Proceedings of Machine Learning Research 103:91–101, 2019 ISIPTA 2019

The Joy of Probabilistic Answer Set Programming

Fabio Gagliardi Cozman FGCOZMAN@USP.BR
Escola Politécnica – Universidade de São Paulo, Brazil

Abstract

Probabilistic answer set programming (PASP) combines rules, facts, and independent probabilistic facts. Often one restricts such programs so that every query yields a sharp probability value. The purpose of this paper is to argue that a very useful modeling language is obtained by adopting a particular credal semantics for PASP, where one associates with each consistent program a credal set. We examine the basic properties of PASP and present an algorithm to compute (upper) probabilities given a program.

Keywords: Logic programming, Answer set programming, Probabilistic programming, Credal sets.

1. Introduction

The purpose of this paper is to show that Probabilistic Answer Set Programming (PASP) actually offers an elegant and enjoyable modeling language. To do so, we review, and modify slightly, the credal semantics of probabilistic logic programming, and we use it to concoct a programming style where one can ask questions about probability distributions satisfying sets of constraints.

The credal semantics is based on the stable model semantics. About twenty years ago it became clear that the stable model semantics for logic programs also offered an attractive programming paradigm. A similar perspective is proposed in this paper: instead of looking at the credal semantics of PASP merely as a way to lend meaning to pathological cyclic programs, here the credal semantics is viewed as a probabilistic programming paradigm that goes beyond existing modeling languages.

Section 2 reviews some needed concepts, and Section 3 presents the basic syntax and semantics of PASP. Sections 4 and 5 detail the semantics and discuss the proposed programming paradigm. Section 6 presents an algorithm that returns (upper) probabilities given a PASP program.

2. Background: Answer Set Programming

Answer Set Programming (ASP) is a programming paradigm that emerged from research in logic programming; we here review basic syntactic and semantic notions [18].

First: a *literal* is either an *atom* $r(t_1,\dots,t_k)$ 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*; that is, the logical value of the expression is given by usual Boolean negation).

Syntactically, an ASP program is a set of *rules* such as

$$H_1 \vee \cdots \vee H_m \;:-\; B_1,\dots,B_n\textbf{.},$$

where $H_1 \vee \cdots \vee H_m$ is the *head* and $B_1,\dots,B_n$ is the *body*, and where each H_i is a literal and each *subgoal* B_j can be either a literal A or a literal A preceded by **not** (that is, **not** A). Here is a rule:

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

meaning that if some individual X is a node that is not known to be barred, then X is red or green or blue. If a rule is such that $m=1$, it is said to be *nondisjunctive*; a nondisjunctive rule with $n=0$ is called a *fact*, and instead of writing $H :- \textbf{.}$, we just write $H\textbf{.}$. A program that is nondisjunctive (it only contains nondisjunctive rules) and contains no **not** and no $\neg$ is said to be *definite*.

A *propositional atom* is an atom without logical variables. The *Herbrand base* of a program is the set of propositional literals (propositional 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 a literal and its strong negation). A propositional literal is true/false with respect to an interpretation when it is/isn't in the interpretation. Similarly, a propositional subgoal A, where A is a literal, is true/false with respect to an interpretation when A is/isnt't in the interpretation, while **not** A is true/false when A isn't/is in the interpretation. A rule is *satisfied* by an interpretation iff 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. A *model* of a program is an interpretation that satisfies all the rules of the program. A model $\mathscr{I}$ of a program is *minimal* iff there exists no model $\mathscr{J}$ of the program such that $\mathscr{J} \subset \mathscr{I}$.

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 [12].

The *stable model* semantics is based on reducts, defined as follows. Given a program $\mathbf{P}$ and an interpretation $\mathscr{I}$, their reduct $\mathbf{P}^{\mathscr{I}}$ is the propositional program obtained by

© 2019 F.G. Cozman.

first removing all grounded rules with **not** A in their body and $A \in \mathscr{I}$, and then by removing each subgoal **not** A from all remaning grounded rules. A stable model of **P** is an interpretation $\mathscr{I}$ that is a minimal model of the reduct $\mathbf{P}^{\mathscr{I}}$. 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 $A :-$ **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.

Originally proposed for nondisjunctive programs [22], the stable model semantics was later extended to general programs and stable models were renamed as *answer sets*. Answer sets were then used to propose a new programming paradigm [37, 40], 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 [30]. An example should illustrate the idea.

Suppose we are given a graph, encoded by a predicate node and a predicate edge (respectively unary and binary predicates). Suppose we assign to each node a color, as specified by the following rule:

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

Suppose also that no two adjacent nodes can be red. We might use the rule:

$$\mathsf{auxiliar} :- \textbf{not}\ \mathsf{auxiliar}, \mathsf{edge}(X,Y), \mathsf{red}(X), \mathsf{red}(Y)..$$

This rule forces auxiliar to be false *and* the remainder of the body to be false. Usually such a *constraint* is written simply as follows:

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

We might then force the colors to be assigned so as to generate a three-coloring (that is, no adjacent nodes have the same color) by adding two more constraints:

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

Given a set of atoms that define a graph, say $\mathsf{node}(n1)., \mathsf{node}(n2)., \dots, \mathsf{edge}(n1,n2)., \dots,$ any answer set for the latter 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.

Other features of popular ASP packages are *aggregates* [19] and numeric domains. We avoid these features here as they would take us too far.

3. PASP: Probabilistic Answer Set Programming

Combinations of logic programming and probabilities have been pursued for some time [21, 34, 39, 42, 49]. Many different proposals can be found in the literature [7, 45, 15, 14, 46]; several recent surveys cover very relevant material [5, 47, 27]. A popular scheme is “Sato’s distribution semantics”; both Sato’s original proposal [49] and Poole’s similar Probabilistic Horn Abduction [42] focused on particular cases (definite/acyclic programs), and they have been greatly extended through the years [43, 44, 50, 51]. The basic idea is to allow for *probabilistic facts*.

A probabilistic fact consists of a fact A. that is associated with a probability α, assumed to be a rational number in $[0,1]$. The syntax of the ProbLog package [20] for a such a probabilistic fact is

$$\alpha :: A..$$

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

Probabilistic Answer Set Programming (PASP) deals with programs consisting of probabilistic facts, facts, and rules. Sato’s distribution semantics adopts the following interpretation for a grounded probabilistic fact $\alpha :: A.$:
• with probability α, the fact A. is kept together with all other facs and rules;
• with probability $1-\alpha$, the fact is simply removed from the program.
This semantics induces a probability distribution over logical programs. However, a different semantics for probabilistic facts is adopted in this paper: here we add A. to the program with probability α, and we add $\neg A$. to the program with probability $1-\alpha$ (if necessary, $\neg\neg A$ is replaced by A). We discuss the reason for departing from Sato’s semantics in Section 4.

So, take the set of probabilistic facts of the program of interest, and ground them as needed to obtain a set of probabilistic propositional facts $\mathscr{F} = \{\alpha_i :: A_i\}_{i \in I}$. A *total choice* θ is a subset of these latter probabilistic propositional facts. Once we collect the facts in θ and the strongly negated facts $\mathscr{F} \backslash \theta$, and add them to the facts and rules of the original program, we obtain an ASP program. So, if we have N probabilistic propositional facts, we have 2^N total choices, and we thus have 2^N 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: probabilistic propositional facts

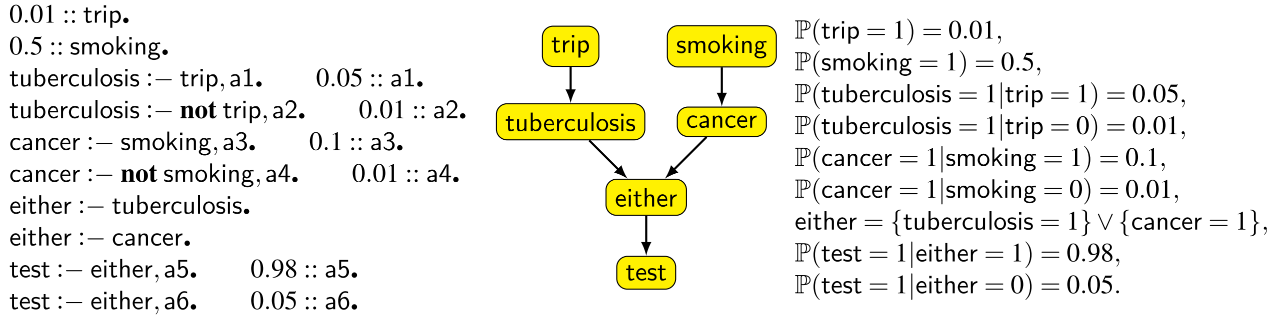

Figure 1: Left: An acyclic PASP program, based on a popular Bayesian network [31]. Middle: the acyclic graph of the program; also the structure of the corresponding Bayesian network. Right: the probability assessments specified by the program (they are the probabilities for the Bayesian network).

are supposed stochastically independence, hence

$$\mathbb{P}(\theta) = \left(\prod_{j':(\alpha_{j'}::A_{j'})\in\theta} \alpha_{j'} \right) \left(\prod_{j'':(\alpha_{j''}::A_{j''})\notin\theta} (1-\alpha_{j''}) \right). \quad (1)$$

It is worth noting that there are other ways to combine ASP and probabilities in the literature. The semantics of P-log assumes a uniform distribution over answer sets given the (equivalent of a) total choice [3], while the language LP^{MLN} resorts to Markov logic to distribute probabilities over answer sets. Other proposals introduce three-valued atoms and various logical devices so as to mix answer sets and probabilities [6, 52]. Another relevant proposal, by Michels et al [38], also resorts to credal sets, but not in the same way we do here. These semantics deserve further treatment in future work.

4. The Joy of PASP

In this section we show how PASP can be used to represent many nontrivial probabilistic problems. We start with nondisjunctive acyclic programs, then look at nondisjunctive stratified ones, and then face the general case.

4.1. Nondisjunctive Acyclic Programs

The short nondisjunctive PASP program in Figure 1 (left) is *acyclic*: intuitively, no atom depends on itself. To present a more formal definition, it is necessary to define the *dependency graph* of the program: this is a graph where each grounded (propositional) 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 graph drawn in Figure 1 (middle) is the dependency graph of the program in the same figure. An acyclic program is a program with an acyclic dependency graph.

Because it is acyclic, the propositional program in Figure 1 (left) is quite easy to read, as its probabilistic facts and rules translate directly into probabilities. The probabilities can be seen in Figure 1 (right); note that here we conflate atoms and the random variables they correspond to (random variables that are defined over the space of all interpretations).

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 [44]. In fact, the program in Figure 1 has been generated from a well-known Bayesian network [31].

There are several specification languages that can compactly describe Bayesian network over repetitive domains, by parameterizing random variables and resorting to various forms of quantification. For instance, the language of *plates*, introduced with the BUGS package [25], uses simple geometric figures to specify parameterized random variables that are associated with template probability distributions. Similarly, Probabilistic Relational Models resort to diagrams containing classes and template probability distributions [11, 23], and several textual languages use elements of functional programming to code repetitive probabilistic structures [26, 28]

Acyclic programs can be used to specify “relational Bayesian networks” as well. For instance, here is an acyclic PASP program that captures part of the well-known University World [24], where we have courses, students, and

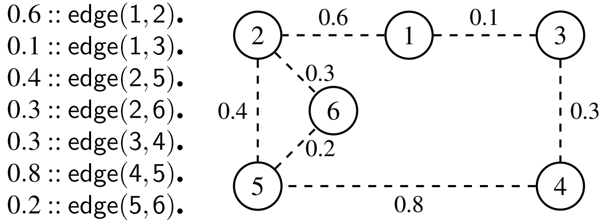

Figure 2: A random undirected graph.

grades:

```
apt(X) :− student(X), a1.        0.7 :: a1.
easy(Y) :− course(X), a2.        0.4 :: a2.
pass(X,Y) :− student(X), apt(X), course(Y), easy(Y).
pass(X,Y) :− student(X), apt(X),
             course(Y), not easy(Y), a3.        0.8 :: a3..
```

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* — that is, if there is no cycle in the dependency graph that contains a negative edge (that is, an edge created by a subgoal containin **not**). A nondisjunctive stratified program has a unique minimal model [12]; consequently any total choice induces a unique model and therefore the probabilistic program induces a unique probability distribution over interpretations. Due to this, most of the literature on probabilistic logic program is restricted to this class of programs [20].

Here is a prototypical example of nondisjunctive stratified program:

```
edge(X,Y) :− edge(Y,X).
path(X,Y) :− edge(X,Y).
path(X,Y) :− edge(X,Z), path(Z,Y)..
```

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 Figure 2 (the drawing depicts a visual representation of the random graph, with edges annotated with probabiliies), then $\mathbb{P}(\mathsf{path}(1,4)) = 0.2330016$.

As illustrated by this example, stratified PASP programs can encode recursion. This is a feature that cannot be reproduced with Probabilistic Relational Models based on first-order logic, as recursion goes beyond the resources of first-order logic [16].

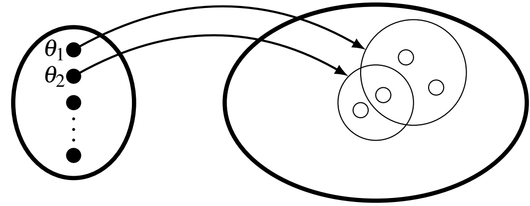

Figure 3: The credal semantics. The set of total choices is shown to the left; there is single distribution over them. Each total choice defines a program that maps to a set of answer sets, shown to the right: total choice θ_1 maps to three answer sets, θ_2 to two answer sets, and so on.

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 [6], or perhaps resort to some three-valued semantics that can accommodate such failures. We leave such strategies to future work.

The more interesting case is the one in which some total choices lead to many answer sets. Note that a total choice θ is associated with a probability $\mathbb{P}(\theta)$; the individual answer sets induced by θ collectively get mass $\mathbb{P}(\theta)$, but no other stipulations are present. One possibility is to distribute $\mathbb{P}(\theta)$ over the answer sets in some prescribed way (for instance, P-log adopts a uniform distribution over answer sets [3]). Another possibility is to consider the set of all possible probability distributions that can be generated by distributing $\mathbb{P}(\theta)$ over the answer sets. This strategy, first proposed by Lukasiewicz [35, 36], is the one we adopt. Because the semantics is then a set of probability distributions, we refer to it as the *credal semantics*, as sets of probability distributions with epistemic import are called *credal sets* [32] (Lukasiewicz referred to his proposal simply as "stable model semantics"). Figure 3 shows the structure of the credal semantics.

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

$$\begin{array}{l} \mathsf{red}(X) \vee \mathsf{green}(X) \vee \mathsf{blue}(X) \;\text{:-}\; \mathsf{node}(X). \\ \mathsf{edge}(X,Y) \;\text{:-}\; \mathsf{edge}(Y,X). \\ \text{:-}\; \mathsf{edge}(X,Y), \mathsf{red}(X), \mathsf{red}(Y). \\ \text{:-}\; \mathsf{edge}(X,Y), \mathsf{green}(X), \mathsf{green}(Y). \\ \text{:-}\; \mathsf{edge}(X,Y), \mathsf{blue}(X), \mathsf{blue}(Y). \end{array} \qquad (2)$$

plus the probabilistic facts in Figure 2 and the facts

$$\mathsf{red}(1).,\qquad \mathsf{green}(4).,\qquad \mathsf{green}(6).. \qquad (3)$$

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 θ, the probability mass $\mathbb{P}(\theta)$ can be attached to each one of the answer sets induced by θ. Take for instance node 3. If both edges to 1 and 4 are present (with probability 0.03), then all answer sets must contain $\mathsf{blue}(3)$. And if both edges are absent, then there are answer sets containing either $\mathsf{red}(3)$ or $\mathsf{blue}(3)$ or $\mathsf{green}(3)$. Other configurations ensue if only one edge is absent.

Under the credal semantics we cannot necessarily associate each propositional 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 distributions; likewise, the upper probability is the supremum over these probability values.

Thus 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 Figure 2 and hard facts in Expression (3), we also have the probabilistic fact

$$0.2 :: \mathsf{blue}(5).. \qquad (4)$$

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 $\mathsf{blue}(5)$ is set to hold. To have a consistent program, we must work differently. Consider the PASP program:

$$\begin{array}{l} \mathsf{red}(X) \vee \mathsf{green}(X) \vee \mathsf{blue}(X) \;\text{:-}\; \mathsf{node}(X). \\ \mathsf{edge}(X,Y) \;\text{:-}\; \mathsf{edge}(Y,X). \\ \neg\mathsf{colorable} \;\text{:-}\; \mathsf{edge}(X,Y), \mathsf{red}(X), \mathsf{red}(Y). \\ \neg\mathsf{colorable} \;\text{:-}\; \mathsf{edge}(X,Y), \mathsf{green}(X), \mathsf{green}(Y). \\ \neg\mathsf{colorable} \;\text{:-}\; \mathsf{edge}(X,Y), \mathsf{blue}(X), \mathsf{blue}(Y). \\ \mathsf{red}(X) \;\text{:-}\; \neg\mathsf{colorable}, \mathsf{node}(X), \mathbf{not}\ \neg\mathsf{red}(X). \\ \mathsf{green}(X) \;\text{:-}\; \neg\mathsf{colorable}, \mathsf{node}(X), \mathbf{not}\ \neg\mathsf{green}(X). \\ \mathsf{blue}(X) \;\text{:-}\; \neg\mathsf{colorable}, \mathsf{node}(X), \mathbf{not}\ \neg\mathsf{blue}(X).. \end{array} \qquad (5)$$

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 iff 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 Sato's semantics takes probabilistic fact $\alpha :: A.$ to mean that we should impose $A.$ with probability α and discard it with probability $1-\alpha$. Our approach is different in that $A.$ 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 why we propose this departure from Sato's proposal, take again the three-coloring program in Expression (5) with probabilistic facts in Expressions (3) and (4). 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 adopt Sato'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 Sato's semantics may be manageable in acyclic programs, where the effect of probabilistic facts is relatively easy to grasp.[1] 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. It may be sometimes inconvenient to get inconsistency, but we find that it is a small price to pay to avoid the perplexing behavior of Sato's semantics.

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

1. Even for acyclic programs may raise difficulties. 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 Sato's semantics (not $\mathbb{P}(\mathsf{r}(\mathsf{a})) = 0.2$ as one might think).

Suppose we have a collection of companies $\mathscr{C} = \{c_1, \ldots, c_m\}$, 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 owed by three other companies, as specified by $\mathsf{control}(c_{i_1}, c_{i_2}, c_{i_3}, c)$ (a company may be controlled by more than one triple). A *strategic set* $\mathscr{C}'$ of companies is a minimal subset of $\mathscr{C}$ (minimal with respect to inclusion) such that: 1) the set of all products manufactured by $\mathscr{C}$ is identical to the set of all products manufactured by $\mathscr{C}'$; 2) if the three controlling companies of a company c is in $\mathscr{C}'$, then c is in $\mathscr{C}'$. The question is, given a company c, is it in some strategic set? This problem is $\mathsf{NP}^{\mathsf{NP}}$-complete,[2] 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 [17]:

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

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

$$0.88 :: \mathsf{control}(\mathsf{comp1}, \mathsf{comp2}, \mathsf{comp3}, \mathsf{comp})\textbf{.}.$$

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.

5. Some Comments on Interpretation

We have so far posed questions as computational exercises: What is the probability that there is a three-coloring? That this 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 (2)). Although total choices may be associated with non-singleton credal sets, the atom $\mathsf{colorable}$ 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 [9].

In fact we may choose to attach a more active role to the agent, supposing that it 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. There is then an important point to make: lower/upper probabilities can be viewed as *sharp* probabilities with respect to appropriate questions. If one asks, "What is the probability that I will be able to select a three-ordering where node 2 is red?", the answer is exactly $\overline{\mathbb{P}}(\mathsf{red}(2))$. Note that the query is actually $\mathbb{P}$(there is answer set such that $\mathsf{red}(2)$). 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 as it asks whether for the probability that all answer sets have node 2 painted red.

Thus 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 $\overline{\mathbb{P}}(\mathsf{strategic}(\mathsf{comp}))$ 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 $\mathsf{colorable}$ 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, $\overline{\mathbb{P}}(\mathsf{strategic}(\mathsf{comp}))$ is also the probability of finding a positive solution to the problem (that is, $\mathbb{P}$(there exists an answer set such that $\mathsf{strategic}(\mathsf{comp})$).

6. Computing (Lower/Upper) Probabilities

Obviously, a powerful programming language is only useful if its instances can be run in reasonable time. As ASP can encode problems that are $\mathsf{NP}^{\mathsf{NP}}$-complete, we cannot expect every PASP program to run quickly. But we must at least know how to exploit problem descriptions to speed up the calculation of lower/upper probabilities.

Note that a stratified nondisjunctive programs specifies a single probability distribution and any query is answered by a sharp probability value. The computation of probabil-

2. That is, it is a representative of those computational problems whose solution requires a nondeterministic Turing machine with a polynomial time bound *and* access to an oracle that is also a nondeterministic Turing machine with a polynomial time bound [41].

ities for stratified programs has been explored through a variety of counting techniques [2, 20, 47]. We thus focus on lower/upper probabilities that arise in connection with general 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 that we now investigate. This is due to the following result, alluded to before [10] in connection with Sato's semantics for probabilistic facts:

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.*

To sketch the proof of this result, refer to Figure 3: 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. The importance of Theorem 1 is that we immediately obtain expressions for lower/upper conditional probabilities. We have, for events $\mathscr{A}$ and $\mathscr{B}$:

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

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

Thus in the remainder of this section we focus on the computation of $\overline{\mathbb{P}}(A_1,\ldots,A_n)$, where each A_i is a literal. It should be clear that the computation of lower probabilities follows similar lines.

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

1. First ground the program.

2. Then the resulting propositional PASP program is turned into a (possibly long) propositional formula. The transformation of an ASP program into a propositional formula in Conjunctive Normal Form (CNF) is well-known [29]; we assume that such an algorithm is run.[3] All literals that appear in probabilistic facts must be left in the program; at the end the propositions that are associated with those facts must be marked with the corresponding probabilities.

3. Finally run an adapted solution counting algorithm that computes the desired probabilities. The counting problem for $\mathsf{NP}^{\mathsf{NP}}$ problems has received relatively little attention in the literature, but some clever algorithms have been proposed [1, 2]. 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. [1]. 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 ϕ; then select a proposition C; simplify ϕ as if C were true, and recurse until it is possible to prove that the new formula is satisfiable or not; and simplify ϕ 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 [4]. 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-fomulas, 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, Aziz et al. [1] use a simple example that we reproduce in Figure 4. The goal is to count the number of assignments of A, B and C for the original formula ϕ. Only branching on these priority propositions is shown; when a formula contains only non-priority propositions, the branching required to determine satisfiability

3. Propositional formulas in Conjunctive Normal Form are written as conjunctions of clauses, where a clause is a disjunction of literals. A package that generates a CNF formula out of an ASP program an be found at research.ics.aalto.fi/software/asp.

is not shown. The counts obtained by branching are the numbers shown in the leaves. Note that when A is applied to formula ϕ, the result is a formula with two "disjoint" sub-formulas ϕ_1 and ϕ_2; the conjunction of these sub-formulas is represented by the left dot. And when $\neg A$ is applied to ϕ, "disjoint" sub-formulas ϕ_3 and ϕ_4 are obtained; their conjunction is represented by the right dot. Thus ϕ_5 "gets" 1, and ϕ_1 gets 2; similarly, ϕ_7 gets 1 and ϕ_8 gets 0 (at ϕ_8 is not satisfiable), so ϕ_2 gets 1. The left dot gets 2 and 1, and multiplies them, "sending" 2 to ϕ. By the same procedure, the right dot sends 2 to ϕ; at ϕ 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, such as done for Bayesian networks [8, 13]. 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 mutiply 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 Figure 4. 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 ϕ_1 gets β and $1-\beta$; it adds both and sends 1 to the left dot. Node ϕ_2 gets γ from its left child and 0 from its right child; thus it sends γ to the left dot. Similarly, the right dot gets β from ϕ_3 and 1 from ϕ_4. Consequently ϕ 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 ϕ 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 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 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 [1].

The real challenge ahead is to build PASP solvers that can take on practical problems. The scheme we have outlined in this paper 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 [30]. 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 [33]. 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 [1] and that deserve attention.

Acknowledgments

The author has been partially supported by the Conselho Nacional de Desenvolvimento Cientifico e Tecnológico (CNPq), grant 312180/2018-7. The work was also supported by the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), grant 2016/18841-0, and also by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior (CAPES) - finance code 001. The work was conducted at the Center for AI and Machine Learning (CIAAM) with funding by Itaú Unibanco.

References

[1] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. #∃SAT: Projected model counting. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 121–137, 2015.

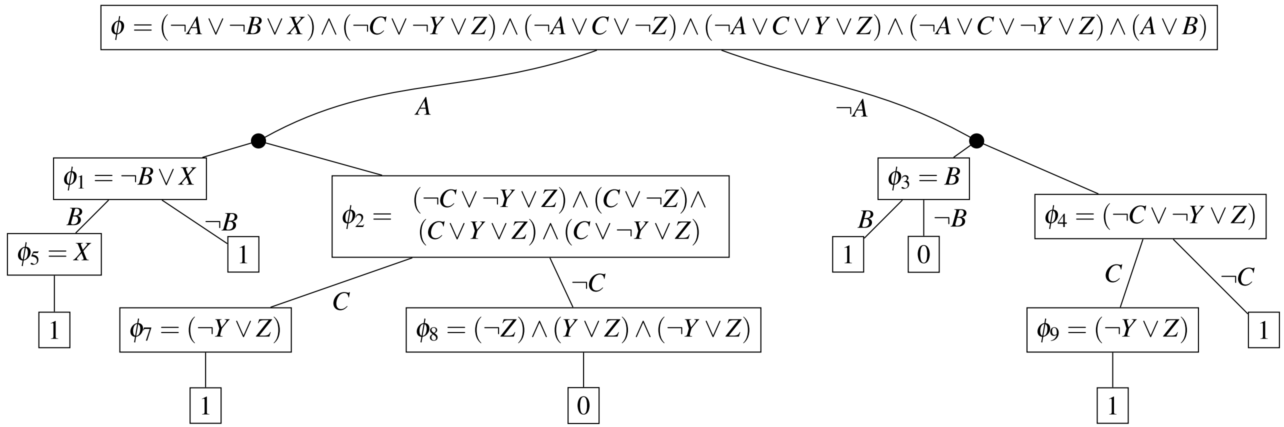

Figure 4: The counting example from Aziz et al. [1].

[2] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. Stable model counting and its application in probabilistic logic programming. In *AAAI*, pages 3468–3474, 2015.

[3] Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming*, 9(1):57–144, 2009.

[4] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[5] Stefan Borgwardt, Ismail Ilkan Ceylan, and Thomas Lukasiewicz. Recent advances in querying probabilistic knowledge bases. *International Joint Conference on Artificial Intelligence*, pages 5420–5426, 2018.

[6] Ísmail Ílkan Ceylan, Thomas Lukasiewicz, and Rafael Peñaloza. Complexity results for probabilistic Datalog$^{\pm}$. In *European Conference on Artificial Intelligence*, pages 1414–1422, 2016.

[7] Ísmail Ílkan Ceylan, and Rafael Peñaloza. The Bayesian Ontology Language BEL. *Journal of Automated Reasoning*, 58(1): 67–95 (2017)

[8] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

[9] Robert T. Clemen. *Making Hard Decisions: An Introduction to Decision Analysis*. Duxbury Press, 1997.

[10] Fabio G. Cozman and Denis D. Mauá. On the semantics and complexity of probabilistic logic programs. *Journal of Artificial Intelligence Research*, 60:221–262, 2017.

[11] Fabio G. Cozman and Denis D. Mauá. The complexity of Bayesian networks specified by propositional and relational languages. *Artificial Intelligence*, 262:96–141, 2018.

[12] Evgeny Dantsin, Thomas Eiter, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[13] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[14] Luc De Raedt. *Logical and Relational Learning*. Springer, 2008.

[15] Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton. *Probabilistic Inductive Logic Programming*. Springer, 2008.

[16] Heinz-Dieter Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1995.

[17] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the DLV system. In *Logic-based Artificial Intelligence*, pages 79–103. Springer, 2000.

[18] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwalner. Answer set programming: a primer. In *Reasoning Web: Semantic Technologies for Information Systems*. 2009.

[19] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175: 278–298, 2011.

[20] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shrerionov, Bernd Gutmann, Gerda Janssens,

and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2014.

[21] Norbert Fuhr. Probabilistic Datalog – a logic for powerful retrieval methods. In *Conference on Research and Development in Information Retrieval*, pages 282–290, Seattle, Washington, 1995.

[22] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *International Logic Programming Conference and Symposium*, volume 88, pages 1070–1080, 1988.

[23] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007. ISBN 0262072882.

[24] Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, and Ben Taskar. Probabilistic relational models. In *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[25] Walter R. Gilks, Andrew Thomas, and David Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43:169–178, 1993.

[26] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajmani. Probabilistic programming. In *Future of Software Engineering*, pages 167–181. ACM, 2014. doi:10.1145/2593882.2593900.

[27] Christian Theil Have, and Riccardo Zese. Probabilistic logic programming. *International Journal of Approximate Reasoning*, 106:88, 2019.

[28] Manfred Jaeger. Complex probabilistic modeling with recursive relational Bayesian networks. *Annals of Mathematics and Artificial Intelligence*, 32:179–220, 2001.

[29] Tomi Janhunen. Representing normal programs with clauses. In *European Conference on Artificial Intelligence (ECAI)*, page 358—362, 2004.

[30] Tomi Janhunen and Ilkka Niemelä. The answer set programming paradigm. *AI Magazine*, 37(3):13–24, 2016.

[31] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal Royal Statistical Society B*, 50(2):157–224, 1988.

[32] Isaac Levi. *The Enterprise of Knowledge*. MIT Press, Cambridge, Massachusetts, 1980.

[33] Vladimir Lifschitz and Alexander Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.

[34] Thomas Lukasiewicz. Probabilistic logic programming. In *European Conference on Artificial Intelligence*, pages 388–392, 1998.

[35] Thomas Lukasiewicz. Probabilistic description logic programs. In *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU 2005)*, pages 737–749, Barcelona, Spain, July 2005. Springer.

[36] Thomas Lukasiewicz. Probabilistic description logic programs. *International Journal of Approximate Reasoning*, 45(2):288–307, 2007.

[37] Victor Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, page 375–398. Springer Verlag, 1999.

[38] Steffen Michels, Arjen Hommersom, Peter J. F. Lucas, and Marina Velikova. A new probabilistic constraint logic programming language based on a generalised distribution semantics. *Artificial Intelligence Journal*, 228:1–44, 2015.

[39] Raymond Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[40] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25: 241–273, 1999.

[41] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing, 1994.

[42] David Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

[43] David Poole. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1/2):7–56, 1997.

[44] David Poole. The Independent Choice Logic and beyond. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming*, volume 4911 of *Lecture Notes in Computer Science*, pages 222–243. Springer, 2008.

[45] Gian Luca Pozzato. Typicalities and probabilities of exceptions in nonmotonic description logics. *International Journal of Approximate Reasoning*, 107:81–100, 2019.

[46] Fabrizio Riguzzi. The distribution semantics is well-defined for all normal programs. In Fabrizio Riguzzi and Joost Vennekens, editors, *International Workshop on Probabilistic Logic Programming*, volume 1413 of *CEUR Workshop Proceedings*, pages 69–84, 2015.

[47] Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Evelina Lamma. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *International Journal of Approximate Reasoning*, 80:313–333, 2017.

[48] Fabrizio Riguzzi, Jan Wielemaker, and Riccardo Zese. Probabilistic inference in SWI-Prolog. *PLP at ILP*, pages 15–27, 2018.

[49] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Conference on Logic Programming*, pages 715–729, 1995.

[50] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15: 391–454, 2001.

[51] Taisuke Sato, Yoshitaka Kameya, and Neng-Fa Zhou. Generative modeling with failure in PRISM. In *International Joint Conference on Artificial Intelligence*, pages 847–852, 2005.

[52] Joost Vennekens, Marc Denecker, and Maurice Bruynoogue. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3):245–308, 2009.