asymptoticplp: Approximating probabilistic logic programs on large domains

Bao Loi Quach[1], Felix Q. Weitkämper[2,*]

[1]Ludwig-Maximilians-Universität München, Oettingenstr. 67, 80538 München, Deutschland

Abstract

Probabilistic logic programs are logic programs in which some of the clauses are annotated with probabilistic facts. The behaviour of relations in these clauses can be very complex, leading to scalability issues. Asymptotic representations, in which queries are completely independent of the domain size and which approximate a probabilistic logic program on large domains, allow us to gain an understanding on how a probabilistic logic programs will behave for increasing domain sizes, and can be computed without actually having to execute the logic program. In particular, every probabilistic logic program under the distribution semantics is asymptotically equivalent to an acyclic probabilistic logic program consisting only of determinate clauses over probabilistic facts. We present asymptoticplp, a Prolog implementation of an algorithm which computes this. The transformation proceeds in several, modular steps which are of independent interest. These steps include rewriting the probabilistic logic program to a formula of least fixed point logic and then applying asymptotic quantifier elimination on the formula. Quantifier-free first-order formulas are then rewritten as acyclic determinate stratified DATALOG formulas, which together with the original probabilistic facts form a (probabilistic) logic program.

Keywords

Probabilistic logic programming, Quantifier elimination, Datalog, Least Fixed Point Logic, SWI-Prolog

1. Introduction

The asymptotic theory of finite structures has been studied for more than half a century, beginning with the 1969 work of Glebskii et al. [1], who showed that as the size of domains under consideration increases, every sentence of first-order logic without constants is eventually either almost surely true or almost surely false. In fact, Glebskii et al. showed more: they demonstrated that every formula of first-order logic, including those with constants and free varables, is almost surely equivalent to one without quantifiers. This result was extended by Blass et al. in 1985 [2], who showed that it remains true even if we allow special quantifiers to compute fixed points of formulas.

In the succeeding years, least fixed point logic as investigated by Blass et al. was related to logic programs, and in 1991 Abiteboul and Vianu [3] demonstrated a correspondence between stratified Datalog programs and formulas in least fixed point logic.

Probabilistic logic programming under the distribution semantics, established independently

PLP 2023: 9th Workshop on Probabilistic Logic Programming, July 09th, 2023, London, UK

*Corresponding author.

felix.weitkaemper@lmu.de (F. Q. Weitkämper)

0000-0002-3895-8279 (F. Q. Weitkämper)

in the early 1990s by Poole [4] and Sato [5], has blossomed since then into an independent research discipline and a promising paradigm, useful in probabilistic programming for allowing recursion, in artificial intelligence as a modelling language in the presence of uncertainty and in machine learning for combining statistical learning methods with inductive logic programming.

By viewing a probabilistic logic program as a deterministic Datalog program over independent probabilistic facts, the distribution semantics allows the rich theory developed by logicians over decades of studying finite models to be brought to bear on the analysis of probabilistic logic programs. In particular, the second author [6, Theorem 27] recently showed that the analysis of Blass et al. [2] is directly applicable to probabilistic logic programs. In the language of probabilistic logic programming this means that on domains of increasing size, every probabilistic logic program is asymptotically equivalent to one in which all clauses are determinate, that is, all atoms occuring in the body also occur in the head.

The relevance of this can be seen by considering the complexity of inference: In general, computing the probability of atomic queries even in acyclic probabilistic logic programs is intractable in the size of the underlying domain, as it generalises first-order model counting, which is known to be intractable. However, if the probabilistic logic program is determinate, then the situation changes completely. In this case, the probability of an atomic query is actually completely independent of the size of the underlying domain and can therefore be calculated in constant time, disregarding the domain entirely.

In this contribution, we report on asymptoticplp, an implementation in SWI-Prolog of the transformation of a probabilistic logic program into an asymptotically equivalent determinate program. We outline the relevant theory, describe the system as it is available and with its current limitations as a research prototype, explain the rationale behind the algorithms used for its individual steps and briefly discuss our experiences and the relevance of the implementation to probablistic logic programming and beyond.

2. Preliminaries

Our outline of the theory closely follows [6], whose theoretical results are implemented by the system presented here.

2.1. First-order logic

This paper follows the notation of Ebbinghaus and Flum [7], which we outline here. Full information can be found in Chapter 1 there. A *vocabulary*, sometimes called a *relational* vocabulary for emphasis, is a finite set of relation symbols, each of which are assigned a natural number arity, and of constant symbols, but does not contain function symbols. We also assume an infinite set of first-order variables, customarily referred to by lower-case letters from the end of the alphabet, i. e. u to z. For a vocabulary $\mathcal{S}$, an *atomic $\mathcal{S}$-formula* or *$\mathcal{S}$-atom* is an expression of the form $R(t_1, \ldots, t_n)$, where R is a relation symbol of arity n and every t_i is either a variable or a constant. An *$\mathcal{S}$-literal* is either an atom φ or its negation $\neg\varphi$. A *quantifier-free $\mathcal{S}$-formula* is a Boolean combination of atoms, where conjunction is indicated by $\wedge$, disjunction by $\vee$ and logical implication by $\rightarrow$. A *first-order formula* is made up from atoms using Boolean

connectives as well as existential and universal quantifiers over variables x, indicated by $\exists_x$ and $\forall_x$ respectively. To simplify the notation for longer strings of quantifiers, we use the shorthand $\forall_{x_1,\ldots,x_n}$ for $\forall_{x_1} \ldots \forall_{x_n}$, and analogously for $\exists$. In Subsection 2.4 we also refer to second-order formulas, which are introduced there.

Let φ be a first-order formula. An occurrence of a variable x in φ is called *bound* if it is in the scope of a quantifier annotated with that variable and *free* otherwise. x is called *free in* φ if it occurs freely. We use the notation $\varphi(x_1, \ldots, x_n)$ for φ to assert that every free variable in φ is from $x_1, \ldots, x_n$. We also abuse notation and write $R(\vec{x}, \vec{c})$ for an atomic formula with constants $\vec{c}$ and free variables $\vec{x}$, even though they don't necessarily appear in that order. A formula with no free variables is called a *sentence*, and a set of sentences is called a *theory*. Sentences making up a given theory are also called its *axioms*.

Since tuples such as $x_1, \ldots, x_n$ occur frequently and their exact length is often not important, we use the notation $\vec{x}$ to indicate a tuple of arbitrary finite length in many contexts.

If $\mathcal{S}$ is a vocabulary, then an *$\mathcal{S}$-structure* ω consists of a finite non-empty set D, the *domain* of ω, along with an interpretation of the relation symbols and constants of $\mathcal{S}$ as relations and elements of D respectively. If R is a relation symbol and c a constant, then we R_ω and c_ω for their respective interpretations in ω. Let ω be an $\mathcal{S}$-structure, let $\varphi(x_1, \ldots, x_n)$ be a first-order $\mathcal{S}$-formula and let $a_1, \ldots, a_n$ be a tuple of elements of the domain D of ω. Then we write $\omega \models \varphi(a_1, \ldots, a_n)$ whenever $\varphi(x_1, \ldots, x_n)$ *holds* with respect to the interpretation of $a_1, \ldots, a_n$ for $x_1, \ldots x_n$, and call ω a *model* of $\varphi(a_1, \ldots, a_n)$.

2.2. Logic Programming

Our terminology for logic programs is taken from Ebbinghaus and Flum [7], where one can find a more detailed exposition.

A *general logic program* in a vocabulary $\mathcal{S}$ is a finite set Π of clauses of the form $\gamma \leftarrow \gamma_1, \ldots, \gamma_n$, where $n \geq 0$, γ is an atomic formula and $\gamma_1, \ldots, \gamma_n$ are literals. We call γ the *head* and $\gamma_1, \ldots, \gamma_n$ the *body* of the logic program. The *intensional* relation symbols are those that occur in the head of any clause of a program, while the relation symbols occurring only in the body of clauses are called *extensional*. We write $(\mathcal{S}, \Pi)_{\text{ext}}$ for the extensional vocabulary, or $\mathcal{S}_{\text{ext}}$ where the logic program is clear from context.

An *acyclic* logic program is one in which no intensional relation symbol occurs in the body of any clause (This is at first glance a stronger condition than the usual definition of acyclicity, but by successively unfolding head atoms used in the body of a clause, every acyclic logic program in the usual sense can easily be brought to this form).

A *pure Datalog program* is a general logic program in which no intensional relation symbol appears within any negated literal. To affix a meaning to a pure Datalog program, consider it as a function which take as input a finite $\mathcal{S}_{\text{ext}}$-structure ω and returns as output an extension ω_Π of ω to $\mathcal{S}$. Starting from an empty interpretation of the relation symbols not in $\mathcal{S}_{\text{ext}}$, we successively expand them by applying the rules of Π. We give an informal description of this process:

Let $\gamma(\vec{x})$ be the head of a clause and $\gamma_1(\vec{x}, \vec{y}), \ldots, \gamma_n(\vec{x}, \vec{y})$ the body, and let $\vec{a}$ be a tuple of elements of the domain D of length equal to $\vec{x}$. Then whenever there is a tuple $\vec{b}$ such that $\omega \models \gamma_i(\vec{a}, \vec{b})$ for every i, we add $\gamma(\vec{a})$ to the interpretation of the relation symbol R of the

atom γ. Successively proceed in this manner until nothing can be added by applying any of the clauses of Π. Since the domain of ω is finite, this is bound to happen eventually.

As the restriction for no intensional relation symbol to occur negated in the body of a clause turns out to be too strong for many practical applications, we consider a generalisation to *stratifiable* Datalog programs. A general logic program Π in a vocabulary $\mathcal{S}$ is called a *stratifiable Datalog program* if there is a partition of $\mathcal{S}$ into subvocabularies $\mathcal{S}_{\text{ext}} = \mathcal{S}_0, \dots \mathcal{S}_n$ such that the following holds:

The corresponding logic programs $\Pi_1, \dots, \Pi_n$, where Π_i is defined as the set of clauses whose head atom starts with a relation symbol from $\mathcal{S}_i$, are pure Datalog programs, and the extensional vocabulary of Π_i is contained in $\mathcal{S}_0 \cup \dots \cup \mathcal{S}_{i-1}$.

If Π is a stratifiable Datalog program in $\mathcal{S}$ and ω an $\mathcal{S}_{\text{ext}}$-structure, then ω is obtained by applying $\Pi_1, \dots, \Pi_n$ successively.

A *(pure/stratifiable) Datalog formula* is an expression of the form $(\Pi, P)\vec{x}$, where Π is a (pure/stratifiable) Datalog program, P an intensional relation symbol and $\vec{x}$ a tuple of variables. We say that a Datalog formula *holds* in an $\mathcal{S}_{\text{ext}}$-structure ω for a tuple of elements $\vec{a}$ of the same length as $\vec{x}$, written $\omega \models (\Pi, P)\vec{a}$, if $P(\vec{a})$ is true in ω_Π.

2.3. Probabilistic logic programming

Our discussion on probabilistic logic programming considers a probabilistic logic program under the *distribution semantics*, as a stratifiable Datalog program over probabilistic facts. This covers several different equally expressive formalisms [8]. Note that in particular, probabilistic logic programs as used here do not involve function symbols, unstratified negation or higher-order constructs.

Definition 1. A *probabilistic logic program* consists of probabilistic facts and deterministic rules, where the deterministic part is a stratifiables Datalog program. We consider it in our framework of abstract distribution semantics as follows:

$\mathcal{R}$ is given by relation symbols R' for every probabilistic fact $p_R :: R(\vec{x})$, with $q_{R'} := p_R$. Their arity is just the arity of R.

$\mathcal{S}$ is given by the vocabulary of the probabilistic logic program and additionally the R' in $\mathcal{R}$.

Let Π be the stratifiable Datalog program obtained by prefixing the program $\{R'(\vec{x}) \leftarrow R(\vec{x}) | R' \in \mathcal{R}\}$ to the deterministic rules of the probabilistic logic program.

Then ϕ_P for a $P \in \mathcal{S} \backslash \mathcal{R}$ is given by $(\Pi, P)\vec{x}$.

Notation. Although we have allowed second-order variables in the inductive definitions above, we will assume from now on unless mentioned otherwise that LFP formulas do not have free second-order variables.

2.4. Least fixed point logic

We will now proceed briefly to discuss fixed point logics, which extend ordinary first-order logic with support for recursion. Modelling Datalog programs within fixed-point logic has a long history in database theory, starting with work by Abiteboul and Vianu in the late 1980s

[3]. Our presentation follows the book by Ebbinghaus and Flum (2006, Chapter 8), to which we refer the reader for a more detailed exposition. We begin by introducing the syntax.

As atomic second-order formulas occur, as subformulas of least fixed point formulas, we will introduce second-order variables.

Definition 2. Assume an infinite set of second-order variables, indicated customarily by uppercase letters from the end of the alphabet, each annotated with a natural number arity. Then an *atomic second-order formula* φ is either a (first-order) atomic formula, or an expression of the form $X(t_1, \ldots, t_n)$, where X is a second-order variable of arity n and $t_1, \ldots, t_n$ are constants or (first-order) variables.

We now proceed to least fixed point formulas.

Definition 3. A formula φ is called *positive in a variable* x if x is in the scope of an even number of negation symbols in φ.

A *formula in least fixed point logic* or *LFP formula* over a vocabulary $\mathcal{R}$ is defined inductively as follows:

1. Any atomic second-order formula is an LFP formula.
2. If φ is an LFP formula, then so is $\neg\varphi$.
3. If φ and ψ are LFP formulas, then so is $\varphi \vee \psi$
4. If φ is an LFP formula, then so is $\exists x\varphi$ for a first-order variable x.
5. If φ is an LFP formula, then so is $[\mathrm{LFP}_{\vec{x},X}\varphi]\vec{t}$, where φ is positive in the second-order variable X and the lengths of the string of first-order variables $\vec{x}$ and the string of terms $\vec{t}$ coincide with the arity of X.

An occurrence of a second-order variable X is *bound* if it is in the scope of an LFP quantifier $\mathrm{LFP}_{\vec{x},X}$ and *free* otherwise.

Fixed point semantics have been used extensively in logic programming theory, and we will exploit this when relating the model theory of LFP to probabilistic logic programming below.

We first associate an operator with each LFP formula φ:

Definition 4. Let $\varphi(\vec{x}, \vec{u}, X, \vec{Y})$ be an LFP formula, with the length of $\vec{x}$ equal to the arity of X, and let ω be an $\mathcal{R}$-structure with domain D. Let $\vec{b}$ and $\vec{S}$ be an interpretation of $\vec{u}$ and $\vec{Y}$ respectively. Then we define the operator $F^{\varphi} : \mathfrak{P}(D^k) \to \mathfrak{P}(D^k)$ as follows:

$$F^{\varphi}(R) \coloneqq \left\{ \vec{a} \in D^k | \omega \models \varphi(\vec{a}, \vec{b}, R, \vec{S}) \right\}.$$

Since we have restricted Rule 5 in Definition 3 to positive formulas, F^{φ} is monotone for all φ (i. e. $R \subseteq F^{\varphi}(R)$ for all $R \subseteq D^k$). Therefore we have:

Fact 1. *For every* LFP *formula* $\varphi(\vec{x}, \vec{u}, X, \vec{Y})$ *and every* $\mathcal{R}$*-structure on a domain* D *and interpretation of variables as in Definition 4, there is a relation* $R \subseteq D^k$ *such that* $R = F^{\varphi}(R)$ *and that for all* R' *with* $R' = F^{\varphi}(R')$ *we have* $R \subseteq R'$.

Definition 5. We call the R from Fact 1 the *least fixed point* of $\varphi(\vec{x}, \vec{u}, X, \vec{Y})$

Now we are ready to define the semantics of least fixed point logic:

Definition 6. By induction on the definition of an LFP formula, we define when an LFP formula $\varphi(\vec{X}, \vec{x})$ is said to *hold* in an $\mathcal{R}$-structure ω for a tuple $\vec{a}$ from the domain of ω and relations $\vec{A}$ of the correct arity:

The first-order connectives and quantifiers $\neg$, $\vee$ and $\exists$ as well as $\wedge$ and $\forall$ defined from them in the usual way are given the usual semantics.

An atomic second order formula $X(\vec{x}, \vec{c})$ holds if and only if $(\vec{a}, \vec{c_\omega}) \in A$.

$[\mathrm{LFP}_{\vec{x},X}\varphi]\vec{t}$ holds if and only if $\vec{a}$ is in the least fixed point of $F^{\varphi(\vec{x},X)}$.

Least fixed point logics are related to logic programming by the following fact [7, Theorem 9.1.1]:

Fact 2. *For every stratifiable Datalog formula $(\Pi, P)\vec{x}$ as above, there exists an* LFP *formula $\varphi(\vec{x})$ over the extensional vocabulary $\mathcal{R}$ of Π such that for every $\mathcal{R}$-structure ω and every tuple $\vec{a}$ of elements of ω of the same length as $\vec{x}$, $\omega \models \varphi(\vec{a})$ if and only if $\omega \models (\Pi, P)\vec{a}$.*

2.5. Asymptotic quantifier elimination

As we are interested in the asymptotic behaviour of formulas, we introduce the notion of abstract families of distributions:

Definition 7. A *family of distributions* for a relational vocabulary $\mathcal{S}$ is a sequence $\left(Q^{(n)}\right)_{n\in\mathbb{N}}$ of probability distributions on the set Ω_n of all $\mathcal{S}$-structures with domain $\{1, \ldots, n\} \subseteq \mathbb{N}$.

Such families are induced by logical formulas (or logic programs) and probabilistic facts as follows.

Definition 8. Let $\mathcal{S}$ be a relational vocabulary, $\mathcal{R} \subseteq \mathcal{S}$, and let $L(\mathcal{R})$ be a logical language over $\mathcal{R}$. Then an *abstract L-distribution over $\mathcal{R}$ (with vocabulary $\mathcal{S}$)* consists of the following data:

For every $R \in \mathcal{R}$ a number $q_R \in \mathbb{Q} \cap [0, 1]$.

For every $R \in \mathcal{S}\backslash\mathcal{R}$, an $L(\mathcal{R})$-formula ϕ_R of the same arity as R.

In the following we will assume that all vocabularies are finite. The semantics of an abstract distribution is only defined relative to a domain D, which we will also assume to be finite. The formal definition is as follows:

Definition 9. Let $L(\mathcal{R})$ be a logical language over $\mathcal{R}$ and let D be a finite set. Let T be an abstract L-distribution over $\mathcal{R}$. Let Ω_D be the set of all $\mathcal{R}$-structures with domain D.

Then the *probability distribution on Ω_D induced by T, written $Q_T^{(D)}$*, is defined as follows:

For all $\omega \in \Omega_D$, if $\exists_{\vec{a}\in\vec{D}}\exists_{R\in\mathcal{S}\setminus\mathcal{R}} : R(\vec{a}) \nLeftrightarrow \phi_R(\vec{a})$, then $Q_T^{(D)}(\{\omega\}) \coloneqq 0$

Otherwise, $Q_T^{(D)}(\{\omega\}) \coloneqq \prod_{R\in\mathcal{R}} (q_R^{|\{\vec{a}\in\vec{D}|R(\vec{a})\}|}) \times \prod_{R\in\mathcal{R}} (1 - q_R)^{|\{\vec{a}\in\vec{D}|\neg R(\vec{a})\}|}$

In other words, all the relations in $\mathcal{R}$ are independent with probability q_R and the relations in $\mathcal{S} \setminus \mathcal{R}$ are defined deterministically by the $L(\mathcal{R})$-formulas ϕ_R. We will illustrate that with an example.

Example 1. Let $\mathcal{R} = \{R, P\}$, $\mathcal{S} = \{R, P, S\}$, for a unary relation R a binary relation P and a unary relation S. Then an abstract distribution over (R) has numbers q_R and q_P which encode probabilities. Consider the first-order distribution T with $\varphi_S = \exists_y (R(x) \wedge P(x, y))$. For any domain D, $Q_T^{(D)}$ is obtained by making an independent choice of $R(a)$ or $\neg R(a)$ for every $a \in D$, with a q_R probability of $R(a)$. Similarly, an independent choice of $P(a, b)$ or $\neg P(a, b)$ is made for every pair (a, b) from D^2, with a q_P probability of $P(a, b)$. Then, for any possible $\mathcal{R}$-structure, the interpretation of S is determined by $\forall_x S(x) \leftrightarrow \varphi_S(x)$.

We introduce our notion of asymptotic equivalence for families of distributions:

Definition 10. Two families of distributions $(Q^{(n)})$ and $(Q'^{(n)})$ are *asymptotically equivalent* if $\lim_{n\to\infty} \sup_{A\subseteq\Omega_n} |Q^{(n)}(A) - Q'^{(n)}(A)| = 0$

This gives us the following setting for asymptotic quantifier elimination:

Definition 11. Let $L(\mathcal{R})$ be an extension of the class of quantifier-free $\mathcal{R}$-formulas. Then $L(\mathcal{R})$ *has asymptotic quantifier elimination* if every abstract $L(\mathcal{R})$ distribution is asymptotically equivalent to a quantifier-free distribution over $L(\mathcal{R})$.

Building on prior work by Glebskii et al. [1] on asymptotic quantifier elimination for first-order logic, Blass et al. [2] showed (phrased in the language adopted here):

Theorem 1. *Least fixed point logic has asymptotic quantifier elimination.*

To obtain a characterisation within probabilistic logic programming, however, we need to translate quantifier-free first order formulas back to stratifiable Datalog.

In fact, they can be mapped to a subset of stratifiable Datalog that is well-known from logic programming:

Definition 12. A Datalog program, Datalog formula or probabilistic logic program is called *determinate* if every variable occurring in the body of a clause also occurs in the head of that clause.

Example 2. Examples of determinate clauses in this sense are $R(x) :- P(x)$ or $Q(x, y) :- R(x)$. Indeterminate clauses include $R(x) :- P(y)$ or $R(x) :- Q(x, y)$.

Determinacy corresponds exactly to the fragment of probabilistic logic programs identified as projective by Jaeger and Schulte [9, Proposition 4.3].

Indeed, Ebbinghaus and Flum's [7] proof of their Theorem 9.1.1 shows:

Fact 3. *Every quantifier-free first order formula is equivalent to an acyclic determinate stratifiable Datalog formula.*

This leads to the main result of this subsection, the implemented transformation.

Theorem 2. *Every probabilistic logic program is asymptotically equivalent to an acyclic determinate probabilistic logic program.*

3. Description of the system

asymptoticplp is an SWI-Prolog program, available at
https://swish.swi-prolog.org/p/asymptoticPLP.pl and entirely written in SWI-Prolog.

The main entry point into the program is the predicate compute_lp_form/3, to be used with the first argument instantiated to the deterministic part of the program to be approximated, and the second argument instantiated to the query predicate. The third argument is then bound to a determinate logic program such that when readding the probabilistic facts, the resulting probabilistic logic program is asymptotically equivalent to the original program on large domains with respect to the query predicate.

Programs are represented as lists of clauses of the form head−vars−body, where variables are rendered as ordinary lowercase Prolog atoms.

Example 3. Consider the example of connectedness in a probabilistic graph. Here, we have a random graph with edge relation edge/2, true for any pair of nodes with fixed probability $\alpha \in (0, 1)$. This can be expressed with a probabilistic fact such as `0.5 :: edge(X,Y)`. Connectedness of two nodes is then expressed by the usual Datalog clauses,

```
connected(X,Y) :- edge(X,Y).
connected(X,Z) :- connected(X,Y), edge(Y,Z).
```

In the representation used by the system, the program is given by the list

```
[connected−[x,y]−[edge−[x,y]],
 connected−[x,z]−[connected−[x,y], edge−[y,z]]]
```

Informally, we can argue that in a sufficiently large random graph, there will always be for any two nodes a and c a third node b such that a is adjacent to b and b is adjacent to c. Thus, we expect that asymptotically, any two nodes will be connected.

This is confirmed by computation in the system, which returns connected−[x,y]−[], its notation for a clause with empty body which is unconditionally true.

Example 4. A probabilistic logic programming classic is the "friends-smokers model". In this program, there is a random graph of friendship connections between different people. Additionally, there is a certain baseline probability of anyone starting smoking. Then, whenever a person befriends another and that person smokes, there is a certain probability that this person also takes up smoking. We thus have a random graph with edge relation `friends/2`, true for any pair of nodes with fixed probability $\alpha \in (0, 1)$. This can be expressed with a probabilistic fact such as `0.3 :: friends(X,Y)`.

Using the syntactic sugar of probabilistic clauses, we can write the remaining model as follows:

```
0.1 :: smokes(X).
0.2 :: smokes(X) :- friends(X,Y), smokes(Y).
```

This encodes a probabilistic logic program with auxiliary predicates u/1 and v/2 as follows:

```
smokes(X) :- u(X).
smokes(X) :- friends(X,Y), smokes(Y), v(X,Y).
```

```
0.1 :: u(X).
0.2 :: v(X,Y).
0.3 :: friends(X,Y).
```

Informally, we can see that in a sufficiently large group of people in which everyone has a certain chance to smoke and to be your friend, there is bound to be someone who smokes and is your friend; in fact, there are bound to be any number of such people, and one of them will convince you to take up smoking.

This is confirmed by computation in the system, which returns smokes−[x]−[], its notation for a clause with empty body which is unconditionally true.

However, consider an altered model where only nice people have friends:

```
smokes(X)      :- u(X).
smokes(X)      :- friends(X,Y), smokes(Y), v(X,Y).
friends(X,Y) :- nice(X), w(X,Y).
0.1 :: u(X).
0.2 :: v(X,Y).
0.3 :: w(X,Y).
```

In this model, the reasoning above only works for nice people. This leads us to conclude that the only people who smoke are either those who take up the habit on their own accord, or are nice enough to have friends.

This is again confirmed by the system, which delivers a program equivalent to [smokes−[x]−[u−[x]], smokes−[x]−[nice−[x]]].

As explained in greater detail in Section 4, the system procedes by converting the input program into an equivalent least fixed point formula and then using asymptotic quantifier elimination to eliminate all uses of the fixed point quantifier and all classical existential and universal quantifiers in the resulting formula. Since to the best of our knowledge this is the first implementation of both conversion to formulas and asymptotic quantfier elimination, asymptoticplp also provides predicates that perform those steps in isolation.

To understand how these are invoked, one needs only to understand the representation used for formulas of first-order logic and for least fixed point quantifiers. We tried to facilitate use by adopting a natural syntax, which is best understood from example:

```
lfp(connected,
    edge−[x, y] or
    exists([at],connected−[x, at] and edge−[at, y]))
```

Existential and universal quantifiers are written using the term constructors exists/2 and forall/2, which take as their first argument a list of bound variables (potentially more than one to avoid unnecessary nesting of quantifiers) and as their second argument a formula. Boolean connectives use the infix constructors and/2 and **or**/2 and the term constructor **not**/1.

The least fixed point quantifier is represented by the constructor lfp/2, which takes as its first argument the predicate with respect to which the fixed point is computed, and as its second argument a formula.

So the example formula above is the translation of Example 3 to a formula of least fixed point logic, and can be computed in the system using the predicate compute_lp_lfp/3, whose first

argument is the logic program to be translated and whose second argument holds the target predicate, in this case connected.

Similarly, the system allows direct access to the asymptotic quantifier elimination algorithm for least fixed point formulas using the predicate compute_lfp_form/2, which takes as its first argument the least fixed point formula to be converted and binds the second argument to an asymptotically equivalent quantifier-free formula. The predicate reduced_quant_elim/2 behaves identically, but only works for first-order formulas. It is provided separately because of the relatively greater interest in the asymptotic theory of first-order logic by logicians and combinatoricists, and allows for optimisations to be added for this case in particular.

Current limitations

The system is currently a research prototype, with some limitations that should be overcome before it is more widely deployed. Constants in clause heads or bodies are not currently supported by the system. Furthermore, the single-letter atoms a, b, c, d, e, f, g and h are used by the system for substituting variables, and no two variable names should differ from each other by merely prefixing one of those letters. To be safe, it is recommended to begin variable names with letters later in the alphabet. Alternatively, or if in larger examples the system requires more variables to substitute, the system can also be invoked with the query predicate compute_lp_form/4, where the third argument holds the list of names to be used for forming substitute variables. Also, the system currently does not support overloading of predicate names within the probabilistic logic program, as it refers to predicates simply by name rather than with associated arities.

4. Description of the algorithm

The algorithm proceeds in several stages, which have been described separately in the classical finite model theory literature. The overarching structure can be seen in the pseudo-code below.

$(\Pi_1, \ldots, \Pi_n)$ = Stratification(Π)
for i=1 **to** n
    For intensional predicates P **of** Π_i,
    intensional predicates $Q_1, \ldots, Q_k$ **of** Π_j **for** $j < i$
        $\Phi_aux(P)$ = LP_to_SLFP(Π_i, P)
        $\Phi_(P)$ = substitute($Q_1, \ldots, Q_k, \Phi_aux(Q_1), \ldots, \Phi_aux(Q_n), \Phi_aux(P)$)
Φ = $\Phi(R)$
Ψ = SLFP_to_LFP(Φ)
Ψ_0 = $\bot$
While $\Psi^+ \not\equiv \Psi$
    Ψ^+ = AQE(Step(Ψ))
    Ψ = Ψ^+

In the first part of the process, the logic program and query predicate are converted into a formula of least fixed point logic, a realisation of Fact 2.

This procedure has several stages. First, the program is explicitly stratified. Then, LP_to_SLFP converts every stratum to a formula in *simultaneous least fixed point logic*. Simultaneous least fixed point logic is an intermediate formalism, in which the fixed point iteration is applied to a number of first-order formulas simultaneously. Since this aligns closely with the operational semantics of Datalog, constructing the equivalent formula in simultaneous least fixed point logic reduces to an essentially syntactic rewrite. The procedure employed by us is a generalisation of Clark's Completion [10] to stratified logic programs. It was first presented by Abiteboul and Vianu [3], and a good exposition is found in Ebbinghaus and Flum [7, Theorem 9.1.1].

Predicates from lower strata are then successively substituted until the final stratum has been converted into a simultaneous least fixed point logic formula all of whose predicates lie in the extensional vocabulary of Π.

SLFP_to_LFP then converts the simultaneous least fixed point formula into an ordinary least fixed point formula in the sense of Subsection 2.4.

While simultaneous fixed points do not increase the expressivity of fixed point logic, their removal comes at a cost. There are two classical ways to replace them with ordinary fixed point quantifiers, either increasing the arities of the predicates involved or increasing the nesting depth of the fixed point quantifiers. Following Kreutzer [11, page 40], we have decided to implement the latter algorithm going back to Bekić [12], as lower arities prove more favourable for the remaining steps of the algorithm.

Following the asymptotic analysis of Blass et al. [2], it remains to compute a first-order formula asymptotically equivalent to the least fixed point formula just constructed, and to compute a quantifier-free formula asymptotically equivalent to that first-order one.

We first turn to the elimination of the least fixed point constructor. The algorithm implicit in Blass et al. [2] proceeds by unfolding the iteration rule represented by the least fixed point formula. This is represented by the Step function in the pseudocode above. The promise of asymptotic quantifier-elimination guarantees that this procedure will eventually no longer result in asymptotically distinguishable formulas, since there is only a finite number of possible quantifier-free formulas available. Note that this subproblem is already EXPTIME-complete [2].

Given a first-order formula, AQE computes an asymptotically equivalent quantifier-free one. Several classical algorithms for this can be found in the theoretical literature. In terms of complexity, an optimal approach was given by Grandjean [13], who showed that the theory of almost all finite structures is decidable in polynomial space using a semantic method that makes indirect use of quantifier elimination. However, we decided to implement the direct quantifier elimination algorithm going back to Gaifman [14] and elaborated in the seminal paper by Glebskii et al. [1] which first established the completeness of the first-order theory of almost all relational structures. We followed Lingsch-Rosenfeld's modern exposition [15] of the algorithm in our implementation.

Although Grandjean [13] contends that this algorithm is inefficient and it is indeed not optimal in its worst-case complexity, we found it very well suited to our application. This is because it enables us to interleave the iteration steps of the transformation algorithm of Blass et al. [2] with the quantifier elimination procedure.

Thus, we simplify the formulas considerably as we unfold the iteration rule and can reduce checking the termination condition of the iteration to checking the satisfiability of a quantifier-free formula, which can easily be implemented using SAT-solving techniques.

5. Discussion and conclusion

The system presented here allows users to test the asymptotic behaviour of probabilistic logic programs on large domains statically without running them on large databases. This is particularly important for probabilistic logic programs since the complexity of probabilistic inference makes querying large domains with general programs intractable. The asymptotically equivalent determinate programs computed by the system evaluate queries independently of domain size, thereby allowing queries to be approximated statically without access to a particular domain. Thus, one could see the present system as a contribution to scalable approximate inference, with an approximation that converges to the correct result with exponential speed as the domain size increases.

Equally, static analysis of a probabilistic logic program can also be seen as a form of verification of certain properties. From this point of view, asymptoticplp allows for a "sanity check" that the behaviour of a program on large domains matches the behaviour expected by the programmer. For instance, a program that predicts disease transmission in a group of people may well be expected to predict increased infection rate with an increase in population size. This can easily be verified with asymptoticplp. Overall, static analysis tools for probabilistic logic programs are scarce, and therefore we hope that this system may inspire other approaches to be implemented effectively.

While the worst-case complexity of computing the asymptotically equivalent quantifier-free formulas for a given formula in least fixed point logic is EXPTIME-complete [2], we have seen that naturally occurring examples stemming from probabilistic logic programs can actually often be evaluated very quickly. The reason for this gap can be seen by comparing the number of iterations required until the termination condition is satisfied with the maximal number of iterations guaranteed by the upper bound in the complexity proof [2, Theorem 4.1], computed within our system by the compute_alpha/4 predicate. For instance, the theoretical maximal number of iterations in the highly recursive Example 3 is 256, but the algorithm terminates already after 2 iterations. In Example 4, which is typical of a probabilistic logic program written using probabilistic clauses, the theoretical maximal number of iterations is 16, while the algorithm in fact terminates after just 3 iterations.

Beyond the relevance for probabilistic logic programming proper, this system is to the best of our knowledge the first practical implementation of any decision or quantifier elimination procedure for the theory of almost all finite structures, whether for first-order logic or for fixed point logic. It can be hoped that researchers in finite model theory, database theory and the combinatorics of random graphs find it useful to be able to evaluate the asymptotics of properties of random graphs effectively.

The encouraging runtime for sample programs also points to the importance of interleaving iteration and quantifier elimination steps, an insight that may inspire similar efforts aimed at evaluating least fixed point formulas in other theories with first-order quantifier elimination but daunting complexity, such as the theory of the rational order or the theory of real-closed and algebraically closed fields.

Similar results on limiting approximations have been obtained for formalisms outside of probabilistic logic programming. This includes Cozman and Maua's Bayesian networks specifications [16], for which the first-order asymptotic quantifier elimination algorithm implemented

here is immediately applicable, and varous formalisms based on lifting Bayesian networks to a relational language. Jaeger proved that as long as their combination functions satisfy a certain property, exponential convergence, the probabilities induced by a relational Bayesian network will always converge with increasing domain size [17]. More recently, the second author demonstrated convergence for domain-size aware relational logistic regression networks [18], and Koponen and the second author showed convergence for formalisms with a range of statistical quantifiers [19, 20, 21, 22].

Our approach to implementation is not immediately transferable to relational Bayesian networks since it is based on the clear distinction between logical and probabilistic part inherent in probabilistic logic programs, as well as on the sophisticated logical description of Datalog semantics developed in the literature. Additionally, arguably the key contribution of our implementation concept is the ability to deal with recursive programs via fixed point logic, while Bayesian networks and similar formalisms based on directed graphical models are inherently acyclic.

5.1. Further work

Currently, an important limitation of the system is that it does not natively support named domain constants in probabilistic logic programs. As explained by Ebbinghaus and Flum [7], clauses involving constants can be refactored as a preprocessing step, where say p[const(min)]−[] would be replaced with p[x]−[const(x,min)], indicating that the variable x is equal to the constant min. This is very similar to how clauses with duplicate head variables are treated already and should fit well into the existing architecture.

Performance could be improved by replacing the current satisfiability check within the termination condition with a state-of-the-art SAT solver, which could simply be grafted into the definition of the notsat/1 predicate.

5.2. Conclusion

Overall, the system provides a tool for modellers and deployers of probabilistic logic programs statically to evaluate the behaviours of programs on large domains, while at the same time providing insights into the relevance of quantifier elimination for the efficient evaluation of least fixed point formulas. While the current limitations, particularly the lack of support for constants in formulas, somewhat limit its scope, the prototype shows the promise of the method despite the seemingly prohibitive worst-case complexity of the problem.

Acknowledgments

During the work leading up to this publication, the second author was supported by LMUexcellent, funded by the Federal Ministry of Education and Research (BMBF) and the Free State of Bavaria under the Excellence Strategy of the Federal Government and the Länder.

References

[1] Y. V. Glebskii, D. I. Kogan, M. I. Liogon'kii, V. A. Talanov, Range and degree of realizability of formulas in the restricted predicate calculus, Cybernetics 5 (1969) 142–154. doi:10.1007/BF01071084.

[2] A. Blass, Y. Gurevich, D. Kozen, A Zero-One Law for Logic with a Fixed-Point Operator, Information and Control 67 (1985) 70–90. doi:10.1016/S0019-9958(85)80027-9.

[3] S. Abiteboul, V. Vianu, Datalog extensions for database queries and updates, Journal of Computer and System Sciences 43 (1991) 62–124. doi:10.1016/0022-0000(91)90032-Z.

[4] D. Poole, Probabilistic Horn abduction and Bayesian networks, Artificial Intelligence 64 (1993) 81–129. doi:10.1016/0004-3702(93)90061-F.

[5] T. Sato, A statistical learning method for logic programs with distribution semantics, in: L. Sterling (Ed.), Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995, MIT Press, 1995, pp. 715–729.

[6] F. Weitkämper, An asymptotic analysis of probabilistic logic programming with implications for expressing projective families of distributions, Theory and Practice of Logic Programming (2021). doi:10.1017/S1471068421000314.

[7] H.-D. Ebbinghaus, J. Flum, Finite model theory, Springer Science & Business Media, 1999.

[8] F. Riguzzi, T. Swift, A survey of probabilistic logic programming, Declarative Logic Programming: Theory, Systems, and Applications (2018) 185–228.

[9] M. Jaeger, O. Schulte, Inference, learning, and population size: Projectivity for SRL models, in: Eighth International Workshop on Statistical Relational AI (StarAI), 2018. arXiv:1807.00564.

[10] K. L. Clark, Negation as Failure, Springer US, Boston, MA, 1978, pp. 293–322. doi:10.1007/978-1-4684-3384-5_11.

[11] S. Kreutzer, Pure and applied fixed-point logics, Dissertation, RWTH Aachen, 2002. URL: https://publications.rwth-aachen.de/record/59225.

[12] H. Bekić, Definable operations in general algebras, and the theory of automata and flowcharts, in: C. B. Jones (Ed.), Programming Languages and Their Definition: H. Bekič (1936–1982), Springer Berlin Heidelberg, Berlin, Heidelberg, 1984, pp. 30–55. doi:10.1007/BFb0048939.

[13] E. Grandjean, Complexity of the first-order theory of almost all finite structures, Information and Control 57 (1983) 180–204. doi:10.1016/S0019-9958(83)80043-6.

[14] H. Gaifman, Concerning measures in first order calculi, Israel Journal of Mathematics 2 (1964) 1–18. doi:10.1007/BF02759729.

[15] M. Lingsch Rosenfeld, Asymptotic Quantifier elimination, Bachelorarbeit, LMU Munich, 2022. URL: https://www.en.pms.ifi.lmu.de/publications.

[16] F. G. Cozman, D. D. Mauá, The finite model theory of Bayesian network specifications: Descriptive complexity and zero/one laws, Int. J. Approx. Reason. 110 (2019) 107–126. doi:10.1016/j.ijar.2019.04.003.

[17] M. Jaeger, Convergence results for relational bayesian networks, in: Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998, IEEE Computer Society, 1998, pp. 44–55. doi:10.1109/LICS.1998.705642.

[18] F. Weitkämper, Scaling the weight parameters in markov logic networks and relational

logistic regression models, CoRR abs/2103.15140 (2021). arXiv:2103.15140.

[19] V. Koponen, Conditional probability logic, lifted bayesian networks, and almost sure quantifier elimination, Theor. Comput. Sci. 848 (2020) 1–27. doi:10.1016/j.tcs.2020.08.006.

[20] F. Weitkämper, Statistical relational artificial intelligence with relative frequencies: A contribution to modelling and transfer learning across domain sizes, CoRR abs/2202.10367 (2022). arXiv:2202.10367.

[21] V. Koponen, F. Weitkämper, Asymptotic elimination of partially continuous aggregation functions in directed graphical models, Information and Computation 293 (2023) 105061. doi:10.1016/j.ic.2023.105061.

[22] V. Koponen, F. Weitkämper, On the relative asymptotic expressivity of inference frameworks, CoRR abs/2204.09457 (2022). doi:10.48550/arXiv.2204.09457. arXiv:2204.09457.