Foundations of Statistical Relational Models

Paulius Dilkas

Supervisors: Dr Vaishak Belle and Dr Ron Petrick

School of Informatics, University of Edinburgh

26th February 2020

1 Introduction

2 Work Plan

3 Progress

A Submitted Papers

On the Equivalence of Constants in Relational Knowledge Bases

Paulius Dilkas[1*] and **Vaishak Belle**[1,2]
[1]University of Edinburgh, UK
[2]Alan Turing Institute, UK
p.dilkas@sms.ed.ac.uk, vaishak@ed.ac.uk

Abstract

Driven by the powerful promises of statistical relational artificial intelligence, relational structures are as important as ever. Various notions of equivalence have contributed to relational models becoming more tractable and usable. We present a fresh (function-based) perspective on relational knowledge bases and use it to formally consider equivalence of constants. We show how logic programs, acting as maps between knowledge bases, are fundamentally defined by their effects on equivalence classes. We also consider how the results can be extended to a probabilistic setting. The results unveil properties of the equivalence structure induced by knowledge bases and have important implications for lifted inference, inductive logic programming, and relational auto-encoders.

1 Introduction

Various definitions of symmetry and equivalence have demonstrated their importance in statistics [Kingman, 1978; Diaconis and Freedman, 1980], and, more recently, neural networks [Ravanbakhsh *et al.*, 2017] and statistical relational artificial intelligence [Raedt *et al.*, 2016; Bui *et al.*, 2013; Niepert and den Broeck, 2014]. These notions can enable enormous computational leverage for inference tasks, and can also lead to more robust learning of representations as entities are not treated in an i.i.d. manner.

In this paper, we consider equivalence classes of constants (and tuples of constants) in *relational knowledge bases* (KBs), defined analogously to orbits induced by renaming permutations [Bui *et al.*, 2013] in Markov logic networks [Richardson and Domingos, 2006]. Our motivation is twofold. First, we define a relation between KBs that describes whether there exists a logic program that can transform one KB to another. This provides the theoretical background to recent work in auto-encoding KBs using logic programs [Dumancic *et al.*, 2019]. The same framework can also be used to conceptualise a version of inductive logic programming [Muggleton, 1991]. Second, equivalence of constants relates to domain abstraction [Belle, 2018; Holtzen *et al.*, 2017] and exchangeability [Diaconis and Freedman, 1980], and is of great importance to efficient lifted inference [Poole, 2003; Niepert and den Broeck, 2014; Bui *et al.*, 2012; de Salvo Braz *et al.*, 2005; Kersting, 2012; Gogate and Domingos, 2010; Gogate and Domingos, 2016]. While most of this paper is dedicated to the purely logical setting, in Section 6 we briefly discuss how the work can be extended to probabilistic KBs and probabilistic logic programs. In this context, reasoning about equivalences of constants can be an efficient way to discover other types of symmetries such as symmetries of ground atoms, formulas, and assignments [Niepert, 2012].

After a review of preliminaries in Section 2, we begin with Section 3 where we present a new way to interpret logical constructs such as atoms and clauses by introducing *realisation functions*. Section 4 then defines equivalence between (tuples of) constants and outlines some key properties. Our main contribution is Theorem 2 in Section 5 that shows how the existence of a logic program that transforms one KB into another is fundamentally related to refinements of equivalence relations. We also demonstrate a concrete way of constructing such a logic program.

2 Preliminaries

In this section, we review the terminology of logic programming, introduce our notation for various constructs, and outline several key results on equivalence.

2.1 Logic and Logic Programming

The primitive building blocks of KBs and logic programs are constants (e.g., $a, b, \dots$), variables (e.g., $X, Y, \dots$), and predicates (e.g., $\mathsf{P}, \mathsf{Q}, \dots$). A *term* is either a constant or a variable. The *arity* of a predicate is the number of terms that it can be applied to (a predicate P of arity n is denoted by P/n). An *atom* is a predicate (say, of arity n) applied to n terms (e.g., $\mathsf{P}(X, a)$). A *literal* is either an atom or its negation. We define a *formula* as a conjunction of literals. An atom or a literal is *ground* if all its terms are constants. In Section 3, we will redefine some of these terms in a more rigorous way.

Definition 1. Let P and C be sets of predicates and constants, respectively. The *Herbrand base* of P and C is the set of all ground atoms that can be constructed from elements of P and C. We will use $\mathcal{KB}(P, C)$ to denote the power set of the Herbrand base. A *knowledge base* is a subset of the Herbrand

[*]Contact Author

base, containing all atoms that evaluate to true (with no additional structural restrictions), i.e., an element of $\mathcal{KB}(P, C)$. If a ground atom is in the KB, it is a *fact*.

Definition 2. A *clause* is a pair of an atom A and a formula F (written as $\mathsf{A} \leftarrow \mathsf{F}$) with an implication that for all possible ways of replacing variables in A and F with constants, if F is true, then A is also true. We say that A is the *head* of the clause, and F is the *body*.

Definition 3. Let C be a set of constants, and let P_1, P_2 be two disjoint sets of predicates. A *logic program* $\mathcal{L}$ from $\mathcal{KB}(P_1, C)$ to $\mathcal{KB}(P_2, C)$ (written $\mathcal{L} : \mathcal{KB}(P_1, C) \to \mathcal{KB}(P_2, C)$) is a set of clauses such that all head predicates are in P_2, all body predicates are in P_1, and all constants are in C.

2.2 Equivalence and Set Partitions

We briefly review some well-known results on equivalence (see, e.g., [Brualdi, 1977; Bourbaki, 2004] for more information). Let A be a non-empty set, and let $\sim$ and $\approx$ be two equivalence relations over A.

Notation. For any positive integer n, $[n] \coloneqq \{1, \ldots, n\}$ while, by abuse of notation, for any constant c, $[c]$ refers to the equivalence class of c. We let

$$A^\infty \coloneqq \bigcup_{n=1}^{\infty} A^n,$$

denote the set of all tuples of all lengths constructed from elements of A[1]. We let $A/\!\sim$ denote the *quotient set* (i.e., the set of equivalence classes) of $\sim$. Finally, let $\mathbb{B} \coloneqq \{\bot, \top\}$ be the set with two values corresponding to false and true.

Definition 4. We say that $\approx$ is *coarser* than $\sim$ (equivalently, $\sim$ is *finer* than $\approx$, or $\sim$ is a *refinement* of $\approx$) if, for any $a, b \in A$, if $a \sim b$, then $a \approx b$.

Definition 5. Let P and Q be two partitions of A. Then P is *coarser* than Q if, for every $q \in Q$, there is a $p \in P$ such that $q \subseteq p$.

Theorem 1 (Fundamental Theorem of Equivalence Relations)**.** *If $\sim$ is an equivalence relation on A, then $A/\!\sim$ is a partition of A. If P is a partition of A then there is an equivalence relation $\equiv$ on A such that $A/\!\equiv\, = P$.*

Lemma 1. *$\approx$ is coarser than $\sim$ if and only if $A/\!\approx$ is coarser than $A/\!\sim$.*

3 Variables: What Are They Made Of?

In this section, we will outline the interpretation of logical entities such as variables, atoms, and clauses that will be used throughout the paper. We will show how formulas can be characterised as compositions of functions and define a new type of functions that act as links between predicates and atoms. We will use the following as a running example:

Example 1. Let $C = \{a, b, c\}$ be a set of constants, and let $P_1 = \{\mathsf{Q}/2, \mathsf{R}/1\}$ and $P_2 = \{\mathsf{P}/2\}$ be two sets of predicates. Let $\mathcal{L} : \mathcal{KB}(P_1, C) \to \mathcal{KB}(P_2, C)$ be a logic program with

$$\mathsf{P}(X, a) \leftarrow \mathsf{Q}(X, Y) \wedge \neg \mathsf{R}(X) \tag{1}$$

as its only clause.

First, list the variables in alphabetical (or any other) order and count the number of variables. In this case, we have two variables: X and Y. Thus, our initial domain is C^2, X represents the first element of the pair, and Y the second.

Definition 6. Let A be a set, and let n, m be positive integers. Let $\{p_i\}_{i=1}^m$ be a set of projections $A^m \to A$. A function $\rho : A^n \to A^m$ is a *realisation function* if, for $i \in [m]$, $p_i \circ \rho$ is either a projection $A^n \to A$ or a constant function[2].

Definition 7. Let Δ be a KB, and let n be a positive integer. An *atom acting on n variables* in Δ is a composition

$$C^n \xrightarrow{\ \rho\ } C^m \xrightarrow{\ \mathsf{P}\ } \mathbb{B}$$

of a realisation function ρ and the evaluation function for a predicate P.

Section 3 shows how these definitions apply to the atoms in Example 1. We will now show how the same ideas can be extended to literals and formulas. To represent a literal such as $\neg\mathsf{R}(X)$, we can compose the representation of $\mathsf{R}(X)$ with $\neg$, interpreted as a function:

$$C^2 \xrightarrow{\ p_1\ } C \xrightarrow{\ \mathsf{R}\ } \mathbb{B} \xrightarrow{\ \neg\ } \mathbb{B}.$$

Keeping the number of arrows (i.e., composed functions) the same, we can formalise the literal $\mathsf{Q}(X, Y)$ as

$$C^2 \xrightarrow{\ \mathrm{id}\ } C^2 \xrightarrow{\ \mathsf{Q}\ } \mathbb{B} \xrightarrow{\ \mathrm{id}\ } \mathbb{B}.$$

Conjunction $\wedge$ then takes a product of $\mathbb{B}$'s and maps it to another $\mathbb{B}$ in the obvious way. The entire body of Clause (1) can then be visualised as[3]

$$C^2 \xrightarrow{\ \langle \mathrm{id}, p_1 \rangle\ } C^2 \times C \xrightarrow{\ \langle \mathsf{Q}, \mathsf{R} \rangle\ } \mathbb{B}^2 \xrightarrow{\ \langle \mathrm{id}, \neg \rangle\ } \mathbb{B}^2 \xrightarrow{\ \wedge\ } \mathbb{B}.$$

Applying the same reasoning to $\mathsf{P}(X, a)$ gives us two maps with the same domain and codomain:

$$\begin{array}{ccccccccc} C^2 & \xrightarrow{\langle \mathrm{id}, p_1 \rangle} & C^2 \times C & \xrightarrow{\langle \mathsf{Q}, \mathsf{R} \rangle} & \mathbb{B}^2 & \xrightarrow{\langle \mathrm{id}, \neg \rangle} & \mathbb{B}^2 & \xrightarrow{\ \wedge\ } & \mathbb{B} \\ & \searrow^{(x,y) \mapsto (x,a)} & & & & & & \nearrow_{\mathsf{P}} & \\ & & & & C^2. & & & & \end{array} \tag{2}$$

The only semantic connection between the two maps, however, is that of implication: if the top path from C^2 to $\mathbb{B}$ leads to $\top$, then so should the bottom path.

Example 1 (continued)**.** Applying the logic program $\mathcal{L}$ to the KB

$$\Delta = \{\mathsf{Q}(a, a), \mathsf{Q}(b, c), \mathsf{Q}(c, c), \mathsf{R}(b)\} \in \mathcal{KB}(P_1, C)$$

gives us

$$\mathcal{L}(\Delta) = \{\mathsf{P}(a, a), \mathsf{P}(c, a)\} \in \mathcal{KB}(P_2, C).$$

[1] While the definition implies that A^∞ is infinite, in practice, it is enough to consider a truncated (finite) version of A^∞. More generally, we implicitly assume that all sets (of constants, predicates, etc.) are finite.

[2] To make examples and computations simpler, we add an additional requirement that there must be at least one projection.

[3] Note that while $\mathsf{Q}(X, Y)$ and $\mathsf{R}(X)$ have different arities, in the context of a larger formula, they both have C^2 as the domain.

Representation of an atom			Realisation function
In a program	Diagrammatic	Algebraic	
$\mathsf{P}(X, a)$	$C^2 \xrightarrow{\rho} C^2 \xrightarrow{\mathsf{P}} \mathbb{B}$	$\mathsf{P} \circ \rho$	$\rho(x, y) = (x, a)$
$\mathsf{Q}(X, Y)$	$C^2 \xrightarrow{\mathrm{id}} C^2 \xrightarrow{\mathsf{Q}} \mathbb{B}$	$\mathsf{Q} \circ \mathrm{id}$ (or just Q)	$\mathrm{id}(x, y) = (x, y)$
$\mathsf{R}(X)$	$C^2 \xrightarrow{p_1} C \xrightarrow{\mathsf{R}} \mathbb{B}$	$\mathsf{R} \circ p_1$	$p_1(x, y) = x$

Table 1: A summary of representations of atoms from Example 1 and the associated realisation functions

However, we ought to note that there are some clauses that cannot be represented in the described manner. Assuming *negation as failure* [Clark, 1977], these are clauses that have a variable which only appears in negative literals (and not in positive literals or the head atom). For example,

$$\mathsf{S}(X) \leftarrow \neg\mathsf{T}(X, Y), \tag{3}$$

while a perfectly valid clause, cannot be represented in a manner similar to Diagram (2). However, we *can* represent a clause equivalent to Clause (3) with Y instantiated with every possible value, i.e.,

$$\mathsf{S}(X) \leftarrow \bigwedge_{y \in C} \neg\mathsf{T}(X, y).$$

The number of values y required for this clause can also be significantly reduced by only considering constants that appear as the second argument to T. We end the section by defining instantiation.

Definition 8. Let Δ be a KB with its set of constants C, and let P/n be a predicate in Δ. Let $a \in C^n$ be a tuple of constants. We say that a *instantiates* P (and write $\Delta \models \mathsf{P}(a)$ or $\mathsf{P}(a) = \top$) if the fact $\mathsf{P}(a)$ is in Δ.

Let A be an atom acting on n variables $(X_i)_{i=1}^n$. We say that $a = (a_1, \ldots, a_n) \in C^n$ *instantiates* A (and write $\Delta \models \mathsf{A}(a)$ or $\mathsf{A}(a) = \top$) if $\mathsf{A}[X_1/a_1, \ldots, X_n/a_n]$ (i.e., the atom A with variables replaced with their corresponding constants) is in Δ. This definition can be further extended to literals and formulas using the usual interpretations of negation and conjunction.

4 Equivalence in Knowledge Bases

Definition 9. Let Δ be a KB with its set of constants C. Let n be a positive integer, and let $a, b \in C^n$ be two tuples of constants. Then a and b are *equivalent*[4] if

$$(\mathsf{P} \circ \rho)(a) = (\mathsf{P} \circ \rho)(b) \tag{4}$$

for all atoms $\mathsf{P} \circ \rho$ acting on n variables in Δ. Let $\sim$ denote this equivalence relation. We can also extend this to an equivalence relation for C^∞ by adding that tuples of different lengths are never equivalent. Finally, a *projection* is a map $\pi : C^n \to C^n/\!\sim$ such that $\pi(a) = [a]$ for any $a \in C^n$ (for $n = 1, \ldots, \infty$).

Informally, two tuples of constants are equivalent if they have the same length, and, given any fact in the KB, we can replace any combination of constants from one tuple with the corresponding constants from the other tuple and get another fact in the KB.

Example 2. Let Δ_1 and Δ_2 be KBs defined as follows:

$$\begin{aligned}\Delta_1 \coloneqq \{&\mathsf{Husband}(\mathit{joffrey}, \mathit{margaery}), \\ &\mathsf{Husband}(\mathit{tommen}, \mathit{margaery}), \\ &\mathsf{Husband}(\mathit{renly}, \mathit{margaery}), \\ &\mathsf{Parent}(\mathit{cersei}, \mathit{joffrey}), \mathsf{Parent}(\mathit{cersei}, \mathit{myrcella}), \\ &\mathsf{Parent}(\mathit{cersei}, \mathit{tommen}), \mathsf{Parent}(\mathit{tywin}, \mathit{cersei})\},\end{aligned} \tag{5}$$

$$\begin{aligned}\Delta_2 \coloneqq \{&\mathsf{Female}(\mathit{cersei}), \mathsf{Female}(\mathit{margaery}), \\ &\mathsf{Female}(\mathit{myrcella})\}.\end{aligned} \tag{6}$$

Let C be the set of all constants mentioned in Eqs. (5) and (6), and let $\sim$ and $\approx$ be the equivalence relations of Δ_1 and Δ_2, respectively. Finally, let $\pi_1 : C^\infty \to C^\infty/\!\sim$ and $\pi_2 : C^\infty \to C^\infty/\!\approx$ be the respective projections of $\sim$ and $\approx$.

To efficiently identify equivalence classes, we can look at pairs of constants that appear as arguments at the same position of the same predicate. For example, $\mathit{joffrey}$ and tommen both appear as the first argument to the predicate $\mathsf{Husband}$. We then look for a fact in Δ_1 that would contradict their equivalence, e.g., an atom with constant $\mathit{joffrey}$ such that replacing $\mathit{joffrey}$ with tommen results in a fact not in Δ_1. In this case, there is no such fact, so we have that

$$\mathit{joffrey} \sim \mathit{tommen}. \tag{7}$$

If we take $\mathit{joffrey}$ and renly, for example, their equivalence is contradicted by the fact $\mathsf{Parent}(\mathit{cercei}, \mathit{joffrey})$. Equivalence (7) is, indeed, the only equivalence of individual constants in Δ_1. The situation with Δ_2 is much more straightforward where $C/\!\approx$ consists of two equivalence classes:

$$\pi_2^{-1}(c) = \{\mathit{cersei}, \mathit{margaery}, \mathit{myrcella}\}$$

and $\pi_2^{-1}(d) = C \setminus \pi_2^{-1}(c)$.

In the rest of this section, we will develop a few propositions, some of which are interesting on their own, and some of which will be used in Section 5. Let n, m be positive integers, and let Δ be a KB with its set of constants C, equivalence relation $\sim$, and its projection $\pi : C^\infty \to C^\infty/\!\sim$.

Lemma 2. *Any realisation function can be represented as a composition of four types of* elementary *functions:*

duplication: $(x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_i, x_i, \ldots, x_n)$,
omission: $(x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$,
permutation: $(x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_{i+1}, x_i, \ldots, x_n)$,
insertion: $(x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_n, a)$.

[4] One can easily check that this is indeed an equivalence relation.

Proposition 1. *Let $\sigma : C^n \to C^m$ be a realisation function. Then, for any $a, b \in C^n$, if $a \sim b$, then*[5] *$\sigma(a) \sim \sigma(b)$.*

Proof sketch. For any predicate P/l in Δ and any elementary function $\sigma : C^n \to C^m$ from Lemma 2, one can construct the set of all possible realisation functions $\rho' : C^m \to C^l$ such that

$$(\mathsf{P} \circ \rho' \circ \sigma)(a) = (\mathsf{P} \circ \rho' \circ \sigma)(b)$$

from the set of all realisation functions $\rho : C^n \to C^l$ such that $(\mathsf{P} \circ \rho)(a) = (\mathsf{P} \circ \rho)(b)$. □

Remark. Note that the converse statement is not necessarily true because the omission operation can destroy crucial information. For example, if $(a, b) \sim (a', b')$, then $a \sim a'$, but if $a \sim a'$, it may or may not be the case that $(a, b) \sim (a', b')$.

Proposition 2. *Let $c \in C^n/\!\sim$ be an equivalence class of n-tuples for $n > 1$. Then, for any positive integer $l < n$, there exist unique equivalence classes $d \in C^l/\!\sim$ and $e \in C^{n-l}/\!\sim$ such that*

$$\pi^{-1}(c) = \pi^{-1}(d) \times \pi^{-1}(e).$$

Proof. Let $a \in \pi^{-1}(c)$ be arbitrary, and let $\rho : C^n \to C^l$ and $\sigma : C^n \to C^{n-l}$ be realisation functions composed purely of omission operations such that $a = (\rho(a), \sigma(a))$. Then set $d \coloneqq [\rho(a)]$ and $e \coloneqq [\sigma(a)]$. We will first show that

$$\pi^{-1}(c) \subseteq \pi^{-1}(d) \times \pi^{-1}(e).$$

Clearly $a \in \pi^{-1}(d) \times \pi^{-1}(e)$. For any other $b \sim a$, by Proposition 1 we have that $\rho(b) \sim \rho(a)$ and $\sigma(b) \sim \sigma(a)$, so $\rho(b) \in \pi^{-1}(d)$ and $\sigma(b) \in \pi^{-1}(e)$, so $b \in \pi^{-1}(d) \times \pi^{-1}(e)$. Finally, note that d and e are unique by definition, as a single element such as $\rho(a)$ cannot belong to multiple equivalence classes.

Conversely, let $r, s \in C^\infty$ be such that $r \sim \rho(a)$ and $s \sim \sigma(a)$. Then $(r, s) \in \pi^{-1}(d) \times \pi^{-1}(e)$, so we just need to show that $(r, s) \sim a$. Let $\mathsf{P} \circ \tau$ be any atom in Δ, and let $\upsilon(x) = (x, \sigma(a))$ be a realisation function composed solely of insertions. Then

$$\begin{aligned}(\mathsf{P} \circ \tau)(a) &= (\mathsf{P} \circ \tau)(\rho(a), \sigma(a)) = (\mathsf{P} \circ \tau \circ \upsilon)(\rho(a)) \\ &= (\mathsf{P} \circ \tau \circ \upsilon)(r) = (\mathsf{P} \circ \tau)(r, \sigma(a)) \\ &= \cdots = (\mathsf{P} \circ \tau)(r, s),\end{aligned}$$

where the skipped steps replace $\sigma(a)$ with s the same way $\rho(a)$ was replaced with r. □

In other words, Proposition 1 says that each equivalence class of tuples is a Cartesian product of 'smaller' equivalence classes. Despite this observation, it remains convenient to reason about equivalences of tuples in many occasions. Having defined constant equivalence and outlined some of the key properties, we end the section with two lemmas that will be useful in the next section.

[5]Formally, this means that realisation functions are morphisms from $\sim$ to itself.

Lemma 3. *Let $\pi : C^n \to C^n/\!\sim$ be the projection map of $\sim$. For any atom $\mathsf{P} \circ \rho$ acting on n variables, we can find a collection of equivalence classes $(c_i)_{i \in I}$ in $C^n/\!\sim$ such that the subset of C^n instantiating $\mathsf{P} \circ \rho$ can be expressed as*

$$\bigcup_{i \in I} \pi^{-1}(c_i).$$

Lemma 4. *Let S be a set of literals acting on n variables, and let F be a formula defined by*

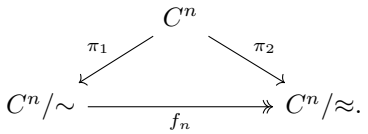
$$\mathsf{F} \coloneqq \bigwedge_{\mathsf{L} \in S} \mathsf{L}.$$

Then, for any $a, b \in C^n$, if $\Delta \models \mathsf{F}(a)$ and $a \sim b$, then $\Delta \models \mathsf{F}(b)$.

5 Refinements and Logic Programs

Now that we have developed the basic tools for reasoning about equivalences in KBs, we can build up to Theorem 2 via Propositions 3 and 4 where we represent each refinement relationship by a map between quotient sets and consider how each equivalence class can be captured with a formula.

Proposition 3. *Let Δ be a KB with its set of constants C and two equivalence relations $\sim$, $\approx$ (as defined in Definition 9) with their respective projections π_1 and π_2. Then $\approx$ is coarser than $\sim$ if and only if, for every positive integer n, there is a map $f_n : C^n/\!\sim \to C^n/\!\approx$ that makes the following diagram commute:*



We can also extend this to $f : C^\infty/\!\sim \to C^\infty/\!\approx$ by considering C^∞ as a disjoint union, i.e.,

$$C^\infty = \coprod_{n=1}^{\infty} C^n.$$

Then f is just a coproduct, i.e., $f = [f_1, f_2, \dots]$.

Proof. Given such an $f : C^\infty/\!\sim \to C^\infty/\!\approx$,

$$\begin{aligned}a \sim b &\iff \pi_1(a) = \pi_1(b) \\ &\implies (f \circ \pi_1)(a) = (f \circ \pi_1)(b) \\ &\iff \pi_2(a) = \pi_2(b) \iff a \approx b.\end{aligned}$$

Conversely, suppose that $\approx$ is coarser than $\sim$, and let n be an arbitrary positive integer. We need to define $f_n : C^n/\!\sim \to C^n/\!\approx$ such that

$$(f_n \circ \pi_1)(c) = \pi_2(c)$$

for all $c \in C^n$. Let $c \in C^n/\!\sim$ be arbitrary. By Theorem 1 and Lemma 1, there is a unique $d \in C^n/\!\approx$ such that $\pi_1^{-1}(c) \subseteq \pi_2^{-1}(d)$, so we can set $f_n(c) \coloneqq d$. □

Remark. Note that f_n must be surjective because otherwise there would be a nonempty equivalence class defined by $\approx$ contradicting the commutativity of the triangle.

Algorithm 1: Capturing an equivalence class

Data:

- a KB Δ with:
  - its set of constants C,
  - and its equivalence relation $\sim$,
- and an n-tuple of constants a.

Result: a set of literals S.

```
1 S ← ∅;
2 foreach atom[6] P ∘ ρ acting on n variables in Δ do
3     if Δ ⊨ (P ∘ ρ)(a) and[7] ∃b ∈ C^n s.t. b ≁ a and
        Δ ⊭ (P ∘ ρ)(b) then
4         add P ∘ ρ to S;
5     else if Δ ⊭ (P ∘ ρ)(a) and ∃b ∈ C^n s.t. b ≁ a
        and Δ ⊨ (P ∘ ρ)(b) then
6         add ¬P ∘ ρ to S;
```

Example 2 (continued)**.** As $\{joffrey, tommen\} \subset \pi_2^{-1}(d)$, $\approx$ is coarser than $\sim$. We can then define $f_1 : C/{\sim} \to C/{\approx}$ as

$$f_1(\pi_1(a)) \coloneqq c \quad \text{for } a \in \pi_2^{-1}(c)$$

and

$$f_1(\pi_1(a)) \coloneqq d \quad \text{for } a \in \pi_2^{-1}(d).$$

Note that a similar definition would not work in the opposite direction ($C/{\approx} \to C/{\sim}$) as some values in the domain would be mapped to multiple values in the codomain. One could similarly define $f_2 : C^2/{\sim} \to C^2/{\approx}$ as well.

Corollary 1. *Let C be a set of constants with two equivalence relations $\sim$ and $\approx$ and their respective projections π_1 and π_2. Furthermore, suppose that $\approx$ is coarser than $\sim$ as exemplified by $f : C^\infty/{\sim} \to C^\infty/{\approx}$. Then, for any $c \in C^\infty/{\approx}$,*

$$\pi_2^{-1}(c) = \bigcup_{c' \in f^{-1}(c)} \pi_1^{-1}(c').$$

Proof sketch. An immediate consequence of Proposition 3. □

Proposition 4. *Let Δ be a KB over a set of constants C inducing an equivalence relation $\sim$. For any positive integer n and equivalence class $c \in C^n/{\sim}$, one can construct a formula that is instantiated by c and only c.*

Proof. Let $a \in C^n$ be any n-tuple of constants such that $[a] = c$, and consider the set S of predicates composed with realisation functions generated by Algorithm 1. We claim that c and only c instantiates

$$\mathsf{R} = \bigwedge_{\mathsf{L} \in S} \mathsf{L}.$$

By the definition of S, $\Delta \models \mathsf{R}(a)$, and so Lemma 4 already tells us that every element of c instantiates R. It remains to show that nothing outside c can instantiate R.

[6] The loop terminates because the number of such atoms is finite.

[7] The second part of the condition is not necessary, but it makes the set S much smaller.

We will show that if $b \not\sim a$, then it cannot be the case that $\Delta \models \mathsf{R}(b)$ using a proof by contradiction and splitting the proof into cases. Let $b \in C^n$ be such that $\Delta \models \mathsf{R}(b)$, and suppose there is an atom $\mathsf{P} \circ \rho$ acting on n variables such that

$$(\mathsf{P} \circ \rho)(a) \neq (\mathsf{P} \circ \rho)(b). \tag{8}$$

Case 1. $\mathsf{P} \circ \rho \in S$. Then, by the definition of S, we have that $\Delta \models (\mathsf{P} \circ \rho)(a)$. Since $\Delta \models \mathsf{R}(b)$, we also have that $\Delta \models (\mathsf{P} \circ \rho)(b)$. But then

$$(\mathsf{P} \circ \rho)(a) = (\mathsf{P} \circ \rho)(b)$$

which contradicts Assumption (8).

Case 2. $\neg\mathsf{P} \circ \rho \in S$. By the same argument as in Case 1,

$$(\mathsf{P} \circ \rho)(a) = (\mathsf{P} \circ \rho)(b) = \bot$$

which also contradicts Assumption (8).

Case 3. $\mathsf{P} \circ \rho \notin S$ and $\neg\mathsf{P} \circ \rho \notin S$. In this case, we know that conditions on Lines 3 and 5 of Algorithm 1 must be false.

Case 3.1. $\Delta \models (\mathsf{P} \circ \rho)(a)$. Line 3 of the algorithm then says that for any $b' \in C^n$, either $b' \sim a$ or $\Delta \models (\mathsf{P} \circ \rho)(b')$. Since $b \not\sim a$, we must have that $\Delta \models (\mathsf{P} \circ \rho)(b)$. But then

$$(\mathsf{P} \circ \rho)(a) = (\mathsf{P} \circ \rho)(b) = \top$$

which contradicts Assumption (8).

Case 3.2. $\Delta \not\models (\mathsf{P} \circ \rho)(a)$. Similarly, Line 5 says that for any $b' \in C^n$, either $b' \sim a$ or $\Delta \models (\mathsf{P} \circ \rho)(b')$, and a similar argument ensures a contradiction.

□

Theorem 2. *Let C be a set of constants, and let P_1, P_2 be two sets of predicates. Let $\Delta_1 \in \mathcal{KB}(P_1, C)$ and $\Delta_2 \in \mathcal{KB}(P_2, C)$ be two KBs, and let $\sim$, $\approx$ be their respective equivalence relations on C^∞. Then there is a logic program $\mathcal{L} : \mathcal{KB}(P_1, C) \to \mathcal{KB}(P_2, C)$ such that $\mathcal{L}(\Delta_1) = \Delta_2$ if and only if $\approx$ is coarser than $\sim$.*

Proof. Suppose that $\approx$ is coarser than $\sim$ as exemplified by $f : C^\infty/{\sim} \to C^\infty/{\approx}$. Let P/n be an arbitrary predicate in P_2. Then, by Lemma 3, there is a collection of equivalence classes $(c_i)_{i \in I}$ in $C^n/{\approx}$ such that the tuples of constants instantiating P can be expressed as

$$P = \bigcup_{i \in I} \pi_2^{-1}(c_i).$$

Then, by Corollary 1,

$$P = \bigcup_{i \in I} \bigcup_{c_i' \in f^{-1}(c_i)} \pi_1^{-1}(c_i').$$

For every such $\pi_1^{-1}(c_i')$, we can construct a clause with P as the head and the formula given by Proposition 4 as the body. These clauses collectively define P to be instantiated by precisely the required tuples of constants. Repeating the process for other predicates in P_2 produces a logic program $\mathcal{L}$ such that $\mathcal{L}(\Delta_1) = \Delta_2$.

Conversely, suppose there is a logic program $\mathcal{L} : \mathcal{KB}(P_1, C) \to \mathcal{KB}(P_2, C)$ such that $\mathcal{L}(\Delta_1) = \Delta_2$. Let n be a positive integer, and consider two tuples of constants $a, b \in C^n$ such that $a \sim b$. We want to show that $a \approx b$. Let $\mathsf{P} \circ \rho$ be an atom acting on n variables in Δ_2 such that $\Delta_2 \models (\mathsf{P} \circ \rho)(a)$, and let m be the arity of P. Because of symmetry, it is enough to show that $\Delta_2 \models (\mathsf{P} \circ \rho)(b)$. If $\Delta_2 \models (\mathsf{P} \circ \rho)(a)$, then there must be a clause in $\mathcal{L}$ that generated this fact. Let

$$\mathsf{P} \circ \sigma \leftarrow \bigwedge_{i=1}^{L} \mathsf{L}_i \tag{9}$$

be such a clause. Here, $(\mathsf{L}_i)_{i=1}^{L}$ are literals in Δ_1, and $\mathsf{P} \circ \sigma$ is an atom in Δ_2. Suppose that Clause (9) acts on l variables, for some positive integer l that is in no way related to n or m. The input to this clause that generates $(\mathsf{P} \circ \rho)(a)$ can be represented as $\tau(a)$ for a realisation function with constants $\tau : C^n \to C^l$ such that $\rho = \sigma \circ \tau$. Indeed, only one restriction is imposed by this choice, namely that at least one element of a is assigned to a variable in Clause (9). Clause (9) can then be represented by the following diagram:

$$\begin{array}{ccccccc} C^n & \xrightarrow{\tau} & C^l & \xrightarrow{\prod_{i=1}^{L} \mathsf{L}_i} & \mathbb{B}^L & \xrightarrow{\wedge} & \mathbb{B} \\ & \searrow^{\rho} & \downarrow{\scriptstyle \sigma} & & & \nearrow_{\mathsf{P}} & \\ & & C^m & & & & \end{array} \tag{10}$$

with the property that, for any $c \in C^n$, if the top path leads to $\top$, then $(\mathsf{P} \circ \sigma \circ \tau)(c) = \top$, and, since the left triangle commutes by definition, this also implies that $(\mathsf{P} \circ \rho)(c) = \top$. Therefore, it remains to show that the top path in Diagram (10) leads to $\top$ for $b \in C^n$, but this follows directly from $a \sim b$ since, for each i, $\mathsf{L}_i \circ \tau$ is just another literal in Δ_1, so b instantiates it if and only if a does. □

Example 2 (continued)**.** Given the KBs Δ_1 and Δ_2, one possible program $\mathcal{L}$ such that $\mathcal{L}(\Delta_1) = \Delta_2$—as generated by Theorem 2 and Algorithm 1—is:

$$\mathsf{Female}(X) \leftarrow \mathsf{Husband}(\mathit{joffrey}, X), \tag{11}$$

$$\mathsf{Female}(X) \leftarrow \mathsf{Parent}(X, \mathit{joffrey}),$$

$$\mathsf{Female}(X) \leftarrow \mathsf{Parent}(\mathit{cersei}, X) \wedge \neg\mathsf{Husband}(X, \mathit{margaery}). \tag{12}$$

Let us remark that the constants in Clauses (11) and (12) could be replaced with new variables to make the program more general (e.g., writing $\mathsf{Husband}(Y, X)$ instead of $\mathsf{Husband}(\mathit{joffrey}, X)$), but such literals would not be considered by Algorithm 1.

6 From Logic to Probabilities

An important extension of logic programs to reason about uncertainty takes the form of probabilistic logic programs, as represented by languages such as ProbLog [Raedt *et al.*, 2007]. Our work can be extended to probabilistic KBs with a small extension to the language. One would have to make the following changes:

- A KB now assigns a probability to each fact.
- Each clause $\mathsf{A} \leftarrow \mathsf{F}$ also has an associated probability p such that $\Pr(\mathsf{A}) = p \times \Pr(\mathsf{F})$.
- We replace $\mathbb{B}$ with $[0, 1]$, so that each predicate can now be represented by a map that assigns a probability to each tuple of constants.

This way, the definition of a logic program extends to the definition of a probabilistic logic program, and the definition of equivalence remains valid. In order to adapt Algorithm 1 to a probabilistic setting, one would have to be able to add a condition to any clause $\mathsf{A} \leftarrow \mathsf{F}$, saying that the clause is only 'activated' if $\Pr(\mathsf{F})$ is in a given interval because otherwise there would be no way to distinguish structurally equivalent facts with different probabilities, e.g., as in $\{0.5 : \mathsf{P}(a), 0.49 : \mathsf{P}(b)\}$. With this addition, we can construct clauses that only apply to tuples of constants with the same $\Pr(\mathsf{F})$. This probability can then be increased or decreased as needed by changing the probability associated with the clause and considering multiple copies of the same clause if needed[8].

Our work is related to exchangeability of ground atoms (interpreted as random variables) [Niepert and den Broeck, 2014], although the two settings are difficult to compare because in this paper we do not consider the structure (e.g., a probabilistic logic program or a Markov logic network) that generated the KB. However, by identifying two constants as equivalent, Definition 9 gives us a set of pairs of ground atoms such that each pair is guaranteed to have the same probability. By considering how equivalence of constants relates to the structure generating the KB, one should be able to show how the equivalence also induces an exchangeable decomposition, although the details are left for future work.

7 Conclusion

In this paper, we described how the equivalence relation of constants induced by a KB Δ can be used to describe Δ in a fundamental way. We also developed a new way to interpret acyclic logic programs, and identified a number of important properties of these equivalence relations. Our work proved the existence of and developed a way to construct logic programs that transform a refinement of a KB to its coarser version. In the context of auto-encoding logic programs [Dumancic *et al.*, 2019] among other cases, it would also be interesting to consider how one could efficiently find an approximate transformation back, i.e., a logic program that takes the coarser KB to something 'similar' to the original (finer) KB.

Acknowledgments

The authors would like to thank Sebastijan Dumančić for his comments. This work was supported by the EPSRC Centre for Doctoral Training in Robotics and Autonomous Systems, funded by the UK Engineering and Physical Sciences Research Council (grant EP/S023208/1).

[8] In an extended report / journal paper, we will expand on these observations and also report on the application of our framework for auto-encoding logic programs [Dumancic *et al.*, 2019].

References

[Belle, 2018] Vaishak Belle. Abstracting probabilistic models. *CoRR*, abs/1810.02434, 2018.

[Bourbaki, 2004] Nicolas Bourbaki. *Theory of Sets*. Springer, 2004.

[Brualdi, 1977] Richard A. Brualdi. *Introductory combinatorics*. Pearson Education India, 1977.

[Bui *et al.*, 2012] Hung B. Bui, Tuyen N. Huynh, and Rodrigo de Salvo Braz. Exact lifted inference with distinct soft evidence on every object. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. AAAI Press, 2012.

[Bui *et al.*, 2013] Hung Hai Bui, Tuyen N. Huynh, and Sebastian Riedel. Automorphism groups of graphical models and lifted variational inference. In Ann Nicholson and Padhraic Smyth, editors, *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013*. AUAI Press, 2013.

[Clark, 1977] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pages 293–322, New York, 1977. Plemum Press.

[de Salvo Braz *et al.*, 2005] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1319–1325. Professional Book Center, 2005.

[Diaconis and Freedman, 1980] Persi Diaconis and David Freedman. Finite exchangeable sequences. *The Annals of Probability*, pages 745–764, 1980.

[Dumancic *et al.*, 2019] Sebastijan Dumancic, Tias Guns, Wannes Meert, and Hendrik Blockeel. Learning relational representations with auto-encoding logic programs. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6081–6087. ijcai.org, 2019.

[Gogate and Domingos, 2010] Vibhav Gogate and Pedro M. Domingos. Exploiting logical structure in lifted probabilistic inference. In *Statistical Relational Artificial Intelligence, Papers from the 2010 AAAI Workshop, Atlanta, Georgia, USA, July 12, 2010*, volume WS-10-06 of *AAAI Workshops*. AAAI, 2010.

[Gogate and Domingos, 2016] Vibhav Gogate and Pedro M. Domingos. Probabilistic theorem proving. *Commun. ACM*, 59(7):107–115, 2016.

[Holtzen *et al.*, 2017] Steven Holtzen, Todd D. Millstein, and Guy Van den Broeck. Probabilistic program abstractions. In Gal Elidan, Kristian Kersting, and Alexander T. Ihler, editors, *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*. AUAI Press, 2017.

[Kersting, 2012] Kristian Kersting. Lifted probabilistic inference. In Luc De Raedt, Christian Bessiere, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas, editors, *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 33–38. IOS Press, 2012.

[Kingman, 1978] John F. C. Kingman. Uses of exchangeability. *The Annals of Probability*, 6(2):183–197, 1978.

[Muggleton, 1991] Stephen Muggleton. Inductive logic programming. *New Generation Comput.*, 8(4):295–318, 1991.

[Niepert and den Broeck, 2014] Mathias Niepert and Guy Van den Broeck. Tractability through exchangeability: A new perspective on efficient probabilistic inference. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2467–2475. AAAI Press, 2014.

[Niepert, 2012] Mathias Niepert. Lifted probabilistic inference: An MCMC perspective. In *Statistical Relational AI Workshop at UAI*, 2012.

[Poole, 2003] David Poole. First-order probabilistic inference. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 985–991. Morgan Kaufmann, 2003.

[Raedt *et al.*, 2007] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007.

[Raedt *et al.*, 2016] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2016.

[Ravanbakhsh *et al.*, 2017] Siamak Ravanbakhsh, Jeff G. Schneider, and Barnabás Póczos. Equivariance through parameter-sharing. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2892–2901. PMLR, 2017.

[Richardson and Domingos, 2006] Matthew Richardson and Pedro M. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

Generating Random Logic Programs Using Constraint Programming

Paulius Dilkas
University of Edinburgh, UK

Vaishak Belle
University of Edinburgh, UK
Alan Turing Institute, UK

Abstract

We present a novel approach to generating random logic programs and random probabilistic logic programs using constraint programming. The generated programs are useful in empirical testing of inference algorithms, random data generation, and program learning. This approach has a major advantage in that one can easily add additional conditions for the generated programs. As an example of this, we introduce a new constraint for predicate independence with efficient propagation and entailment algorithms, allowing one to generate programs that have a certain independence structure. To generate valid probabilistic logic programs, we also present a new constraint for negative cycle detection. Finally, we provide a combinatorial argument for correctness and describe how the parameters of the model affect the empirical difficulty of the program generation task.

1 INTRODUCTION

Unifying logic and probability is a long-standing challenge in artificial intelligence (Russell, 2015), and, in that regard, statistical relational learning (SRL) has developed into an exciting area that mixes machine learning and symbolic (logical and relational) structures. In particular, logic programs and probabilistic logic programs—including languages such as PRISM (Sato and Kameya, 1997), ICL (Poole, 1997), and ProbLog (De Raedt et al., 2007)—are promising frameworks for codifying complex SRL models. With the enhanced structure, however, inference becomes more challenging as algorithms have to correctly handle hard and soft logical constraints. At the moment, we have no precise way of evaluating and comparing different inference algorithms. Incidentally, if one were to survey the literature, one often finds that an inference algorithm is only tested on 1–4 data sets (Kimmig et al., 2011; Bruynooghe et al., 2010; Vlasselaer et al., 2015), originating from areas such as social networks, citation patterns, and biological data. But how confidently can we claim that an algorithm works well if it is only tested on a few types of problems?

About thirty years ago, SAT solving technology was dealing with a similar lack of clarity (Selman et al., 1996). This changed with the study of generation of random SAT instances against different input parameters (e.g., clause length and total number of variables) to better understand the behaviour of algorithms and their ability to solve random synthetic problems. Unfortunately, in the context of probabilistic logic programming, most current approaches to random instance generation are very restrictive, e.g., limited to clauses with only two literals (Namasivayam and Truszczynski, 2009), or to clauses of the form $a \leftarrow \neg b$ (Wen et al., 2016), although some are more expressive, e.g., defining a program only by the (maximum) number of atoms in the body and the total number of rules (Zhao and Lin, 2003).

In this work, we introduce a constraint model for generating random logic programs according to a number of user-specified parameters on the structure of the program. In fact, the same model can generate both probabilistic programs directly in the syntax of ProbLog (De Raedt et al., 2007) and non-probabilistic Prolog programs. For generated probabilistic programs to be valid, we use a custom constraint to detect negative cycles. A major advantage of our constraint-based approach is that one can easily add additional constraints to the model. To demonstrate that, we present a custom constraint with propagation and entailment algorithms that can ensure predicate independence. We also present a combinatorial argument for correctness, counting the number of programs that the model produces for various parame-

ter values. Finally, we show how the model scales when tasked with producing more complicated programs and identify the relationships between parameter values and the empirical hardness of the program generation task.

Overall, our main contributions are concerned with logic programming-based languages and frameworks, which capture a major fragment of SRL (De Raedt et al., 2016). However, since probabilistic logic programming languages are closely related to other areas of machine learning, including (imperative) probabilistic programming (De Raedt and Kimmig, 2015), our results can lay the foundations for exploring broader questions on generating models and testing algorithms in machine learning.

2 PRELIMINARIES

The basic primitives of logic programs are *constants*, *(logic) variables*, and *predicates*. Each predicate has an *arity* that defines the number of terms that it can be applied to. A *term* is either a variable or a constant, and an *atom* is a predicate of arity n applied to n terms. A *formula* is a grammatically-valid expression that connects atoms using conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). A *clause* is a pair of a *head* (which is an atom) and a *body* (which is a formula). A *(logic) program* is a multiset of clauses. Given a program $\mathscr{P}$, a *subprogram* $\mathscr{R}$ of $\mathscr{P}$ is a subset of the clauses of $\mathscr{P}$ and is denoted by $\mathscr{R} \subseteq \mathscr{P}$.

In the world of constraint satisfaction, we also have *(constraint) variables*, each with its own *domain*, whose values are restricted using *constraints*. All constraint variables in the model are integer or set variables, however, if an integer refers to a logical construct (e.g., a logical variable or a constant), we will make no distinction between the two and often use names of logical constructs to refer to the underlying integers. We say that a constraint variable is *(fully) determined* if its domain (at the given moment in the execution) has exactly one value. We will often use $\square$ as a special domain value to indicate a 'disabled' (i.e., fixed and ignored) part of the model. We write $\texttt{a}[b] \in c$ to mean that $\texttt{a}$ is an array of variables of length b such that each element of $\texttt{a}$ has domain c. Similarly, we write $\texttt{c} : \texttt{a}[b]$ to denote an array $\texttt{a}$ of length b such that each element of $\texttt{a}$ has type $\texttt{c}$. Finally, we assume that all arrays start with index zero.

2.1 PARAMETERS OF THE MODEL

We begin defining the parameters of our model by initialising sets and lists of the primitives used in constructing logic programs: a list of predicates $\mathcal{P}$, a list of their corresponding arities $\mathcal{A}$ (so $|\mathcal{A}| = |\mathcal{P}|$), a set of variables $\mathcal{V}$, and a set of constants $\mathcal{C}$. Either $\mathcal{V}$ or $\mathcal{C}$ can be empty, but we assume that $|\mathcal{C}| + |\mathcal{V}| > 0$. Similarly, the model supports zero-arity predicates but requires at least one predicate to have non-zero arity. For notational convenience, we also set $\mathcal{M}_{\mathcal{A}} \coloneqq \max \mathcal{A}$.

We also define a measure of how complicated a body of a clause can become. As each body is represented by a tree (see Section 4), we set $\mathcal{M}_{\mathcal{N}} \geq 1$ to be the maximum number of nodes in the tree representation of any clause. We also set $\mathcal{M}_{\mathcal{C}}$ to be the maximum number of clauses in a program. We must have that $\mathcal{M}_{\mathcal{C}} \geq |\mathcal{P}|$ because we require each predicate to have at least one clause that defines it. The model supports eliminating either all cycles or just negative cycles (see Section 8) and enforcing predicate independence (see Section 7), so a set of independent pairs of predicates is another parameter. Since this model can generate probabilistic as well as non-probabilistic programs, each clause is paired with a probability which is randomly selected from a given multiset (i.e., our last parameter). For generating non-probabilistic programs, one can set this list equal to $\{1\}$. Finally, we define $\mathcal{T} = \{\neg, \wedge, \vee, \top\}$ as the set of tokens that (together with atoms) form a clause. All decision variables of the model can now be divided into $2 \times \mathcal{M}_{\mathcal{C}}$ separate groups, treating the body and the head of each clause separately. We say that the variables are contained in two arrays: $\texttt{Body} : \texttt{bodies}[\mathcal{M}_{\mathcal{C}}]$ and $\texttt{Head} : \texttt{heads}[\mathcal{M}_{\mathcal{C}}]$. Since the order of the clauses does not change the meaning of the program, we can also state our first constraint:

Constraint 1. *Clauses are sorted.*

Here and henceforth, the exact ordering is immaterial: we only impose an order to eliminate permutation symmetries.

3 HEADS OF CLAUSES

Definition 1. The *head* of a clause is composed of a $\texttt{predicate} \in \mathcal{P} \cup \{\square\}$, and $\texttt{arguments}[\mathcal{M}_{\mathcal{A}}] \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$.

Here, we use $\square$ to denote either a disabled clause that we choose not to use or disabled arguments if the arity of the $\texttt{predicate}$ is less than $\mathcal{M}_{\mathcal{A}}$. The reason why we need a separate value for the latter (i.e., why it is not enough to fix disabled arguments to a single already-existing value) will become clear in Section 5.

Definition 2. The $\texttt{predicate}$'s $\texttt{arity} \in [0, \mathcal{M}_{\mathcal{A}}]$ can then be defined using the $\texttt{table}$ constraint as the arity of the $\texttt{predicate}$ if $\texttt{predicate} \in \mathcal{P}$, and zero otherwise.

Having defined arity, we can now fix the superfluous arguments:

Constraint 2. *For* $i = 0, \ldots, \mathcal{M}_{\mathcal{A}} - 1$*,*

$$\texttt{arguments}[i] = \square \iff i \geq \texttt{arity}.$$

We can also add a constraint that each predicate $\mathsf{P} \in \mathcal{P}$ should have at least one clause with P at its head:

Constraint 3. *Let*

$$P = \{h.\texttt{predicate} \mid h \in \texttt{heads}\}.$$

Then $\texttt{nValues}(P) = |\mathcal{P}|$ *if* $\texttt{count}(\square, P) = 0$ *and* $|\mathcal{P}| + 1$ *otherwise, where* $\texttt{nValues}(P)$ *counts the number of unique values in* P*.*

4 BODIES OF CLAUSES

As was briefly mentioned before, the body of a clause is represented by a tree.

Definition 3. The *body* of a clause has two parts. First, we have the $\texttt{structure}[\mathcal{M}_{\mathcal{N}}] \in [0, \mathcal{M}_{\mathcal{N}} - 1]$ array that encodes the structure of the tree using the following two rules: $\texttt{structure}[i] = i$ means that the i-th node is a root, and $\texttt{structure}[i] = j$ (for $j \neq i$) means that the i-th node's parent is node j. The second part is the array $\texttt{Node} : \texttt{values}[\mathcal{M}_{\mathcal{N}}]$ such that $\texttt{values}[i]$ holds the value of the i-th node.

We can use the `tree` constraint (Fages and Lorca, 2011) to forbid cycles in the `structure` array and simultaneously define $\texttt{numTrees} \in \{1, \ldots, \mathcal{M}_{\mathcal{N}}\}$ to count the number of trees. We will view the tree rooted at the zeroth node as the main tree and restrict all other trees to single nodes. For this to work, we need to make sure that the zeroth node is indeed a root:

Constraint 4. $\texttt{structure}[0] = 0$.

Definition 4. For convenience, we also define $\texttt{numNodes} \in \{1, \ldots, \mathcal{M}_{\mathcal{N}}\}$ to count the number of nodes in the main tree. We define it as

$$\texttt{numNodes} = \mathcal{M}_{\mathcal{N}} - \texttt{numTrees} + 1.$$

Example 1. Let $\mathcal{M}_{\mathcal{N}} = 8$. Then

$$\neg\mathsf{P}(X) \vee (\mathsf{Q}(X) \wedge \mathsf{P}(X))$$

corresponds to the tree in Fig. 1 and can be encoded as:

$$\begin{aligned} \texttt{structure} &= [0, 0, 0, \quad 1, \quad 2, \quad 2, \quad 6, 7], \\ \texttt{values} &= [\vee, \neg, \wedge, \mathsf{P}(X), \mathsf{Q}(X), \mathsf{P}(X), \top, \top], \\ \texttt{numNodes} &= 6, \\ \texttt{numTrees} &= 3. \end{aligned}$$

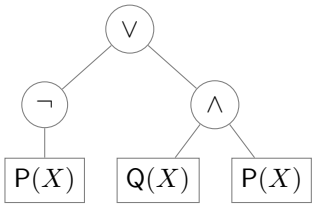


Figure 1: A tree representation of the formula from Example 1

Here, $\top$ is the value we use for the remaining one-node trees. The elements of the `values` array are nodes:

Definition 5. A *node* has a $\texttt{name} \in \mathcal{T} \cup \mathcal{P}$ and $\texttt{arguments}[\mathcal{M}_{\mathcal{A}}] \in \mathcal{V} \cup \mathcal{C} \cup \{\square\}$. The node's `arity` can then be defined analogously to Definition 2.

Furthermore, we can use Constraint 2 to again disable the extra arguments.

Example 2. Let $\mathcal{M}_{\mathcal{A}} = 2$, $X \in \mathcal{V}$, and let P be a predicate with arity 1. Then the node representing atom $\mathsf{P}(X)$ has:

$$\begin{aligned} \texttt{name} &= \mathsf{P}, \\ \texttt{arguments} &= [X, \square], \\ \texttt{arity} &= 1. \end{aligned}$$

We need to constrain the forest represented by the `structure` array together with its `values` to eliminate unnecessary symmetries and adhere to our desired format. First, we can recognise that the order of the elements in the `structure` array does not matter, i.e., the structure is only defined by how the elements link to each other, so we can add a constraint saying that:

Constraint 5. `structure` *is sorted.*

Next, since we already have a variable that counts the number of nodes in the main tree, we can fix the structure and the values of the remaining trees to some constant values:

Constraint 6. *For* $i = 1, \ldots, \mathcal{M}_{\mathcal{N}} - 1$*, if* $i \geq \texttt{numNodes}$*, then*

$$\texttt{structure}[i] = i, \quad \textit{and} \quad \texttt{values}[i].\texttt{name} = \top,$$

else $\texttt{structure}[i] < i$*.*

The second part of this constraint states that every node in the main tree except the zeroth node cannot be a root and must have its parent located to the left of itself. Next, we classify all nodes into three classes: predicate (or empty) nodes, negation nodes, and conjunction/disjunction nodes based on the number of children (zero, one, and two, respectively).

Constraint 7. *For $i = 0, \ldots, \mathcal{M}_\mathcal{N} - 1$, let C_i be the number of times i appears in the* `struture` *array with index greater than i. Then*

$$C_i = 0 \iff \texttt{values}[i].\texttt{name} \in \mathcal{P} \cup \{\top\},$$
$$C_i = 1 \iff \texttt{values}[i].\texttt{name} = \neg,$$
$$C_i > 1 \iff \texttt{values}[i].\texttt{name} \in \{\wedge, \vee\}.$$

The value $\top$ serves a twofold purpose: it is used as the fixed value for nodes outside the main tree, and, when located at the zeroth node, it can represent a clause with no body. Thus, we can say that only root nodes can have $\top$ as the value:

Constraint 8. *For $i = 0, \ldots, \mathcal{M}_\mathcal{N} - 1$,*

$$\texttt{structure}[i] \neq i \implies \texttt{values}[i].\texttt{name} \neq \top.$$

Finally, we add a way to disable a clause by setting its head predicate to $\square$:

Constraint 9. *For $i = 0, \ldots, \mathcal{M}_\mathcal{C} - 1$, if* `heads`$[i]$.`predicate` $= \square$*, then*

$$\texttt{bodies}[i].\texttt{numNodes} = 1,$$

and

$$\texttt{bodies}[i].\texttt{values}[0].\texttt{name} = \top.$$

5 VARIABLE SYMMETRIES

Given any clause, we can permute the variables in that clause without changing the meaning of the clause or the entire program. Thus, we want to fix the order of variables to eliminate unnecessary symmetries. Informally, we can say that variable X goes before variable Y if the first occurrence of X in either the head or the body of the clause is before the first occurrence of Y. Note that the constraints described in this section only make sense if $|\mathcal{V}| > 1$. Also, note that all definitions and constraints here are on a per-clause basis.

Definition 6. Let $N = \mathcal{M}_\mathcal{A} \times (\mathcal{M}_\mathcal{N} + 1)$, and let $\texttt{terms}[N] \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$ be a flattened array of all arguments in a particular clause.

Then we can use a channeling constraint to define $\texttt{occ}[|\mathcal{C}|+|\mathcal{V}|+1]$ as an array of subsets of $\{0, \ldots, N-1\}$ such that for all $i = 0, \ldots, N-1$, and $t \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$,

$$i \in \texttt{occ}[t] \iff \texttt{terms}[i] = t$$

Next, we introduce an array that, for each variable, holds the position of its first occurrence:

Definition 7. Let $\texttt{intros}[|\mathcal{V}|] \in \{0, \ldots, N\}$ be such that for $v \in \mathcal{V}$,

$$\texttt{intros}[v] = \begin{cases} 1 + \min \texttt{occ}[v] & \text{if } \texttt{occ}[v] \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Here, a value of zero means that the variable does not occur in the clause. The reason why we want to use specifically zero for this will become clear with Constraint 12. Because of this choice, the definition of `intros` shifts all indices by one. Lastly, we add the constraint that eliminates variable symmetries:

Constraint 10. `intros` *are sorted.*

In other words, we constrain the model so that the variable listed first in whatever order $\mathcal{V}$ is presented in has to occur first in our representation of a clause.

Example 3. Let $\mathcal{C} = \emptyset$, $\mathcal{V} = \{X, Y, Z\}$, $\mathcal{M}_\mathcal{A} = 2$, $\mathcal{M}_\mathcal{N} = 3$, and consider the clause

$$\mathsf{sibling}(X, Y) \leftarrow \mathsf{parent}(X, Z) \wedge \mathsf{parent}(Y, Z).$$

Then

$$\texttt{terms} = [X, Y, \square, \square, X, Z, Y, Z],$$
$$\texttt{occ} = [\{0, 4\}, \{1, 6\}, \{5, 7\}, \{2, 3\}],$$
$$\texttt{intros} = [0, 1, 5],$$

where the $\square$'s correspond to the conjunction node.

5.1 REDUNDANT CONSTRAINTS

We add a number of redundant constraints to make search more efficient. First, we can state that the positions occupied by different terms must be different:

Constraint 11. *For $u \neq v \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$,*

$$\texttt{occ}[u] \cap \texttt{occ}[v] = \emptyset.$$

The reason why we used zero to represent an unused variable is so that we could efficiently rephrase Constraint 11 for the `intros` array:

Constraint 12. `allDifferentExcept0(intros)`.

We can also add another link between `intros` and `occ` that essentially says that the smallest element of a set is an element of the set:

Constraint 13. *For $v \in \mathcal{V}$,*

$$\texttt{intros}[v] \neq 0 \iff \texttt{intros}[v] - 1 \in \texttt{occ}[v].$$

Finally, we define an auxiliary set variable to act as a set of possible values that `intros` can take:

Definition 8. Let $\texttt{potentials} \subseteq \{0, \ldots, N\}$ be such that for $v \in \mathcal{V}$, $\texttt{intros}[v] \in \texttt{potentials}$.

Using this new variable, we can add a constraint saying that non-predicate nodes in the tree representation of a clause cannot have variables as arguments:

Constraint 14. *For* $i = 0, \ldots, \mathcal{M}_\mathcal{N} - 1$*, let*

$$S = \{\mathcal{M}_\mathcal{A} \times (i+1) + j + 1 \mid j = 0, \ldots, \mathcal{M}_\mathcal{A} - 1\}.$$

If $\texttt{values}[i].\texttt{name} \notin \mathcal{P}$*, then* $\texttt{potentials} \cap S = \emptyset$.

6 COUNTING PROGRAMS

To demonstrate the correctness of the model and explain it in more detail, in this section we are going to derive combinatorial expressions for counting the number of programs with up to $\mathcal{M}_\mathcal{C}$ clauses and up to $\mathcal{M}_\mathcal{N}$ nodes per clause, and arbitrary $\mathcal{P}$, $\mathcal{A}$, $\mathcal{V}$, and $\mathcal{C}$[1]. To simplify the task, we only consider clauses without probabilities and disable (negative) cycle elimination. We also introduce the term *total arity* of a body of a clause to refer to the sum total of arities of all predicates in the body.

We will first consider clauses with gaps, i.e., without taking variables and constants into account. Let $T(n, a)$ denote the number of possible clause bodies with n nodes and total arity a. Then $T(1, a)$ is the number of predicates in $\mathcal{P}$ with arity a, and the following recursive definition can be applied for $n > 1$:

$$T(n,a) = T(n-1,a) + 2 \sum_{\substack{c_1+\cdots+c_k=n-1,\\ 2\le k\le \frac{a}{\min \mathcal{A}},\\ c_i\ge 1 \text{ for all } i}} \sum_{\substack{d_1+\cdots+d_k=a,\\ d_i\ge \min \mathcal{A} \text{ for all } i}} \prod_{i=1}^{k} T(c_i, d_i).$$

The first term here represents negation, i.e., negating a formula consumes one node but otherwise leaves the task unchanged. If the first operation is not negation, then it must be either conjunction or disjunction (hence the coefficient '2'). In the first sum, k represents the number of children of the root node, and each c_i is the number of nodes dedicated to child i. Thus, the first sum iterates over all possible ways to partition the remaining $n - 1$ nodes. Similarly, the second sum considers every possible way to partition the total arity a across the k children nodes.

[1]We checked that our model agrees with the derived combinatorial formula in close to a thousand different scenarios. The details of this empirical investigation are omitted as they are not crucial to the thrust of this paper.

We can then count the number of possible clause bodies with total arity a (and any number of nodes) as

$$C(a) = \begin{cases} 1 & \text{if } a = 0 \\ \sum_{n=1}^{\mathcal{M}_\mathcal{N}} T(n,a) & \text{otherwise.} \end{cases}$$

Here, the empty clause is considered separately.

The number of ways to select n terms is

$$P(n) = |\mathcal{C}|^n + \sum_{\substack{1\le k\le|\mathcal{V}|,\\ 0=s_0<s_1<\cdots<s_k<s_{k+1}=n+1}} \prod_{i=0}^{k} (|\mathcal{C}| + i)^{s_{i+1}-s_i-1}.$$

The first term is the number of ways select n constants. The parameter k is the number of variables used in the clause, and $s_1, \ldots, s_k$ mark the first occurrence of each variable. For each gap between any two introductions (or before the first introduction, or after the last introduction), we have $s_{i+1} - s_i - 1$ spaces to be filled with any of the $|\mathcal{C}|$ constants or any of the i already-introduced variables.

Let us order the elements of $\mathcal{P}$, and let a_i be the arity of the i-th predicate. The number of programs is then:

$$\sum_{\substack{\sum_{i=1}^{|\mathcal{P}|} h_i = n,\\ |\mathcal{P}|\le n\le \mathcal{M}_\mathcal{C},\\ h_i\ge 1 \text{ for all } i}} \prod_{i=1}^{|\mathcal{P}|} \left(\!\!\binom{\sum_{a=0}^{\mathcal{M}_\mathcal{A}\times\mathcal{M}_\mathcal{N}} C(a)P(a+a_i)}{h_i}\!\!\right),$$

where

$$\left(\!\!\binom{n}{k}\!\!\right) = \binom{n+k-1}{k}$$

counts the number of ways to select k out of n items with repetition (and without ordering). Here, we sum over all possible ways to distribute $|\mathcal{P}| \le n \le \mathcal{M}_\mathcal{C}$ clauses among $|\mathcal{P}|$ predicates so that each predicate gets at least one clause. For each predicate, we can then count the number of ways to select its clauses out of all possible clauses. The number of possible clauses can be computed by considering each possible arity a, and multiplying the number of 'unfinished' clauses $C(a)$ by the number of ways to select the required $a + a_i$ terms in the body and the head of the clause.

7 PREDICATE INDEPENDENCE

In this section, we define a notion of predicate independence as a way to constrain the probability distributions defined by the generated programs. We also describe efficient algorithms for propagation and entailment checking.

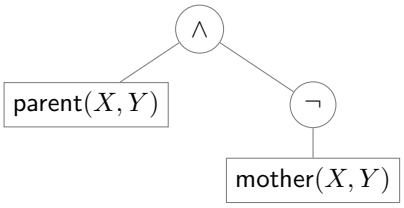


Figure 2: A tree representation of the body of Clause (1)

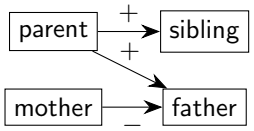


Figure 3: The predicate dependency graph of the program in Example 4. Positive edges are labelled with '+', and negative edges with '−'.

Definition 9. Let $\mathscr{P}$ be a probabilistic logic program. Its *predicate dependency graph* is a directed graph $G_{\mathscr{P}} = (V, E)$ with the set of nodes V consisting of all predicates in $\mathscr{P}$. For any two different predicates P and Q, we add an edge from P to Q if there is a clause in $\mathscr{P}$ with Q as the head and P mentioned in the body. We say that the edge is *negative* if there exists a clause with Q as the head and at least one instance of P at the body such that the path from the root to the P node in the tree representation of the clause passes through at least one negation node. Otherwise, it is *positive*. We say that $\mathscr{P}$ (or $G_{\mathscr{P}}$) has a *negative cycle* if $G_{\mathscr{P}}$ has a cycle with at least one negative edge.

Labelling the edges as positive/negative will be immaterial for predicate independence, but the same graph will play a crucial role in negative cycle detection in the next section.

Definition 10. Let P be a predicate in a program $\mathscr{P}$. The set of *dependencies* of P is the smallest set D_{P} such that $\mathsf{P} \in D_{\mathsf{P}}$, and, for every $\mathsf{Q} \in D_{\mathsf{P}}$, all direct predecessors of Q in $G_{\mathscr{P}}$ are in D_{P}.

Definition 11. Two predicates P and Q are *independent* if $D_{\mathsf{P}} \cap D_{\mathsf{Q}} = \emptyset$.

Example 4. Consider the following (fragment of a) program:

$$\mathsf{sibling}(X, Y) \leftarrow \mathsf{parent}(X, Z) \wedge \mathsf{parent}(Y, Z),$$
$$\mathsf{father}(X, Y) \leftarrow \mathsf{parent}(X, Y) \wedge \neg \mathsf{mother}(X, Y) \quad (1)$$

Its predicate dependency graph is in Fig. 3. Because of the negation in Clause (1) (as seen in Fig. 2), the edge from mother to father is negative, while the other two edges are positive.

We can now list the dependencies of each predicate:

$$D_{\mathsf{parent}} = \{\mathsf{parent}\},\ D_{\mathsf{sibling}} = \{\mathsf{sibling}, \mathsf{parent}\},$$
$$D_{\mathsf{mother}} = \{\mathsf{mother}\}, D_{\mathsf{father}} = \{\mathsf{father}, \mathsf{mother}, \mathsf{parent}\}.$$

Hence, we have two pairs of independent predicates, i.e., mother is independent of parent and sibling.

We can now add a constraint to define an adjacency matrix for the predicate dependency graph but without positivity/negativity:

Definition 12. An $|\mathcal{P}| \times |\mathcal{P}|$ adjacency matrix $\mathbf{A}$ with $\{0, 1\}$ as its domain is defined by stating that $\mathbf{A}[i][j] = 0$ if and only if, for all $k \in \{0, \ldots, \mathcal{M}_{\mathcal{C}} - 1\}$, either

$$\texttt{heads}[k].\texttt{predicate} \neq j$$

or $i \notin \{a.\texttt{name} \mid a \in \texttt{bodies}[k].\texttt{values}\}$.

Given an undetermined model, we can classify all dependencies of a predicate P into three categories based on how many of the edges on the path from the dependency to P are undetermined. In the case of zero, we call the dependency *determined*. In the case of one, we call it *almost determined*. Otherwise, it is *undetermined*. In the context of propagation and entailment algorithms, we define a *dependency* as the sum type:

$$\langle \textit{dependency} \rangle ::= \Delta(p) \mid \Upsilon(p) \mid \Gamma(p, s, t)$$

where each alternative represents a determined, undetermined, and almost determined dependency, respectively. Here, $p \in \mathcal{P}$ is the name of the predicate which is the dependency of P, and—in the case of Γ—$(s, t) \in \mathcal{P}^2$ is the one undetermined edge in $\mathbf{A}$ that prevents the dependency from being determined. For a dependency d—regardless of its exact type—we will refer to its predicate p as d.predicate. In describing the algorithms, we will use an underscore to replace any of p, s, t in situations where the name is unimportant.

Algorithm 1: Entailment for independence

Data: predicates p_1, p_2
$D \leftarrow \{(d_1, d_2) \in \texttt{deps}\,(p_1,\, 1) \times \texttt{deps}\,(p_2,\, 1)$
$\quad \mid d_1.\mathsf{predicate} = d_2.\mathsf{predicate}\}$;
if $D = \emptyset$ **then return** TRUE;
if $\exists(\Delta\ \_, \Delta\ \_) \in D$ **then return** FALSE;
return UNDEFINED;

Each entailment algorithm returns one out of three different values: TRUE if the constraint is guaranteed to hold, FALSE if the constraint is violated, and UNDEFINED if whether the constraint will be satisfied or not depends on

the future decisions made by the solver. Algorithm 1 outlines a simple entailment algorithm for the independence of two predicates p_1 and p_2. First, we separately calculate all dependencies of p_1 and p_2 and look at the set D of dependencies that p_1 and p_2 have in common. If there are none, then the predicates are clearly independent. If they have a dependency in common that is already fully determined (Δ) for both predicates, then they cannot be independent. Otherwise, we return UNDEFINED.

Algorithm 2: Propagation for independence

Data: predicates p_1, p_2; adjacency matrix $\mathbf{A}$

1 **for** $(d_1, d_2) \in \texttt{deps}(p_1, 0) \times \texttt{deps}(p_2, 0)$ *such that* d_1.predicate $= d_2$.predicate **do**

2 **if** d_1 **is** $\Delta(\_)$ **and** d_2 **is** $\Delta(\_)$ **then** `fail()`;

3 **if** d_1 **is** $\Delta(\_)$ **and** d_2 **is** $\Gamma(\_, s, t)$ **or** d_2 **is** $\Delta(\_)$ **and** d_1 **is** $\Gamma(\_, s, t)$ **then**

4 $\mathbf{A}[s][t]$.`removeValue(1)`;

Propagation algorithms have two goals: causing a contradiction (failing) in situations where the corresponding entailment algorithm would return FALSE, and eliminating values from domains of variables that are guaranteed to cause a contradiction. Algorithm 2 does the former on Line 2. Furthermore, for any dependency shared between predicates p_1 and p_2, if it is determined (Δ) for one predicate and almost determined (Γ) for another, then the edge that prevents the Γ from becoming a Δ cannot exist–Lines 3 and 4 handle this possibility.

Algorithm 3: Dependencies of a predicate

Data: adjacency matrix $\mathbf{A}$

Function `deps`$(p,$ allDependencies$)$**:**

 $D \leftarrow \{\Delta(p)\}$;

 while `true` **do**

  $D' \leftarrow \emptyset$;

  for $d \in D$ **and** $q \in \mathcal{P}$ **do**

   edge $\leftarrow \mathbf{A}[q][d.\text{predicate}] = \{1\}$;

   if edge **and** d **is** $\Delta(\_)$ **then**

    $D' \leftarrow D' \cup \{\Delta(q)\}$

   else if edge **and** d **is** $\Gamma(\_, s, t)$ **then**

    $D' \leftarrow D' \cup \{\Gamma(q, s, t)\}$;

   else if $|\mathbf{A}[q][d.\text{predicate}]| > 1$ **and** d **is** $\Delta(r)$ **then**

    $D' \leftarrow D' \cup \{\Gamma(q, q, r)\}$;

   else if $|\mathbf{A}[q][d.\text{predicate}]| > 1$ **and** allDependencies **then**

    $D' \leftarrow D' \cup \{\Upsilon(q)\}$;

  if $D' = D$ **then return** D;

  $D \leftarrow D'$;

$$\begin{array}{c} \\ \text{father} \\ \text{mother} \\ \text{parent} \\ \text{sibling} \end{array} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & \{0, 1\} & \{0, 1\} & \{0, 1\} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 4: The adjacency matrix defined using Definition 12 for Example 5

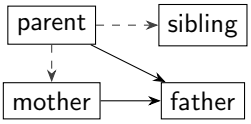


Figure 5: The predicate dependency graph that corresponds to Fig. 4. Dashed edges are undetermined—they may or may not exist.

The function `deps` in Algorithm 3 calculates D_p for any predicate p. It has two versions: `deps`$(p, 1)$ returns all dependencies, while `deps`$(p, 0)$ returns only determined and almost-determined dependencies. It starts by establishing the predicate p itself as a dependency and continues to add dependencies of dependencies until the set D stabilises. For each dependency $d \in D$, we look at the in-links of d in the predicate dependency graph. If the edge from some predicate q to d.predicate is fully determined and d is determined, then q is another determined dependency of p. If the edge is determined but d is almost determined, then q is an almost-determined dependency. The same outcome applies if d is fully determined but the edge is undetermined. Finally, if we are interested in collecting all dependencies regardless of their status, then q is a dependency of p as long as the edge from q to d.predicate is possible. Note that if there are multiple paths in the dependency graph from q to p, Algorithm 3 could include q once for each possible type (Δ, Υ, and Γ), but Algorithms 1 and 2 would still work as intended.

Example 5. Consider this partially determined (fragment of a) program:

$$\square(X, Y) \leftarrow \mathsf{parent}(X, Z) \wedge \mathsf{parent}(Y, Z),$$
$$\mathsf{father}(X, Y) \leftarrow \mathsf{parent}(X, Y) \wedge \neg\mathsf{mother}(X, Y)$$

where $\square$ indicates an unknown predicate with domain

$$D_\square = \{\mathsf{father}, \mathsf{mother}, \mathsf{parent}, \mathsf{sibling}\}.$$

The predicate dependency graph without positivity/negativity (as defined in Definition 12) is represented in Figs. 4 and 5.

Suppose we have a constraint that mother and parent must be independent. The lists of potential dependencies

for both predicates are:

$$D_{\mathsf{mother}} = \{\Delta(\mathsf{mother}), \Gamma(\mathsf{parent}, \mathsf{parent}, \mathsf{mother})\},$$
$$D_{\mathsf{parent}} = \{\Delta(\mathsf{parent})\}.$$

An entailment check at this stage would produce UNDEFINED, but propagation replaces the boxed value in Fig. 4 with zero, eliminating the potential edge from parent to mother. This also eliminates mother from $D_{\square}$, and, although some undetermined variables remain, this is enough to make Algorithm 1 return TRUE.

8 NEGATIVE CYCLES

Having no negative cycles in the predicate dependency graph is a requirement of ProbLog that makes the program well-defined (Kimmig et al., 2009). Ideally, we would like to design a constraint for negative cycles similar to the constraint for independence in the previous section. However, the difficulty with creating a propagation algorithm for negative cycles is that there seems to be no good way to extend Definition 12 so that the adjacency matrix captures positivity/negativity. Thus, we settle for an entailment algorithm with no propagation.

Algorithm 4: Entailment for negative cycles

Data: a program $\mathscr{P}$
Let $\mathscr{R} \subseteq \mathscr{P}$ be the largest subprogram of $\mathscr{P}$ with its `structure` and predicates in both body and head fully determined[2];
if `hasNegativeCycles` $(G_{\mathscr{R}})$ **then**
  return FALSE;
if $\mathscr{R} = \mathscr{P}$ **then return** TRUE;
return UNDEFINED;

The algorithm takes all clauses whose structure and predicates have been fully determined and uses them to construct a full dependency graph. In our implementation, `hasNegativeCycles` function is just a simple extension of the backtracking cycle detection algorithm that 'travels' around the graph following edges and checking if each vertex has already been visited or not. Alternatively, one could assign weights to the edges (e.g., 1 for positive and $-\infty$ for negative edges), thus reducing our negative cycle detection problem to what is typically known as the negative cycle detection problem in the literature, and use an algorithm such as Bellman-Ford (Shimbel, 1954).

If the algorithm finds a negative cycle in this fully-determined part of the program, then we must return FALSE. If there was no negative cycle and the entire program is (sufficiently) determined, then there cannot be any negative cycles. In all other cases, it is too early to tell.

[2]The arguments (whether variables or constants) are irrelevant to our definition of independence.

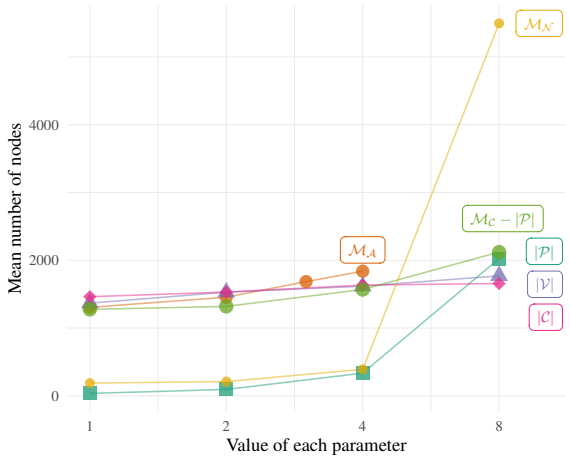


Figure 6: The mean number of nodes in the binary search tree for each value of each experimental parameter. Note that the horizontal axis is on a $\log_2$ scale.

9 EMPIRICAL PERFORMANCE

Along with constraints, variables, and their domains, two more design decisions are needed to complete the model: heuristics and restarts. By trial and error, the variable ordering heuristic was devised to eliminate sources of thrashing, i.e., situations where a contradiction is being 'fixed' by making changes that have no hope of fixing the contradiction. Thus, we partition all decision variables into an ordered list of groups, and require the values of all variables from one group to be determined before moving to the next group. Within each group, we use the 'fail first' variable ordering heuristic. The first group consists of all head predicates. Afterwards, we handle all remaining decision variables from the first clause before proceeding to the next. The decision variables within each clause are divided into: 1. the `structure` array, 2. body predicates, 3. head arguments, 4. (if $|\mathcal{V}| > 1$) the `intros` array, 5. body arguments. For instance, in the clause from Example 3, all visible parts of the clause would be decided in this order:

$$\overset{1}{\mathsf{sibling}}(\overset{3}{X}, \overset{3}{Y}) \leftarrow \overset{2}{\mathsf{parent}}(\overset{4}{X}, \overset{4}{Z}) \overset{2}{\wedge} \overset{2}{\mathsf{parent}}(\overset{4}{Y}, \overset{4}{Z}).$$

We also employ a geometric restart policy, restarting after $10, 20, 40, 80, \ldots$ contradictions.

We ran close to 400 000 experiments, investigating whether the model is efficient enough to generate

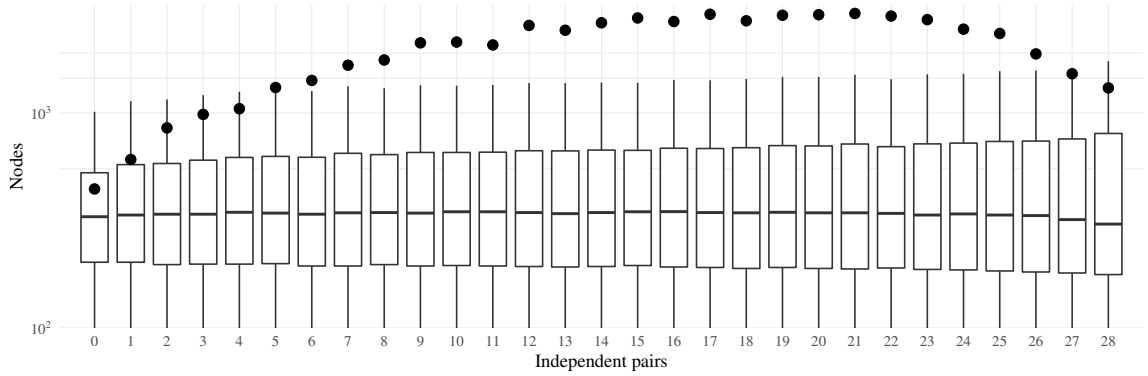


Figure 7: The distribution of the number of nodes in the binary search tree as a function of the number of independent pairs of predicates for $|\mathcal{P}| = 8$. Outliers are hidden, the dots denote mean values, and the vertical axis is on a $\log_{10}$ scale.

reasonably-sized programs and gaining insight into what parameter values make the constraint satisfaction problem harder. For these experiments, we use Choco 4.10.2 (Prud'homme et al., 2017) with Java 8. For $|\mathcal{P}|$, $|\mathcal{V}|$, $|\mathcal{C}|$, $\mathcal{M}_\mathcal{N}$, and $\mathcal{M}_\mathcal{C} - |\mathcal{P}|$ (i.e., the number of clauses in addition to the mandatory $|\mathcal{P}|$ clauses), we assign all combinations of 1, 2, 4, 8. $\mathcal{M}_\mathcal{A}$ is assigned to values 1–4. For each $|\mathcal{P}|$, we also iterate over all possible numbers of independent pairs of predicates, ranging from 0 up to $\binom{|\mathcal{P}|}{2}$. For each combination of the above-mentioned parameters, we pick ten random ways to assign arities to predicates (such that $\mathcal{M}_\mathcal{A}$ occurs at least once) and ten random combinations of independent pairs. We then run the solver with a $60\,\mathrm{s}$ timeout.

The majority ($97.7\,\%$) of runs finished in under $1\,\mathrm{s}$, while four instances timed out: all with $|\mathcal{P}| = \mathcal{M}_\mathcal{C} - |\mathcal{P}| = \mathcal{M}_\mathcal{N} = 8$ and the remaining parameters all different. This suggests that—regardless of parameter values—most of the time a solution can be identified instantaneously while occasionally a series of wrong decisions can lead the solver into a part of the search space with no solutions.

In Fig. 6, we plot how the mean number of nodes in the binary search tree grows as a function of each parameter (the plot for the median is very similar). The growth of each curve suggest how well/poorly the model scales with higher values of the parameter. From this plot, it is clear that $\mathcal{M}_\mathcal{N}$ is the limiting factor. This is because some tree structures can be impossible to fill with predicates without creating either a negative cycle or a forbidden dependency, and such trees become more common as the number of nodes increases. Likewise, a higher number of predicates complicates the situation as well.

Fig. 7 takes the data for $|\mathcal{P}| = 8$ (almost $300\,000$ observations) and shows how the number of nodes in the search tree varies with the number of independent pairs of predicates. The box plots show that the median number of nodes stays about the same while the dots (representing the means) draw an arc. This suggests a type of phase transition, but only in mean rather than median, i.e., most problems remain easy, but with some parameter values hard problems become more likely. On the one hand, with few pairs of independent predicates, one can easily find the right combination of predicates to use in each clause. On the other hand, if most predicates must be independent, this leaves fewer predicates that can be used in the body of each clause (since all of them have to be independent with the head predicate), and we can either quickly find a solution or identify that there is none.

10 CONCLUSIONS

We were able to design an efficient model for generating both logic programs and probabilistic logic programs. The model avoids unnecessary symmetries, generates valid programs, and can ensure predicate independence. Our constraint-driven approach is advantageous in that one can easily add additional conditions on the structure and properties of the program, although the main disadvantage is that there are no guarantees about the underlying probability distribution from which programs are sampled.

In addition, note that our model treats logically equivalent but syntactically different formulas as different. This is so in part because designing a constraint for logical equivalence fell outside the scope of this work and in part because in some situations one might want to enumerate all ways to express the same probability distribution or knowledge base, e.g., to investigate whether inference algorithms are robust to changes in representation.

Acknowledgements

This work was supported by the EPSRC Centre for Doctoral Training in Robotics and Autonomous Systems, funded by the UK Engineering and Physical Sciences Research Council (grant EP/S023208/1). The authors would like to thank Fazl Barez for his comments.

References

M. Bruynooghe, T. Mantadelis, A. Kimmig, B. Gutmann, J. Vennekens, G. Janssens, and L. De Raedt. ProbLog technology for inference in a probabilistic first order logic. In H. Coelho, R. Studer, and M. J. Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 719–724. IOS Press, 2010. ISBN 978-1-60750-605-8. doi: 10.3233/978-1-60750-606-5-719.

L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015. doi: 10.1007/s10994-015-5494-z.

L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In M. M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007.

L. De Raedt, K. Kersting, S. Natarajan, and D. Poole. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2016. doi: 10.2200/S00692ED1V01Y201601AIM032.

J. Fages and X. Lorca. Revisiting the tree constraint. In J. H. Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 2011. ISBN 978-3-642-23785-0. doi: 10.1007/978-3-642-23786-7_22.

A. Kimmig, B. Gutmann, and V. Santos Costa. Trading memory for answers: Towards tabling ProbLog. In *International Workshop on Statistical Relational Learning, Date: 2009/07/02-2009/07/04, Location: Leuven, Belgium*, 2009.

A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *TPLP*, 11(2-3):235–262, 2011. doi: 10.1017/S1471068410000566.

G. Namasivayam and M. Truszczynski. Simple random logic programs. In E. Erdem, F. Lin, and T. Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2009. ISBN 978-3-642-04237-9. doi: 10.1007/978-3-642-04238-6_20.

D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell.*, 94(1-2):7–56, 1997. doi: 10.1016/S0004-3702(97)00027-1.

C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. URL `http://www.choco-solver.org`.

S. J. Russell. Unifying logic and probability. *Commun. ACM*, 58(7):88–97, 2015. doi: 10.1145/2699411.

T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 1330–1339. Morgan Kaufmann, 1997.

B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artif. Intell.*, 81(1-2): 17–29, 1996. doi: 10.1016/0004-3702(95)00045-3.

A. Shimbel. Structure in communication nets. In *Proceedings of the symposium on information networks*, pages 119–203. Polytechnic Institute of Brooklyn, 1954.

J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt. Anytime inference in probabilistic logic programs with Tp-compilation. In Q. Yang and M. J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1852–1858. AAAI Press, 2015. ISBN 978-1-57735-738-4.

L. Wen, K. Wang, Y. Shen, and F. Lin. A model for phase transition of random answer-set programs. *ACM Trans. Comput. Log.*, 17(3):22:1–22:34, 2016. doi: 10.1145/2926791.

Y. Zhao and F. Lin. Answer set programming phase transition: A study on randomly generated programs. In C. Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2003. ISBN 3-540-20642-6. doi: 10.1007/978-3-540-24599-5_17.

A EXAMPLE PROGRAMS

In this appendix, we provide examples of probabilistic logic programs generated by various combinations of parameters. In all cases, we use $\{0.1, 0.2, \ldots, 0.9, 1, 1, 1, 1, 1\}$ as the multiset of probabilities. Each clause is written on a separate line and ends with a full stop. The head and the body of each clause are separated with `:-` (instead of $\leftarrow$). The probability of each clause is prepended to the clause, using `::` as a separator. Probabilities equal to one and empty bodies of clauses can be omitted. Conjunction, disjunction, and negation are denoted by commas, semicolons, and '`\+`', respectively. Parentheses are used to demonstrate precedence, although many of them are redundant.

By setting $\mathcal{P} = [\mathtt{p}]$, $\mathcal{A} = [1]$, $\mathcal{V} = \{\mathtt{X}\}$, $\mathcal{C} = \emptyset$, $\mathcal{M}_\mathcal{N} = 4$, and $\mathcal{M}_\mathcal{C} = 1$, we get fifteen one-line programs, six of which are without negative cycles (as highlighted below). Only the last program has no cycles at all.

1. `0.5 :: p(X) :- (\+(p(X))), (p(X)).`
2. `0.8 :: p(X) :- (\+(p(X))); (p(X)).`
3. `0.8 :: p(X) :- (p(X)); (p(X)).`
4. `0.7 :: p(X) :- (p(X)), (p(X)).`
5. `0.6 :: p(X) :- (p(X)), (\+(p(X))).`
6. `p(X) :- (p(X)); (\+(p(X))).`
7. `0.1 :: p(X) :- (p(X)); (p(X)); (p(X))`
8. `0.8 :: p(X) :- (p(X)), (p(X)), (p(X))`
9. `p(X) :- \+(p(X)).`
10. `0.1 :: p(X) :- \+(\+(p(X))).`
11. `p(X) :- \+((p(X)); (p(X))).`
12. `0.4 :: p(X) :- \+((p(X)), (p(X))).`
13. `0.4 :: p(X) :- \+(\+(\+(p(X)))).`
14. `0.7 :: p(X) :- p(X).`
15. `p(X).`

Note that:

- A program such as Program 14, because of its cyclic definition, defines a predicate that has probability zero across all constants. This can more easily be seen as solving equation $0.7x = x$.
- Programs 10 and 14 are not equivalent (i.e., double negation does not cancel out) because Program 10 has a negative cycle and is thus considered to be ill-defined.

To demonstrate variable symmetry reduction in action, we set $\mathcal{P} = [\mathtt{p}]$, $\mathcal{A} = [3]$, $\mathcal{V} = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$, $\mathcal{C} = \emptyset$, $\mathcal{M}_\mathcal{N} = 1$, $\mathcal{M}_\mathcal{C} = 1$, and forbid all cycles. This gives us the following five programs:

- `0.8 :: p(Z, Z, Z).`
- `p(Y, Y, Z).`
- `p(Y, Z, Z).`
- `p(Y, Z, Y).`
- `0.1 :: p(X, Y, Z).`

This is one of many possible programs with $\mathcal{P} = [\mathtt{p}, \mathtt{q}, \mathtt{r}]$, $\mathcal{A} = [1, 2, 3]$, $\mathcal{V} = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$, $\mathcal{C} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\mathcal{M}_\mathcal{N} = 5$, $\mathcal{M}_\mathcal{C} = 5$, and without negative cycles:

```
p(b) :- \+((q(a, b)), (q(X, Y)), (q(Z, X))).
0.4 :: q(X, X) :- \+(r(Y, Z, a)).
q(X, a) :- r(Y, Y, Z).
q(X, a) :- r(Y, b, Z).
r(Y, b, Z).
```

Finally, we set $\mathcal{P} = [\mathtt{p}, \mathtt{q}, \mathtt{r}]$, $\mathcal{A} = [1, 1, 1]$, $\mathcal{V} = \emptyset$, $\mathcal{C} = \{\mathtt{a}\}$, $\mathcal{M}_\mathcal{N} = 3$, $\mathcal{M}_\mathcal{C} = 3$, forbid negative cycles, and constrain predicates `p` and `q` to be independent. The resulting search space contains thousands of programs such as:

- ```
  0.5 :: p(a) :- (p(a)); (p(a)).
  0.2 :: q(a) :- (q(a)), (q(a)).
  0.4 :: r(a) :- \+(q(a)).
  ```
- ```
  p(a) :- p(a).
  0.5 :: q(a) :- (r(a)); (q(a)).
  r(a) :- (r(a)); (r(a)).
  ```
- ```
  p(a) :- (p(a)); (p(a)).
  0.6 :: q(a) :- q(a).
  0.7 :: r(a) :- \+(q(a)).
  ```
  ```