Generating Random Logic Programs Using Constraint Programming

No Author Given

No Institute Given

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.

Keywords: Probabilistic logic programming · Statistical relational artificial intelligence.

1 Introduction

Unifying logic and probability is a long-standing challenge in artificial intelligence [13], 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 [14], ICL [11], and ProbLog [5]—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 [2, 7, 17], 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 [15]. 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 [10], or to clauses of the form $a \leftarrow \neg b$ [18], 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 [19].

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 in the syntax of ProbLog [5] and non-probabilistic Prolog programs. 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 parameter values. Finally, we show how the model scales when tasked with producing more complicated programs.

A common restriction put on both logic programs and probabilistic logic programs in order to ensure that there is a unique answer to every query is known as *stratification* [1, 9]. We extend this idea to programs that have arbitrary well-formed formulas in clause bodies (as ProbLog supports this) and describe a custom constraint that ensures stratification by checking for negative cycles in the predicate dependency graph.

Overall, our main contributions are concerned with logic programming-based languages and frameworks, which capture a major fragment of SRL [3]. However, since probabilistic logic programming languages are closely related to other areas of machine learning, including (imperative) probabilistic programming [4], 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 any well-formed 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 Sect. 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 enforcing predicate independence (see Sect. 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 according to a lexicographic increasing order*[1].

3 Heads of Clauses

We define the *head* of a clause as 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

[1] The exact order is immaterial: we only impose an order to eliminate duplicates and permutation symmetries.

to fix disabled arguments to a single already-existing value) will become clear in Sect. 5.

This `predicate` variable has a corresponding arity that depends on the value of `predicate`. We can define $\texttt{arity} \in [0, \mathcal{M}_\mathcal{A}]$ as the arity of the `predicate` if $\texttt{predicate} \in \mathcal{P}$ and zero otherwise using the table constraint [8]. This constraint uses a set of pairs of the form (p, a), where p ranges over all possible values of `predicate`, and a is either the arity of predicate p or zero. 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* $P = \{h.\texttt{predicate} \mid h \in \texttt{heads}\}$ *be a multiset. Then*

$$\texttt{nValues}(P) = \begin{cases} |\mathcal{P}| & \text{if } \texttt{count}(\square, P) = 0 \\ |\mathcal{P}| + 1 & \text{otherwise,} \end{cases}$$

where $\texttt{nValues}(P)$ *counts the number of unique values in* P*, and* $\texttt{count}(\square, P)$ *counts how many times* $\square$ *appears in* P*.*

4 Bodies of Clauses

As was briefly mentioned before, the *body* of a clause is represented by a tree. It has two parts. First, there is 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 [6] 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. $\texttt{structure}[0] = 0$.

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:

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

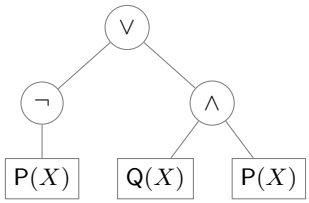


Fig. 1. A tree representation of the formula from Example 1

Here, $\top$ is the value we use for the remaining one-node trees. The elements of the `values` array are nodes. A *node* has a `name` $\in \mathcal{T} \cup \mathcal{P}$ and `arguments`$[\mathcal{M}_\mathcal{A}] \in \mathcal{V} \cup \mathcal{C} \cup \{\square\}$. The node's `arity` can then be defined in the same way as in Sect. 3. 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: `name` $= \mathsf{P}$, `arguments` $= [X, \square]$, `arity` $= 1$.

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* $i \geq$ `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* `structure` *array with index greater than* i*. Then*

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

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* $\texttt{heads}[i].\texttt{predicate} = \square$*, then*

$$\texttt{bodies}[i].\texttt{numNodes} = 1, \quad \textit{and} \quad \texttt{bodies}[i].\texttt{values}[0].\texttt{name} = \top.$$

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 1. *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* $t \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$*,*

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

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

Definition 2. *Let* $\texttt{intros}[|\mathcal{V}|] \in \{0, \ldots, N\}$ *be such that for* $v \in \mathcal{V}$*,*

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

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

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

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* $v \in \mathcal{V}$,

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

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

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

[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.

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:

$$\sum_{\substack{\sum_{i=1}^{|\mathcal{P}|} h_i = n, \\ |\mathcal{P}| \leq n \leq \mathcal{M}_\mathcal{C}, \\ h_i \geq 1 \text{ for all } i}} \prod_{i=1}^{|\mathcal{P}|} \left(\left(\begin{matrix} \sum_{a=0}^{\mathcal{M}_\mathcal{A} \times \mathcal{M}_\mathcal{N}} C(a) P(a + a_i) \\ h_i \end{matrix} \right) \right),$$

where

$$\left(\left(\begin{matrix} n \\ k \end{matrix} \right) \right) = \binom{n + k - 1}{k}$$

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 $|\mathcal{P}| \leq n \leq \mathcal{M}_\mathcal{C}$ 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. 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 designing a constraint for stratification in the next section. In fact, a program $\mathscr{P}$ is *stratified* if $G_\mathscr{P}$ has no negative cycles.

Definition 3. *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*. Two predicates* P *and* Q *are* independent *if* $D_\mathsf{P} \cap D_\mathsf{Q} = \emptyset$*.*

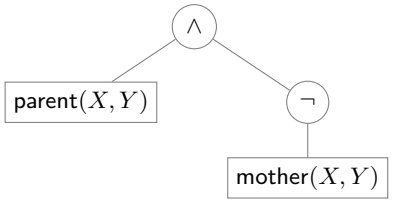


Fig. 2. A tree representation of the body of (1)

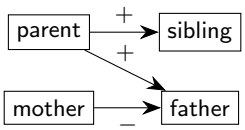


Fig. 3. The predicate dependency graph of the program in Example 4. Positive edges are labelled with '+', and negative edges with '−'.

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

$$\begin{aligned} \mathsf{sibling}(X,Y) &\leftarrow \mathsf{parent}(X,Z) \wedge \mathsf{parent}(Y,Z), \\ \mathsf{father}(X,Y) &\leftarrow \mathsf{parent}(X,Y) \wedge \neg\mathsf{mother}(X,Y). \end{aligned} \tag{1}$$

Its predicate dependency graph is in Fig. 3. Because of the negation in (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:

$$\begin{aligned} D_{\mathsf{parent}} &= \{\mathsf{parent}\}, & D_{\mathsf{sibling}} &= \{\mathsf{sibling}, \mathsf{parent}\}, \\ D_{\mathsf{mother}} &= \{\mathsf{mother}\}, & D_{\mathsf{father}} &= \{\mathsf{father}, \mathsf{mother}, \mathsf{parent}\}. \end{aligned}$$

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 4. *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

Algorithm 1: Entailment for independence

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

as $d.\mathsf{predicate}$. In describing the algorithms, we will use an underscore to replace any of p, s, t in situations where the name is unimportant.

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** $(d_1, d_2) \in \texttt{deps}(p_1,\ 0) \times \texttt{deps}(p_2,\ 0)$ *s.t.* $d_1.\mathsf{predicate} = d_2.\mathsf{predicate}$ **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.

The function `deps` in Algorithm 3 calculates D_p for any predicate p. It has two versions: $\texttt{deps}(p, 1)$ returns all dependencies, while $\texttt{deps}(p, 0)$ 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 $d.\mathsf{predicate}$ 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,

Algorithm 3: Dependencies of a predicate

```
Data: adjacency matrix A
Function deps(p, allDependencies):
    D ← {Δ(p)};
    while true do
        D′ ← ∅;
        for d ∈ D and q ∈ 𝒫 do
            edge ← A[q][d.predicate] = {1};
            if edge and d is Δ(_) then
                D′ ← D′ ∪ {Δ(q)}
            else if edge and d is Γ(_, s, t) then
                D′ ← D′ ∪ {Γ(q, s, t)};
            else if |A[q][d.predicate]| > 1 and d is Δ(r) then
                D′ ← D′ ∪ {Γ(q, q, r)};
            else if |A[q][d.predicate]| > 1 and allDependencies then
                D′ ← D′ ∪ {ϒ(q)};
        if D′ = D then return D;
        D ← D′;
```

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 $d.\mathsf{predicate}$ 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 4) is represented in Fig. 4.

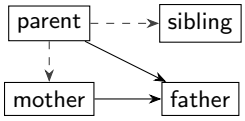
Suppose we have a constraint that mother and parent must be independent. The lists of potential dependencies for both predicates are:

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

An entailment check at this stage would produce UNDEFINED, but propagation replaces the boxed value in Fig. 4a with zero, eliminating the potential edge

$$\begin{array}{l} \mathsf{father} \\ \mathsf{mother} \\ \mathsf{parent} \\ \mathsf{sibling} \end{array} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & \boxed{\{\ 0,\ 1\ \}} & \{0,1\} & \{0,1\} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

(a) The adjacency matrix of the graph. The boxed value is the decision variable that will be propagated by Algorithm 2.



(b) A drawing of the graph. Dashed edges are undetermined—they may or may not exist.

Fig. 4. The predicate dependency graph of Example 5

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

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 4 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

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

known as the negative cycle detection problem in the literature, and use an algorithm such as Bellman-Ford [16].

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.

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 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 [12] 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 s timeout.

The majority (97.7 %) of runs finished in under 1 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. 5, 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.

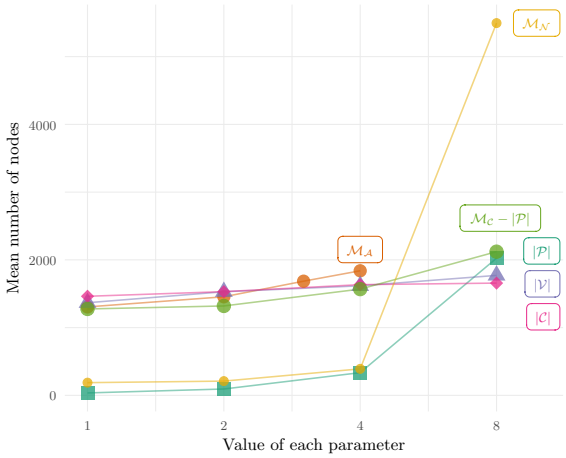


Fig. 5. The mean number of nodes in the binary search tree for each value of each experimental parameter. Note that the horizontal axis is on a $\log_2$ scale.

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.

References

1. Balbin, I., Port, G.S., Ramamohanarao, K., Meenakshi, K.: Efficient bottom-up computation of queries on stratified databases. J. Log. Program. **11**(3&4), 295–344 (1991). https://doi.org/10.1016/0743-1066(91)90030-S
2. Bruynooghe, M., Mantadelis, T., Kimmig, A., Gutmann, B., Vennekens, J., Janssens, G., De Raedt, L.: ProbLog technology for inference in a probabilistic first order logic. In: Coelho, H., Studer, R., Wooldridge, M.J. (eds.) ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 719–724. IOS Press (2010). https://doi.org/10.3233/978-1-60750-606-5-719
3. De Raedt, L., Kersting, K., Natarajan, S., Poole, D.: Statistical Relational Artificial Intelligence: Logic, Probability, and Computation. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2016). https://doi.org/10.2200/S00692ED1V01Y201601AIM032
4. De Raedt, L., Kimmig, A.: Probabilistic (logic) programming concepts. Machine Learning **100**(1), 5–47 (2015). https://doi.org/10.1007/s10994-015-5494-z
5. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 2462–2467 (2007)
6. Fages, J., Lorca, X.: Revisiting the tree constraint. In: Lee, J.H. (ed.) Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6876, pp. 271–285. Springer (2011). https://doi.org/10.1007/978-3-642-23786-7_22
7. Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. TPLP **11**(2-3), 235–262 (2011). https://doi.org/10.1017/S1471068410000566
8. Mairy, J., Deville, Y., Lecoutre, C.: The smart table constraint. In: Michel, L. (ed.) Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9075, pp. 271–287. Springer (2015). https://doi.org/10.1007/978-3-319-18008-3_19
9. Mantadelis, T., Rocha, R.: Using iterative deepening for probabilistic logic inference. In: Lierler, Y., Taha, W. (eds.) Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10137, pp. 198–213. Springer (2017). https://doi.org/10.1007/978-3-319-51676-9_14
10. Namasivayam, G., Truszczynski, M.: Simple random logic programs. In: Erdem, E., Lin, F., Schaub, T. (eds.) Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5753, pp. 223–235. Springer (2009). https://doi.org/10.1007/978-3-642-04238-6_20
11. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artif. Intell. **94**(1-2), 7–56 (1997). https://doi.org/10.1016/S0004-3702(97)00027-1
12. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017), http://www.choco-solver.org

13. Russell, S.J.: Unifying logic and probability. Commun. ACM **58**(7), 88–97 (2015). https://doi.org/10.1145/2699411
14. Sato, T., Kameya, Y.: 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. pp. 1330–1339. Morgan Kaufmann (1997)
15. Selman, B., Mitchell, D.G., Levesque, H.J.: Generating hard satisfiability problems. Artif. Intell. **81**(1-2), 17–29 (1996). https://doi.org/10.1016/0004-3702(95)00045-3
16. Shimbel, A.: Structure in communication nets. In: Proceedings of the symposium on information networks. pp. 119–203. Polytechnic Institute of Brooklyn (1954)
17. Vlasselaer, J., Van den Broeck, G., Kimmig, A., Meert, W., De Raedt, L.: Anytime inference in probabilistic logic programs with Tp-compilation. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 1852–1858. AAAI Press (2015)
18. Wen, L., Wang, K., Shen, Y., Lin, F.: A model for phase transition of random answer-set programs. ACM Trans. Comput. Log. **17**(3), 22:1–22:34 (2016). https://doi.org/10.1145/2926791
19. Zhao, Y., Lin, F.: Answer set programming phase transition: A study on randomly generated programs. In: Palamidessi, C. (ed.) Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2916, pp. 239–253. Springer (2003). https://doi.org/10.1007/978-3-540-24599-5_17