Approximate Inference in Probabilistic Answer Set Programming for Statistical Probabilities

Damiano Azzolini[1][0000−0002−7133−2673], Elena Bellodi[2][0000−0002−3717−3779], and Fabrizio Riguzzi[3][0000−0003−1654−9703]

[1] Dipartimento di Scienze dell'Ambiente e della Prevenzione – Università di Ferrara
[2] Dipartimento di Ingegneria – Università di Ferrara
[3] Dipartimento di Matematica e Informatica – Università di Ferrara
{damiano.azzolini,elena.bellodi,fabrizio.riguzzi}@unife.it

Abstract. "Type 1" statements were introduced by Halpern in 1990 with the goal to represent statistical information about a domain of interest. These are of the form "x% of the elements share the same property". The recently proposed language PASTA (Probabilistic Answer set programming for STAtistical probabilities) extends Probabilistic Logic Programs under the Distribution Semantics and allows the definition of this type of statements. To perform exact inference, PASTA programs are converted into probabilistic answer set programs under the Credal Semantics. However, this algorithm is infeasible for scenarios when more than a few random variables are involved. Here, we propose several algorithms to perform both conditional and unconditional approximate inference in PASTA programs and test them on different benchmarks. The results show that approximate algorithms scale to hundreds of variables and thus can manage real world domains.

Keywords: Probabilistic Answer Set Programming, Credal Semantics, Statistical Statements, Approximate Inference.

1 Introduction

In [14] Halpern discusses the difference between "Type 1" (T1) and "Type 2" (T2) statements: the former describes a statistical property of the world of interest while the latter represents a degree of belief. "The probability that a random person smokes is 20%" is an example of "Type 1" statement while "John smokes with probability 30%", where John is a particular individual, is an example of "Type 2" statement.

Answer Set Programming (ASP) [7] is a powerful language that allows to easily encode complex domains. However, ASP does not allow uncertainty on the data. To handle this, we need to consider Probabilistic ASP (PASP) where the uncertainty is expressed through probabilistic facts, as done in Probabilistic Logic Programming [10]. We focus here on PASP under the Credal Semantics [9], where each query is associated with a probability interval defined by a lower and an upper bound.

Recently, the authors of [3] introduced PASTA ("Probabilistic Answer set programming for STAtistical probabilities"), a new language (and software) where statistical statements are translated into PASP rules and inference is performed by converting the PASP program into an equivalent answer set program. However, performing exact inference is exponential in the number of probabilistic facts, and thus it is infeasible in the case of more than a few dozens of variables. In this paper, we propose four algorithms to perform approximate inference in PASTA programs: one for unconditional sampling and three for conditional sampling that adopt rejection sampling, Metropolis Hastings sampling, and Gibbs sampling. Empirical results show that our algorithms can handle programs with hundreds of variables. Moreover, we compare our algorithms with PASOCS [23], a solver able to perform approximate inference in PASP program under the Credal Semantics, showing that our algorithms reach a comparable accuracy in a lower execution time.

The paper is structured as follows: Section 2 discusses some related works and Section 3 introduces background concepts. Section 4 describes our algorithms for approximate inference in PASTA programs that are tested in Section 5. Section 6 concludes the paper.

2 Related Work

PASTA [3] extends Probabilistic Logic Programming [20] under the Distribution Semantics [21] by allowing the definition of Statistical statements. Statistical statements, also referred to as "Probabilistic Conditionals", are discussed in [16], where the authors give a semantics to T1 statements leveraging the maximum entropy principle. Under this interpretation, they consider the unique model that yields the maximum entropy. Differently from them, we consider all the models, thus obtaining a more general framework [3].

T1 statements are also studied in [15] and [24]: the former adopts the cross entropy principle to assign a semantics to T1 statements while the latter identifies only a specific model and a sharp probability value, rather than all the models and an interval for the probability, as we do.

We adopt the credal semantics [9] for PASP, where the probability of a query is defined by a range. To the best of our knowledge, the only work which performs inference in PASP under the Credal Semantics is PASOCS [23]. They propose both an exact solver, which relies on the generation of all the possible combinations of facts, and an approximate one, based on sampling. We compare our approach with it in Section 5.

Other solutions for inference in PASP consider different semantics that assign to a query a sharp probability value, such as [6,17,19,22].

3 Background

We assume that the reader is familiar with the basic concepts of Logic Programming. For a complete treatment of the field, see [18].

An Answer Set Programming (ASP) [7] rule has the form `h1 ; ... ; hm :- b1, ... , bn.` where each `hi` is an atom, each `bi` is a literal and `:-` is called the neck operator. The disjunction of the `hi`s is called the *head* while the conjunction of the `bi`s is called the *body* of the rule. Particular configurations of the atoms/literals in the head/body identify specific types of rules: if the head is empty and the body is not, the rule is a *constraint*. Likewise, if the body is empty and the head is not, the rule is a *fact*, and the neck operator is usually omitted. We consider only rules where every variable also appears in a positive literal in the body. These rules are called *safe*. Finally, a rule is called *ground* if it does not contain variables.

In addition to atoms and literals, we also consider *aggregate atoms* of the form $\gamma_1 \omega_1 \ \#\zeta\{\epsilon_1, \ldots, \epsilon_l\} \ \omega_2 \gamma_2$ where γ_1 and γ_2 are constants or variables called *guards*, ω_1 and ω_2 are arithmetic comparison operators (such as $>$, $\geq$, $<$, and $\leq$), ζ is an aggregate function symbol, and each ϵ_i is an expression of the form $t_1, \ldots, t_i \ : \ F$ where each t_j is a term, F is a conjunction of literals, and $i > 0$. Moreover, each variable in $t_1, \ldots, t_i$ also appears in F.

We denote an answer set program with $\mathcal{P}$ and its Herbrand base, i.e., the set of atoms that can be constructed with all the symbols in it, as $B_{\mathcal{P}}$. An *interpretation* $I \subset B_{\mathcal{P}}$ satisfies a ground rule when at least one of the `hi`s is true in I when the body is true in I. A *model* is an interpretation that satisfies all the ground rules of a program $\mathcal{P}$. The *reduct* [11] of a ground program $\mathcal{P}_g$ with respect to an interpretation I is a new program $\mathcal{P}_g^r$ obtained from $\mathcal{P}_g$ by removing the rules in which a `bi` is false in I. Finally, an interpretation I is an *answer set* for $\mathcal{P}$ if it is a minimal model of $\mathcal{P}_g^r$. We consider minimality in terms of set inclusion and denote with $AS(\mathcal{P})$ the set of all the answer sets of $\mathcal{P}$.

Probabilistic Answer Set Programming (PASP) [8] is to Answer Set Programming what Probabilistic Logic Programming [20] is to Logic Programming: it allows the definition of uncertain data through probabilistic facts. Following the ProbLog [10] syntax, these facts can be represented with $\Pi :: f$ where f is a ground atom and Π is its probability. If we assign a truth value to every probabilistic fact (where $\top$ represents true and $\bot$ represents false) we obtain a *world*, i.e., an answer set program. There are 2^n worlds for a probabilistic answer set program, where n is the number of ground probabilistic facts. Many Probabilistic Logic Programming languages rely on the distribution semantics [21], according to which the probability of a world w is computed with the formula

$$P(w) = \prod_{i|f_i=\top} \Pi_i \cdot \prod_{i|f_i=\bot} (1 - \Pi_i)$$

while the probability of a *query* q (conjunction of ground literals), is computed with the formula

$$P(q) = \sum_{w \models q} P(w)$$

when the world has a single answer set.

For performing inference in PASP we consider the *Credal Semantics* [8], where every query q is associated with a probability range: the upper probability

bound $\overline{\mathrm{P}}(q)$ is given by the sum of the probabilities of the worlds w where there is *at least one* answer set of w where the query is present. Conversely, the lower probability bound $\underline{\mathrm{P}}(q)$ is given by the sum of the probabilities of the worlds w where the query is present in *all* the answer sets of w, i.e.,

$$\overline{\mathrm{P}}(q) = \sum_{w_i \mid \exists m \in AS(w_i),\ m \models q} P(w_i), \quad \underline{\mathrm{P}}(q) = \sum_{w_i \mid |AS(w_i)|>0\ \wedge\ \forall m \in AS(w_i),\ m \models q} P(w_i)$$

Note that the credal semantics requires that every world has at least one answer set. In the remaining part of the paper we consider only programs where this requirement is satisfied.

Example 1 (PASP Example). We consider 3 objects whose components are unknown and suppose that some of them may be made of iron with a given probability. An object made of iron may get rusty or not. We want to know the probability that a particular object is rusty. This can be modelled with:

```
1 0.2::iron(1). 0.9::iron(2). 0.6::iron(3).
2
3 rusty(X) ; not_rusty(X):- iron(X).
4 :- #count{X:rusty(X), iron(X)} = RI,
5     #count{X:iron(X)} = I, 10*RI < 6*I.
```

The constraint states that at least 60% of the object made of iron are rusty. This program has $2^3 = 8$ worlds. For example, the world where all the three probabilistic facts are true has 4 answer sets. If we consider the query q `rusty(1)`, this world only contributes to the upper probability since the query is present only in 3 of the 4 answer sets. By considering all the worlds, we get $\underline{\mathrm{P}}(q) = 0.092$ and $\overline{\mathrm{P}}(q) = 0.2$, so the probability of the query lies in the range $[0.092, 0.2]$.

If we want to compute the conditional probability for a query q given evidence e, $P(q \mid e)$, we need to consider two different formulas for the lower and upper probability bounds [8]:

$$\overline{\mathrm{P}}(q \mid e) = \frac{\overline{\mathrm{P}}(q, e)}{\overline{\mathrm{P}}(q, e) + \underline{\mathrm{P}}(\neg q, e)}, \quad \underline{\mathrm{P}}(q \mid e) = \frac{\underline{\mathrm{P}}(q, e)}{\underline{\mathrm{P}}(q, e) + \overline{\mathrm{P}}(\neg q, e)} \tag{1}$$

Clearly, these are valid if the denominator is different from 0, otherwise the value is undefined. If we consider again Example 1 with query q `rusty(1)` and evidence e `iron(2)`, we get $\underline{\mathrm{P}}(q \mid e) = 0.08$ and $\overline{\mathrm{P}}(q \mid e) = 0.2$.

Following the syntax proposed in [3], a *probabilistic conditional* is a formula of the form $(C \mid A)[\Pi_l, \Pi_u]$ stating that the fraction of As that are also Cs is between Π_l and Π_u. Both C and A are two conjunctions of literals. To perform inference, a conditional is converted into three answer set rules: i) `C ; not_C :- A`, ii) `:- #count{X : C, A} = V0, #count{X : A} = V1, 10*V0 < 10*`Π_l`*V1`, and iii) `:- #count{X : C, A} = V0, #count{X : A} = V1, 10*V0 > 10*`Π_u`*V1`, where `X` is a vector of elements containing all the variables in `C` and `A`. If Π_l or Π_u are respectively 0 or 1, the rules ii) or iii) can be omitted.

Moreover, if the probability values Π_l and Π_u have n decimal digits, the 10 in the multiplications above should be replaced with 10^n, because ASP cannot deal with floating point values.

A PASTA program [3] is composed of a set of probabilistic facts, a set of ASP rules, and a set of probabilistic conditionals.

Example 2 (Probabilistic Conditional (PASTA program)). The following program

```
1 0.2::iron(1). 0.9::iron(2). 0.6::iron(3).
2 (rusty(X) | iron(X))[0.6,1].
```

is translated into the PASP program shown in Example 1. The rule iii) is omitted since $\Pi_u = 1$.

In [3] an exact inference algorithm was proposed to perform inference with probabilistic conditionals, that basically requires the enumeration of all the worlds. This is clearly infeasible when the number of variables is greater than 20-30. To overcome this issue, in the following section we present different algorithms that compute the probability interval in an approximate way based on sampling techniques.

4 Approximate Inference for PASTA Programs

To perform approximate inference in PASTA programs, we developed four algorithms: one for unconditional sampling (Algorithm 1) and three for conditional sampling that adopt rejection sampling (Algorithm 2), Metropolis Hastings sampling (Algorithm 3), and Gibbs sampling (Algorithm 4) [4,5]. Algorithm 1 describes the basic procedure to sample a query (without evidence) in a PASTA program. First, we keep a list of sampled worlds. Then, for a given n number of times (number of samples), we sample a world id with function SampleWorld by choosing a truth value for every probabilistic fact according to its probability. For every probabilistic facts, the process is the following: we sample a random value between 0 and 1, call it r. If $r < \Pi_i$ for a given probabilistic fact f_i with associated probability Π_i, f_i is set to true, otherwise false. id is a binary string representing a world where, if the nth digit is 0, the nth probabilistic fact (in order of appearance in the program) is false, true otherwise. To clarify this, if we consider the program shown in Example 2, a possible world id could be 010, indicating that `iron(1)` is not selected, `iron(2)` is selected, and `iron(3)` is not selected. The probability of this world is $(1 - 0.2) \cdot 0.9 \cdot (1 - 0.6) = 0.288$. If we have already considered the currently sampled world, we look in the list of sampled worlds whether it contributes to the lower or upper counters (function GetContribution) and update the lower (lp) and upper (up) counters accordingly. In particular, GetContribution returns two values, one for the lower and one for the upper probability, each of which can be either 0 (the world id does not contribute to the probability) or 1 (the world id contributes to the probability). If, instead, the world had never been encountered before, we assign

a probability value to the probabilistic facts in the program according to the truth value (probability Π for $\top$, $1-\Pi$ for $\bot$) that had been sampled (function SetFacts), we compute its contribution to the lower and upper probabilities (function CheckLowerUpper, with the same output as GetContribution), and store the results in the list of already encountered worlds (function InsertContribution). In this way, if we sample again the same world, there is no need to compute again its contribution to the two probability bounds. Once we have a number of samples equal to *Samples*, we simply return the number of samples computed for the lower and upper probability divided by *Samples*.

Algorithm 1 Function Sample: computation of the unconditional probability from a PASTA program.

```
1: function Sample(Query, Samples, Program)
2:     sampled ← {}                                   ▷ list of sampled worlds
3:     lp ← 0, up ← 0, n ← 0
4:     while n ≤ Samples do                 ▷ Samples is the number of samples
5:         id ←SampleWorld(Program)
6:         n ← n + 1
7:         if id ∈ sampled then                   ▷ a world was already sampled
8:             up_0, lp_0 ← GetContribution(sampled, id)
9:             up ← up + up_0
10:            lp ← lp + lp_0
11:        else
12:            Program_d ←SetFacts(Program, id)
13:            lp_0, up_0 ← CheckLowerUpper(Program_d)
14:            lp ← lp + lp_0
15:            up ← up + up_0
16:            InsertContribution(sampled, id, lp_0, up_0)
17:        end if
18:    end while
19:    return lp/Samples, up/Samples
20: end function
```

When we need to account also for the evidence, other algorithms should be applied, such as rejection sampling. It is described in Algorithm 2: as in Algorithm 1, we maintain a list with the already sampled worlds. Moreover, we need 4 variables to store the joint lower and upper counters of q and e (*lpqe* and *upqe*) and $\neg q$ and e (*lpnqe* and *upnqe*), see Equation 1. Then, with the same procedure as before, we sample a world. If we have already considered it, we retrieve its contribution from the *sampled* list. If not, we set the probabilistic facts according to the sampled choices, compute the contribution to the four values, update them accordingly, and store the results. $lpqe_0$ is 1 if both the evidence and the query are present in all the answer sets of the current world, 0 otherwise. $upqe_0$ is 1 if both the evidence and the query are present in at least one answer set of the current world, 0 otherwise. $lpnqe_0$ is 1 if the evidence is present and the query is absent in all the answer sets of the current world, 0 otherwise. $upnqe_0$ is 1 if the evidence is present and the query is absent in at least one answer set of the current world, 0 otherwise. As before, we return the ratio between the number of samples combined as in Equation 1.

Algorithm 2 Function RejectionSample: computation of the conditional probability from a PASTA program using Rejection sampling.

```
1: function RejectionSample(Query, Evidence, Samples, Program)
2:     lpqe ← 0, upqe ← 0, lpnqe ← 0, upnqe ← 0, n ← 0, sampled ← {}
3:     while n ≤ Samples do
4:         id ← SampleWorld(Program)
5:         n ← n + 1
6:         if id ∈ sampled then
7:             lpqe_0, upqe_0, lpnqe_0, upnqe_0 ← GetContribution(sampled, id)
8:             lpqe ← lpqe + lpqe_0, upqe ← upqe + upqe_0
9:             lpnqe ← lpnqe + lpnqe_0, upnqe ← upnqe + upnqe_0
10:        else
11:            Program_d ← SetFacts(Program, id)
12:            lpqe_0, upqe_0, lpnqe_0, upnqe_0 ← CheckLowerUpper(Program_d)
13:            lpqe ← lpqe + lpqe_0, upqe ← upqe + upqe_0
14:            lpnqe ← lpnqe + lpnqe_0, upnqe ← upnqe + upnqe_0
15:            InsertContribution(sampled, id, lpqe_0, upqe_0, lpnqe_0, upnqe_0)
16:        end if
17:    end while
18:    return lpqe / (lpqe + upnqe), upqe / (upqe + lpnqe)
19: end function
```

In addition to rejection sampling, we developed two other algorithms that mimic Metropolis Hastings sampling (Algorithm 3) and Gibbs sampling (Algorithm 4). Algorithm 3 proceeds as follows. The overall structure is similar to Algorithm 2. However, after sampling a world, we count the number of probabilistic facts set to true (function CountTrueFacts). Then, with function CheckContribution we check whether the current world has already been considered. If so, we accept it with probability $min(1, N_0/N_1)$ (line 18), where N_0 is the number of true probabilistic facts in the previous iteration and N_1 is the number of true probabilistic facts in the current iteration. If the world was never considered before, we set the truth values of the probabilistic facts in the program (function SetFacts), compute its contribution with function CheckLowerUpper, save the values (function InsertContribution), and check whether the sample is accepted or not (line 27) with the same criteria just discussed. As for rejection sampling, we return the ratio between the number of samples combined as in Equation 1.

Finally, for Gibbs sampling (Algorithm 4), we first sample a world until e is true (function TrueEvidence), saving, as before, the already encountered worlds. Once we get a world that satisfies this requirement, we switch the truth values of *Block* random probabilistic facts (function SwitchBlockValues, line 19) and we check the contribution of this new world as in Algorithm 2. Also there, the return value is the one described by Equation 1.

5 Experiments

We implemented the previously described algorithms in Python 3 and we integrated them into the PASTA[4] solver [3]. We use clingo [12] to compute the

[4] Source code and datasets available at https://github.com/damianoazzolini/pasta.

Algorithm 3 Function MHSample: computation of the conditional probability from a PASTA program using Metropolis Hastings sampling.

```
1: function MHSample(Query, Evidence, Samples, Program)
2:     sampled ← {}
3:     lpqe ← 0, upqe ← 0, lpnqe ← 0, upnqe ← 0, n ← 0, trueFacts_0 ← 0
4:     while n ≤ Samples do
5:         id ←SampleWorld(Program)
6:         n ← n + 1
7:         trueFacts_1 ← CountTrueFacts(id)
8:         lpqe_0, upqe_0, lpnqe_0, upnqe_0 ←
9:             CheckContribution(Program_d, trueFacts_0, trueFacts_1, id, sampled)
10:         lpqe ← lpqe +lpqe_0,  upqe ← upqe + upqe_0
11:         lpnqe ← lpnqe + lpnqe_0,  upnqe ← upnqe + upnqe_0
12:         trueFacts_0 ← trueFacts_1
13:     end while
14:     return lpqe/(lpqe + upnqe), upqe/(upqe + lpnqe)
15: end function
16: function CheckContribution(Program_d, N_0, N_1, id, sampled)
17:     if id ∈ sampled then
18:         if random < min(1, N_0/N_1) then        ▷ random is a random value ∈ [0, 1]
19:             return GetContribution(id, sampled)
20:         else
21:             return 0, 0, 0, 0
22:         end if
23:     else
24:         Program_d ← SetFacts(Program, id)
25:         lpqe_0, upqe_0, lpnqe_0, upnqe_0 ← CheckLowerUpper(Program_d)
26:         InsertContribution(sampled, id, lpqe_0, upqe_0, lpnqe_0, upnqe_0)
27:         if random < min(1, N_0/N_1) then
28:             return lpqe_0, upqe_0, lpnqe_0, upnqe_0
29:         else
30:             return 0, 0, 0, 0
31:         end if
32:     end if
33: end function
```

answer sets. To assess the performance, we ran multiple experiments on a computer with Intel® Xeon® E5-2630v3 running at 2.40 GHz with 16 Gb of RAM. Execution times are computed with the bash command `time`. The reported values are from the `real` field.

We consider two datasets with different configurations. The first one, `iron`, contains programs with the structure shown in Example 2. In this case, the size of an instance indicates the number of probabilistic facts. The second dataset, `smoke`, describes a network where some people are connected by a probabilistic friendship relation. In this case the size of an instance is the number of involved people. Some of the people in the network smoke. A conditional states that at least 40% of the people that have a friend that smokes are smokers. An example of instance of size 5 is

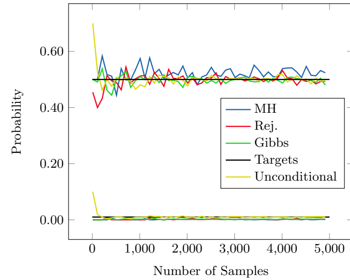
```
1 0.5::friend(a,b). 0.5::friend(b,c).
2 0.5::friend(a,d). 0.5::friend(d,e).
3 0.5::friend(e,c).
4 smokes(b). smokes(d).
5 (smokes(Y) | smokes(X), friend(X,Y))[0.4,1].
```

Algorithm 4 Function GibbsSample: computation of the conditional probability from a PASTA program using Gibbs sampling.

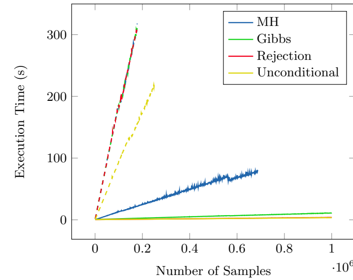
```
1: function GibbsSample(Query, Evidence, Samples, Block, Program)
2:     sampledEvidence ← {}, sampledQuery ← {}
3:     lpqe ← 0, upqe ← 0, lpnqe ← 0, upnqe ← 0, n ← 0
4:     while n ≤ Samples do
5:         ev ← false, n ← n + 1
6:         while ev is false do
7:             id ←SampleWorld(Program)
8:             if id ∈ sampledEvidence then
9:                 ev ← sampledEvidence[id]
10:            else
11:                Program_d ← SetFacts(Program, id)
12:                if TrueEvidence(Program_d) then
13:                    ev ← true, sampledEvidence[id] ← true
14:                else
15:                    sampledEvidence[id] ← false
16:                end if
17:            end if
18:        end while
19:        id_s ←SwitchBlockValues(id, Block, Program, Evidence)
20:        if id_s ∈ sampled then
21:            lpqe_0, upqe_0, lpnqe_0, upnqe_0 ← GetContribution(sampled, id)
22:            lpqe ← lpqe +lpqe_0,  upqe ← upqe + upqe_0
23:            lpnqe ← lpnqe + lpnqe_0,  upnqe ← upnqe + upnqe_0
24:        else
25:            Program_d ← SetFacts(Program, id)
26:            lpqe_0, upqe_0, lpnqe_0, upnqe_0 ← CheckLowerUpper(Program_d)
27:            lpqe ← lpqe +lpqe_0,  upqe ← upqe + upqe_0
28:            lpnqe ← lpnqe + lpnqe_0,  upnqe ← upnqe + upnqe_0
29:            InsertContribution(sampled, id, lpqe_0, upqe_0, lpnqe_0, upnqe_0)
30:        end if
31:    end while
32:    return lpqe/(lpqe + upnqe), upqe/(upqe + lpnqe)
33: end function
```

The number of probabilistic facts follows a Barabási-Albert preferential attachment model generated with the `networkx` [13] Python package. The initial number of nodes of the graph, n, is the size of the instance while the number of edges to connect a new node to an existing one, m, is 3.

In a first set of experiments, we fixed the number of probabilistic facts, for `iron`, and the number of people, for `smoke`, to 10 and plotted the computed lower and upper probabilities and the execution time by increasing the number of samples. All the probabilistic facts have probability 0.5. The goal of these experiments is to check how many samples are needed to converge and how the execution time varies by increasing the number of samples, with a fixed program. For the `iron` dataset, the query q is `rusty(1)` and the evidence e is `iron(2)`. Here, the exact values are $\underline{\mathrm{P}}(q) = 0.009765625$, $\overline{\mathrm{P}}(q) = 0.5$, $\underline{\mathrm{P}}(q \mid e) = 0.001953125$, and $\overline{\mathrm{P}}(q \mid e) = 0.5$. For the `smoke` dataset, the program has 21 connections (probabilistic facts): node 0 is connected to all the other nodes, node 2 with 4, 6, and 8, node 3 with 4, 5, and 7, node 4 with 5, 6, 7, and 9, and node 7 with 8 and 9. All the connections have probability 0.5. Nodes 2, 5, 6, 7, and 9 certainly smoke. The query q is `smokes(8)` and the evidence is `smokes(4)`. The targets are $\underline{\mathrm{P}}(q) = 0.158$, $\overline{\mathrm{P}}(q) = 0.75$, $\underline{\mathrm{P}}(q \mid e) = 0$, and $\overline{\mathrm{P}}(q \mid e) = 0.923$. Results for all the four algorithms are shown in Figures 1 (`iron`) and 2 (`smoke`).

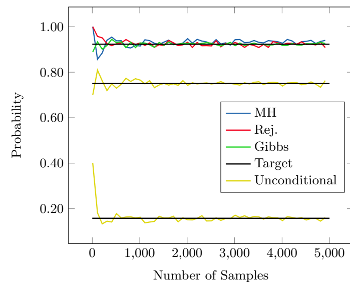


(a) Lower and upper probabilities.

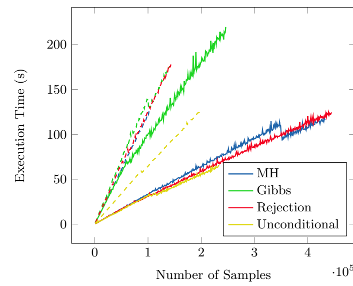


(b) Execution time.

Fig. 1: Comparison of the sampling algorithms on the `iron` dataset. Straight lines are the results for PASTA while dashed lines for PASOCS.



(a) Lower and upper probabilities.

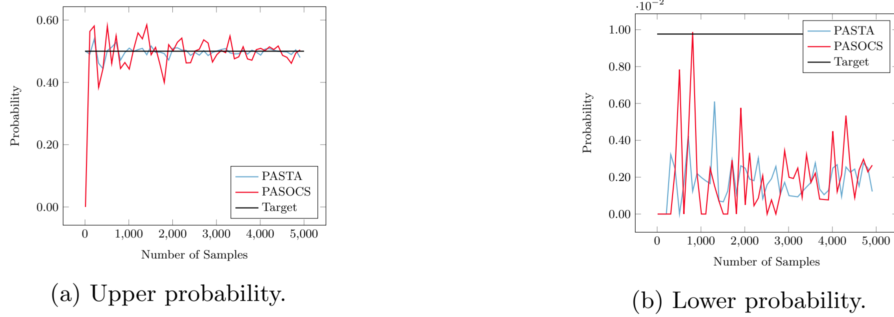


(b) Execution times.

Fig. 2: Comparison of the sampling algorithms on the `smoke` dataset. Straight lines are the results for PASTA while dashed lines for PASOCS. In Figure 2a the target line at 0.75 is for the upper unconditional probability.

For Gibbs sampling, we set the number *Block* (i.e, number of probabilistic facts to resample), to 1. All the algorithms seem to stabilize after a few thousands of samples for both datasets. For `iron`, MH seems to slightly overestimate the upper probability. Gibbs and rejection sampling require a few seconds to take 10^6 samples, while Metropolis Hastings (MH) requires almost 100 seconds. However, for the `smoke` dataset, MH and Rejection sampling have comparable execution times (more than 100 seconds for $5 \cdot 10^5$ samples) while Gibbs is the slowest among the three. This may be due to a low probability of the evidence.

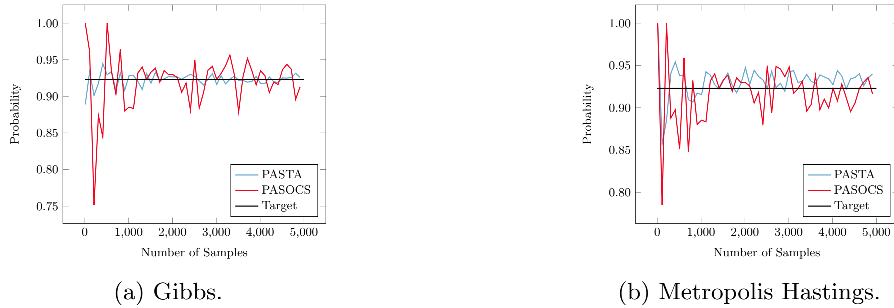
We compared our results with PASOCS [23] (after translating by hand the probabilistic conditionals in PASP rules). We used the following settings: `-n_min n -n_max -1 -ut -1 -p 300 -sb 1 -b 0` where `n` is the number of considered samples, `n_min` is the minimum number of samples, `n_max` is the maximum number of samples (`-1` deactivates it), `ut` is the uncertainty threshold (`-1` deactivates it), `p` is the percentile (since they estimate values with gaussians), `sb` is the number of samples to run at once during sampling, and `b` is the burnin value for Gibbs and Metropolis Hastings sampling (0 deactivates it). We do not select



(a) Upper probability.

(b) Lower probability.

Fig. 3: Comparison of Gibbs sampling on the `iron` dataset.

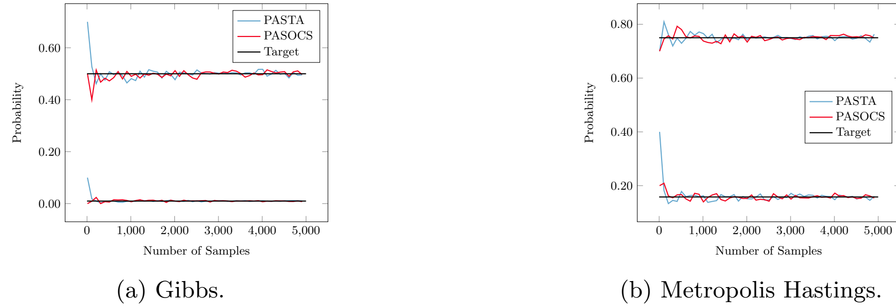


(a) Gibbs.

(b) Metropolis Hastings.

Fig. 4: Comparison of Gibbs sampling and MH on the `smoke` dataset.

parallel solving, since PASTA is not parallelized yet (this may be the subject of a future work). PASOCS adopts a different approach for conditional inference: at each iteration, instead of sampling a world, it updates the probabilities of the probabilistic facts and samples a world using these values. In Figure 1b, the execution times of PASOCS for all the tested algorithms are comparable and seem to grow exponentially with the number of samples. The lines for rejection and unconditional sampling for PASTA overlap. This also happens for the lines for MH, Gibbs, and rejection sampling for PASOCS. PASOCS seems to be slower also on the `smoke` dataset (Figure 2b), but the difference with PASTA is smaller. We also plotted how PASTA and PASOCS perform in terms of number of samples required to converge. In Figure 3, we compare Gibbs sampling on the `iron` dataset. Here, PASTA seems to be more stable on both lower and upper probability. However, even with 5000 samples, both still underestimate the lower probability, even if the values are considerably small. In Figure 4 we compare PASOCS and PASTA on Gibbs sampling and Metropolis Hastings sampling on the `iron` dataset. Also here, PASTA seems more stable, but both algorithms are not completely settled on the real probability after 5000 samples. Finally, Figure 5 compares the unconditional sampling of PASTA and PASOCS on both datasets. Here, the results are similar: after approximately 3000 samples, the



(a) Gibbs.

(b) Metropolis Hastings.

Fig. 5: Comparison of unconditional sampling on the `iron` and the `smoke` datasets.

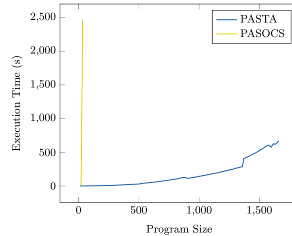


Fig. 6: Comparison between PASTA and PASOCS by increasing the number of probabilistic facts for the `iron` dataset.

computed probability seems to be stabilized. In another experiment, we fixed the number of samples to 1000, increased the size of the instances for the `iron` dataset, and plot how the execution time varies with PASTA and PASOCS. The goal is to check how the execution time varies by increasing the number of samples. The query is `rusty(1)`. Results are shown in Figure 6. For PASOCS, we get a memory error starting from size 32. PASTA requires approximately 500 seconds to take 1000 samples on a program with the structure of Example 2 with 1500 probabilistic facts. Note again that, during sampling, we assume that every world has at least one answer set, since if we need to check this, all the worlds must be generated and clearly the inference will not scale.

6 Conclusions

In this paper, we propose four algorithms to perform approximate inference, both conditional and unconditional, in PASTA programs. We tested the execution time and the accuracy also against the PASOCS solver (after manually performing the conversion of probabilistic conditionals). Empirical results show that our algorithms reach a comparable accuracy in a lower execution time. As future work, we plan to better investigate the convergence of the algorithms and to develop approximate methods for abduction [1,2] in PASTA programs.

References

1. Azzolini, D., Bellodi, E., Ferilli, S., Riguzzi, F., Zese, R.: Abduction with probabilistic logic programming under the distribution semantics. International Journal of Approximate Reasoning **142**, 41–63 (2022). https://doi.org/10.1016/j.ijar.2021.11.003
2. Azzolini, D., Bellodi, E., Riguzzi, F.: Abduction in (probabilistic) answer set programming. In: Calegari, R., Ciatto, G., Omicini, A. (eds.) Proceedings of the 36th Italian Conference on Computational Logic. CEUR Workshop Proceedings, vol. 3204, pp. 90–103. Sun SITE Central Europe, Aachen, Germany (2022)
3. Azzolini, D., Bellodi, E., Riguzzi, F.: Statistical statements in probabilistic logic programming. In: Gottlob, G., Inclezan, D., Maratea, M. (eds.) Logic Programming and Nonmonotonic Reasoning. pp. 43–55. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-15707-3_4
4. Azzolini, D., Riguzzi, F., Lamma, E.: An analysis of Gibbs sampling for probabilistic logic programs. In: Dodaro, C., Elder, G.A., Faber, W., Fandinno, J., Gebser, M., Hecher, M., LeBlanc, E., Morak, M., Zangari, J. (eds.) Workshop on Probabilistic Logic Programming (PLP 2020). CEUR-WS, vol. 2678, pp. 1–13. Sun SITE Central Europe, Aachen, Germany (2020)
5. Azzolini, D., Riguzzi, F., Lamma, E., Masotti, F.: A comparison of MCMC sampling for probabilistic logic programming. In: Alviano, M., Greco, G., Scarcello, F. (eds.) Proceedings of the 18th Conference of the Italian Association for Artificial Intelligence (AI*IA2019), Rende, Italy 19-22 November 2019. Lecture Notes in Computer Science, vol. 11946, pp. 18–29. Springer, Heidelberg, Germany (2019). https://doi.org/10.1007/978-3-030-35166-3_2
6. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. Theor. Pract. Log. Prog. **9**(1), 57–144 (2009). https://doi.org/10.1017/S1471068408003645
7. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (Dec 2011). https://doi.org/10.1145/2043174.2043195
8. Cozman, F.G., Mauá, D.D.: On the semantics and complexity of probabilistic logic programs. J. Artif. Intell. Res. **60**, 221–262 (2017). https://doi.org/10.1613/jair.5482
9. Cozman, F.G., Mauá, D.D.: The joy of probabilistic answer set programming: Semantics, complexity, expressivity, inference. Int. J. Approx. Reason. **125**, 218–239 (2020). https://doi.org/10.1016/j.ijar.2020.07.004
10. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI 2007. vol. 7, pp. 2462–2467. AAAI Press/IJCAI (2007)
11. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. **175**(1), 278–298 (2011). https://doi.org/10.1016/j.artint.2010.04.002
12. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot asp solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019). https://doi.org/10.1017/S1471068418000054
13. Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using networkx. In: Varoquaux, G., Vaught, T., Millman, J. (eds.) Proceedings of the 7th Python in Science Conference. pp. 11–15. Pasadena, CA USA (2008)

14. Halpern, J.Y.: An analysis of first-order logics of probability. Artif. Intell. **46**(3), 311–350 (1990)
15. Jaeger, M.: Probabilistic reasoning in terminological logics. In: Doyle, J., Sandewall, E., Torasso, P. (eds.) 4th International Conference on Principles of Knowledge Representation and Reasoning. pp. 305–316. Morgan Kaufmann (1994). https://doi.org/10.1016/B978-1-4832-1452-8.50124-X
16. Kern-Isberner, G., Thimm, M.: Novel semantical approaches to relational probabilistic conditionals. In: Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning. pp. 382–392. AAAI Press (2010)
17. Lee, J., Wang, Y.: A probabilistic extension of the stable model semantics. In: AAAI Spring Symposia (2015)
18. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987)
19. Nickles, M.: A tool for probabilistic reasoning based on logic programming and first-order theories under stable model semantics. In: Michael, L., Kakas, A. (eds.) Logics in Artificial Intelligence. pp. 369–384. Springer International Publishing, Cham (2016). https://doi.org/doi.org/10.1007/978-3-319-48758-8_24
20. Riguzzi, F.: Foundations of Probabilistic Logic Programming: Languages, semantics, inference and learning. River Publishers, Gistrup, Denmark (2018)
21. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP 1995. pp. 715–729. MIT Press (1995). https://doi.org/10.7551/mitpress/4298.003.0069
22. Totis, P., Kimmig, A., Raedt, L.D.: Smproblog: Stable model semantics in problog and its applications in argumentation. arXiv **abs/2110.01990** (2021). https://doi.org/10.48550/ARXIV.2110.01990
23. Tuckey, D., Russo, A., Broda, K.: Pasocs: A parallel approximate solver for probabilistic logic programs under the credal semantics. arXiv **abs/2105.10908** (2021). https://doi.org/10.48550/ARXIV.2105.10908
24. Wilhelm, M., Kern-Isberner, G., Finthammer, M., Beierle, C.: Integrating typed model counting into first-order maximum entropy computations and the connection to markov logic networks. In: Barták, R., Brawner, K.W. (eds.) Proceedings of the Thirty-Second International Florida Artificial Intelligence Research Society Conference. pp. 494–499. AAAI Press (2019)