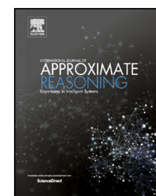
International Journal of Approximate Reasoning 125 (2020) 218–239

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

www.elsevier.com/locate/ijar



# The joy of Probabilistic Answer Set Programming: Semantics, complexity, expressivity, inference

Fabio Gagliardi Cozman [a,*], Denis Deratani Mauá [b]

[a] *Center for Artificial Intelligence (C4AI) and Escola Politécnica, Universidade de São Paulo, Brazil*
[b] *Center for Artificial Intelligence (C4AI) and Instituto de Matemática e Estatística, Universidade de São Paulo, Brazil*

A R T I C L E I N F O

*Article history:*
Received 31 December 2019
Received in revised form 14 May 2020
Accepted 17 July 2020
Available online 31 July 2020

*Keywords:*
Logic programming
Answer Set Programming
Probabilistic programming
Credal sets
Computational complexity
Descriptive complexity

A B S T R A C T

Probabilistic Answer Set Programming (PASP) combines rules, facts, and independent probabilistic facts. We argue that a very useful modeling paradigm is obtained by adopting a particular semantics for PASP, where one associates a credal set with each consistent program. We examine the basic properties of PASP under this credal semantics, in particular presenting novel results on its complexity and its expressivity, and we introduce an inference algorithm to compute (upper) probabilities given a program.

© 2020 Elsevier Inc. All rights reserved.

## 1. Introduction

Combinations of probabilities and rules have been investigated for decades under the banner of “probabilistic logic programming”. Most research and practice has gradually concentrated on Sato’s distribution semantics and its variants. The key idea in Sato’s distribution semantics is to have probabilities associated with a set of independent events in such a way that a unique probability measure is induced over all interpretations of ground atoms.

However, if we try to apply Sato’s strategy to general answer set programs, the distribution semantics fails to guarantee the specification of a single probability distribution over ground atoms under the usual two-valued logic. One possible solution is to extend Sato’s semantics so that sets of probability measures can be specified by answer set programs.

The main goal of this paper is to show that Probabilistic Answer Set Programming (PASP) offers an elegant and enjoyable modeling language when coupled with a particular semantics based on sets of probability measures. As such sets are often referred to as *credal sets*, we refer to the latter extended semantics as the *credal semantics*. Instead of looking at the credal semantics of PASP merely as a way to lend meaning to pathological logic programs that defy Sato’s semantics, here the credal semantics is viewed as a probabilistic programming paradigm that goes beyond existing modeling languages. In short, we propose a programming style where one can ask questions about probability distributions satisfying sets of constraints. A similar movement happened some twenty years ago in logic programming, when it became clear that the stable model semantics was not just a strategy to handle pathological programs, but in fact an attractive programming paradigm in itself.

We also examine some key properties of PASP. We develop analogues to combined and data complexity that take into account the probabilistic character of PASP. We derive the descriptive complexity of PASP, proving in essence that it captures

* Corresponding author.
*E-mail address:* fgcozman@usp.br (F.G. Cozman).

https://doi.org/10.1016/j.ijar.2020.07.004
0888-613X/© 2020 Elsevier Inc. All rights reserved.

the class $\mathsf{PP}^{\Sigma_2^p}$; that is, the class of polynomial nondeterministic Turing machines that count accepting paths with the support of $\mathsf{NP}^{\mathsf{NP}}$ oracles.

Finally, we introduce an inference algorithm that calculates upper probabilities given a PASP program. The algorithm uses the fact that answer set programs can be translated into satisfiability problems, and builds upon existing counting techniques for extended satisfiability problems.

The paper is organized as follows. Section 2 reviews basic concepts of logic programming, and fixes some needed terminology and notation. Section 3 surveys the relevant literature on probabilistic logic programming from a particular historical perspective. Section 4 discusses the proposed programming paradigm. Section 5 examines properties of PASP and Section 6 presents our inference algorithm. A few concluding comments are collected in Section 7.

## 2. A bit of Answer Set Programming

In this section we present relevant syntactic and semantic notions related to Answer Set Programming (ASP). More detailed technical discussion can be found in Ref. [28].

In this paper we employ *atoms*, where an atom is written as $r(t_1, \ldots, t_k)$ with $r$ a predicate of arity $k$ and each $t_j$ either a constant or a logical variable. We do not use functions in this paper. An atom without variables is a *ground atom*. A *literal* is either an *atom* $r(t_1, \ldots, t_k)$ where $r$ is a predicate of arity $k$ and each $t_i$ is either a constant or a logical variable, or an atom preceded by $\neg$ (then we say the atom is *strongly negated*; the logical value of the expression is given by usual Boolean negation).

An ASP program is a set of *rules* such as

$$H_1 \vee \cdots \vee H_m \;:-\; S_1, \ldots, S_n\textbf{.},$$

where each $H_i$ is a literal, each $S_j$ is either a literal $A$ or a literal $A$ preceded by **not** (that is, **not** $A$), and $m + n > 0$. The expression $H_1 \vee \cdots \vee H_m$ is called the *head* of the rule, and $S_1, \ldots, S_n$ is the *body*; each $S_i$ is called a *subgoal*. For instance, here is a rule meaning that if some individual $X$ is a node that is not known to be barred, then $X$ is red or green or blue:

$$\mathsf{red}(X) \vee \mathsf{green}(X) \vee \mathsf{blue}(X) :- \mathsf{node}(X), \textbf{not}\ \mathsf{barred}(X)\textbf{.}.$$

If a rule is such that $m \leq 1$, it is said to be *nondisjunctive*; if $m = 1$ it is said to be *normal*. A rule with $m = 0$ is called a *constraint*. An example of constraint is:

$$:- \mathsf{edge}(X, Y), \mathsf{red}(X), \mathsf{red}(Y)\textbf{.},$$

meaning that two adjacent nodes cannot both be colored red. A normal rule with $n = 0$ is called a *fact*, and instead of writing $H :- \textbf{.}$, we just write $H\textbf{.}$.

The *dependency graph* of a program is a graph where each grounded atom is a node, and where, for each grounded rule, there are edges from the atoms in the body to the atoms in the head. The edge is negative if the atom is the body is preceded by **not** (for some rule), and is positive otherwise. An *acyclic program* is a program with an acyclic dependency graph.

A program with normal rules is said to be *normal*. A program that is normal and contains no **not** and no $\neg$ is said to be *definite*. A program is *stratified* if and only if there is no cycle in the dependency graph that contains a negative edge (that is, an edge that goes through a negation). Every definite program is stratified. A *propositional* program is a program without logical variables.

The *Herbrand base* of a program is the set of ground literals (ground atoms and their strongly negated versions) that can be produced by combining all predicates and constants in the program. An *interpretation* is a consistent subset of the Herbrand base of the program (that is, it does not contain an atom and its strong negation). A ground literal is true (resp., false) with respect to an interpretation when it is (resp., is not) in the interpretation. Similarly, a ground subgoal $A$, where $A$ is a ground literal, is true (resp., false) with respect to an interpretation when $A$ is (resp., is not) in the interpretation, while **not** $A$ is true (resp., false) when $A$ is not (resp., is) in the interpretation. It is worth examining the semantics of **not**: intuitively, **not** $A$ means that literal $A$ is not known explicitly to be true (in fact we may have **not** $\neg A$ where $A$ is an atom, meaning that atom $A$ is not known explicitly to be false). A ground rule is *satisfied* by an interpretation if and only if either some of the subgoals in the body are false or all the subgoals in the body and some of the literals in the head are true with respect to the interpretation. In that definition we assume that the head of a constraint is never satisfied (i.e., a constraint is satisfied if and only if some of its subgoals are false).

A *model* of a program is an interpretation that satisfies all the rules of the program. A model $\mathcal{I}$ of a program is *minimal* if and only if there exists no model $\mathcal{J}$ of the program such that $\mathcal{J} \subset \mathcal{I}$.

Every definite program has a unique minimal model; hence it is natural to take this model as the semantics of the program. With negation, there may be no unique minimal model, and there are several proposed semantics [18].

The *stable model* semantics is based on reducts, defined as follows. Given a program $\mathbf{P}$ and an interpretation $\mathcal{I}$, their reduct $\mathbf{P}^{\mathcal{I}}$ is the propositional program obtained by first removing all grounded rules with **not** $A$ in their body and $A \in \mathcal{I}$, and then by removing each subgoal **not** $A$ from all remaining grounded rules. A stable model of $\mathbf{P}$ is an interpretation $\mathcal{I}$

that is a minimal model of the reduct $\mathbf{P}^{\mathcal{I}}$. The set of stable models is the semantics of **P**. Note that a program may fail to have a stable model: an example is the single-rule program $A :- \textbf{not}\ A$.

Intuitively: if we think of an interpretation as the set of atoms that are assumed true/false, then the stable models of a program are those interpretations that, once assumed, are again obtained by applying the rules of the program. The stable models of a logic program are its *answer sets*.

Answer sets were proposed as a new programming paradigm in 1999 [68,78], where one writes down rules and constraints that characterize a problem in such a way that answer sets are solutions of the problem. Solvers that find answer sets for ASP are now popular, usually operating within a "Guess & Check" methodology. The idea is to use disjunctive rules to guess solutions nondeterministically, and constraints to check whether interpretations are actually solutions [48]. An example should illustrate the idea.

Suppose we are given a graph, encoded by specifying its nodes and edges, say $\mathsf{node}(n1).$, $\mathsf{node}(n2).$, $\ldots$, $\mathsf{edge}(n1, n2).$, $\ldots$. The following program asserts that each node must have one color, and that no two adjacent nodes may share the same color:

$$\begin{array}{l}\mathsf{red}(X) \vee \mathsf{green}(X) \vee \mathsf{blue}(X) :- \mathsf{node}(X).\\ :- \mathsf{edge}(X, Y), \mathsf{red}(X), \mathsf{red}(Y).\\ :- \mathsf{edge}(X, Y), \mathsf{green}(X), \mathsf{green}(Y).\\ :- \mathsf{edge}(X, Y), \mathsf{blue}(X), \mathsf{blue}(Y)..\end{array}$$

Any answer set for the above program is a three-coloring; failure to have an answer set signals failure to have a three-coloring. One can think of the program as first *guessing* a color for each node, and then *checking* whether the required constraint is respected.

Popular ASP packages have additional features such as *aggregates* [29]. We avoid these features here as they would take us too far.

## 3. From the origins of Prolog to the credal semantics

In this section we review some of the history behind probabilistic logic programming, emphasizing design decisions that have gradually moved in the direction of the credal semantics. The reader who wishes to skip historical matters may jump to the next section.

### 3.1. Mixing logic programs and uncertainty

A large part of logic programming, whose origins can be traced back to the sixties [55], is based on rules such as:

$$H \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n, \tag{1}$$

where all $H$ and $B_i$ are atoms. Programs with rules such as Expression (1) have a single minimal model.

During the seventies rules were enlarged with negation as failure, where each atom $B_i$ in Expression (1) may appear negated as $\textbf{not}\ B_i$. There was then a long debate on the best semantics to adopt in the presence of negation as failure.

Another relevant feature of the seventies was the gradual development of expert systems, many of them based on rules. One notable expert system was MYCIN, where "certainty factors" were coupled to if-then rules, on the grounds that probability theory would face difficulties against scarce data and uncertain information [94].[1] The same term "certainty factor" was used by Shapiro in 1983 to attach numbers to atoms in Prolog-style rules [93], later to be given a probabilistic interpretation [52]. Say for instance that a single rule $A \leftarrow B \wedge C$ is present with both $B$ and $C$ getting 0.5; then the rule assigns 0.25 to $A$.

During the eighties various facets of uncertainty were captured by "annotated" extensions of logic programming, often with connections to possibilistic and fuzzy logics [53,100]. The typical annotated logic rule has syntax:

$$H : \nu \leftarrow B_1 : \mu_1 \wedge B_2 : \mu_2 \wedge \cdots \wedge B_n : \mu_n, \tag{2}$$

with $H$ and each $B_i$ as before, perhaps with negation in the body, while $\nu$ and each $\mu_i$ are *annotations*. Usually the annotations are numbers, but many other possibilities have been contemplated; for instance, annotations may be intervals carrying belief and plausibility as prescribed by Dempster-Shafer theory [105].

At the cost of sacrificing some chronological consistency, in the remainder of this section we review a long line of proposals that mix rules and probability intervals.

A notable approach by Ng and Subrahmanian appeared in 1992 [76] by pursuing interval-valued annotations. They considered annotated clauses as in Expression (2), but where annotations are probability intervals. If $A : [0.2, 0.3]$ is an

[1] The semantics of certainty factors generated much debate and eventually was linked to Dempster-Shafer theory [40], where each event is associated with an interval between its *belief* and its *plausibility*. The belief functions employed in Dempster-Shafer theory are infinitely monotone Choquet capacities, objects that will be important later in this paper.

annotated atom, the intended reading is "The probability of $A$ belongs to the interval $[0.2, 0.3]$". To use an example by Ng and Subrahmanian, the following annotated clause

$$\mathsf{path}(X,Y):[0.85,1] \leftarrow \mathsf{connect}(X,Z):[1,1] \wedge \mathsf{path}(Z,Y):[0.75,1]$$

says that if there is a connection between $X$ and $Z$, and a path from $Z$ and $Y$ with probability at least 0.75, then there is a path between $X$ and $Y$ with probability at least 0.85. Ng and Subrahmanian studied the models of such annotated clauses and techniques to compute probability intervals, in work that was later expanded in many directions [95].

Lakshmanan and Sadri proposed a somewhat intimidating syntax for rules [56]:

$$H \xleftarrow{\langle [a,b],[c,d] \rangle} B_1 \wedge B_2 \wedge \cdots \wedge B_n,$$

where $[a, b]$ conveys the "belief" in favor of $H$ and $[c, d]$ conveys the "doubt" against $H$, both of them endowed with probabilistic meaning. A more recent effort by Dekhtyar and Subrahmanian investigated *hybrid* probabilistic programs, where one can specify relations of dependence or independence as well as operations to combine probabilities in annotated rules [23].

We must also mention the probabilistic logic programming scheme proposed by Lukasiewicz [63], where the syntax $(H|B)[a, b]$ is used to indicate that the conditional probability of $H$ given $B$ is in $[a, b]$. Thus each "rule" is in fact a constraint on conditional probabilities. Similarly, another research program that started during the nineties [42,51] looked into rules of the form

$$H \xleftarrow{[a,b]} B_1 \wedge B_2 \wedge \cdots \wedge B_n,$$

where $a$ and $b$ are interpreted respectively as lower and upper bounds on conditional probabilities for the head given the body. That work emphasized independence relations and connections with Bayesian networks (eventually investigating Bayesian networks endowed with probability intervals [96]).

### 3.2. The path to the distribution semantics and its close relatives

Dantsin proposed in 1990 to divide a logic program in two parts [17]. The first part consists of a set of rules as Expression (1), enlarged with negation as failure. The second part consists of facts that are associated with probabilities (referred to as *uncertain clauses*). At the time there was considerable debate over the semantics of negation, thus to simplify matters Dantsin assumed that clauses were *stratified*.[2] Dantsin showed that the maximum entropy distribution over uncertain facts is a product measure that makes all uncertain facts independent (Dantsin adopted maximum entropy as suggested by Nilsson in the context of probabilistic logic [79]). This single product probability measure in turn induces a distribution over all ground atoms, given that the rules are stratified and thus induce a single truth-value assignment over the (non-uncertain) atoms. Hence every program within Dantsin's guidelines specifies a single probability measure over all ground atoms.

Poole proposed, in 1991, a language consisting of rules and *assumables*, where an assumable is an atom associated with a probability value [81]. The assumables are taken to be independent, thus a product probability measure is imposed over them. Poole shows that such a language can specify Bayesian networks, in some cases producing very natural descriptions. This scheme was refined by Poole in a very influential 1993 paper, with added material on abduction, explanations, causation [82].[3] A critical decision in Poole's original proposal was to focus on sets of rules where dependencies over atoms are acyclic (hence stratified), thus guaranteeing that any selection of assumables induces a single minimal model over all ground atoms. Consequently, the product probability measure over assumables induces a single probability measure over all ground atoms. Consider a trivial example, written in a simple syntax. Suppose we have assumables $A$, $B$, $C$ with associated probabilities

$$\mathbb{P}(A) = 0.2, \qquad \mathbb{P}(B) = 0.4, \qquad \mathbb{P}(C) = 0.6,$$

and rules

$$D \leftarrow A \wedge B \qquad \text{and} \qquad D \leftarrow B \wedge C.$$

Then $D$ is true whenever $A$ and $B$ are true or $B$ and $C$ are true; that is, $\mathbb{P}(D) = 0.2 \times 0.4 + 0.4 \times 0.6 - 0.2 \times 0.4 \times 0.4 \times 0.6 = 0.3008$.

Similar ideas were pursued almost simultaneously by Fuhr and by Sato.

[2] We discuss stratified programs later; for now it suffices to say that a stratified program has a single minimal model that assigns either true or false to every ground atom.

[3] Recent work has returned to abduction techniques in the context of probabilistic logic programming [98].

Fuhr's Probabilistic Datalog[4] [33] used stratified rules and allowed for additional ground facts to be associated with probabilities such as

$$0.9 \;\; \mathsf{indterm}(\mathsf{d1}, \mathsf{db}).$$

Fuhr assumes that these facts are independent, so they behave as Dantsin's uncertain facts and as Poole's assumables, thus inducing a single probability measure over all interpretations of ground atoms.

Likewise, Sato proposed to have two distinct sets of sentences [90]. One of them consists of usual rules. The other consists of *random facts* that are atoms associated with probability values. Sato's *distribution semantics* prescribes the following interpretation. Each random fact is associated with a *switch*; these switches are independent and they select one of a set of associated random facts with its prescribed probability. Thus we have a product probability measure over selections of random facts. For each such selection, we obtain a complete logic program (selected facts plus original rules). Thus the product probability measure over switches induces a probability measure over programs. As Sato did not use negation, each realization of random facts produces a single minimal model, hence any probabilistic logic program induces a single probability measure. Sato implemented his distribution semantics in the popular PRISM package [91], and his ideas were gradually adopted by most of the related literature.

It is perhaps fair to say that Dantsin, Poole, Fuhr, and Sato proposed the same overall strategy to guarantee that a probabilistic logic program specifies a single probability measure. However, to guarantee uniqueness they employed different assumptions; Sato discarded negation, while Poole demanded acyclicity, and Dantsin and Fuhr required stratification.

### 3.3. Probabilistic relational models and probabilistic inductive logic programming

The nineties also witnessed a growing, and eventually explosive, interest in probabilistic models specified by relational means, with particular interest in enlarging Bayesian networks with the ability to handle uncertainty over relations and individuals. Starting from template-based languages [5,37,45,66,106], many specification schemes were devised by importing features of first-order logic. The term "Probabilistic Relational Model" (PRM) was coined [32,54]; related specification languages were variously diagramatic [44] or textual, sometimes relying on logic programming — rules were used primarily as template specification devices [11,16,30,38,50]. There are also useful languages based on functional programming [39,46] that are less closely related to the topic of this paper. The vast ensuing literature was united in one purpose: to find ways to specify a single probability distribution, preferably one that could be viewed as a Bayesian or a Markov network, from a set of predicates and individuals. Recently there have been efforts to analyze complexity of such languages in an abstract manner [14]. There was particular excitement around learning techniques for these models. Several excellent surveys about that material are available, some in book format [12,36,49,71,85,99].

Interest in learning probabilistic logic programs also surged after 1995, often under the banner of "Probabilistic Inductive Logic Programming". Many such learning techniques gravitated around the distribution semantics, while others contemplated alternative semantics based on probabilities over proofs. Again the literature is vast but there are excellent surveys available, several in book format [20,21,88,89].

As a digression, we mention an idiosyncratic proposal that appeared in the mid-nineties: Ngo and Haddawy's combination of annotated rules and probabilistic relational assessments [77], where one can use *probabilistic sentences* of the form

$$(\mathbb{P}(H_0|H_1,\ldots,H_m)=\alpha) \;\leftarrow\; B_1 \wedge B_2 \wedge \cdots \wedge B_n,$$

where each $H_i$ is (in essence) an atom, and each $B_j$ is (in essence) an atom possibly preceded by negation [77]. Intuitively, the probabilistic sentence reads "if the context given by the conjunction $B_1 \wedge \ldots \wedge B_n$ holds, then the conditional probability for $H_0$ holds". They introduced assumptions of independence and *combining functions* to merge information from two or more rules with identical heads and non-independent bodies. The proposal was very rich and required heavy machinery to produce inferences.

### 3.4. The evolution and the challenges of the distribution semantics

Through the more than twenty years of the distribution semantics, several languages have been proposed and several implementations have been released. (During that time, Answer Set Programming matured, first through the consensus over the semantics for negation [34,101], then over disjunctive programs [72], and also over additional features such as strong negation [28].)

A language that emerged around 2007, and whose conventions and syntax we later adopt, is ProbLog [31]. ProbLog has been implemented in a freely available package with excellent interface and state-of-art inference algorithms for stratified probabilistic logic programs.

[4] Datalog does not allow functions. As noted already, we do not allow for functions in this paper.

Another example of evolution within the distribution semantics is the language of Logic Programs with Annotated Disjuctions (LPADs), where a rule is written in the form [103]:

$$H_1 : \nu_1; H_2 : \nu_2; \ldots; H_m : \nu_m \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n,$$

where all $H_i$ and $B_j$ are as before (each $B_j$ may be preceded by negation), and the $\nu_i$ are probability values that add up to one. The interpretation is simple: if the body is satisfied, then an atom in the head is selected with the associated probability. And despite its apparent distance from previous languages, LPADs can be translated into ProbLog, thus satisfying the distribution semantics [87]. It is not necessary to go over other extensions here, specially because the excellent textbook by Riguzzi [87] presents a detailed picture of the state-of-art in probabilistic logic programming based on the distribution semantics.

However, despite all of this progress, the distribution semantics still has a weak spot. For the distribution semantics to yield a probability measure we must have a language such that any realization of random facts produces a logic program with a single minimal model. If one has rules that contain negation and that are not stratified, a program may have many minimal models, or no model; then one cannot generate a single probability measure. Riguzzi reserves the adjective "non-sound" for probabilistic logic programs that, for at least one realization of the random facts, have more than one or no minimal models.

Refined assumptions or techniques have been proposed to go beyond acyclic and stratified logic programs [43,83,84,86,91,92], either by keeping restrictions that avoid the "non-sound" status, or by resorting to three-valued semantics. In this latter scheme, we should allow three truth-values to be associated with atoms: "true", "false", "undefined". The extent to which one can make sense of the mixture between probabilities and undefined values is unclear [13]. We prefer not to follow this route.

Matters get more complicated for the distribution semantics when one considers disjunctive heads. There is overwhelming support in favor of answer set semantics for disjunctive heads, and even the simplest logic programs with disjunctive heads yield several answer sets. Thus a direct extension of Sato's semantics to Answer Set Programming seems elusive.

There are a few formalisms that deal with several answer sets per realization of random facts by somehow selecting one probability measure over the answer sets. This idea is pursued in the P-log language [6], where a uniform distribution is adopted over answer sets whenever more than one answer set is produced. Another strategy is adopted by the language $\text{LP}^{MLN}$, where a unique Markov random field is always generated over all ground atoms [58,59]. The challenge there, of course, is to justify the selection of a particular probability measure. The PrASP language, a flexible formalism that even allows first-order sentences and interval-valued probabilistic rules, employs maximum entropy to select a single distribution [74]. The use of PrASP to represent a variety of scenarios has been explored using a full-fledged implementation [75].

### 3.5. Semantics based on credal sets

The semantics for logic programs (with disjunctive heads, negation, cycles) we explore in this paper comes from a proposal by Lukasiewicz [64,65]. The idea is simple: if a probabilistic logic program is such that it has more than a minimal model for some realization of the random facts, we simply build the credal set consisting of *all* possible distributions induced by the product measure over random facts. An example is perhaps the best way to clarify this idea [13, Example 9]; we formalize the credal semantics later.

**Example 1.** Suppose we have a random fact $A$ associated with probability 0.3, and rules

$$S \;:-\; \textbf{not}\, W, \textbf{not}\, A., \qquad W \;:-\; \textbf{not}\, S..$$

This logic program is not stratified and it fails to have a distribution semantics. With probability 0.3, $A$ is included in the program; using the stable model semantics defined later, the resulting logic program has a single model where $W$ is true and $S$ is false. And with probability 0.7, $A$ is not included in the program and is false; then we get two models, one where $W$ is true and $S$ is false, and another where $W$ is false and $S$ is true. Thus $\mathbb{P}(A = \text{true}) = 0.3$ but $\mathbb{P}(W = \text{true}) \in [0.3, 1]$ and $\mathbb{P}(S = \text{true}) \in [0, 0.7]$. In fact all probability measures that satisfy both the probabilistic fact and the rules can be parameterized as

$$\mathbb{P}(A = \text{true}) = 0.3, \quad \mathbb{P}(W = \text{true}) = 0.3 + 0.7\alpha, \quad \mathbb{P}(S = \text{true}) = 0.7(1 - \alpha),$$

for $\alpha \in [0, 1]$. □

This semantics was referred to simply as the "stable model" semantics by Lukasiewicz, but this is perhaps confusing as the answer set semantics is associated with (pure) logic programs. We instead use "credal semantics", inspired by the term "credal set" [60].

We have previously studied the complexity of inferences under the credal semantics [13]; in that investigation we have studied the credal semantics under many features of Answer Set Programming, such as disjunctive heads, strong negation, and even aggregates [69].

Despite the flexibility of the credal semantics, one might ask whether it is only a technical device to lend meaning to pathological probabilistic logic programs. The main purpose of this paper is to show that the credal semantics allows one to specify and to solve interesting problems.

To conclude this section, we note that related interest in credal sets can be found in recent literature; for instance, Michels et al. [70] consider a language whose semantics also relies on credal sets, even though they are built in different ways. Similarly, Antonucci and Facchini [1] enlarge Poole's language to allow for more general specification, this way obtaining a semantics based on credal sets. An even more ambitious language is explored by Dekhtyar and Dekhtyar [22] as they in essence combine Ng and Subrahmanian's annotated logic [76] with disjunctive heads — thus obtaining very general sets of probability measures as semantics.

## 4. The joy of Probabilistic Answer Set Programming

In this section we explore Answer Set Programming enriched with the idea that probabilities are associated with a selected number of facts. We adopt the conventions and syntax of the ProbLog package [31]. Thus we refer to the facts associated with probabilities as *probabilistic facts*; they in essence correspond to Sato's random facts.

A probabilistic fact consists of an atom $A$ associated with a probability $\alpha$, assumed to be a rational number in $[0, 1]$; the probabilistic fact is written as follows:

$$\alpha :: A\textbf{.}.$$

A probabilistic fact may contain logical variables; for instance we may write $0.25 :: \mathsf{edge}(X, Y)\textbf{.}$. If we then have constants node1 and node2, the latter probabilistic fact can be grounded into

$$0.25 :: \mathsf{edge}(\mathsf{node1}, \mathsf{node1})\textbf{.}, \qquad 0.25 :: \mathsf{edge}(\mathsf{node1}, \mathsf{node2})\textbf{.},$$
$$0.25 :: \mathsf{edge}(\mathsf{node2}, \mathsf{node1})\textbf{.}, \qquad 0.25 :: \mathsf{edge}(\mathsf{node2}, \mathsf{node2})\textbf{.}.$$

Probabilistic Answer Set Programming (PASP) deals with programs consisting of probabilistic facts, probabilistic facts, and rules. In ProbLog, the interpretation of the probabilistic fact $\alpha :: A\textbf{.}$ is as follows: with probability $\alpha$, the probabilistic fact is replaced by the fact $A\textbf{.}$; with probability $1 - \alpha$, the probabilistic fact is simply removed from the program. This semantics induces a probability distribution over logic programs.

However, a different semantics for probabilistic facts is adopted in this paper, where:

- we add $A\textbf{.}$ to the program with probability $\alpha$; and
- we add $\neg A\textbf{.}$ to the program with probability $1 - \alpha$.

We discuss the rationale for this semantics in Section 4.3.

So, take the set of probabilistic facts of the program of interest, and ground them as needed to obtain a set of ground probabilistic facts $\mathcal{F} = \{\alpha_i :: A_i\textbf{.}\}_{i \in I}$. A *total choice* $\theta$ is a subset of these latter probabilistic facts. Once we collect the facts from $\theta$ and the strongly negated facts from $\mathcal{F} \backslash \theta$, and add them to the facts and rules of the original program, we obtain an ASP program. So, if we have $N$ ground probabilistic facts, we have $2^N$ total choices, and we thus have $2^N$ possible ASP programs. The remaining bit is to define the probability of each total choice (hence the probability that each ASP program is generated), but this is simple enough: ground probabilistic facts are supposed stochastically independent, hence

$$\mathbb{P}(\theta) = \left( \prod_{i:(\alpha_i::A_i) \in \theta} \alpha_i \right) \left( \prod_{i:(\alpha_i::A_i) \notin \theta} (1 - \alpha_i) \right). \tag{3}$$

Now say that a PASP program is *consistent* if and only if there is at least one answer set for the logic program generated by each total choice of probabilistic facts.

**Definition 1.** A *probability model* for a consistent PASP program is a probability measure $\mathbb{P}$ over interpretations of the ground atoms of the program, such that for each total choice $\theta$ we have $\mathbb{P}(\theta)$ given by Expression (3) and $\mathbb{P}(\cdot|\theta)$ a probability distribution over the answer sets induced by $\theta$.

**Definition 2.** The *credal semantics* of a PASP program is the credal set of all its probability models.

Under the credal semantics we cannot necessarily associate each ground atom $A$ with a sharp probability value. Instead, we can associate $A$ with a *lower probability* $\underline{\mathbb{P}}(A)$ and an *upper probability* $\overline{\mathbb{P}}(A)$. The lower probability is the infimum over all probability values for $A$, for all possible probability models; likewise, the upper probability is the supremum over these probability values.

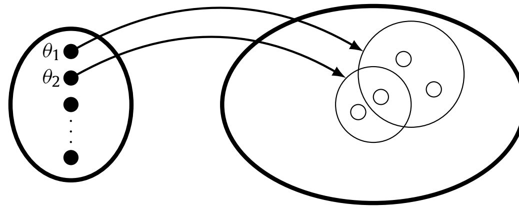



**Fig. 1.** The credal semantics. The set of total choices is shown to the left; there is a single product probability measure over them. Each total choice defines an ASP program that maps to a set of answer sets, shown to the right: total choice $\theta_1$ maps to three answer sets, $\theta_2$ to two answer sets, and so on.

```
0.01 :: trip.      0.5 :: smoking.
tuberculosis :− trip, a1.      0.05 :: a1.
tuberculosis :− not trip, a2.      0.01 :: a2.
cancer :− smoking, a3.      0.1 :: a3.
cancer :− not smoking, a4.      0.01 :: a4.
or :− tuberculosis.      or :− cancer.
test :− or, a5.      0.98 :: a5.
test :− not or, a6.      0.05 :: a6.
```

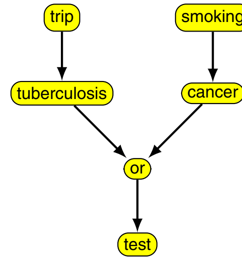



**Fig. 2.** Left: An acyclic PASP program, based on a popular Bayesian network [57]. Right: the acyclic dependency graph of the program (except auxiliary predicates); the graph is also the structure of the corresponding Bayesian network.

The following result will be useful later.[5] Basically, it clarifies the structure of the credal semantics by connecting it with the theory of infinitely monotone Choquet capacities [2,73].

**Theorem 1.** *Suppose we have a PASP program whose semantics is a credal set $\mathbb{K}$. Then: 1) the lower probability with respect to $\mathbb{K}$ is an infinitely monotone Choquet capacity; 2) $\mathbb{K}$ is the largest credal set dominating this capacity; 3) $\mathbb{K}$ is closed and convex.*

**Proof.** Refer to Fig. 1: we have a space endowed with a single probability distribution, and a multi-valued mapping into a second space. Results from the theory of Choquet capacities then lead to the desired representation through credal sets [2,73]. □

One valuable consequence of Theorem 1 is that we immediately obtain expressions for lower/upper conditional probabilities by applying well-known results from the theory of infinitely monotone Choquet capacities. We have, for events $\mathcal{A}$ and $\mathcal{B}$:

$$\overline{\mathbb{P}}(\mathcal{A}|\mathcal{B}) = \frac{\overline{\mathbb{P}}(\mathcal{A}\cap\mathcal{B})}{\overline{\mathbb{P}}(\mathcal{A}\cap\mathcal{B}) + \underline{\mathbb{P}}(\mathcal{A}^c\cap\mathcal{B})},$$
$$\underline{\mathbb{P}}(\mathcal{A}|\mathcal{B}) = \frac{\underline{\mathbb{P}}(\mathcal{A}\cap\mathcal{B})}{\underline{\mathbb{P}}(\mathcal{A}\cap\mathcal{B}) + \overline{\mathbb{P}}(\mathcal{A}^c\cap\mathcal{B})}.$$

These expressions hold because with the multi-valued mapping in the proof of Theorem 1 we can place as much probability as possible in $\mathcal{A}\cap\mathcal{B}$ by removing as much probability as possible from $\mathcal{A}^c\cap\mathcal{B}$ (and vice-versa). In fact, $\overline{\mathbb{P}}(\mathcal{A}\cap\mathcal{B}) + \underline{\mathbb{P}}(\mathcal{A}^c\cap\mathcal{B})$ is the value of $\mathbb{P}(\mathcal{B})$ for the probability measure $\mathbb{P}$ that attains $\overline{\mathbb{P}}(\mathcal{A}|\mathcal{B})$, a result we use later.

In the remainder of this section we show how the credal semantics of PASP can be used to represent many nontrivial probabilistic problems. We start with nondisjunctive acyclic programs, a very simple and well-known case. We then look at nondisjunctive stratified ones, and then face the general case.

### *4.1. Nondisjunctive acyclic programs*

The short nondisjunctive PASP program in Fig. 2 is *acyclic*. Because it is acyclic, the propositional program in Fig. 2 is quite easy to read, as its probabilistic facts and rules translate directly into probabilities. The probabilities are as follows:

$$\begin{array}{ll} \mathbb{P}(\mathsf{trip}=1)=0.01, & \mathbb{P}(\mathsf{smoking}=1)=0.5, \\ \mathbb{P}(\mathsf{tuberculosis}=1|\mathsf{trip}=1)=0.05, & \mathbb{P}(\mathsf{tuberculosis}=1|\mathsf{trip}=0)=0.01, \\ \mathbb{P}(\mathsf{cancer}=1|\mathsf{smoking}=1)=0.1, & \mathbb{P}(\mathsf{cancer}=1|\mathsf{smoking}=0)=0.01, \\ \mathbb{P}(\mathsf{test}=1|\mathsf{or}=1)=0.98, & \mathbb{P}(\mathsf{test}=1|\mathsf{or}=0)=0.05, \end{array}$$

[5] Note that this result has been alluded to before [13] in connection with Sato's semantics for probabilistic facts.

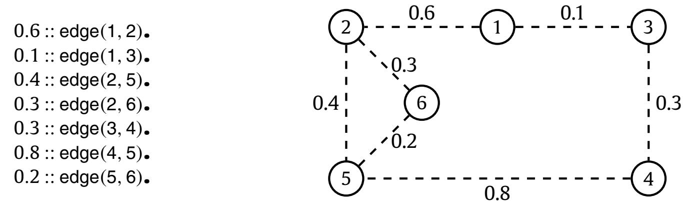



**Fig. 3.** A random undirected graph.

and additionally,

$$\text{or} = \{\text{tuberculosis} = 1\} \vee \{\text{cancer} = 1\}.$$

Note that here we conflate atoms and the random variables they correspond to (random variables that are defined over the space of all interpretations).

In Fig. 2 we employ the pattern

$$A :- B_1, B_2. \qquad \alpha :: B_2..$$

We might take this pair as a "probabilistic rule" (in fact ProbLog accepts the syntax $\alpha :: A :- B_1, B_2.$ to represent this pair). In this paper we do not use special syntax for probabilistic rules for the sake of simplicity.

Any acyclic propositional program can be viewed as the specification of a Bayesian network over binary random variables: the structure of the Bayesian network is the dependency graph; the random variables correspond to the atoms; the probabilities can be read off of the probabilistic facts and rules. Conversely, any Bayesian network over binary variables can be specified by an acyclic nondisjunctive PASP program [84]. In fact, the program in Fig. 2 has been generated from a well-known Bayesian network [57].

Acyclic programs can be used to specify "relational Bayesian networks" as well.[6] For instance, here is an acyclic PASP program that captures part of the well-known University World [35], where we have courses, students, and grades:

$$\begin{aligned}
&\text{apt}(X) :- \text{student}(X), \text{a1}. \qquad 0.7 :: \text{a1}.\\
&\text{easy}(Y) :- \text{course}(X), \text{a2}. \qquad 0.4 :: \text{a2}.\\
&\text{pass}(X, Y) :- \text{student}(X), \text{apt}(X), \text{course}(Y), \text{easy}(Y).\\
&\text{pass}(X, Y) :- \text{student}(X), \text{apt}(X), \text{course}(Y), \textbf{not}\ \text{easy}(Y), \text{a3}. \qquad 0.8 :: \text{a3}..
\end{aligned}$$

This is unrealistically simple but it conveys the idea: apt students do well in easy courses, and even in courses that are not easy with probability 0.8 (and a student is apt with probability 0.7; a course is easy with probability 0.4).

### *4.2. Nondisjunctive stratified programs*

A program still specifies a single probability distribution if it is nondisjunctive and stratified. Because any nondisjunctive stratified program has a unique minimal model [18], any total choice induces a unique model and therefore the probabilistic program induces a unique probability distribution over interpretations. Most of the literature on probabilistic logic programs is restricted to this class of programs [31].

Here is a prototypical example of nondisjunctive stratified program:

$$\begin{aligned}
&\text{edge}(X, Y) :- \text{edge}(Y, X).\\
&\text{path}(X, Y) :- \text{edge}(X, Y).\\
&\text{path}(X, Y) :- \text{edge}(X, Z), \text{path}(Z, Y)..
\end{aligned}$$

These rules can be spelled out as follows: there is a path between two nodes if there is an undirected edge between them, or if there is an undirected edge from one of them to some node from which there is a path to the other node. Coupled with an explicit description of the predicate edge, this program allows one to determine whether it is possible or not to find a path between two given nodes. For instance, if we have a random graph specified as in Fig. 3 (the drawing depicts a visual representation of the random graph, with edges annotated with probabilities), then $\mathbb{P}(\text{path}(1, 4)) = 0.2330016$.

As illustrated by this example, stratified PASP programs can encode recursion. This is a feature that cannot be reproduced with Probabilistic Relational Models (Section 3.3) based on first-order logic, as recursion goes beyond the resources of first-order logic [24].

[6] As noted in Section 3.3, there are several other languages that compactly describe Bayesian networks over repetitive domains.

*4.3. The general case*

If we have a nondisjunctive and non-stratified program, or a disjunctive program, then it may be the case that for some total choices the program has no answer set, or that for some total choices the program has many answer sets. Suppose first that some total choice leads to a program without any answer set; in this case we take that the whole probabilistic program is inconsistent and has no semantics. Other strategies might be possible; we might try to repair the inconsistency [8], or perhaps resort to some three-valued semantics that can accommodate such failures. We leave such strategies to future debate.

The more interesting case is the one in which any total choice induces at least one answer set, and some total choices lead to many answer sets. As a total choice $\theta$ is associated with a probability $\mathbb{P}(\theta)$, the individual answer sets induced by $\theta$ collectively get mass $\mathbb{P}(\theta)$, but no other stipulations are present. Thus we can distribute $\mathbb{P}(\theta)$ arbitrarily over the answer sets, and we can use such a freedom to our advantage. A few examples illustrate possible strategies.

Consider as an example the three-colorability problem for undirected graphs:

$$\begin{array}{l}\mathsf{red}(X) \vee \mathsf{green}(X) \vee \mathsf{blue}(X) \coloneq \mathsf{node}(X).\\ \mathsf{edge}(X,Y) \coloneq \mathsf{edge}(Y,X).\\ \coloneq \mathsf{edge}(X,Y), \mathsf{red}(X), \mathsf{red}(Y).\\ \coloneq \mathsf{edge}(X,Y), \mathsf{green}(X), \mathsf{green}(Y).\\ \coloneq \mathsf{edge}(X,Y), \mathsf{blue}(X), \mathsf{blue}(Y).\end{array} \quad (4)$$

plus the probabilistic facts in Fig. 3 and the facts

$$\mathsf{red}(1)., \qquad \mathsf{green}(4)., \qquad \mathsf{green}(6).. \quad (5)$$

Each total choice fixes a graph; for this particular graph, each answer set is a three-coloring. If it so happens that each total choice induces a single three-coloring, then the program defines a single probability distribution over interpretations. If instead some total choices induce several three-colorings, then the program defines a non-singleton set of probability distributions: for each total choice $\theta$, the probability mass $\mathbb{P}(\theta)$ can be attached to each one of the answer sets induced by $\theta$. Take for instance node 3. If both edges to 1 and 4 are present (with probability 0.03), then all answer sets must contain blue(3). And if both edges are absent, then there are answer sets containing either red(3) or blue(3) or green(3). Other configurations are produced by the presence/absence of edges.

We have that, in the last example, $\underline{\mathbb{P}}(\mathsf{blue}(3)) = 0.03$; $\overline{\mathbb{P}}(\mathsf{blue}(3)) = 1$; $\underline{\mathbb{P}}(\mathsf{red}(3)) = 0.0$; $\overline{\mathbb{P}}(\mathsf{red}(3)) = 0.9$.

Suppose now that on top of probabilistic facts in Fig. 3 and hard facts in Expression (5), we also have the probabilistic fact

$$0.2 :: \mathsf{blue}(5).. \quad (6)$$

In this case some total choices fail to produce a three-coloring: for instance, if all edges are present, then there is no way to produce a three-coloring when blue(5) is set to hold. To have a consistent program, we must work differently. Consider the PASP program:

$$\begin{array}{l}\mathsf{red}(X) \vee \mathsf{green}(X) \vee \mathsf{blue}(X) \coloneq \mathsf{node}(X).\\ \mathsf{edge}(X,Y) \coloneq \mathsf{edge}(Y,X).\\ \neg\mathsf{colorable} \coloneq \mathsf{edge}(X,Y), \mathsf{red}(X), \mathsf{red}(Y).\\ \neg\mathsf{colorable} \coloneq \mathsf{edge}(X,Y), \mathsf{green}(X), \mathsf{green}(Y).\\ \neg\mathsf{colorable} \coloneq \mathsf{edge}(X,Y), \mathsf{blue}(X), \mathsf{blue}(Y).\\ \mathsf{red}(X) \coloneq \neg\mathsf{colorable}, \mathsf{node}(X), \textbf{not}\ \neg\mathsf{red}(X).\\ \mathsf{green}(X) \coloneq \neg\mathsf{colorable}, \mathsf{node}(X), \textbf{not}\ \neg\mathsf{green}(X).\\ \mathsf{blue}(X) \coloneq \neg\mathsf{colorable}, \mathsf{node}(X), \textbf{not}\ \neg\mathsf{blue}(X).\\ \mathsf{colorable} \coloneq \textbf{not}\ \neg\mathsf{colorable}..\end{array} \quad (7)$$

This program is certainly non-trivial. Basically, if there is a three-coloring for the input graph, the corresponding interpretation is an answer set. But if there is no three-coloring, then colorable is set to false, and all groundings of red, blue and green are set to true except for those groundings that are set to false via probabilistic facts. To guarantee the latter behavior the program resorts to the idiom **not** $\neg A$, where $A$ is an atom: basically this is true whenever it is not known explicitly that $A$ is false. As answer sets must be minimal, colorable is true if and only if there is a three-coloring. In our example, $\overline{\mathbb{P}}(\mathsf{colorable}, \mathsf{blue}(3)) = 0.976$. In fact, we can ask for more: we can determine the probability that *there is no coloring at all*; this is precisely 0.024. As colorable indicates the possibility of a three-coloring for each total choice, there is a sharp value for its probability.

Thus in PASP we can formulate a combinatorial problem and ask for the lower/upper probability of its atoms; also, we can ask for the probability that there is a solution at all.

Using the three-coloring example we can analyze in more detail the semantics of probabilistic facts. Recall that ProbLog's semantics takes probabilistic fact $\alpha :: A.$ to mean that we should impose $A.$ with probability $\alpha$ and discard it with probability $1-\alpha$. Our approach is different in that $\alpha :: A.$ means:

$$\begin{cases} \text{take } A \text{ with probability } \alpha; \\ \text{take } \neg A \text{ with probability } 1-\alpha. \end{cases}$$

Of course we can do so because strong negation is available in ASP; however, mere availability of $\neg$ is not the key point. To understand our proposal, take again the three-coloring program in Expression (7) with probabilistic facts in Expressions (5) and (6). If we adopt our proposed semantics for the probabilistic facts, the credal semantics yields $\mathbb{P}(\mathsf{blue}(5)) = 0.2$ and $\mathbb{P}(\mathsf{colorable}, \mathsf{blue}(5)) = 0.1856$ (some probability mass is "lost" to configurations without three-colorings). If we use ProbLog's semantics for the probabilistic facts, and continue with the construction of the credal semantics, we obtain $\underline{\mathbb{P}}(\mathsf{colorable}, \mathsf{blue}(5)) = 0.1856$ but $\overline{\mathbb{P}}(\mathsf{colorable}, \mathsf{blue}(5)) = 0.9280$! This behavior of ProbLog's semantics may be manageable in acyclic programs, where the effect of probabilistic facts is relatively easy to grasp.[7] In the presence of cyclic rules and constraints, such as the ones typically found in ASP programming, it does not seem appropriate to leave this management to the programmer, and it seems better to alert her about problems through inconsistency.

To finish this section, we briefly comment on the ability of PASP to solve complex problems.

Suppose we have a collection of companies $\mathcal{C} = \{c_1, \ldots, c_m\}$, such that each company manufactures a range of products. Each product $g_j$ is manufactured by two companies, as specified by atom $\mathsf{produce}(c_{i_1}, c_{i_2}, g_j)$. Each company $c$ may be owned by three other companies, as specified by the predicate $\mathsf{control}(c_{i_1}, c_{i_2}, c_{i_3}, c)$ (a company may be controlled by more than one triple). A *strategic set* $\mathcal{C}'$ of companies is a minimal subset of $\mathcal{C}$ (minimal with respect to inclusion) such that: 1) the set of all products manufactured by $\mathcal{C}$ is identical to the set of all products manufactured by $\mathcal{C}'$; 2) if the three controlling companies of a company $c$ are in $\mathcal{C}'$, then $c$ is also in $\mathcal{C}'$. The question is, given a company $c$, is it in some strategic set? This problem is $\mathsf{NP}^{\mathsf{NP}}$-complete (Appendix A), hence it is widely believed to be harder than the three-coloring problem. Amazingly, the "strategic company" problem can be solved by a short ASP program [25]:

$$\begin{aligned} &\mathsf{strategic}(C1) \vee \mathsf{strategic}(C2) :- \mathsf{produce}(C1, C2, G). \\ &\mathsf{strategic}(C) :- \mathsf{control}(C1, C2, C3, C), \\ &\qquad\qquad \mathsf{strategic}(C1), \mathsf{strategic}(C2), \mathsf{strategic}(C3).. \end{aligned}$$

The first rule guarantees that all products are still sold by the strategic set. The second rule guarantees that, if three companies are in the strategic set, then a company they control is also in the strategic set. The minimality of answer sets guarantees the required minimality of strategic sets.

Now suppose, as it so happens in real life, that there is some uncertainty as to which company controls which. We may then be interested in the probability that some company comp is in some strategic set. We must simply state the relevant probabilistic facts such as

$$0.9 :: \mathsf{control}(\mathsf{comp1}, \mathsf{comp2}, \mathsf{comp3}, \mathsf{comp})..$$

and then compute $\underline{\mathbb{P}}(\mathsf{strategic}(\mathsf{comp}))$. In so doing we must go through the set of solutions of an $\mathsf{NP}^{\mathsf{NP}}$-complete problem.

### 4.4. Some comments on interpretation

We have so far discussed our suggested programming style by posing questions such as "What is the probability that there is a three-coloring?" or "What is the probability that a certain node is red?". In the remainder of this section we focus on the interpretation of the lower/upper probabilities obtained in answering such questions.

So, take a PASP program like the three-coloring one (Expression (4)). Although total choices may be associated with non-singleton credal sets, the atom colorable has a unique value per total choice; hence the probability of this atom is sharp. However, probabilities on the color of particular nodes may of course fail to be sharp. What is then the import of $\underline{\mathbb{P}}(\mathsf{red}(2))$? Basically, there is *at least* probability $\underline{\mathbb{P}}(\mathsf{red}(2))$ that node 2 gets red if a three-coloring is selected *after* the uncertainty is resolved (that is, after the atoms associated with probabilities have their values set). Similarly, we have *at most* probability $\overline{\mathbb{P}}(\mathsf{red}(2))$ that node 2 gets red if a three-coloring is selected *after* the uncertainty is resolved. We can thus take lower/upper probabilities as values generated by decisions taken *after* the resolution of uncertainty. This is certainly in contrast to most decision-making models where decisions are made *before* dice are rolled [10].

In fact we may choose to attach a more active role to the agent, supposing that she selects a three-coloring after uncertainty is resolved: the lower probability is the probability if the agent is adversarially working against red in node 2, while the upper probability obtains if the agent works on behalf of red in node 2.

In addition lower/upper probabilities can be viewed as *sharp* probabilities with respect to appropriate (quantified) questions. For instance, if one asks "What is the probability that I will be able to select a three-coloring where node 2 is red?",

[7] Even for acyclic programs we produce some surprises. Consider the simple acyclic program consisting of $0.2 :: \mathsf{r}(\mathsf{a}).$ and $0.8 :: \mathsf{r}(X).$; then $\mathbb{P}(\mathsf{r}(\mathsf{a})) = 0.2 + 0.8 - 0.2 \times 0.8 = 0.84$ in ProbLog (not $\mathbb{P}(\mathsf{r}(\mathsf{a})) = 0.2$ as one might think).

the answer is $\overline{\mathbb{P}}(\mathsf{red}(2))$. But the query is actually asking for $\mathbb{P}$(there is some answer set such that red(2)). Similarly, the question "What is the probability that node 2 will be red in a three-coloring, no matter what I do?" leads to a lower probability but it asks for the precise probability that *all* answer sets have node 2 painted red. That is, PASP indeed lets one formulate probabilities that quantify over answer sets, disguised as lower/upper probabilities.

To conclude, the program that encodes the "probabilistic strategic company" problem also illustrates a situation where the computation of the upper probability $\overline{\mathbb{P}}(\mathsf{strategic(comp)})$ actually answers the question "What is the probability that I will be able to place comp in a strategic set?". While in the three-coloring problem it was necessary to introduce a predicate colorable to determine when a total choice induces a solution (a three-coloring), here determining whether a company is in a strategic set is the problem itself. Consequently, $\overline{\mathbb{P}}(\mathsf{strategic(comp)})$ is also the probability of finding a positive solution to the problem. More explicitly, this latter upper probability yields $\mathbb{P}$(there exists an answer set such that strategic(comp)).

## 5. Complexity and expressivity

In this section we first discuss the computational complexity of inferences with respect to PASP programs (Section 5.1). We then discuss the expressivity of PASP (Section 5.2). The results in this section require quite a bit of background in complexity theory. Necessary material is available in textbooks or related publications [15,80], but to clarify the notation we have collected the relevant notions in the Appendix.

### 5.1. The computational complexity of PASP

As indicated in the examples in the previous section, a PASP program can be used to compute the lower/upper probability of some atom $T$. A more general question would be to compute the lower/upper conditional probability of a conjunction of "target" atoms $T_1, \dots, T_t$, given some "evidence" atoms $E_1, \dots, E_e$. To simplify notation, we denote by $\mathbf{T}$ the conjunction $T_1 \wedge \cdots \wedge T_t$ and by $\mathbf{E}$ the conjunction $E_1 \wedge \cdots \wedge E_e$.

We wish to characterize the computational complexity of computing lower and upper probabilities, $\underline{\mathbb{P}}(\mathbf{T}|\mathbf{E})$ and $\overline{\mathbb{P}}(\mathbf{T}|\mathbf{E})$. To turn this into a decision problem, we study the complexity of deciding whether $\underline{\mathbb{P}}(\mathbf{T}|\mathbf{E}) > \gamma$ and whether $\overline{\mathbb{P}}(\mathbf{T}|\mathbf{E}) > \gamma$. We adopt the convention that *any probability is specified as a rational number*, to avoid difficulties with non-computable real-numbers. We assume that $\gamma$ is *always a rational number* as well. We also assume that the *decision is negative (input is rejected) when* $\underline{\mathbb{P}}(\mathbf{E}) = 0$. Finally, we assume that $\mathbf{T}$ and $\mathbf{E}$, and whatever additional atoms in the input, contain only atoms that use the vocabulary of the program.

Note that in logic programming one distinguishes between *combined* and *data* complexity [102]. Combined complexity focuses on whether an input atom is true or not when the program is also in the input, together with whatever facts are deemed relevant. Data complexity takes the program as fixed and considers as input the atom of interest and the list of relevant facts.[8] In our context we can also differentiate between two questions, one where the program is given as input, another one where it is fixed.

By examining the proof of the next theorem one can see that, as we compute lower and upper probabilities, some other questions can be answered without any additional cost (as subproducts of the whole computation). Thus we consider a large number of possible decisions within our problems.

Our first problem consists of getting, as input, a PASP program, a rational $\gamma$, a pair $(\mathbf{T}, \mathbf{E})$, and a flag indicating one of six decisions of interest: (a) whether $\underline{\mathbb{P}}(\mathbf{T}|\mathbf{E}) > \gamma$; (b) whether $\overline{\mathbb{P}}(\mathbf{T}|\mathbf{E}) > \gamma$; (c) whether $\mathbb{P}(\mathbf{T} \cap \mathbf{E}) > \gamma$ where $\mathbb{P}$ attains $\underline{\mathbb{P}}(\mathbf{T}|\mathbf{E})$; (d) whether $\mathbb{P}(\mathbf{T} \cap \mathbf{E}) > \gamma$ where $\mathbb{P}$ attains $\overline{\mathbb{P}}(\mathbf{T}|\mathbf{E})$; (e) whether $\mathbb{P}(\mathbf{E}) > \gamma$ where $\mathbb{P}$ attains $\underline{\mathbb{P}}(\mathbf{T}|\mathbf{E})$; (f) whether $\mathbb{P}(\mathbf{E}) > \gamma$ where $\mathbb{P}$ attains $\overline{\mathbb{P}}(\mathbf{T}|\mathbf{E})$. We refer to the complexity of this problem as the *inferential complexity*.

Our second problem consists of getting, as input, a rational $\gamma$, a pair $(\mathbf{T}, \mathbf{E})$, and a flag indicating one of the decisions listed in the previous problem (note: the program is fixed!). We refer to this complexity as the *query complexity*.

We can work more closely to concepts employed in logic programming by defining a third problem, consisting of getting, as input, a rational $\gamma$, a set of facts, and a flag indicating one of the same decisions listed in the previous problems (note: the program and the pair $(\mathbf{T}, \mathbf{E})$ are fixed!). We refer to this complexity as the *data complexity*.

We have:

**Theorem 2.** *The inferential complexity of PASP programs with a bound on the arity of predicates is* $\mathsf{PP}^{\Sigma_3^p}$*-complete. Both the query complexity and the data complexity of PASP programs are* $\mathsf{PP}^{\Sigma_2^p}$*-complete.*

**Proof.** The membership of inferential complexity for the specific decision $\underline{\mathbb{P}}(\mathbf{T}|\mathbf{E}) > \gamma$ has been derived previously using a very short argument [69]. Here we present an explicit construction to prove membership of the various decisions we contemplate; that is, we build nondeterministic Turing machines with access to a $\Sigma_3^p$ oracle with input $\gamma$ and $(\mathbf{T}, \mathbf{E})$, and a flag indicating the particular inequality to verify. The construction can in essence be found in the proof of Theorem 25 of Ref. [13], so we shorten several steps.

[8] It is also common to define *program* complexity, where the set of facts is fixed and the program is the input. This sort of complexity seems less relevant here but it may be of interest in future work.

We start by proving membership for the decision $\underline{\mathbb{P}}(\mathbf{T}|\mathbf{E}) > \mu/\nu$ where $\mu$ and $\nu$ are the smallest possible integers such that $\mu/\nu = \gamma$. Start by grounding the probabilistic facts to obtain a polynomial number of probabilistic facts without variables (as the arity of predicates is bounded). Clearly if $\gamma = 1$ the machine must return NO; we assume this case is dealt with at once so that we can assume that $\gamma < 1$ in the subsequent computation steps. Deciding $\underline{\mathbb{P}}(\mathbf{T}|\mathbf{E}) > \mu/\nu$ is equivalent to deciding

$$(\nu - \mu)\underline{\mathbb{P}}(\mathbf{T} \cap \mathbf{E}) > \mu\overline{\mathbb{P}}\left(\mathbf{T}^c \cap \mathbf{E}\right).$$

(This covers all special cases discussed in the proof of Theorem 25 of Ref. [13] that address zero probabilities.) So we focus on the latter inequality. Our machine starts by emulating the selection of a total choice, by creating a transition for each probabilistic fact in the program (we can assume the machine makes transitions with arbitrary probabilities, not just 0.5; in any case, it is possible to emulate such probabilities with a machine that only has transitions with probability 0.5, as discussed in the proof of Theorem 25 of Ref. [13]). Denote by $N$ the total number of computation paths the machine can take out of these transitions (that is, the number of paths that select total choices). By selecting a total choice the machine actually has selected an answer set program. The machine then runs cautious inference on this latter program to determine whether $\mathbf{T}$ and $\mathbf{E}$ hold in every stable model of the answer set program (to do so, the machine must use a $\Pi_3^p$ computation [27, Table 5] that can be produced with the $\Sigma_3^p$ oracle and a negation): if so, the machine moves to a state $q_1$. Otherwise, the machine runs brave inference on the answer set program to determine whether $\mathbf{T}$ does not hold while $\mathbf{E}$ holds in some stable model of the answer set program (to do so, the machine calls the $\Sigma_3^p$ oracle [27, Table 5]): if so, the machine moves to a state $q_2$. In all other cases the machine moves to a state $q_3$. Now denote by $N_i$ the number of computation paths that arrive at $q_i$ (for $i \in \{1, 2, 3\}$). From $q_1$ the machine branches into $\nu - \mu$ computation paths that move immediately to the accepting state; from $q_2$ the machine branches into $\mu$ computation paths that move immediately to the rejecting state; from $q_3$ the machine nondeterministically moves either into the accepting or the rejecting state. The number of accepting paths is larger than the number of rejecting paths if and only if

$$\begin{aligned}(\nu - \mu)N_1 + N_3 > \mu N_2 + N_3 &\Leftrightarrow (\nu - \mu)\frac{N_1}{N} > \mu\frac{N_2}{N} \\ &\Leftrightarrow (\nu - \mu)\underline{\mathbb{P}}(\mathbf{T} \cap \mathbf{E}) > \mu\overline{\mathbb{P}}\left(\mathbf{T}^c \cap \mathbf{E}\right)\end{aligned}$$

where the last equivalence follows from the fact that by construction $N_1/N = \underline{\mathbb{P}}(\mathbf{T} \cap \mathbf{E})$ and $N_2/N = \overline{\mathbb{P}}\left(\mathbf{T}^c \cap \mathbf{E}\right)$. This checks the desired inequality.

Now suppose our task is to decide $\overline{\mathbb{P}}(\mathbf{T}|\mathbf{E}) > \mu/\nu$. We build a machine as in the previous paragraph, with a few changes. The decision is now $(\nu - \mu)\overline{\mathbb{P}}(\mathbf{T} \cap \mathbf{E}) > \mu\underline{\mathbb{P}}\left(\mathbf{T}^c \cap \mathbf{E}\right)$; the machine must run brave inference (on the answer set program specified by the selected total choice) to determine whether $\mathbf{T}$ and $\mathbf{E}$ hold in some stable model of the answer set program (so as to move to $q_1$) and cautious inference (on the answer set program specified by the selected total choice) to determine whether $\mathbf{T}$ does not hold while $\mathbf{E}$ holds (to move to $q_2$). This checks the desired inequality.

Now if the decision problem is $\mathbb{P}(\mathbf{T} \cap \mathbf{E}) > \gamma$ with respect to $\mathbb{P}$ that attains the lower / upper probability of $\mathbf{T}$ given $\mathbf{E}$, then proceed respectively as in the first / second machine in the previous two paragraphs, except that from $q_3$ the machine branches into $\mu$ computation paths that move immediately into the rejecting state. Hence the number of accepting paths is larger than the number of rejecting paths if and only if

$$(\nu - \mu)N_1 > \mu N_2 + \mu N_3 \quad \Leftrightarrow \quad (\nu - \mu)\frac{N_1}{N} > \mu\frac{N - N_1}{N} \quad \Leftrightarrow \quad \frac{N_1}{N} > \frac{\mu}{\nu}$$

as desired.

And if the decision problem is $\mathbb{P}(\mathbf{E}) > \gamma$ with respect to the $\mathbb{P}$ that attains the lower / upper probability of $\mathbf{T}$ given $\mathbf{E}$, then proceed respectively as in the first / second machine in the previous paragraphs, except that from $q_2$ the machine branches into $\nu - \mu$ computation paths that move immediately to the accepting state, and from $q_3$ the machine branches into $\mu$ computation paths that move immediately into to the rejecting state. Hence the number of accepting paths is larger than the number of rejecting paths if and only if

$$\begin{aligned}(\nu - \mu)N_1 + (\nu - \mu)N_2 > \mu N_3 &\Leftrightarrow (\nu - \mu)\frac{N_1 + N_2}{N} > \mu\frac{N - (N_1 + N_2)}{N} \\ &\Leftrightarrow \frac{N_1 + N_2}{N} > \frac{\mu}{\nu}\end{aligned}$$

as desired.

This closes the proof of membership for inferential complexity.

The hardness result for the inferential complexity of probabilistic logic programs with all features of PASP has been derived previously [69].

Membership of query complexity follows from the fact that we can ground the fixed program in polynomial time. We must then run propositional inference. The proof is exactly the same as the proof of membership for inferential complexity,

with the *only* difference that whenever the oracle is called to run cautious or brave inference, now we need only an oracle with power $\Sigma_2^p$ (because cautious and brave inference are respectively $\Sigma_2^p$ and $\Pi_2^p$ complete for propositional answer set programs [27, Table 5]).

To prove hardness of query complexity, consider the $\mathsf{PP}^{\Sigma_2^p}$-complete problem of deciding whether there are more than $K$ assignments of $X_1, \dots, X_m$ that satisfy $\exists Y_1, \dots, Y_n : \forall Z_1, \dots, Z_o : \varphi$, where $\varphi$ is in 3-DNF [104]. That is, $\varphi$ is in Disjunctive Normal Form: it is a disjunction of conjuncts $c_1 \vee c_2 \vee \cdots \vee c_k$ where each conjunct $c_i$ has three literals so that it can be written as $A_i \wedge B_i \wedge C_i$ for literals $A_i$, $B_i$, and $C_i$.

Now introduce a predicate $\mathsf{t}(T, V, P)$ whose meaning is this. When $T$ is equal to 1, then $V$ is the index of a logical variable $X_V$; when $T$ is equal to 2, then $V$ is the index of a logical variable $Y_V$; finally, when $T$ is equal to 3, then $V$ is the index of a logical variable $Z_V$. Now if $P$ is equal to 1, then $\mathsf{t}(T, V, P)$ gets the same value as the logical variable indicated by $(T, V)$; if $P$ is equal to 0, then $\mathsf{t}(T, V, P)$ gets the negated value of the logical variable indicated by $(T, V)$. We could obviously write $\varphi$ as

$$\begin{aligned} &\mathsf{t}(t_{1,1}, v_{1,1}, p_{1,1}) \wedge \mathsf{t}(t_{1,2}, v_{1,2}, p_{1,2}) \wedge \mathsf{t}(t_{1,3}, v_{1,3}, p_{1,3})) \vee \dots \\ &\quad \vee(\mathsf{t}(t_{k,1}, v_{k,1}, p_{k,1}) \wedge \mathsf{t}(t_{k,2}, v_{k,2}, p_{k,2}) \wedge \mathsf{t}(t_{k,3}, v_{k,3}, p_{k,3})), \end{aligned} \tag{8}$$

where we choose the $t_{c,j}$, $v_{c,j}$, and $p_{c,j}$ appropriately.

Introduce another predicate $\mathsf{in}(T, V, C, P)$ whose purpose is to "choose" the right $\mathsf{t}(T, V, P)$ in each conjunct. That is, each element of the $c$th conjunct in Expression (8) is to be written as

$$\mathsf{in}(t_{c,j}, v_{c,j}, c, p_{c,j}) \wedge \mathsf{t}(t_{c,j}, v_{c,j}, p_{c,j}).$$

So we can run through all possible conjuncts, selecting only the appropriate atoms in the conjuncts that belong to $\psi$. This is the strategy employed in the following program:

```
0.5 :: x(V).
0.5 :: in(T, V, C, P).
t(T, V, 0) ∨ t(T, V, 1).
t(1, V, 0) :− not x(V).
t(1, V, 1) :− x(V).
phi :− in(T1, V1, C, P1), t(T1, V1, P1),
          in(T2, V2, C, P2), t(T2, V2, P2),
              in(T3, V3, C, P3), t(T3, V3, P3).
t(3, V, P) :− phi.
negin(T, V, C, P) :− not in(T, V, C, P)..
```

The value of each $X_i$, represented by $\mathsf{x}(i)$, is associated with a probability (in the first line of the program) so as to simulate counting. Note that the third line of this program guarantees that $\mathsf{t}(T, V, P)$ does behave in a binary way with respect to $P$, so that the value of each $Y_i$ is selected by the program; values of $\mathsf{t}(1, V, P)$ are automatically complementary given the fourth and fifth lines of the program, and the value of each $Z_k$ is discussed later.

So, given a formula $\varphi$, insert in $\mathbf{E}$ all facts $\mathsf{in}(1, i, c, 0)$ if $X_i$ appears in the $c$th clause nonnegated, $\mathsf{in}(1, i, c, 1)$ if it appears negated, and similarly for $Y_j$ and $Z_k$. All other groundings of $\mathsf{in}(T, V, C, P)$ are forced not to hold by assembling their corresponding grounding of $\mathsf{negin}(T, V, C, P)$ into $\mathbf{E}$.

The counting question about the quantified formula is true if and only if $\overline{\mathbb{P}}(\mathsf{phi}|\mathbf{E}) > K/2^m$. Note that computation of the upper probability of phi collects the probabilities/proportions of the configurations of $X_i$ such that there is a configuration of $Y_j$ that makes $\varphi$ true for some configuration of $Z_k$ (thus taking care of the existential quantifiers). The universal quantifiers over the $Z_k$ are obtained by replicating a technique often used in disjunctive logic programs (for instance, consider the proof of Theorem 6.3 in Ref. [26]): each configuration of $Z_k$ is associated with a configuration of $\mathsf{t}(3, V, P)$ as follows. If the configuration of $Z_k$ is such that $\varphi$ is false (for a fixed configuration of $X_i$ and $Y_j$), then we can select an interpretation for $\mathsf{t}(3, V, P)$ that mirrors the same assignments (e.g., $\mathsf{t}(3, V, 0)$ is true if $Z_V = 0$), satisfies all rules of the program and makes phi false. However, if no such configuration exists (i.e., if for all $Z_k$ we have that $\varphi$ is true), then any model must contain phi and consequently, by the last rule, assign all $\mathsf{t}(3, V, P)$ to true. By minimality of answer sets, the converse is also true: there is a minimal model that assigns all $\mathsf{t}(3, V, P)$ to true only if there is not a configuration of $Z_k$ that can falsify $\varphi$. Hence phi is true if and only if there is no assignment that makes $\varphi$ false.

Finally, note that the same program can be used to show the data complexity; just remove the probabilistic fact $0.5 :: \mathsf{in}(T, V, C, P)$**.**, the auxiliary predicate $\mathsf{negin}(T, V, C, P)$ and its associated rule $\mathsf{negin}(T, V, C, P) :- \textbf{not}\ \mathsf{in}(T, V, C, P)$**.**, and instead add the necessary groundings of $\mathsf{in}(T, V, C, P)$ as input facts. □

### *5.2. The expressivity of PASP*

There are several ways to assess the expressivity of a programming language. For instance, one may compare programming languages by checking whether they can express the same input-output mappings or not. In fact, there has been quite some work on verifying how to translate programs amongst various probabilistic logic languages [87].

Another strategy to assess expressivity, an "absolute" one, is to determine a class of Turing machines that can be emulated within the programming language of interest. For instance, suppose we have a programming language such that its programs can produce exactly the same input-output mappings as nondeterministic polynomial-time Turing machines; we would like to say that the expressivity of the programming language is given by the complexity class NP [41].

In our setting, a fixed PASP program receives as input a string with $(\mathbf{T}, \mathbf{E})$ and $\gamma$, and a flag indicating a particular inequality to be checked. The output indicates whether or not the inequality is satisfied. Now, can we capture the behavior of a class of Turing machines with such a programming language?

To formalize this question, we might say that a programming language $\mathcal{P}$ *captures* complexity class $C$ exactly when [41]: (1) the query complexity of $\mathcal{P}$ is in $C$; (2) for each set $\mathcal{S}$ of strings encoding inputs, so that this set of strings is a language in $C$, there is a program in $\mathcal{P}$ such that all and only those strings in $\mathcal{S}$ yield a positive output.[9]

We have:

**Theorem 3.** *PASP programs capture* $\mathsf{PP}^{\Sigma_2^p}$.

**Proof.** Query complexity is proved by Theorem 2.

Now suppose we have a set of strings $\mathcal{S}$ in $\mathsf{PP}^{\Sigma_2^p}$; that means there is a polynomial-time probabilistic Turing machine, with an oracle in $\Sigma_2^p$, that can decide whether each string is in $\mathcal{S}$ with error strictly less than half. We must code a PASP program with the same behavior.

Note first that the base probabilistic Turing machine is not required to call a full-fledged Turing machine (itself with another Turing machine as oracle); rather, the base probabilistic Turing machine can use any $\Sigma_2^p$-complete problem as oracle. Thus we can use for instance the decision as to whether $\exists Y_1, \ldots, Y_n : \forall Z_1, \ldots, Z_o : \varphi$ holds, where $\varphi$ is in Disjunctive Normal Form with three literals per conjunct [80].

Moreover, we can assume that the base probabilistic Turing machine calls the oracle *only once* per computation path, using a transformation described by Toran [97]. Toran's transformation does not guarantee that the call to the oracle is the very last operation of the base machine, but we can move the call to the very last computation steps as follows. Build a base probabilistic Turing machine that operates until the point where Toran's transformation would call the oracle. Instead of calling the oracle at that point, make the machine repeat the subsequent operations twice, once assuming the oracle accepts its input, once assuming the oracle rejects its input, and storing the output of each one of these computations in two positions in the tape. Then there are two possibilities. If each computation produces the same answer, then simply return that answer (YES or NO). If however the computations differ, we have two possibilities. First, it may be the case that the output is YES in the path corresponding to the oracle returning YES and NO in the path corresponding to the oracle returning NO. In this case the machine must simply call the oracle at the very end and return the output of the oracle. Second, it may be the case that the output is YES in the path corresponding to the oracle returning NO and NO in the path corresponding to the oracle returning YES. In this case the machine must simply call the oracle with a negated version of the query originally required for the oracle, and then return the output of the oracle. Hence we can focus on a probabilistic Turing machine that makes a unique call to a $\Sigma_2^p$ oracle and returns its output. The whole computation is depicted in Fig. 4.

So, first build a program that reproduces the operations of the base probabilistic Turing machine using a set of random bits (these can be produced using probabilistic facts). A Turing machine can be encoded through facts and rules using for instance the encoding by Marek and Remmel [67]. Note that their encoding makes a few assumptions about the Turing machine; for example, that the machine runs for a precise number of steps that is a (known) polynomial of the size of the input. These assumptions can be easily imposed on our Turing machine. There are however changes that must be made to Marek and Remmel's encoding to account for the oracle. Instead of encoding the oracle explicitly (oracle tape, oracle head, and so on), we benefit from the previous discussion to simplify matters. Thus, when the program reaches the point where both Computations A and B have finished, the program is in one of four possible cases (larger rectangle in Fig. 4). If it is in Case 1, then it sets predicates case1 and output to true. If it is in Case 4, then it sets predicate case4 to true and the predicate output to false.

If the program is in Case 2 or in Case 3, it must respectively set to true predicate case2 or case3. We do not need an explicit oracle tape; rather, we simply use a predicate $\mathsf{in}(T, V, C, P)$ (and a few additional predicates introduced later) to encode the formula processed by the oracle (similarly to the proof of Theorem 2). Thus we assume that an appropriate set of groundings for $\mathsf{in}(T, V, C, P)$ has been fixed (the precise meaning of these groundings is described shortly).

Suppose first that we have reached Case 2; we must return whether or not some formula $\exists Y_1, \ldots, Y_n : \forall Z_1, \ldots, Z_o : \varphi$, where $\varphi$ is in 3-DNF, holds, (in this proof the particular formula $\varphi$ is built by the program and it can depend on the probabilistic facts). We already built rules that handle such a situation in the proof of Theorem 2:

[9] There are differences between showing that a programming language captures a complexity class and showing that its query complexity is complete for that class; the latter requires finding at least one hard query, while the former requires expressing all queries within the language [18].

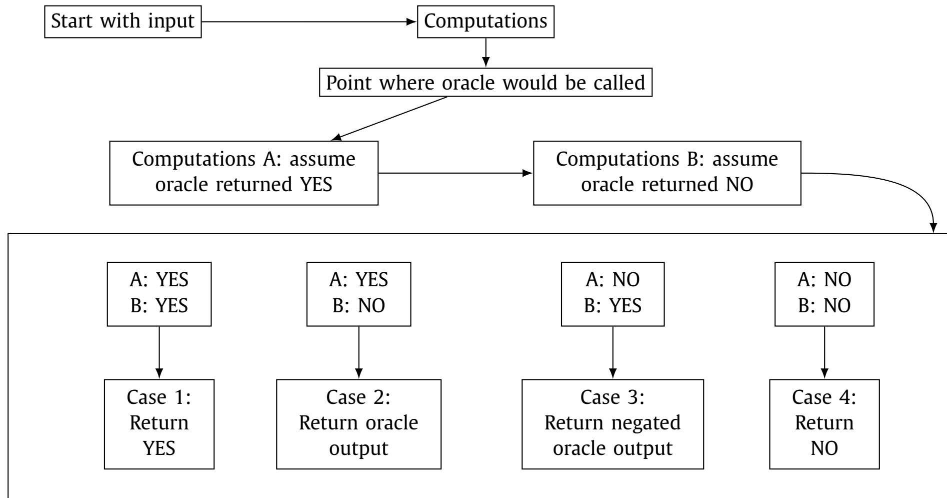



**Fig. 4.** The operations in the Turing machine. Computations A and B are conceptually run in parallel, even though they run sequentially in the machine. After computations B finish, there are four possible situations; two of them actually require calling the oracle.

$$\begin{aligned}
&\mathsf{t}(T,V,0) \vee \mathsf{t}(T,V,1).\\
&\mathsf{phi} :- \mathsf{case2},\\
&\qquad \mathsf{in}(T1,V1,C,P1), \mathsf{t}(T1,V1,P1),\\
&\qquad\quad \mathsf{in}(T2,V2,C,P2), \mathsf{t}(T2,V2,P2),\\
&\qquad\qquad \mathsf{in}(T3,V3,C,P3), \mathsf{t}(T3,V3,P3).\\
&\mathsf{t}(3,V,P) :- \mathsf{case2}, \mathsf{in}(3,V,C,P), \mathsf{phi}.,
\end{aligned} \tag{9}$$

where

- $\mathsf{t}(T,V,P)$ is an auxiliary predicate for which we need to handle only $T=2$ (indicating the $Y_j$) and $T=3$ (indicating the $Z_k$), and possibly $T=1$ (indicating literals associated with probabilistic facts), and
- $\mathsf{in}(T,V,C,P)$ encodes the formula $\varphi$.

The program then closes by moving to the accepting state when phi is true and to the rejecting state otherwise. Thus phi is indeed the output; indeed, we add:

$$\mathsf{output} :- \mathsf{case2}, \mathsf{phi}..$$

Note that if we were to compute the upper probability of output, then in this particular computation path we would obtain $\exists Y_1,\dots,Y_n : \forall Z_1,\dots,Z_o : \varphi$ as desired (because the computation of upper probability forces the inner formula to be true as long as there is a stable model specified by the $Y_1,\dots,Y_n$).

Now suppose we have reached Case 3. Because we need to negate the oracle output, we add:

$$\mathsf{output} :- \mathsf{case3}, \textbf{not}\ \mathsf{phi}..$$

By itself, this will not do: if we compute the upper probability of output, we actually obtain $\exists Y_1,\dots,Y_n : \exists Z_1,\dots,Z_o : \neg\varphi$ in this computation path (instead of $\forall Y_1,\dots,Y_n : \exists Z_1,\dots,Z_o : \neg\varphi$). And if we compute the (unconditional) lower probability of output we will face a mirror problem when case2 holds. What we need is to compute the lower probability in a computation path that reaches Case 3 while computing the upper probability in a computation path that reaches Case 2.

So, we introduce a predicate through the rules:

$$\mathsf{target} :- \mathsf{case1}. \qquad \mathsf{target} :- \mathsf{case2}.$$

and ask the PASP program to decide whether $\mathbb{P}(\mathsf{output}) > \gamma$ where $\mathbb{P}$ is the probability measure that attains $\overline{\mathbb{P}}(\mathsf{target}|\mathsf{output})$. The probability $\mathbb{P}(\mathsf{output})$ for this particular probability measure $\mathbb{P}$ is equal to

$$\mathbb{P}(\mathsf{case1}) + \overline{\mathbb{P}}(\mathsf{case2}, \mathsf{phi}) + \underline{\mathbb{P}}(\mathsf{case3}, \neg\mathsf{phi})$$

given the properties of infinitely monotone Choquet capacities. And this latter expression is exactly the probability that the original Turing machine returns YES. So the behavior of the machine is captured, as desired. □

## 6. Computing upper probabilities

Obviously, a powerful programming language is only useful if its programs can be run in reasonable time. Given the expressivity of PASP, we cannot expect every PASP program to run quickly. But we must at least discuss how to exploit problem descriptions to speed up the calculation of lower/upper probabilities.

A stratified nondisjunctive program specifies a single probability distribution and any query is answered by a sharp probability value. As the computation of probabilities for stratified programs has been explored through a variety of counting techniques [4,31,89], we do not discuss stratified programs in any detail.

We focus on lower/upper probabilities that arise in connection with general (non-stratified, disjunctive) programs. The techniques we develop resort to algorithms that turn propositional logic programs into classical formulas (it should be noted that such algorithms have been refined in particular for nondisjunctive programs).

The first observation we make is that conditional lower/upper probabilities can be easily calculated from unconditional lower/upper probabilities, due to properties of the credal semantics discussed after Theorem 1. In the remainder of this section we focus on the computation of the upper probability $\overline{\mathbb{P}}(T_1, \ldots, T_t)$, where each $T_i$ is a literal. The computation of lower probabilities follows similar lines.

As input we have a PASP program and literals $\{T_i\}$. Our strategy is:

1. First ground the program.
2. Then the resulting propositional PASP program is turned into a (possibly long) propositional formula, using a conversion algorithm (there are relatively efficient ones for non-disjunctive programs [47] and more complex ones for disjunctive programs [61]).
3. We assume the propositional formula obtained in the previous step is turned to Conjunctive Normal Form. All literals that appear in probabilistic facts must be left in the formula; at the end the propositions that are associated with those facts must be marked with the corresponding probabilities. Even small programs can produce large propositional classical formulas through this step and the previous one.
4. Finally run an adapted solution counting algorithm that computes the desired probabilities. Note that we must count solutions with respect to some propositions while others are used to determine satisfiability; this is a prototypical problem in $\mathsf{NP}^{\mathsf{NP}}$.[10]

To illustrate these steps, consider here a simplified example. Suppose we have the program in Expression (4) together with probabilistic facts in Fig. 3. Take one of the constraints:

$$:-\ \mathsf{edge}(X,Y), \mathsf{red}(X), \mathsf{red}(Y)..$$

By grounding this constraint, we obtain propositional constraints

$$:-\ \mathsf{edge}(1,2), \mathsf{red}(1), \mathsf{red}(2).,$$
$$:-\ \mathsf{edge}(1,3), \mathsf{red}(1), \mathsf{red}(3).,$$

and so on. Each one of these constraints can be translated to a clause; for instance, the first propositional constraint imposes

$$\neg\,(\mathsf{edge}(1,2) \wedge \mathsf{red}(1) \wedge \mathsf{red}(2))$$

and therefore the clause

$$(\neg \mathsf{edge}(1,2) \vee \neg \mathsf{red}(1) \vee \neg \mathsf{red}(2))\,.$$

Now consider the rule

$$\mathsf{red}(X) \vee \mathsf{green}(X) \vee \mathsf{blue}(X) :-\ \mathsf{node}(X)..$$

Consider its grounding

$$\mathsf{red}(1) \vee \mathsf{green}(1) \vee \mathsf{blue}(1) :-\ \mathsf{node}(1).;$$

this is partially translated to clause

$$(\mathsf{red}(1) \vee \mathsf{green}(1) \vee \mathsf{blue}(1) \vee \neg \mathsf{node}(1))\,;$$

this clause does not capture the complete meaning of the disjunctive rule as minimality must be imposed through additional clauses.

[10] Of course we are not reducing the general PASP inference to a problem in $\mathsf{NP}^{\mathsf{NP}}$, because the propositional formula generated in the third step may be exponentially large in the worst case. A guarantee of polynomial size for the formula would imply a collapse of the counting hierarchy, an unlikely event.

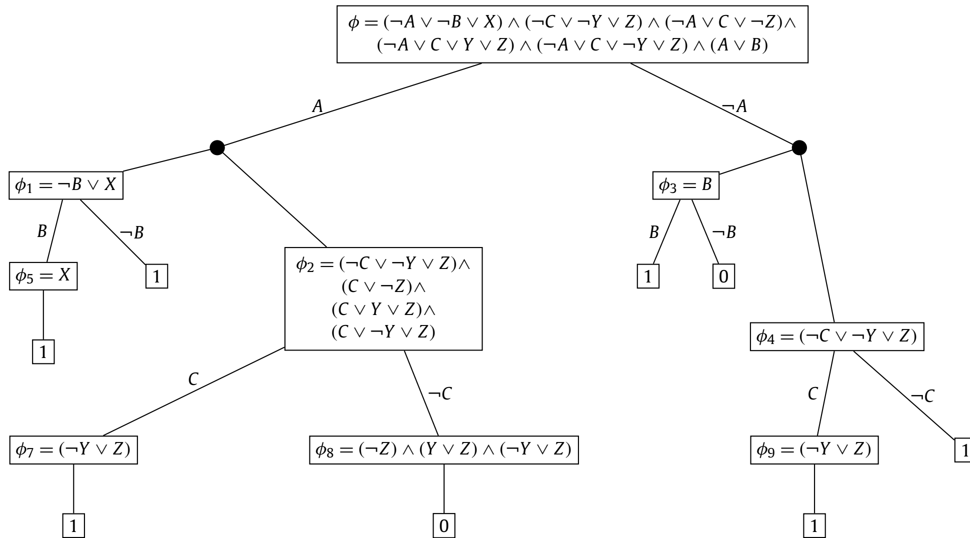



**Fig. 5.** The counting example from Aziz et al. [3].

Now consider the fourth step of the method, where counting is in fact run. The counting problem for $\mathsf{NP}^{\mathsf{NP}}$ problems has received relatively little attention in the literature, but some clever algorithms have been proposed [3,4]. Our contribution in this paper is to adapt one particular algorithm; in the remainder of this section we do so.

We thus have a CNF formula where some propositions are associated with probabilities; we call these the *priority* propositions. The remaining propositions are referred to as *non-priority* ones. We must count the satisfying assignments for priority propositions (non-priority propositions are set as needed). Such a counting problem is solved by the dSharpP algorithm by Aziz et al. [3].

The dSharpP algorithm modifies the celebrated DPLL algorithm. The latter algorithm is, in essence, rather simple: suppose we wish to check the satisfiability of formula $\phi$; then select a proposition $C$; simplify $\phi$ as if $C$ were true, and recurse until it is possible to prove that the new formula is satisfiable or not; and simplify $\phi$ as if $C$ were false, and recurse until it is possible to prove that the new formula is satisfiable or not. The DPLL algorithm implicitly builds a tree that branches on the selected propositions; there are many techniques to select propositions, to detect as early as possible when to stop recursing, to store useful information, and to exploit problem decompositions [7]. Algorithms that count satisfying assignments of CNF formulas are often based on the same sort of computing tree, but instead of exploring the tree only until a satisfying assignment is found, the algorithms must go through the whole tree, explicitly or implicitly.

The basic idea in the dSharpP algorithm is to build an implicit tree as the DPLL algorithm, but to select non-priority propositions only when there are no priority ones in the formula at hand. The counts associated with two distinct assignments of the same priority proposition are added, and at the end the number of satisfying assignments (for the priority propositions) are obtained. Two techniques are used to speed up the whole process. First, the algorithm must detect as early as possible when the formula can be satisfied; when this happens and some priority propositions remain unassigned, the algorithm computes in closed-form the number of assignments for those propositions (a simple calculation: $2^K$ assignments if there are $K$ unassigned propositions). Second, it may be the case that a formula in CNF may be divided into several sub-formulas, each one of them a conjunction of clauses, such that propositions in one sub-formula do not appear in other sub-formulas. In this case the number of satisfying assignments for the original formula is the product of the number of satisfying assignments for each one of the sub-formulas.

To illustrate this process, we use a simple example inspired by a discussion in Ref. [3]. Consider the formula $\phi$ in Fig. 5, and suppose the goal is to count the number of assignments of $A$, $B$ and $C$ (the remaining propositions are non-priority ones). Only branching on these priority propositions is shown; when a formula contains only non-priority propositions, the branching required to determine satisfiability is not shown. The counts obtained by branching are the numbers shown in the leaves. Note that when $A$ is applied to formula $\phi$, the result is a formula with two "disjoint" sub-formulas $\phi_1$ and $\phi_2$; the conjunction of these sub-formulas is represented by the left dot. And when $\neg A$ is applied to $\phi$, "disjoint" sub-formulas $\phi_3$ and $\phi_4$ are obtained; their conjunction is represented by the right dot. Thus $\phi_5$ "gets" 1, and $\phi_1$ gets 2; similarly, $\phi_7$ gets 1 and $\phi_8$ gets 0 (at $\phi_8$ is not satisfiable), so $\phi_2$ gets 1. The left dot gets 2 and 1, and multiplies them, "sending" 2 to $\phi$. By the same procedure, the right dot sends 2 to $\phi$; at $\phi$ we obtain $2 + 2 = 4$ satisfying assignments for priority propositions.

To adapt this algorithm to our setting, note that in PASP probabilities are directly associated with atoms. Thus we do not have to pursue a sophisticated encoding of probability values (as needed for instance when running Bayesian networks inference through counting methods [9,19]). Instead we just have to take into account that each priority proposition is associated with a probability, and all of them are stochastically independent. What happens then is that any leaf node sends a 1 or a 0 up (from the satisfiability of a corresponding sub-formula). These numbers are then sent up across the edges. When a number is sent from a sub-formula to another one through an edge labeled with a literal, the number is

multiplied by the probability of that literal. Also, the numbers must be added at non-leaf nodes containing sub-formulas, and they must be multiplied at dots that represent separation into components. It should be noted that in our setting we do not need to multiply these numbers by any factor when there are unassigned priority propositions: any such proposition would lead to branches containing a number and 1 minus that number; as the numbers in these branches are to be added, the effect of the unassigned proposition is just a factor 1.

To understand the algorithm, consider again Fig. 5. Suppose that the priority propositions are actually generated from three probabilistic facts in a PASP program:

$$\alpha :: A., \qquad \beta :: B., \qquad \gamma :: C..$$

Node $\phi_1$ gets $\beta$ and $1-\beta$; it adds both and sends 1 to the left dot. Node $\phi_2$ gets $\gamma$ from its left child and 0 from its right child; thus it sends $\gamma$ to the left dot. Similarly, the right dot gets $\beta$ from $\phi_3$ and 1 from $\phi_4$. Consequently $\phi$ gets $\alpha\gamma$ from the left dot and $(1-\alpha)\beta$ from the right dot; it adds both numbers thus producing the correct upper probability that $\phi$ is satisfied: $\alpha\gamma + (1-\alpha)\beta$.

## 7. Conclusion

Hopefully the reader is convinced that PASP offers an elegant way to code probabilistic questions. Most of the existing literature on PASP focuses on acyclic or definite or stratified programs that are already quite powerful as they can capture Bayesian networks and their relational variants, and even introduce recursive behavior. General probabilistic logic programs, with disjunctive heads and non-stratified behavior, have been left aside, often viewed as pathological entities devoid of semantics. The point of this paper is that, once a proper credal semantics is given, cyclic programs provide concise encodings for probabilistic combinatorial problems. The semantics is actually straightforward and mathematically simple as it is based on the well-known theory of two-monotone Choquet capacities. Many applications can be considered; just as an example, one may be interested in the robustness of planning policies to uncertainty in initial conditions, with planning problems encoded through ASP [3].

A more detailed study of expressivity should be pursued in future work. We have considered a particular notion of "capture" in our analysis of the expressivity of PASP, but additional analysis may be able to find other interesting results.

The real challenge ahead is to build PASP solvers that can take on practical problems. The scheme we have outlined in this paper is certainly a first step, aimed at demonstrating the basic operations that must still be refined in future work. Our scheme has three steps: grounding, conversion to satisfiability, and (adapted) counting techniques. Each one of these steps deserves further analysis:

1. Many ASP solvers selectively ground rules and facts [48]. In ASP, not all grounded rules and facts are needed in a particular calculation; finding exactly the needed ones is the goal of a grounder. Future work should consider techniques that selectively ground PASP programs.
2. There are several ASP solvers that are based on conversion to propositional satisfiability, but usually they work by constructing formulas gradually, introducing clauses only as needed — this is important because the size of the whole propositional formula may be exponential on the input program [62]. Similar techniques should be examined for PASP in future work.
3. The adapted counting algorithm we have presented should be refined in future work. There are many techniques used in DPLL that should be tested in the context of PASP. And there are entirely different counting techniques that could be appropriate in various scenarios [3] and that deserve attention.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgements

The first author has been partially supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), grant 312180/2018-7 (Pq). The second author has been partially supported by CNPq, grant 304012/2019-0 (Pq). The work was also supported by the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), grants 2016/18841-0 and 2019/07665-4, and also by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - finance code 001.

## Appendix A. A bit of computational complexity theory

The reader can find a more detailed discussion about computational complexity theory in Ref. [80].

A complexity class is a set of languages; a language is a set of strings; a string is a sequence of 0s and 1s.

A (nondeterministic) Turing machine is a tuple $(Q, \Sigma, q_0, q_a, q_r, \delta)$ associated with a device containing a tape with symbols and a head that can move over the tape and read/write symbols, where $Q$ is a set of states, with $q_0/q_a/q_r$ respectively

the initial/accepting/rejecting states; $\Sigma$ is the alphabet that includes 0, 1, and blank, with blank never present in the input; and $\delta$ is the transition function that takes a pair in $Q \times \Sigma$ consisting of the current state and the current symbol in the tape, are returns a subset of $Q \times \Sigma \times \{-1, 0, 1\}$, where each element of this subset indicates the next state, the next symbol to be written in the tape, and the movement of the head ($-1$ means left, 1 means right, 0 means no motion). If the machine reaches $q_a$, the input is accepted; if the machine reaches $q_r$, the input is rejected. A language is decided by a Turing machine if the machine accepts all strings in the language and rejects all strings not in the language. If acceptance/rejection happens within a number of steps that is polynomial in the size of the input, the machine is said to be *polynomial-time*. The well-known complexity class NP consists of those languages that can be decided by polynomial-time (nondeterministic) Turing machines. Now if the transition function $\delta$ is modified such that there is a uniform probability distribution over the set of triples produced for each pair state/symbol, then we obtain a *probabilistic* Turing machine. The complexity class PP is the set of languages that are decided by a probabilistic Turing machine in a number of steps polynomial in the size of the input, with an error probability strictly less than half for all input strings. Equivalently, PP is the class of languages satisfying the following property: there is a polynomial-time Turing machine such that a string is in the language if and only if more than half of the computation paths of the machine end up in the accepting state (all other paths end up in the rejecting state).

If a Turing machine has a second pair tape/head, such that it can write a string to the secondary tape and then, in a single step, it can read whether the string is in a language $\mathcal{L}$ or not, then the machine is said to have an oracle for $\mathcal{L}$. The class of languages that can be decided by machines defining complexity class $C$, with each machine allowed to use an additional oracle for $\mathcal{L}$, is denoted by $C^{\mathcal{L}}$. If $D$ is another complexity class, then $C^D$ is defined to be $\bigcup_{\mathcal{L} \in D} C^{\mathcal{L}}$. The polynomial hierarchy contains classes $\Sigma^p_{k+1} = \mathsf{NP}^{\Sigma^p_k}$ for $k > 1$, with $\Sigma^p_1 = \mathsf{NP}$. Wagner's counting hierarchy contains classes PP and $\mathsf{PP}^{\Sigma^p_k}$ for $k > 0$ [104].

A polynomial-time many-one reduction from language $\mathcal{L}$ to language $\mathcal{L}'$ is an algorithm that, in a number of steps that is polynomial in the size of the input, transforms a string $\ell \in \mathcal{L}$ into a string $\ell' \in \mathcal{L}'$ such that $\ell \in \mathcal{L}$ if and only if $\ell' \in \mathcal{L}'$. If there is such a reduction from any language in a complexity class $C$ to a language $\mathcal{L}$, then $\mathcal{L}$ is $C$-hard. If $\mathcal{L}$ is in $C$ and is $C$-hard, then $\mathcal{L}$ is $C$-complete.

## References


[1] Alessandro Antonucci, Alessandro Facchini, A credal extension of independent choice logic, in: International Conference on Scalable Uncertainty Management, 2018, pp. 35–49.
[2] Thomas Augustin, Frank P.A. Coolen, Gert de Cooman, Matthias C.M. Troffaes, Introduction to Imprecise Probabilities, Wiley, 2014.
[3] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, Peter Stuckey, #∃SAT: projected model counting, in: International Conference on Theory and Applications of Satisfiability Testing, 2015, pp. 121–137.
[4] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, Peter Stuckey, Stable model counting and its application in probabilistic logic programming, in: AAAI, 2015, pp. 3468–3474.
[5] Fahiem Bacchus, Using first-order probability logic for the construction of Bayesian networks, in: Conference on Uncertainty in Artificial Intelligence, 1993, pp. 219–226.
[6] Chitta Baral, Michael Gelfond, Nelson Rushton, Probabilistic reasoning with answer sets, Theory Pract. Log. Program. 9 (1) (2009) 57–144.
[7] Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsh, Handbook of Satisfiability, IOS Press, 2009.
[8] Ísmail Ílkan Ceylan, Thomas Lukasiewicz, Rafael Peñaloza, Complexity results for probabilistic Datalog$^{\pm}$, in: European Conference on Artificial Intelligence, 2016, pp. 1414–1422.
[9] Mark Chavira, Adnan Darwiche, On probabilistic inference by weighted model counting, Artif. Intell. 172 (6–7) (2008) 772–799.
[10] Robert T. Clemen, Making Hard Decisions: An Introduction to Decision Analysis, Duxbury Press, 1997.
[11] Vítor Santos Costa, David Page, Maleeha Qazi, James Cussens, CLP($\mathcal{BN}$): constraint logic programming for probabilistic knowledge, in: Uffe Kjaerulff, Christopher Meek (Eds.), Conference on Uncertainty in Artificial Intelligence, San Francisco, California, Morgan-Kaufmann, 2003, pp. 517–524.
[12] Fabio G. Cozman, Languages for probabilistic modeling over structured and relational domains, in: A Guided Tour of Artificial Intelligence Research Volume 2, 2020, pp. 247–283, chapter 9.
[13] Fabio G. Cozman, Denis D. Mauá, On the semantics and complexity of probabilistic logic programs, J. Artif. Intell. Res. 60 (2017) 221–262.
[14] Fabio G. Cozman, Denis D. Mauá, The complexity of Bayesian networks specified by propositional and relational languages, Artif. Intell. 262 (2018) 96–141.
[15] Fabio G. Cozman, Denis D. Mauá, The finite model theory of Bayesian network specifications: descriptive complexity and zero/one laws, Int. J. Approx. Reason. 110 (2019) 107–126.
[16] Paulo C.G. da Costa, Kathryn B. Laskey, Of Klingons and starships: Bayesian logic for the 23rd century, in: Conference on Uncertainty in Artificial Intelligence, 2005.
[17] E. Dantsin, Probabilistic logic programs and their semantics, in: Proceedings of the First Russian Conference on Logic Programming, Springer, 1990, pp. 152–164.
[18] Evgeny Dantsin, Thomas Eiter, Andrei Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (3) (2001) 374–425.
[19] Adnan Darwiche, Modeling and Reasoning with Bayesian Networks, Cambridge University Press, 2009.
[20] Luc De Raedt, Logical and Relational Learning, Springer, 2008.
[21] Luc De Raedt, Paolo Frasconi, Kristian Kersting, Stephen Muggleton, Probabilistic Inductive Logic Programming, Springer, 2008.
[22] Alex Dekhtyar, Michael Dekhtyar, The theory of interval probabilistic programs, Ann. Math. Artif. Intell. 55 (2009) 355–388.
[23] Alex Dekhtyar, V.S. Subrahmanian, Hybrid probabilistic programs, J. Log. Program. 43 (3) (2000) 187–250.
[24] Heinz-Dieter Ebbinghaus, J. Flum, Finite Model Theory, Springer-Verlag, 1995.
[25] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, Declarative problem-solving using the DLV system, in: Logic-Based Artificial Intelligence, Springer, 2000, pp. 79–103.
[26] Thomas Eiter, Georg Gottlob, Heikki Mannila, Disjunctive datalog, ACM Trans. Database Syst. 22 (3) (1997) 364–418.
[27] Thomas Eiter, Wolfgang Faber, Michael Fink, Stefan Woltran, Complexity results for answer set programming with bounded predicate arities and implications, Ann. Math. Artif. Intell. 51 (2007) 123–165.

[28] Thomas Eiter, Giovambattista Ianni, Thomas Krennwalner, Answer set programming: a primer, in: Reasoning Web: Semantic Technologies for Information Systems, 2009.
[29] Wolfgang Faber, Gerald Pfeifer, Nicola Leone, Semantics and complexity of recursive aggregates in answer set programming, Artif. Intell. 175 (2011) 278–298.
[30] D. Fierens, H. Blockeel, M. Bruynooghe, J. Ramon, Logical Bayesian networks and their relation to other probabilistic logical models, in: Conference on Inductive Logic Pogramming, 2005, pp. 121–135.
[31] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shrerionov, Bernd Gutmann, Gerda Janssens, Luc De Raedt, Inference and learning in probabilistic logic programs using weighted Boolean formulas, Theory Pract. Log. Program. 15 (3) (2014) 358–401.
[32] Nir Friedman, Lise Getoor, Daphne Koller, A. Pfeffer, Learning probabilistic relational models, in: International Joint Conference on Artificial Intelligence, 1999, pp. 1300–1309.
[33] Norbert Fuhr, Probabilistic Datalog – a logic for powerful retrieval methods, in: Conference on Research and Development in Information Retrieval, Seattle, Washington, 1995, pp. 282–290.
[34] Michael Gelfond, Vladimir Lifschitz, The stable model semantics for logic programming, in: International Logic Programming Conference and Symposium, vol. 88, 1988, pp. 1070–1080.
[35] Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, Ben Taskar, Probabilistic relational models, in: Introduction to Statistical Relational Learning, MIT Press, 2007.
[36] Lise Getoor, Ben Taskar, Introduction to Statistical Relational Learning, MIT Press, 2007.
[37] Walter R. Gilks, Andrew Thomas, David Spiegelhalter, A language and program for complex Bayesian modelling, Statistician 43 (1993) 169–178.
[38] Sabine Glesner, Daphne Koller, Constructing flexible dynamic belief networks from first-order probabilistic knowledge bases, in: Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 1995, pp. 217–226.
[39] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, Sriram K. Rajmani, Probabilistic programming, in: Future of Software Engineering, ACM, 2014, pp. 167–181.
[40] J. Gordon, E.H. Shortliffe, The Dempster-Shafer theory of evidence, in: E.H. Shortliffe, B.G. Buchanan (Eds.), Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project, Addison-Wesley, 1984, pp. 272–292, chapter 13.
[41] Erich Grädel, Finite model theory and descriptive complexity, in: Finite Model Theory and Its Applications, Springer, 2007, pp. 125–229.
[42] Ulrich Guntzer, Werner Kiessling, Helmut Thone, New directions for uncertainty reasoning in deductive databases, in: ACM SIGMOD Conference, 1991, pp. 178–187.
[43] Spyros Hadjichristodoulou, David S. Warren, Probabilistic logic programming with well-founded negation, in: International Symposium on Multiple-Valued Logic, 2012, pp. 232–237.
[44] David Heckerman, Christopher Meek, Daphne Koller, Probabilistic entity-relationship models, PRMs, and plate models, in: L. Getoor, B. Taskar (Eds.), Introduction to Statistical Relational Learning, MIT Press, 2007, pp. 201–238.
[45] Michael C. Horsch, David Poole, A dynamic approach to probabilistic inference using Bayesian networks, in: Conference of Uncertainty in Artificial Intelligence, 1990, pp. 155–161.
[46] Manfred Jaeger, Complex probabilistic modeling with recursive relational Bayesian networks, Ann. Math. Artif. Intell. 32 (2001) 179–220.
[47] Tomi Janhunen, Representing normal programs with clauses, in: European Conference on Artificial Intelligence, ECAI, 2004, pp. 358–362.
[48] Tomi Janhunen, Ilkka Niemela, The answer set programming paradigm, AI Mag. 37 (3) (2016) 13–24.
[49] Kristian Kersting, Lifted probabilistic inference, in: L. De Raedt, C. Bessiere, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, P. Lucas (Eds.), European Conference on Artificial Intelligence, IOS Press, 2012.
[50] Kristian Kersting, Luc De Raedt, Stefan Kramer, Interpreting Bayesian logic programs, in: AAAI-2000 Workshop on Learning Statistical Models from Relational Data, 2000.
[51] Werner Kiessling, Helmut Thone, Ulrich Guntzer, Database support for problematic knowledge, in: EDBT, in: LNCS, vol. 580, Springer, 1992, pp. 421–436.
[52] M. Kifer, A. Li, On the semantics of rule-based expert systems with uncertainty, in: ICDT, in: Lecture Notes in Computer Science, Springer Verlag, 1988, pp. 102–117.
[53] M. Kifer, V.S. Subrahmanyan, Theory of generalized annotated logic programming and its applications, J. Log. Program. 12 (4) (1992) 335–367.
[54] Daphne Koller, Probabilistic relational models, in: Inductive Logic Programming, in: LNCS, vol. 1634, Springer, 1999, pp. 3–13.
[55] Robert Kowalski, History of logic programming, in: Joerg Siekmann (Ed.), Computational Logic, Elsevier, 2014, pp. 523–569.
[56] Lars V.S. Lakshmanan, Fereidoon Sadri, Probabilistic deductive databases, in: Symposium on Logic Programming, 1994, pp. 254–268.
[57] Steffen L. Lauritzen, David J. Spiegelhalter, Local computations with probabilities on graphical structures and their application to expert systems, J. R. Stat. Soc. B 50 (2) (1988) 157–224.
[58] Joohyung Lee, Yi Wang, A probabilistic extension of the stable model semantics, in: AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning, 2015, pp. 96–102.
[59] Joohyung Lee, Zhun Yang, $LP^{MLN}$ weak constraints, and P-log, in: AAAI Conference on Artificial Intelligence, 2017, pp. 1170–1177.
[60] Isaac Levi, The Enterprise of Knowledge, MIT Press, Cambridge, Massachusetts, 1980.
[61] Y. Lierler, Disjunctive Answer Set Programming via satisfiability, in: Answer Set Programming, vol. 142, CEUR Workshop Proceedings, 2005.
[62] Vladimir Lifschitz, Alexander Razborov, Why are there so many loop formulas?, ACM Trans. Comput. Log. 7 (2) (2006) 261–268.
[63] Thomas Lukasiewicz, Probabilistic logic programming with conditional constraints, ACM Trans. Comput. Log. 2 (3) (2001) 289–339.
[64] Thomas Lukasiewicz, Probabilistic description logic programs, in: European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU 2005, Springer, Barcelona, Spain, July 2005, pp. 737–749.
[65] Thomas Lukasiewicz, Probabilistic description logic programs, Int. J. Approx. Reason. 45 (2) (2007) 288–307.
[66] Suzanne Mahoney, K.B. Laskey, Network engineering for complex belief networks, in: Conference on Uncertainty in Artificial Intelligence, 1996.
[67] Victor Marek, Jeffrey B. Remmel, On the expressibility of stable logic programming, Theory Pract. Log. Program. 3 (4) (2003) 551–567.
[68] Victor Marek, Miroslaw Truszczynski, Stable models and an alternative logic programming paradigm, in: The Logic Programming Paradigm: A 25-Year Perspective, Springer Verlag, 1999, pp. 375–398.
[69] Denis Deratani Mauá, Fabio Gagliardi Cozman, Complexity results for probabilistic answer set programming, Int. J. Approx. Reason. 118 (2020) 133–154.
[70] Steffen Michels, Arjen Hommersom, Peter J.F. Lucas, Marina Velikova, A new probabilistic constraint logic programming language based on a generalised distribution semantics, Artif. Intell. J. 228 (2015) 1–44.
[71] Brian Milch, Stuart Russell, First-order probabilistic languages: into the unknown, in: Int. Conference on Inductive Logic Programming, 2007.
[72] Jack Minker, Overview of disjunctive logic programming, Ann. Math. Artif. Intell. 12 (1994) 1–24.
[73] Ilya Molchanov, Theory of Random Sets, Springer, 2005.
[74] Matthias Nickles, Alessandra Mileo, A system for probabilistic inductive answer set programming, in: International Conference on Scalable Uncertainty Management, 2015, pp. 99–105.
[75] Matthias Nickles, A tool for probabilistic reasoning based on logic programming and first-order theories under stable model semantics, in: European Conference on Logics in Artificial Intelligence, JELIA, 2016, pp. 369–384.

[76] Raymond Ng, V.S. Subrahmanian, Probabilistic logic programming, Inf. Comput. 101 (2) (1992) 150–201.
[77] Liem Ngo, Peter Haddawy, Answering queries from context-sensitive probabilistic knowledge bases, Theor. Comput. Sci. 171 (1–2) (1997) 147–177.
[78] Ilkka Niemela, Logic programs with stable model semantics as a constraint programming paradigm, Ann. Math. Artif. Intell. 25 (1999) 241–273.
[79] Nils J. Nilsson, Probabilistic logic, Artif. Intell. 28 (1986) 71–87.
[80] Christos H. Papadimitriou, Computational Complexity, Addison-Wesley Publishing, 1994.
[81] David Poole, Representing Bayesian networks within Probabilistic Horn Abduction, in: Conference on Uncertainty in Artificial Intelligence, 1991, pp. 271–278.
[82] David Poole, Probabilistic Horn abduction and Bayesian networks, Artif. Intell. 64 (1993) 81–129.
[83] David Poole, The Independent Choice Logic for modelling multiple agents under uncertainty, Artif. Intell. 94 (1/2) (1997) 7–56.
[84] David Poole, The Independent Choice Logic and beyond, in: Luc De Raedt, Paolo Frasconi, Kristian Kersting, Stephen Muggleton (Eds.), Probabilistic Inductive Logic Programming, in: Lecture Notes in Computer Science, vol. 4911, Springer, 2008, pp. 222–243.
[85] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, David Poole, Statistical Relational Artificial Intelligence: Logic, Probability, and Computation, Morgan & Claypool Publishers, 2016.
[86] Fabrizio Riguzzi, The distribution semantics is well-defined for all normal programs, in: Fabrizio Riguzzi, Joost Vennekens (Eds.), International Workshop on Probabilistic Logic Programming, in: CEUR Workshop Proceedings, vol. 1413, 2015, pp. 69–84.
[87] Fabrizio Riguzzi, Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning, River Publishers, 2018.
[88] Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, A history of probabilistic inductive logic programming, Front. Robot. AI 1 (2014) 1–5.
[89] Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, Evelina Lamma, A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics, Int. J. Approx. Reason. 80 (2017) 313–333.
[90] Taisuke Sato, A statistical learning method for logic programs with distribution semantics, in: Conference on Logic Programming, 1995, pp. 715–729.
[91] Taisuke Sato, Yoshitaka Kameya, Parameter learning of logic programs for symbolic-statistical modeling, J. Artif. Intell. Res. 15 (2001) 391–454.
[92] Taisuke Sato, Yoshitaka Kameya, Neng-Fa Zhou, Generative modeling with failure in PRISM, in: International Joint Conference on Artificial Intelligence, 2005, pp. 847–852.
[93] Ehud Y. Shapiro, Logic programs with uncertainties: a tool for implementing rule-based systems, in: International Joint Conference on Artificial Intelligence, 1983, pp. 529–532.
[94] E.H. Shortliffe, B.G. Buchanan, Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project, Addison-Wesley, Reading, Mass, 1984.
[95] V.S. Subrahmanian, Amalgamating knowledge bases, ACM Trans. Database Syst. 19 (2) (1994) 291–331.
[96] Helmut Thone, Ulrich Guntzer, Werner Kießling, Increased robustness of Bayesian networks through probability intervals, Int. J. Approx. Reason. 17 (1997) 37–76.
[97] Jacobo Tóran, Complexity classes defined by counting quantifiers, J. ACM 38 (3) (1991) 753–774.
[98] Calin Rares Turliuc, Luke Dickens, Alessandra Russo, Krysia Broda, Probabilistic abductive logic programming using Dirichlet priors, Int. J. Approx. Reason. 78 (2016) 223–240.
[99] Guy Van den Broeck, Dan Suciu, Query processing on probabilistic: a survey, Found. Trends® Databases (2017) 197–341.
[100] M.H. Van Emden, Quantitative deduction and its fixpoint theory, J. Log. Program. 3 (1) (1986) 37–53.
[101] Allen Van Gelder, Kenneth A. Ross, John S. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (3) (1991) 620–650.
[102] Moshe Y. Vardi, The complexity of relational query languages, in: Annual ACM Symposium on Theory of Computing, 1982, pp. 137–146.
[103] Joost Vennekens, Sofie Verbaeten, Maurice Bruynooghe, Logic programs with annotated disjunctions, in: Logic Programming - ICLP, in: LNCS, vol. 3132, 2004, pp. 431–445.
[104] Klaus W. Wagner, The complexity of combinatorial problems with succinct input representation, Acta Inform. 23 (1986) 325–356.
[105] Hui Wan, Michael Kifer, Belief logic programming: uncertainty reasoning with correlation of evidence, in: International Conference on Logic Programming and Nonmonotonic Reasoning, 2009, pp. 316–328.
[106] M.P. Wellman, J.S. Breese, R.P. Goldman, From knowledge bases to decision models, Knowl. Eng. Rev. 7 (1) (1992) 35–53.