bounded iterations. In this way, we lose probabilistic information but we ensure that the probability will be a lower/upper bound of the exact probability. This simple strategy gives us lower and upper bounds for AND-OR graphs that do not contain general negation. By memorizing and returning for each AND-OR subgraph whether an assumption was taken to compute the result, we know if the stored result of the AND-OR graph is the equivalent of the exact result for that AND-OR graph. This allows us to detect when we computed an iteration that provides an equivalent to exact Boolean formulae regardless the actual probability results.

Returning to our example, we illustrate how the AND-OR graph of Fig. 1 would appear for the first two iterations of our iterative deepening algorithm. The first iteration would produce the AND-OR graph presented in Fig. 4 and the second iteration would produce the AND-OR graph presented in Fig. 5. For this specific example, on the third iteration, the computed AND-OR graph would be identical with the AND-OR graph computed by the exact method, which is presented in Fig. 3. With octagons we annotated the nodes that the iterative deepening approach assumed true or false.

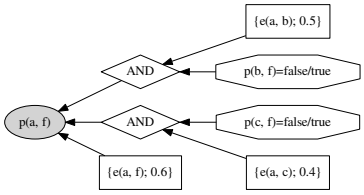

Fig. 4: AND-OR graph after one iteration of the iterative deepening algorithm.

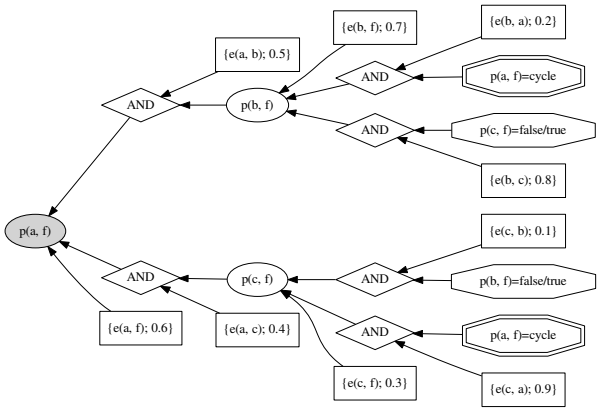

Fig. 5: AND-OR graph after two iterations of the iterative deepening algorithm.

After each iteration of the iterative deepening algorithm, ProbLog compiles the computed Boolean formulae using a knowledge compilation approach in order to efficiently compute the probability. For our implementation and description, we used ROBDDs.

Finally, this algorithm is executed iteratively with the user being able to choose: (i) the starting bound; (ii) the step; and (iii) several ending conditions, such as, specific bound, lower to upper probability difference being less than a value, or reach a time limit. The algorithm also automatically terminates if it detects the computation of the exact probability.

Implementation-wise, we have introduced three different inference methods that the user can use, namely: *lower iterative*; *upper iterative*; and *bounded iterative* (the combination of the two). These new methods are compatible with all features and optimization techniques that exact inference uses.

3.1 General Negation & Conditional Probability

ProbLog programs with general negation impose a complication to the iterative algorithm assumption taking. Whenever the algorithm takes an assumption, the number of enclosing negations of the assumed subgraph define whether the assumption should be false or true. If the enclosed graph is an odd number of times enclosed then we must assume true for lower bound and false for upper

bound. In order to handle the above complication, the moment we encounter a general negation, we need to push it deeper in the computation by using De Morgan's laws, i.e., in practice we are transforming the next AND-OR subgraph. This mechanism causes a small overhead when general negation is encountered.

ProbLog queries that have evidence impose the interesting theoretical question whether the results are bounded. In ProbLog, we define the probability of a query q with evidence e as $P(q|e) = \frac{P(q \cap e)}{P(e)}$. Our iterative method computes lower and upper bounds, $P_l(q|e) = \frac{P_l(q \cap e)}{P_l(e)}$, $P_u(q|e) = \frac{P_u(q \cap e)}{P_u(e)}$, respectively. We would like to prove that $P_l(q|e) \leq P(q|e) \leq P_u(q|e)$. Unfortunately, we can prove the opposite by using set theory. Assume two sets Q, E such that (i) at lower iteration i, $Q_i = Q$ and $\emptyset \neq E_i \subset E$ and that (ii) $Q_i \cap E_i = Q \cap E$ then clearly $P_l(Q_i \cap E_i) = P(Q \cap E)$ and $P_l(E_i) < P(E)$ resulting in $P_l(Q_i|E_i) > P(Q|E)$. For example, if $Q = Q_i = \{pf1 \cup pf2 \cup pf3\}$, $E = \{pf1 \cup (pf2 \cap pf4)\}$ and $E_i = \{pf1 \cup (pf2 \cap false)\} = \{pf1\}$ then $Pl(Q_i|E_i) = \frac{P(\{pf1\})}{P(\{pf1\})} = 1 > P(Q|E) = \frac{P(\{pf1\})}{P(\{pf1 \cup (pf2 \cap pf4)\})}$. Similarly one can miss proof $P_u(q|e) \geq P(q|e)$.

Thus, the proposed algorithm does not compute lower and upper bounded probabilities for queries with evidence. Our solution to this problem is to swap the divisors of the lower and upper iterations and compute the probabilities as follows: $P_l(q|e) = \frac{P_l(q|e)}{P_u(e)}$ and $P_u(q|e) = min(\frac{P_u(q|e)}{P_l(e)}, 1)$. Clearly, as $P_u(e) \geq P(e), P_l(q \cap e) \leq P(q \cap e)$ then $\frac{P_l(q \cap e)}{P_u(e)} \leq \frac{P(q \cap e)}{P(e)}$ (similarly for the upper bound).

4 Experimental Results

For the purpose of this paper, we have implemented the proposed algorithm in MetaProbLog. MetaProbLog is an efficient implementation of ProbLog that is closely integrated with Yap Prolog [5] and is used in SkILL [4]. MetaProbLog supports both ROBDDs and sd-DNNFs as a knowledge compilation language. Previous experimental evaluations have shown that ROBDDs are able to solve more problems than sd-DNNFs in the context of MetaProbLog and sd-DNNFs only perform better in conditional queries [12,17]. For that reason, in our experiments we use ROBDDs.

We have experimented with 3 benchmark sets, namely Alzheimer, Smokers and Grid, which have been previously used for testing the performance of different ProbLog implementations. The Alzheimer benchmark set [8] was generated from a real-world biological dataset of Alzheimer genes. Due to the complexity and importance of this dataset, it has been established as the most used testing ground for ProbLog. We used three different queries (Q1, Q2, Q3) and their reversed instances ($\overline{Q1}$, $\overline{Q2}$, $\overline{Q3}$)[2]. In order to see the scaling of the inference methods based on the size of the graph, we increased the number of edges by

[2] Q1 = p(hgnc_983,hgnc_620), $\overline{Q1}$ = p(hgnc_620,hgnc_983),
Q2 = p(hgnc_582,hgnc_620), $\overline{Q2}$ = p(hgnc_620,hgnc_582),
Q3 = p(hgnc_983,hgnc_582) and $\overline{Q3}$ = p(hgnc_582,hgnc_983).

300 in each scale step starting from 1500 edges until 6000 edges (16 scale steps in total). The Smokers benchmark set [14] was introduced for testing multiple queries and queries with evidence. The scale parameter for Smokers is the number of persons in the social network, currently our dataset has up to 51 people. The Grid benchmark set [17] was constructed as a worst case scenario for ProbLog1 and MetaProbLog inference and is a fully connected grid that always contains the probabilistic information at the deepest step and, as such, it is the worst case scenario for our iterative algorithm.

The environment for our experiments was a Supermicro AS-2042G-72RF4 server with four AMD Opteron(tm) Processor 6376 (16 cores each, 64 cores in total) and 256GB of RAM memory. The benchmarks where executed concurrently and each had a total of one hour for time out.

The foremost target of our approach is to enable us to compute an approximation in queries where exact inference is unable to compute any result. Furthermore, we would like our iterative inference method to be able to compute the exact inference and detect its computation when that is possible. Further than simply comparing our approach with the usual exact inference, we also used variable compaction for the Alzheimer dataset as presented in [12]. We noticed that variable compaction permitted us to compute more upper bounded queries and that, in general, variable compaction improved the performance (decreased the runtime) of the iterative inference method.

Table 1 presents the scaling results of our approach compared with exact inference. The queries Q1 to $\overline{Q3}$ are sorted from easier to hardest. One can notice that, for Alzheimer queries, exact inference usually times out at after 3000 edges. The presented iterative approach almost always computes results for all Alzheimer queries and in most cases computes the exact result for at least one scaling factor lower than what exact inference would compute. From a theoretical point of view, we were expecting iterative deepening to be able to compute the $N-1$ iteration of all benchmarks that exact inference was able to return the result. Theoretically, the complexity of computing iterations from 1 to $N-1$ is equal to $O(N)$ for iterative deepening strategies. Finally, we notice that computing upper bounded Boolean formulae is a significantly harder task than computing lower bounded Boolean formulae and some times even harder than computing exact Boolean formulae.

Regarding the Smokers datasets, we see similar behavior as with the Alzheimer dataset. The exact inference stops at queries with up to 40 people while our iterative deepening approach computed results for all our queries. Finally, for the Grid dataset our approach behaved as we expected. Our iterative deepening approach is able to compute the same queries as with exact inference. This was expected as the Grid problems push the probabilistic information very deep in the iterations and always time out on the knowledge compilation step. The results for the program sampling inference method show that it underperforms in the Alzheimer dataset problems, but it is ideal to solve problems like the ones introduced by the Grid dataset, where it easily solves the 15x15 graph.

Dataset / Query	Exact no	Exact comp	Program Sampling	Lower Iterative no	Lower Iterative comp	Upper Iterative no	Upper Iterative comp
Alzheimer Q1	6000	6000	6000	6000 (6000)	6000 (6000)	6000 (6000)	6000 (6000)
Alzheimer $\overline{Q1}$	5100	3900	3300	6000 (3300)	6000 (3300)	2700 (2700)	3900 (2700)
Alzheimer Q2	3000	3300	6000	6000 (3000)	6000 (2700)	6000 (3000)	6000 (3000)
Alzheimer $\overline{Q2}$	3000	3300	2100	6000 (2400)	6000 (2400)	2400 (2400)	3900 (2700)
Alzheimer Q3	3000	3000	2700	6000 (2400)	6000 (2400)	6000 (2400)	6000 (2400)
Alzheimer $\overline{Q3}$	2400	2400	3900	5700 (2100)	6000 (2100)	6000 (2100)	6000 (2100)
Smokers	40	–	–	51 (40)	–	51 (40)	–
Grid	7x7	–	15x15	7x7 (7x7)	–	7x7 (7x7)	–

Table 1: Highest scaling results for exact, program sampling, lower iterative, and upper iterative inference methods over the three datasets (columns **no**). For the Alzheimer dataset, we also present the results with variable compaction (columns **comp**). In parenthesis, we present the highest scaling factor at which the iterative inference detected that it has computed the exact probability.

The second question we want to answer is how good is our approximations. Theoretically, it is difficult to answer this question, as we do not have a way to compute the amount of probabilistic information the next iterations would add. Practically, for most queries, we are able to compute both an upper and lower bound giving an indication of how good our approximation is. We use the same notion of precision as used in [18], but we also distinguish the queries where we are able to compute the exact probability. Table 2 shows the results.

Precision	Alzheimer no	Alzheimer comp	Smokers no
Exact (< 0.00001)	40 (+9)	41 (+9)	304
Almost Exact ($[0.00001, 0.01)$)	0	2	22
Tight Bounds ($[0.01, 0.25)$)	29	26	27
Loose Bounds ($>= 0.25$)	27 (-9)	27 (-9)	4
# Queries Solved by Iterative Deepening	96	96	357
# Queries Solved by Exact Inference	48	46	305
# Queries Solved by Program Sampling	55	-	-

Table 2: Precision results of computed bounds by iterative deepening inference (columns **no**). For the Alzheimer dataset, we also present the results with variable compaction (column **comp**). In brackets are the results where exact probability is computed but is not detected. For program sampling we count the number of programs that reached a 95% confidence to be within a 0.01 interval.

For the Alzheimer dataset, we can see that there is a beneficial impact from enabling variable compaction. The impact comes from improved results in the upper bound computations. We also want to note that sometimes, even if the computed upper bounds are high, the computed lower bound probabilities are the exact probability but our system could not detect that (in brackets we present how the results would be affected if we could identify those cases).

In this regard, the Alzheimer $\overline{Q1}$ imposes an interesting problem for discussion. For that specific query, exact inference managed to compute the probability for graphs with up to 5100 edges. Our iterative deepening approach is able to compute the exact probability (as a lower bound) of that query for graphs with any number of edges in a identical execution time, but it fails to compute upper bound probabilities and it is also unable to automatically detect that the exact probability has been computed. The underlying reason for the complexity of this specific query is that it contains a complex graph that always leads to cycles but do not contributes to the query. The iterative deepening approach for the lower bound is able to drop this graph but is unable to detect that it is actually computing the exact probability. On the other hand, the upper bound computation is assuming that the complex graph contributes to the probability mass and always returns a probability of 1.0.

5 Related Work

Lately, there has been a growing interest in combining probabilistic information with logic expressions, giving rise to different PLP systems, such as PRISM [16], IBAL [13], Alchemy [14], ProbLog [8] and PITA [15], among others. These systems use both exact and approximate inference methods in order to compute the marginal or/and conditional probabilities.

A similar inference method with our proposal is mentioned in [2]. Their iterative method is not described in detail and the authors only mention that it underperforms exact inference. By examining the provided source code, we have derived that their iterative method is used to generate growing subformulae that are given for knowledge compilation, i.e., their system handles the cycles before, at the logic part, assuming that the probabilistic derivations do not contain any cyclic information. Thus, their method does not generate a cyclic structure when representing the Boolean formulae like is in our proposal. As they conclude, exact inference or a k-best approximation is more appropriate in their setting (lack of cycles). Another similar approach is the *anytime (approximate) inference* method [18]. The main difference to our approach is that anytime inference fully constructs a CNF formula by executing the exact Boolean formula preprocessing step once and, then, performs incrementally the knowledge compilation step to a set of chosen subformulae. Theoretically, iterative deepening inference and anytime inference could be combined in order to improve the results of each other. Iterative deepening could be used in the Boolean formula preprocessing step and anytime inference in the knowledge compilation step and, in that way, the two approaches could be seen as the complement of each other. Finally [13] mentions the use of an iterative deepening algorithm in order to provide anytime inference for IBAL but the details of the algorithm are omitted from the paper.

6 Conclusions & Future Work

We have introduced a new approximate inference method for probabilistic logic programs based on a iterative deepening algorithm that, at each iteration, computes lower and upper bounds. Our algorithm is able to detect when an iteration would compute the exact probability either because the bounds converge or because the iteration examines the complete AND-OR graph. The proposed inference method can be used for any logic based system that collects SLG-derivations and needs to extract a cycle free Boolean formula from the derivations. This includes the PLP systems ProbLog1, ProbLog2, MetaProbLog, PITA and a version of PRISM that uses MTBBDs. Furthermore, some ASP systems use similar technology to handle Non-tight programs [9].

We also discuss how the new inference method is able to handle conditional queries and queries with general negation. Furthermore, our new inference method is compatible with all optimizations that the current system supports and, specifically, when it is combined with variable compaction, it is able to compute deeper iterations that enables us to get better bounds. The current implementation in MetaProbLog provides three new inference methods, namely: `problog_lower_iterative/2`, `problog_upper_iterative/2` and the combination of the two `problog_bounded_iterative/2`.

We performed a set of experiments on important applications of ProbLog that cover a wide range of different problems and we showed how our method enables us to solve queries that for exact inference were intractable. With the Alzheimer dataset, we presented the beneficial impact of variable compaction for our method. We used the Smokers dataset in order to compare our method against exact inference in the tasks of conditional and multiple queries. Our method clearly outperforms the exact inference being able to return results for all queries tests and returning the exact result for all but one query that exact inference could compute. Finally, we used the Grid dataset as a worst case scenario problem for our approach. Using Grid, we showed that in a worst case scenario our method performs similarly with exact inference, as expected.

For future work, we will extend the algorithm to use multi-threading in order to perform multiple knowledge compilations at the same time; investigate how to theoretically tighten the bounds of conditional queries; and take advantage of previous iterations compiled ROBDDs for incremental compilation. The development of this inference method was motivated by the SkILL system [4] and, as such, we intent to integrate the new method in SkILL. Finally, we are studying the combination of our approximate method with $T_P - Compilation$ [18] in order to boost even further the knowledge compilation step.

Acknowledgments

We want to thank the anonymous reviewers for their valuable comments. This work is partially funded by the ERDF through the COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT as part of project UID/EEA/50014/2013.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers 27(6), 509–516 (1978)
2. Bragaglia, S., Riguzzi, F.: Approximate inference for logic programs with annotated disjunctions. In: International Conference on Inductive Logic Programming (ILP). pp. 30–37 (2011)
3. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of ACM 43(1), 20–74 (1996)
4. Côrte-Real, J., Mantadelis, T., de Castro Dutra, I., Rocha, R.: SkILL - a stochastic inductive logic learner. In: International Conference on Machine Learning and Applications (ICMLA). pp. 555–558 (2015)
5. Costa, V.S., Rocha, R., Damas, L.: The YAP prolog system. Theory and Practice of Logic Programming (TPLP) 12(1-2), 5–34 (2012)
6. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: International Joint Conference on Artificial Intelligence (IJCAI). vol. 2, pp. 819–826 (2011)
7. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence and Reasoning (JAIR) 17, 229–264 (2002)
8. Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming (TPLP) 11(2-3), 235–262 (2011)
9. Lierler, Y., Maratea, M.: Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In: Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 346–350 (2004)
10. Mantadelis, T., Janssens, G.: Dedicated tabling for a probabilistic setting. In: International Conference on Logic Programming (ICLP). Leibniz International Proceedings in Informatics (LIPIcs), vol. 7, pp. 124–133 (2010)
11. Mantadelis, T., Rocha, R., Kimmig, A., Janssens, G.: Preprocessing Boolean Formulae for BDDs in a Probabilistic Context. In: European Conference on Logics in Artificial Intelligence (JELIA). pp. 260–272 (2010)
12. Mantadelis, T., Shterionov, D., Janssens, G.: Compacting boolean formulae for inference in probabilistic logic programming. In: Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 425–438 (2015)
13. Pfeffer, A.: IBAL: A probabilistic rational programming language. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 733–740 (2001)
14. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62(1-2), 107–136 (2006)
15. Riguzzi, F., Swift, T.: The PITA system: Tabling and answer subsumption for reasoning under uncertainty. Computing Research Repository abs/1107.4747 (2011)
16. Sato, T., Kameya, Y.: PRISM: A language for symbolic-statistical modeling. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 1330–1339 (1997)
17. Shterionov, D.S., Janssens, G.: Implementation and performance of probabilistic inference pipelines. In: International Symposium on Practical Aspects of Declarative Languages (PADL). pp. 90–104 (2015)
18. Vlasselaer, J., Van den Broeck, G., Kimmig, A., Meert, W., De Raedt, L.: Anytime inference in probabilistic logic programs with Tp-compilation. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 1852–1858 (2015)