Explanation-based Approximate Inference in Stratified Probabilistic Logic Programs

Anahí Coimbra Maciel[1], **Denis Deratani Mauá**[1]

[1]Institute of Mathematics, Statistics and Computer Science, University of São Paulo

anahi@usp.br, ddm@ime.usp.br

Abstract. *We introduce two novel search-based algorithms for approximate inference in stratified probabilistic logic programs. Both methods build upon the concept of explanations: minimal conjunctions of literals over probabilistic atoms that logically entail a query. Our first contribution is an algorithm that computes bounds on query probabilities by simultaneously searching for explanations and refutations. The second contribution is an algorithm that transforms probabilistic logic programs into an ASP encoding and uses solution enumeration by optimality to enumerate explanations in order of probability. Finally, experiments across two domains demonstrate that our approaches often outperform existing methods, offering a tradeoff between reliability and accuracy.*

1. Introduction

Neuro-symbolic (NeSy) Logic Programming is a paradigm that integrates neural network learning and probabilistic logic reasoning, combining the strengths of both approaches [Huang et al. 2021, Li et al. 2023, Li et al. 2024, Geh et al. 2024]. Neural Networks provide effective subsymbolic to symbolic mapping by gradient-based learning, while logic programming provides intuitive high-level expert knowledge representation and explainable and controllable reasoning.

Performing inference or learning in this hybrid neural-logical setting can be computationally expensive, particularly when dealing with large or complex probabilistic logic programs (PLPs) and many neural network components, as these tasks require summing over an exponential number of intended models for a given query. To counter that downside and to make NeSy Logic Programming systems more scalable, researchers have proposed approximate inference and learning methods [Renkens et al. 2012, Manhaeve et al. 2021, van Krieken et al. 2023, Dierckx et al. 2025].

In this work, we center our attention on approximate inference algorithms. Many such methods consist in searching for a high probability subset of derivation proofs for a given query, which can be computationally easy to find by applying backward chaining (a.k.a., SLD resolution) from the query atom.

For example, [Manhaeve et al. 2021] proposed DPLA*, an A*-based search on the SLD tree to find the most probable derivation proof(s) of a query. The SLD tree is a structure where each node corresponds to a partial proof (defined as a conjunction of probabilistic facts); the search is guided by an admissible heuristic estimating the probability of completing a proof from a given node which guarantees that the most probable proof is found first. The neuro-symbolic framework Scallop [Huang et al. 2021] employs *top-k*, a forward chaining inference approach that uses relational algebra operators

to limit the size of intermediate tuple sets to the k most probable items at each execution step. However, because this restriction is applied locally, it does not guarantee that the globally optimal proof set will be found.

Similarly, [Renkens et al. 2012] proposed *k-optimal*, a greedy search over the SLD trees that finds the set of at most k proofs maximizing query probability (which, due to overlap, may be different from the k-best set). On the other hand, [Mantadelis and Rocha 2017] applied an iterative deepening search to handle cycles in probabilistic logic programs and compute bounds on query probabilities. More recently, [Dierckx et al. 2025] proposed an approximate inference method for neuro-symbolic systems where the logical component is encoded as a weighted CNF formula. By constructing approximating arithmetic circuits (ACs) through a DPLL-style search, their approach computes bounds on the weighted model count (WMC), which are then used to derive guarantees on the gradients.

Nevertheless, the syntactic approach presents one important drawback. Because they are tied to the particular form that knowledge is expressed in the program (i.e., its syntax), and not to its semantics, derivation proofs can be unnecessarily large and redundant [Renkens et al. 2012]. A promising alternative is to use a semantic approach by replacing the search for derivation proofs by a search for *explanations*, defined as a minimal conjunction of literals that entails the query.

To that end, [Renkens et al. 2014] proposed an algorithm to search for explanations via solving MAX-SAT instances. The algorithm first converts the probabilistic logic program into an equivalent WMC problem over a CNF formula. This formula is then modified such that the models of the modified formula encode partial interpretations of the original formula (or equivalently, 3-valued interpretations). Then the algorithm successively searches for a next maximum weight assignment (MAX-SAT) by incorporating constraints discarding the previous found explanations.

However, this approach encounters a number of limitations: the conversion from PLP to WMC potentially adds a large number of auxiliary variables to the program; also, the successive calls might create programs that are increasingly difficult to solve due to the addition of constraints that rule out previous found explanations.

Here we avoid the overhead of logic program to CNF conversion by casting the search for maximum probability explanation within the framework of Answer Set Enumeration by Optimality (ASEO) [Pajunen and Janhunen 2021], which acts similar as MAX-SAT but operates directly over Answer Set Solvers (e.g., clingo). ASEO enumerates answer set in order according to an objective function that assigns weights to literals. Starting from the optimal answer set (i.e., the solution with that maximizes the value of the objective function), ASEO recursively enumerates the next-best solutions. We can apply this technique to approximate probabilistic inference by defining the objective function in terms of the probabilities of the probabilistic facts and enumerating the most probable models of a query in order, yielding increasingly accurate lower bounds on the query probability.

In our *Explanation ASEO* algorithm, rather than explicitly searching a proof space, we transform the PLP into an Answer Set Programming (ASP) encoding via the double-variable encoding of probabilistic facts from [Renkens et al. 2014] and delegate the search

for explanations to clingo [Gebser et al. 2019], taking advantage of this highly optimized system. Clingo is an ASP solver with a built-in ASEO implementation, allowing explanations to be enumerated in decreasing order of probability by progressively adding weak constraints that exclude previously found solutions. A key difference between our approach and the one presented by Renkens et al. is the way the disjoint sum problem is handled: while in [Renkens et al. 2012] binary decision diagrams (BDDs) are used, we choose to handle the overlapping explanations algebraically via the inclusion-exclusion principle, without the need for auxiliary structures.

Our second contribution is *Explanation-Refutation Search*, an algorithm that searches directly on the space of explanations and uses Answer Set Solvers only to certify if a candidate set of literals is an explanation, refutation (an explanation for the negated query) or "none". This ternary classification allows us to significantly prune the search space and implement the search in a memory-efficient way by noting that only “none” search nodes need to be further explored at each step and that we only need to memorize explanations/refutations, which are by design limited.

This anytime inference algorithm provides a point estimate with error bounds that converge to the optimum given enough resources. These bounds are computed over the explanation/refutation sets, using the inclusion-exclusion principle to circumvent the disjoint sum problem. Furthermore, the method not only outputs the calculated probabilities but also the sets of explanations and refutations, which can be used to re-compute the query if probabilities associated to atom change, as is needed for learning.

Finally, our experimental results show that *ExpRef* provides a highly robust alternative for approximate inference in stratified PLPs. Our approach significantly outperforms existing methods in terms of reliability while maintaining satisfactory approximation accuracy.

2. Background

We first define some concepts that will be used in the next sections. We assume the reader is familiar with concepts of propositional logic.

2.1. Logic Programs

Fix a vocabulary of predicate names, constants and logical variables. An atom is of the form $p(t_1, \ldots, t_n)$ where p is a *predicate name* of arity n and $(t_1, \ldots, t_n)$ are *terms* (constants or variables). A *literal* is either an atom A or the negation of an atom, written $not\ A$ [Fierens et al. 2015]. A (normal) *logic program* (LP) is defined as a finite set of facts and rules. A rule is a clause:

$$A \leftarrow L_1, \ldots, L_n, \tag{1}$$

where the head A is an atom and the body $L_1, \ldots, L_n$ is a finite conjunction of literals. Rules with empty bodies are called *facts*. A variable-free (*ground*) program is obtained by *grounding*: substituting each variable with every possible constant [De Raedt and Kimmig 2015].

A model is an interpretation (an assignment of a specific truth value to each ground atom in the LP) that satisfies every rule. The model is minimal if it is not a subset of any

other model (when the model is taken as its set of true atoms). For negation-free programs, there exists a unique minimal model called the *Least Herbrand Model* (LHM). For programs with negation, we adopt the *stable model semantics* [Cozman and Mauá 2017]: given an interpretation *I*, the *reduct* P^I is the negation-free program obtained by removing rules whose body contains a negated atom satisfied by *I*, and stripping remaining negative literals. *I* is a *stable model* of *P* if it is the LHM of P^I. In this work, we restrict our scope to *stratified* programs (programs whose dependency graph contains no cycle through a negative edge), whose semantics are defined by their unique stable model.

A common inference mechanism in negation-free logic programs is *SLD resolution*, a backward-chaining proof procedure that starts from a goal clause and repeatedly resolves its literals against matching rule heads, substituting each selected literal with the corresponding rule body until the empty clause is reached [Lloyd 2012]. SLD resolution produces a *SLD tree*, where each branch corresponds to sequence of rule applications and each leaf corresponds to either a successful proof or a failure. SLD resolution is extended for programs with negation to SLDNF resolution, which incorporates the notion of negation as failure [Apt and Doets 1994].

2.2. Probabilistic Logic Programs

A probabilistic logic program (PLP) is composed by a set of probabilistic facts $p :: A$, where $0 < p < 1$ and A is a ground atom, as well as a set of normal logical rules. As usual, we assume that no two probabilistic facts share the same atom and that probabilistic facts do not appear in the head of rules.

Under *Sato's distribution semantics* [Sato 1995], a PLP defines a probability distribution over LPs. Each LP is obtained by a *total choice*, that is, by independently including each probabilistic fact f_i with probability p_i or excluding it with probability $1 - p_i$. The probability of a total choice $F' \subseteq F$, where F is the set of atoms in probabilistic facts in the PLP, is thus

$$Pr_F(F') = \prod_{f_i \in F'} p_i \prod_{f_i \in F \setminus F'} (1 - p_i). \tag{2}$$

Given a PLP, we define an *explanation* $\mathcal{E}$ of a given Boolean query q as a minimal conjunction of literals over probabilistic atoms such that q is true in any logic program induced by a total choice F' that is consistent with $\mathcal{E}$, that is, which contains all positive literals of $\mathcal{E}$ and none of its negative literals. Minimality of an explanation $\mathcal{E}$ refers to set inclusion, meaning that removing any literal from $\mathcal{E}$ makes it cease from being an explanation. Note that the leaves of an SLD/SLDNF resolution tree are hence not necessarily explanations (see example below). Symmetrically, we define a *refutation* as an explanation of $\neg q$.

The *success probability* of a Boolean query q is the probability that it is entailed by a randomly sampled LP $L_{F'}$:

$$Pr(q) = \sum_{\substack{F' \subseteq F \\ L_{F'} \models q}} Pr_F(F') = Pr(\bigcup_{\mathcal{E} \models q} \mathcal{E}), \tag{3}$$

where we union over explanations $\mathcal{E}$ with probabilities $Pr(\mathcal{E}) = \prod_{\ell \in \mathcal{E}} Pr(\ell)$, with $Pr(\ell) = p_i$ if $\ell = f_i$ and $Pr(\ell) = 1 - p_i$ if $\ell = \neg f_i$, given some probabilistic fact f_i.

Calculating the success probability of a query is a computationally difficult task. In particular, simply adding the probabilities of the explanations for the query could give an incorrect result due to overlapping explanations, in what is called the *disjoint sum problem* [De Raedt and Kimmig 2015].

Example. Consider the following stratified PLP written in ProbLog's syntax [Fierens et al. 2015]:

```
0.5::coat.  0.4::sweater.
warm :- coat, sweater.
warm :- coat, not sweater.
warm :- not coat, sweater.
```

The program represents a distribution over $2^2 = 4$ LPs, each associated to a stable model. SLDNF produces proofs `{coat, sweater}`, `{coat, not sweater}` and `{not coat, sweater}`, while `{coat}` and `{sweater}` are explanations under our definition. The refutation of $warm$ is `{not coat, not sweater}`. The success probability of query *warm* is 0.7.

3. Two Novel Approximate Inference Algorithms

We now present our two contributions. We start by the search of explanation space.

3.1. Explanation-Refutation Search

We propose to replace the search of syntactic proofs by explanations and refutations, which are potentially significantly shorter and insensitive to knowledge encoding.

Formally, given a probabilistic logic program and a query atom q, we aim to find a set of explanations $\mathcal{P}$ and a set of refutations $\mathcal{R}$ that bound the probability of q:

$$0 \leq a = Pr(\mathcal{P}) \leq Pr(q) \leq 1 - Pr(\mathcal{R}) = b \leq 1\,. \tag{4}$$

We assume that the PLP and the query q are fixed in the following. Within this setting, we search the lattice of the power set of consistent probabilistic literals simultaneously for explanations and refutations, progressively tightening the interval $[a, b]$ for the probability of q. A search state refers to a set $\mathcal{C} = \{\mu_1, \ldots, \mu_n\}$ where each μ_i is a literal. To maximize computational efficiency and avoid heavy string-manipulation overhead, each literal is mapped to a unique integer identifier. Furthermore, we take advantage of the fact that any superset of an explanation of q cannot be a refutation, and vice versa, to prune branches and significantly accelerate the exploration.

To explore the explanation space without incurring exponential memory overhead associated to breadth-first search, we implement an Iterative Deepening Depth-First Search (IDDFS) strategy.

Algorithm 1 outlines the core depth-bounded depth-first search (DFS) routine, taking as arguments the query q, the list of probabilistic literals $\mathcal{L}$, the set of explanations

$\mathcal{P}$, the set of refutations $\mathcal{R}$, the lower bound a, the upper bound b, the maximum number of explanations/refutations k and the maximum depth parameter *maxDepth*. Rather than running as a standalone search, this DFS function serves as the internal subroutine for the IDDFS algorithm, which calls it successively inside an outer loop.

The process initializes at $maxDepth = 1$ with empty sets $\mathcal{P} = \mathcal{R} = \emptyset$ and initial bounds $a = 0$ and $b = 1$. Algorithm 1 is run sequentially, incrementing the $maxDepth$ parameter by 1 at each successive iteration. This iterative deepening structurally guarantees that smaller, more concise explanations and refutations are discovered before larger ones. Sets $\mathcal{P}$ and $\mathcal{R}$, along with corresponding probability bounds a and b, are preserved at the end of each DFS pass and passed back as inputs into the next execution. This cycle continues until $maxDepth$ reaches a user-defined threshold tailored to the problem's characteristics, or until the bounds converge.

Function `DepthBoundedDFS(`*q, $\mathcal{L}$,$\mathcal{P}$,$\mathcal{R}$,a,b,k,maxDepth*`)`
  $S \leftarrow$ *newStack() ;* `// Stack of current set, depth`
  $\mathcal{L}' \leftarrow$ *sort $\mathcal{L}$ by probabilities;*
  $S.push(\emptyset, 0)$
  while *S is not empty and ($|\mathcal{P}| < k$ or $|\mathcal{R}| < k$)* **do**
    $(\mathcal{C}, depth) \leftarrow Stack.pop()$
    if $\exists p \in (\mathcal{P} \cup \mathcal{R})$ *such that* $p \subseteq \mathcal{C}$ **then**
      continue *;* `// Pruning superset branches`
    end
    `/* Updating probabilities */`
    if `IsExplanation(`$\mathcal{C}, q$`)` *and* $depth == maxDepth$ **then**
      `UpdateLowerBound(`*a,* $\mathcal{C}$*,* $\mathcal{P}$*,*$\mathcal{L}'$`)`*;*
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{C}\}$*;*
    else if `IsExplanation(`$\mathcal{C}, \neg q$`)` *and* $depth == maxDepth$ **then**
      `UpdateUpperBound(`*b,* $\mathcal{C}$*,* $\mathcal{R}$*,*$\mathcal{L}'$`)`*;*
      $\mathcal{R} \leftarrow \mathcal{R} \cup \{\mathcal{C}\}$*;*
    else if *depth* **<** *maxDepth* **then**
      $S.push($`Expand(`$\mathcal{C}, depth + 1$`)`$)$ *;* `// Generate new candidate`
    end
  end
  return $(\mathcal{P}, \mathcal{R}, a, b)$
end

Algorithm 1: Bounded-depth DFS (inner loop of IDDFS).

The function `Expand` generates next candidates through the expansion of $\mathcal{C}$ with a new literal. To avoid the generation of redundant states, we maintain probabilistic literals ordered according to a specific heuristic and track the index i of the last literal added to $\mathcal{C}$. A new literal can only be appended if its index in the order is strictly greater than i. In this formulation, this index corresponds directly to the search depth. Although more refined heuristics, such as heuristics based on the dependency graph of the program, may be used, for the purpose of this paper simply considering the literals by the decreasing order of probability was found to suffice, as demonstrated in section 4.

```
Function IsExplanation(C, q)
    program ← BuildASP(C, q);
    if SolveASP(program) is unsatisfiable then
        return True;
    end
    return False;
end
```

Algorithm 2: Check if $\mathcal{C}$ is an explanation for q.

Algorithm 2 determines if candidate set $\mathcal{C}$ constitutes an explanation using the off-the-shelf ASP logic solver *clingo*. It is worth noting that any symbolic inference engine can be used, providing additional flexibility to the system. The function `BuildASP` assembles an ASP program consistent with $\mathcal{C}$ and places a constraint on q. The ASP solver is used to assess the satisfiability of this program; the program being unsatisfiable implies that $\mathcal{C}$ entails q, confirming it as an explanation. The implementation of `isRefutation` is analogous.

As the search progresses, the functions `UpdateLowerBound` and `UpdateUpperBound` update respectively the lower and upper bounds a and b. Note that due to IDDFS this update step is only necessary in the nodes of depth $maxDepth$. The inclusion-exclusion principle is used to approximate the change in $Pr(\mathcal{P})$ (resp. $Pr(\mathcal{R})$) caused by the addition of $\mathcal{C}$. We thus have:

$$Pr(\mathcal{P}) \approx Pr(\mathcal{P} \setminus \{\mathcal{C}\}) + Pr(\mathcal{C}) - \sum_{\substack{\mathcal{D} \in \mathcal{P} \\ \mathcal{D} \neq \mathcal{C}}} Pr(\mathcal{C} \wedge \mathcal{D}) + \sum_{\substack{\mathcal{D}, \mathcal{G} \in \mathcal{P} \\ \mathcal{D} \neq \mathcal{C},\ \mathcal{D} \neq \mathcal{G} \neq \mathcal{C}}} Pr(\mathcal{C} \wedge \mathcal{D} \wedge \mathcal{G}) + \cdots . \quad (5)$$

Each time a new explanation (resp. refutation) is found, it is added to $\mathcal{P}$ (resp. $\mathcal{R}$), and a (resp. b) is updated according to the above equation, with at most k terms. By the end of the search, we have a set of explanations, a set of refutations and, respectively, an upper and a lower bound for q. This is specially useful in learning settings, for example, to recalculate queries when input probabilities change. The search can be run until completion, obtaining the exact value of $a = Pr(q) = b$, or, more conveniently, be interrupted when the gap $b - a$ is smaller than a given tolerance or some maximum amount of memory or time is exceeded. With minor changes, the pseudocode above may be altered to include a timeout parameter.

3.2. Explanation ASEO

Inspired by the work done in [Renkens et al. 2014], we propose to guide the search for explanations through the double-variable encoding present in that paper. Given a PLP with probabilistic facts $\{p_1 :: f_1, \ldots, p_n :: f_n\}$ and a Boolean query q, we transform it into an ASP encoding by applying the following process.

Each probabilistic fact $p_i :: f_i$ is replaced by two atoms f_{i_pos} and f_{i_neg}, representing its positive and negative occurrences respectively, as well as a choice rule ensuring at most one holds. The probabilities of the probabilistic facts are encoded as weak constraints weighted by $-\log(p_i)$ and $-\log(1 - p_i)$, so that minimizing the total weight of

active constraints (a task that can be done with a solver such as clingo) is equivalent to maximizing the probability of the corresponding total choice. Finally, a hard constraint $:- \texttt{not}\ q$ enforces that only total choices entailing q are considered. This encoding also ensures that the solutions found by the solver are equivalent to our definition of explanations, rather than full models.

To enumerate all solutions in decreasing order of probability, we use an approach similar to ASEO [Pajunen and Janhunen 2021] with the use of the off-the-shelf ASP solver clingo. At each step of the process, clingo returns the current optimal set, that is, the set of variables that maximizes the probability of the corresponding explanation. Note that if there is more than one optimal set, clingo will return all optimal sets. We then add a constraint (or constraints, in the case of multiple sets) to the transformed program explicitly excluding the found set(s) from the search. This forces the enumeration of the next-best solutions until no more solutions can be found or a stopping criterion is met, such as a timeout or a maximum number of solutions.

Similar to the previous approach, this method uses the inclusion-exclusion principle to circumvent the disjoint-sum problem and exhibits anytime behavior.

4. Experiments

The performance of *ExpRef* (Section 3.1) and *ExpASEO* (Section 3.2) on the task of query success probability approximation was evaluated on a common ProbLog application, *Smokers* [Domingos et al. 2008], and on the *WebKB*[1] dataset. As our baseline for accuracy, we used the exact inference algorithm implemented by ProbLog. We compare the performance of the proposed algorithms to that of other approximate inference algorithms: *k-best* as implemented in ProbLog, ASEO as implemented in Plingo [Hahn et al. 2022] and *top-k* as implemented in Scallop [Li et al. 2023]. Below, these will be referred simply as *k-best*, *plingo* and *top-k*.

The Smokers domain models how the habit of smoking is reproduced through the nodes of social networks. For the experiments the following set of rules was used:

```
smokes(X) :- stress(X).
smokes(X) :- friend(X,Y), smokes(Y), influences(Y,X).
```

The predicates `stress/1` and `influences/2` are probabilistic facts that model the stress upon a node and the influence of a node on another, respectively. While in other usages of this application, such as [De Raedt et al. 2007], the probabilities of both predicates are considered uniform over all nodes and node pairs, to better suit the characteristics of real social networks, we have chosen to randomly generate values for these probabilities.

On the other hand, the predicate `friend/1` is a non-probabilistic predicate that represents the edges along the social network. To generate the values for the predicate, a random graph generator with power-law degree distribution was used. These random graphs have been shown to be similar to empiric social networks [De Raedt et al. 2007].

We generated 10 different instances of the *Smokers* programs, ranging from 32 to 73 probabilistic facts. In each instance, half of the nodes were assigned a `stress/1`

[1] www.webkb.org

predicate and the other half were used as the base for queries on the `smokes/1` predicate, resulting in between 4 to 9 queries per instance. Each query was run with a timeout of 60 seconds.

The *WebKB* dataset consists in a collection of HTML university webpages separated into classes such as student or faculty pages. The classification depends on the content of the pages as well as on the classes of linked pages. The datasets contains 7 classes: *course*, *department*, *faculty*, *other*, *project*, *staff* and *student*. Borrowing from [Gutmann et al. 2011], we define this relationship with the following rules:

```
class(P, C) :- hasWord(P, W), pfWoCla(P, C, W).
class(P, C) :- linksTo(P2, P), pfLiCla(P2, P, C2, C),
                    class(P2, C2).
```

The predicates `hasWord/2` and `linksTo/2` are deterministic predicates which model content (i.e., whether a page P has a certain word W) and link structure (i.e., whether a page P links to page $P2$), respectively. These are taken directly from the structure of the HTML files. On the other hand, `pfWoCla/3` and `pfLiCla/4` are probabilistic non-ground predicates with probabilities chosen randomly between $[0.1, 0.4]$. These model how likely page P is to be class C given that it has word W and how likely page P is to be class C given that it links to page $P2$ of class $C2$, respectively. Because the predicates are non-ground, these probabilities are the same for all pages, and the number of probabilistic facts increases with an increase in the number of considered words.

Utilizing a subset of 150 webpages, we generated 10 different instances using the 10 to 100 more common words, corresponding to between 119 and 749 probabilistic facts. In each instance, half of the webpages had their classes included as facts using `class/2` predicates, and the other half were used as a base for queries. Of the possible queries, 10 were chosen randomly for each instance. Each query was run three times, with timeouts of 5, 30 and 60 seconds.

Our experiments investigated the accuracy and reliability of the proposed algorithms, as well as the impact restrictions imposed on the maximum number of explanations (parameter k) or refutations, on the maximum depth of search (parameter $maxDepth$) and on the time allotted had on those factors. To allow for comparison, the number of explanations used by *k-best*, *plingo* and *top-k* was also restricted, using built-in features when possible, and the same timeout was applied across all algorithms. For the parameter k, the values used were $(3, 5, 7, 11)$, and for $maxDepth$, the values $(5, 7, 9)$ were used.

However, varying the maximum depth parameter for the *ExpRef* algorithm did not exert a significant influence on the overall outcomes. Furthermore, increasing the maximum number of explanations beyond 5 or extending the allocated execution time did not alter the general behavior of the results. For this reason, we report the results exclusively for $k = 3$ and $k = 11$ under a fixed timeout of 60 seconds.

The experimental results are depicted in Figures 1 and 2. The experiments were conducted on an AMD Ryzen Threadripper 5965WX processor with 128 GB of RAM. Each data point corresponds to an average across the queries of each instance, representing the relative error with respect to the exact ProbLog inference. In the *WebKB* domain, out of 600 total executions, exact ProbLog inference failed 210; consequently, the analysis

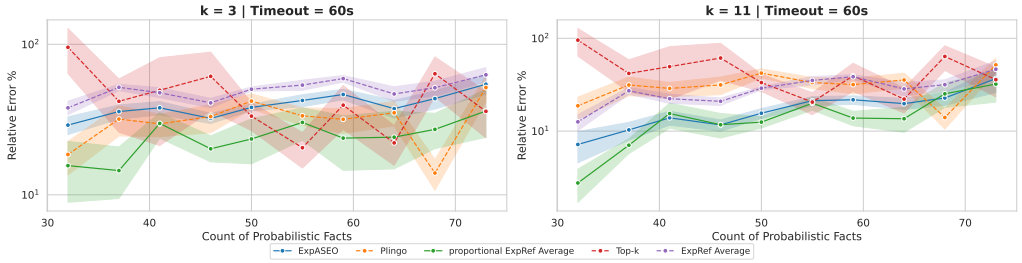


Figure 1. Results for the *Smokers* domain.

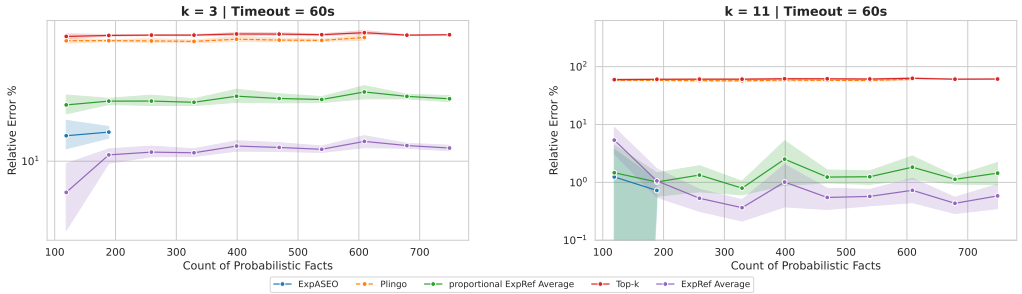


Figure 2. Results for the *WebKB* domain.

spans the remaining 390 successful experiments.

For the *Smokers* domain, the proportional average of the upper and lower bounds computed by *ExpRef* consistently provided the best approximation for the query success probability across all tested parameter combinations, followed by *ExpASEO* as the second-best approach in most settings. The proportional average is defined as the average of the bounds, weighted by the number of explanations and refutations found on their respective searches.

In the *WebKB* domain, the arithmetic average was the second-best approximation on average, while the proportional average ranked third, showing progressive improvement as k increased. Although the *k-best* approach yielded the highest average accuracy in the subset of experiments where it successfully terminated, it proved to be the second-least robust method due to long formula compilation times. Specifically, *k-best* suffered from a 53.17% timeout rate, failing in 319 out of 600 experiments. In contrast, *ExpRef* demonstrated high reliability, successfully producing a result in every experiment. Similarly, *top-k* completed every experiment, whereas *Plingo* exhibited a 20% timeout rate.

In contrast, *ExpASEO* struggled on the *WebKB* domain, experiencing an 86.67% timeout rate. However, on the instances where it did complete, it achieved exceptional accuracy, outperforming all other tested algorithms when $k = 11$. These findings point to *ExpASEO*'s utility for medium-sized programs (with less than 190 probabilistic facts) in settings where maximizing accuracy is a priority.

5. Conclusion

In this work, we proposed two novel approximate inference algorithms, *ExpRef* and *ExpASEO*, for query success probability approximation in stratified probabilistic logic programs. Our experimental evaluation across the *Smokers* and *WebKB* domains demon-

strates that searching for semantic explanations and refutations via an IDDFS approach constitutes a highly viable and robust alternative to existing methods. In particular, *ExpRef* presented the highest average accuracy in the *Smokers* domain and, together with *top-k*, the highest reliability in the *WebKB* domain. *ExpASEO*, on the other hand, achieved the highest accuracy in the instances where it terminated in the *WebKB* domain when $k = 11$.

While methods with heavier compilation steps such as *k-best* struggle with scalability and reliability in dense domains like *WebKB*, our proposed *ExpRef* algorithm reliably delivers success probability approximations with good accuracy. Nevertheless, further research is required to evaluate the proposed approaches across a broader and more diverse set of domains, which could reveal different advantages. Additionally, the *ExpASEO* algorithm showed significant potential in our experiments; although it was mostly outperformed by *ExpRef* here, the results for $k = 11$ suggest it may achieve superior performance, especially when dealing with medium-sized programs. Future work will focus on identifying these optimal settings and developing better heuristics to improve inference performance.

Acknowledgments

This work was partly funded by São Paulo Research Agency (FAPESP) Grants no. 2025/01335-3 and 2022/02937-9, CNPq Grant 301420/2026-2 and CAPES Finance Code 001.

References

Apt, K. R. and Doets, K. (1994). A new definition of SLDNF-resolution. *The Journal of Logic Programming*, 18(2):177–190.

Cozman, F. G. and Mauá, D. D. (2017). On the semantics and complexity of probabilistic logic programs. *Journal of Artificial Intelligence Research*, 60:221–262.

De Raedt, L. and Kimmig, A. (2015). Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47.

De Raedt, L., Kimmig, A., and Toivonen, H. (2007). ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2462–2467.

Dierckx, L., Dubray, A., and Nijssen, S. (2025). Learning from logical constraints with lower- and upper-bound arithmetic circuits. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5012–5020.

Domingos, P., Kok, S., Lowd, D., Poon, H., Richardson, M., and Singla, P. (2008). Markov logic. In *Probabilistic Inductive Logic Programming: Theory and Applications*, pages 92–117. Springer.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. (2015). Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2019). Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82.

Geh, R. L., Gonçalves, J., Silveira, I. C., Mauá, D. D., and Cozman, F. G. (2024). dPASP: A probabilistic logic programming environment for neurosymbolic learning and reasoning. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, pages 731–742.

Gutmann, B., Thon, I., and De Raedt, L. (2011). Learning the parameters of probabilistic logic programs from interpretations. In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages 581–596. Springer.

Hahn, S., Janhunen, T., Kaminski, R., Romero, J., Rühling, N., and Schaub, T. (2022). Plingo: A system for probabilistic reasoning in clingo based on LP MLN. In *International Joint Conference on Rules and Reasoning*, pages 54–62. Springer.

Huang, J., Li, Z., Chen, B., Samel, K., Naik, M., Song, L., and Si, X. (2021). Scallop: From probabilistic deductive databases to scalable differentiable reasoning. *Advances in Neural Information Processing Systems*, 34:25134–25145.

Li, Z., Huang, J., Liu, J., Zhu, F., Zhao, E., Dodds, W., Velingker, N., Alur, R., and Naik, M. (2024). Relational programming with foundational models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 10635–10644.

Li, Z., Huang, J., and Naik, M. (2023). Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1463–1487.

Lloyd, J. W. (2012). *Foundations of Logic Programming*. Springer Science & Business Media.

Manhaeve, R., Marra, G., and De Raedt, L. (2021). Approximate inference for neural probabilistic logic programming. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pages 475–486.

Mantadelis, T. and Rocha, R. (2017). Using iterative deepening for probabilistic logic inference. In *Practical Aspects of Declarative Languages*, pages 198–213. Springer.

Pajunen, J. and Janhunen, T. (2021). Solution enumeration by optimality in answer set programming. *Theory and Practice of Logic Programming*, 21(6):750–767.

Renkens, J., Kimmig, A., Van den Broeck, G., and De Raedt, L. (2014). Explanation-based approximate weighted model counting for probabilistic logics. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, pages 2490–2496.

Renkens, J., Van den Broeck, G., and Nijssen, S. (2012). k-optimal: A novel approximate inference algorithm for ProbLog. *Machine Learning*, 89(3):215–231.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP)*, pages 715–729.

van Krieken, E., Thanapalasingam, T., Tomczak, J., Van Harmelen, F., and Ten Teije, A. (2023). A-NeSI: A scalable approximate method for probabilistic neurosymbolic inference. *Advances in Neural Information Processing Systems*, 36:24586–24609.