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. Finally, we provide a combinatorial argument for the correctness of the model and describe how the parameters of the model affect the empirical difficulty of the program generation task.

Keywords: Constraint programming · Probabilistic logic programming · Statistical relational learning.

1 Introduction

Unifying logic and probability is a long-standing challenge in artificial intelligence [17], 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, probabilistic logic programs—including languages such as PRISM [18], ICL [16], and ProbLog [8]—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, 12, 21], 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 [19]. This changed with the study of generating random SAT instances against different input parameters (e.g., clause length and the 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 [15], or clauses of the form $a \leftarrow \neg b$ [23], 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 [24].

In this work, we introduce a constraint-based representation for logic programs based on some simple parameters that describe the program's size, what predicates and constants it uses, etc. This representation takes the form of a constraint satisfaction problem, i.e., a set of discrete variables and restrictions on what values they can take. Every solution to this problem (as output by a constraint solver) directly translates into a logic program. One can either use random value-ordering heuristics and restarts to generate random programs or find all (sufficiently small) programs that satisfy the given requirements. In fact, the same model can generate both probabilistic programs in the syntax of ProbLog [8] and non-probabilistic Prolog programs.

A major advantage of a constraint-based approach is the ability to add additional constraints as needed, and to do that efficiently (compared to generate-and-test approaches). As an example of this, we develop a custom constraint that, given two predicates P and Q, ensures that any ground atom with predicate P is independent of any ground atom with predicate Q. In this way, we can easily regulate the independence structure of the underlying probability distribution. We also present a combinatorial argument for correctness, counting the number of programs that the model produces for various parameter values and show how the model scales when tasked with producing more complicated programs.

Overall, our main contributions are concerned with logic programming-based languages and frameworks, which capture a major fragment of SRL [6]. However, since probabilistic logic programming languages are closely related to other areas of machine learning, including (imperative) probabilistic programming [7], 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 $\neg$[1]. A *clause* is a pair of a *head* (which is an atom) and a *body* (which is a formula). A *(logic) program* is a set of clauses, and a *ProbLog program* is a set of clause-probability pairs.

In the world of constraint satisfaction, we also have *(constraint) variables*, each with a *domain*, whose values are restricted using *constraints*. All constraint

[1] One can add a couple of extra constraints to restrict the generated programs to logic programs that only allow a single disjunction with positive and negative literals.

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 $\texttt{a[}b\texttt{]} \in c$ 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 $\texttt{c} : \texttt{a[}b\texttt{]}$ 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.

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 $\mathcal{M}_{\mathcal{A}} = \max \mathcal{A}$. Next, we need a measure of how complex a body of a clause can be. 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 list—our last parameter. For generating non-probabilistic programs, one can set this list 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: $\texttt{Body} : \texttt{bodies[}\mathcal{M}_{\mathcal{C}}\texttt{]}$ and $\texttt{Head} : \texttt{heads[}\mathcal{M}_{\mathcal{C}}\texttt{]}$.

3 Heads of Clauses

We define the *head* of a clause as a $\texttt{predicate} \in \mathcal{P} \cup \{\square\}$ and $\texttt{arguments[}\mathcal{M}_{\mathcal{A}}\texttt{]} \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$. Here, we use $\square$ to denote either a disabled clause that we choose not to use or disabled arguments if the arity of the `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 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 [13]. 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 1. *For* $i = 0, \dots, \mathcal{M}_{\mathcal{A}} - 1$, $\texttt{arguments}[i] = \square \iff i \geq \texttt{arity}$.

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 2. *Let* $P = \{h.\texttt{predicate} \mid h \in \texttt{heads}\}$ *be a multiset. Then*

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

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*.*

Finally, clauses are supposed to constitute a set but with one important exception: there may be more than one disabled clause, i.e., a clause with head $\texttt{predicate} = \square$. Assuming a lexicographic order over entire clauses such that $\square > \mathsf{P}$ for all $\mathsf{P} \in \mathcal{P}$ and the head predicate is the 'first digit' of this representation, we get the following constraint.

Constraint 3. *For* $i = 1, \dots, \mathcal{M}_{\mathcal{C}} - 1$, *if* $\texttt{heads}[i].\texttt{predicate} \neq \square$*, then*

$$(\texttt{heads}[i-1], \texttt{bodies}[i-1]) < (\texttt{heads}[i], \texttt{bodies}[i]).$$

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 $\texttt{structure[}\mathcal{M}_{\mathcal{N}}\texttt{]} \in [0, \mathcal{M}_{\mathcal{N}} - 1]$ 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 $\texttt{Node : values[}\mathcal{M}_{\mathcal{N}}\texttt{]}$ such that $\texttt{values}[i]$ holds the value of the i-th node, i.e., a representation of the atom or logical operator.

We can use the `tree` constraint [9] to forbid cycles in the `structure` array and simultaneously define $\texttt{numTrees} \in \{1, \dots, \mathcal{M}_{\mathcal{N}}\}$ 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, i.e., fix $\texttt{structure}[0] = 0$. For convenience, we also define $\texttt{numNodes} \in \{1, \dots, \mathcal{M}_{\mathcal{N}}\}$ 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 $\neg\mathsf{P}(X) \vee (\mathsf{Q}(X) \wedge \mathsf{P}(X))$ can be encoded as:

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

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 $\texttt{name} \in \mathcal{T} \cup \mathcal{P}$ and $\texttt{arguments[}\mathcal{M}_{\mathcal{A}}\texttt{]} \in \mathcal{V} \cup \mathcal{C} \cup \{\square\}$. The node's `arity` can then be defined in the same way as in Sect. 3. Furthermore, we can use Constraint 1 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: $\texttt{name} = \mathsf{P}$, $\texttt{arguments} = [X, \square]$, $\texttt{arity} = 1$.

We need to constrain the forest represented by the `structure` array together with its `values` to eliminate 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 `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 4. *For $i = 1, \ldots, \mathcal{M}_{\mathcal{N}} - 1$, if $i <$ `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 5. *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*

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

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 6. *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 7. *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 Symmetry Breaking

Ideally, we want to avoid generating programs that are equivalent in the sense that they produce the same answers to all queries. Even more importantly, we want to avoid generating multiple internal representations that ultimately result in the same program. This is the purpose of *symmetry-breaking constraints*, another important benefit of which is that the constraint solving task becomes easier [22]. 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. 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 $|\mathcal{V}| > 1$ and 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* $\{0, \dots, N-1\}$ *such that for all* $i = 0, \dots, N-1$*, 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 holds the first occurrence of each variable.

Definition 2. *Let* $\texttt{intros[}|\mathcal{V}|\texttt{]} \in \{0, \dots, N\}$ *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 (this choice is motivated by subsequent constraints). This means that the definition of `intros` shifts all indices by one. Lastly, we add the variable symmetry-breaking constraint for sorting `intros`. 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 $\mathsf{sibling}(X, Y) \leftarrow \mathsf{parent}(X, Z) \land \mathsf{parent}(Y, Z)$. 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.

We end the section with several redundant constraints. First, we can state that the positions occupied by different terms must be different.

Constraint 8. *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 now use the 'all different except zero' constraint for the `intros` array. 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 9. *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. Let $\texttt{potentials} \subseteq \{0, \dots, N\}$ 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 10. *For* $i = 0, \dots, \mathcal{M}_{\mathcal{N}} - 1$, *let*

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

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 $\mathcal{C}$[2]. Being able to establish two ways to generate the same sequence of numbers (i.e., numbers of programs with certain properties and parameters) allows us to gain confidence that the constraint model accurately matches our intentions. For this section, we 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 $n > 1$:

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

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').

[2] 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.

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 $s_1, \dots, s_k$ 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:

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

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 Stratification and Independence

A ProbLog program has to be stratified in order to ensure a unique solution to every query [1, 14]. Independence is a fundamental concept in probability theory that imposes structure on the probability distribution and makes probabilistic inference easier. In the context of probabilistic logic programming languages, these two seemingly disparate concepts can be defined using the same building block, i.e., a predicate dependency graph. In this section, we define both notions and describe custom constraints meant to ensure that our generated programs are valid and have the desired independence structure. A new constraint is usually presented with two algorithms: one for *entailment* checking, i.e., checking if the (partially solved) constraint model satisfies the constraint or not, and one for *propagation*, i.e., removing values from domains that are incompatible with the current state of the model.

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. A program $\mathscr{P}$ is *stratified* if $G_{\mathscr{P}}$ has no negative cycles. Thus a simple entailment algorithm for stratification can be constructed by selecting all clauses, all predicates of which are fully determined, and looking for negative cycles in the dependency graph constructed based on those clauses using an algorithm such as Bellman-Ford.

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 $D_{\mathsf{P}} \cap D_{\mathsf{Q}} = \emptyset$.

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

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

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

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

Hence, we have two pairs of independent predicates, i.e., mother is independent of parent and $\mathsf{sibling}$.

Next, we can define a $|\mathcal{P}| \times |\mathcal{P}|$ adjacency matrix $\mathbf{A}$ with a binary domain (i.e., without positivity/negativity). It can be defined element-wise 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,

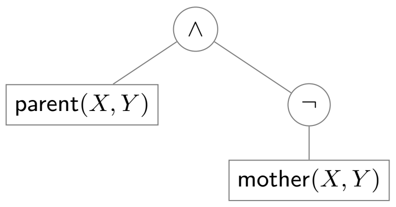

Fig. 1. A tree representation of the body of (1)

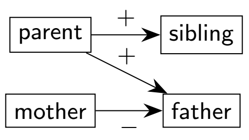

Fig. 2. The predicate dependency graph of the program in Example 4. Positive edges are labelled with '+', and negative edges with '−'.

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;
```

it is *undetermined*. In the context of propagation and entailment algorithms, we define a *dependency* as the sum type: $\langle dependency \rangle ::= \Delta(p) \mid \Upsilon(p) \mid \Gamma(p, s, t)$. 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 $d.\mathsf{predicate}$. 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.

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

Algorithm 2: Propagation for independence

```
Data: predicates $p_1$, $p_2$; adjacency matrix $\mathbf{A}$
1 for $(d_1, d_2) \in$ deps($p_1$, 0) $\times$ deps($p_2$, 0) $s.t.$ $d_1$.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);
```

Algorithm 3: Dependencies of a predicate

```
Data: adjacency matrix $\mathbf{A}$
Function deps($p$, allDeps):
    $D \leftarrow \{\Delta(p)\}$;
    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 allDeps then $D' \leftarrow D' \cup \{\Upsilon(q)\}$;
        if $D' = D$ then return $D$ else $D \leftarrow D'$;
```

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, 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:

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

where $\Box$ indicates an unknown predicate with domain

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

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

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

$$\begin{tikzcd} \boxed{\textsf{parent}} \arrow[r, dashed] \arrow[d, dashed] \arrow[dr] & \boxed{\textsf{sibling}} \\ \boxed{\textsf{mother}} \arrow[r] & \boxed{\textsf{father}} \end{tikzcd}$$

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

Fig. 3. The predicate dependency graph of Example 5

The predicate dependency graph is pictured in Fig. 3. Suppose we have a constraint that $\textsf{mother}$ and $\textsf{parent}$ must be independent. The lists of potential dependencies for both predicates are:

$$\begin{aligned} D_{\textsf{mother}} &= \{\Delta(\textsf{mother}), \Gamma(\textsf{parent}, \textsf{parent}, \textsf{mother})\}, \\ D_{\textsf{parent}} &= \{\Delta(\textsf{parent})\}. \end{aligned}$$

An entailment check at this stage would produce UNDEFINED, but propagation replaces the boxed value in Fig. 3a with zero, eliminating the potential edge from $\textsf{parent}$ to $\textsf{mother}$. This also eliminates $\textsf{mother}$ from $D_{\square}$, and this is enough to make Algorithm 1 return TRUE.

8 Experimental Results

This section presents two empirical investigations: in Sect. 8.1 we examine the scalability of our constraint model with respect to its parameters and in Sect. 8.2 we demonstrate how the model can be used to compare inference algorithms and describe their behaviour across a wide range of programs—something that has not been done before. The experiments were run on a system with Intel Core i5-8250U processor, 8 GB RAM, and Arch Linux 5.6.11-arch1-1 operating system. The constraint model was implemented in Java 8 with Choco 4.10.2. All inference algorithms are implemented in ProbLog 2.1.0.39 and were run using Python 3.8.2 with PySDD 0.2.10 and PyEDA 0.28.0. For both sets of experiments, we generate programs without negative cycles and use a 60 s timeout.

8.1 Empirical Performance of the Model

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 $10, 10 \times 1.1, 10 \times 1.1^2, \ldots$ contradictions[3]. 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 $|\mathcal{P}|$, $|\mathcal{V}|$, $|\mathcal{C}|$, $\mathcal{M}_{\mathcal{N}}$, 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.

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. 4, 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.

8.2 Experimental Comparison of Inference Algorithms

For this experiment, we consider clauses of two types: *rules* are clauses such that the head atom has at least one variable, and *facts* are clauses with no bodies and no variables. We use our constraint model to generate the rules according to the following parameter values: $|\mathcal{P}|, |\mathcal{V}|, \mathcal{M}_{\mathcal{N}} \in \{2, 4, 8\}$, $\mathcal{M}_{\mathcal{A}} \in \{1, 2, 3\}$, $\mathcal{M}_{\mathcal{C}} = |\mathcal{P}|$, $\mathcal{C} = \emptyset$. Just like before, we explore all possible numbers of independent predicate pairs. We also add a constraint that forbids empty bodies. For

[3] Restarts help overcome early mistakes in the search process but can be disabled if one wants to find all solutions, in which case search is complete regardless of the variable ordering heuristic.

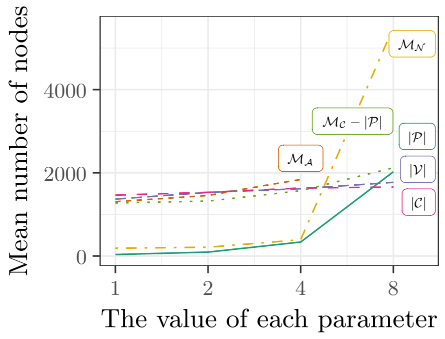

Fig. 4. The mean number of nodes in the binary search tree for each value of each experimental parameter. Note that the horizontal axis is on a $\log_2$ scale.

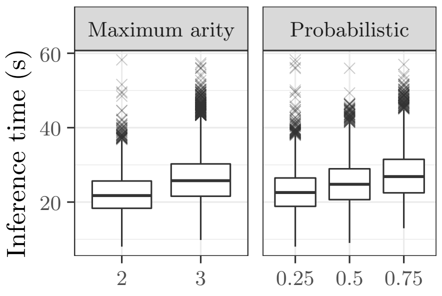

Fig. 5. Inference time for different values of $\mathcal{M}_\mathcal{A}$ and proportions of facts that are probabilistic. The total number of facts is fixed at 10^5.

both rules and facts, probabilities are uniformly sampled from $\{0.1, 0.2, \ldots, 0.9\}$. Furthermore, all rules are probabilistic, while we vary the proportion of probabilistic facts among 25 %, 50 %, and 75 %. For generating facts, we consider $|\mathcal{C}| \in \{100, 200, 400\}$ and vary the number of facts among 10^3, 10^4, and 10^5 but with one exception: the number of facts is not allowed to exceed 75 % of all possible facts with the given values of $\mathcal{P}$, $\mathcal{A}$, and $\mathcal{C}$. Facts are generated using a simple procedure that randomly selects a predicate, combines it with the right number of constants, and checks whether the generated atom is already included or not. We randomly select configurations from the description above and generate ten programs with a complete restart of the constraint solver before the generation of each program, including choosing different arities and independent pairs. Finally, we set the query of each program to a random fact not explicitly included in the program and consider six natively supported algorithms and knowledge compilation techniques: binary decision diagrams (BDDs) [3], negation normal form (NNF), deterministic decomposable NNF (d-DNNF) [5], K-Best [11], and two encodings based on sentential decision diagrams [4], one of which encodes the entire program (SDDX), while the other one encodes only the part of the program relevant to the query (SDD)[4].

Out of 11 310 generated problem instances, about 35 % were discarded because one or more algorithms were not able to ground the instance in an unambiguous way. The first observation (pictured in Fig. 6) is that the algorithms are remarkably similar, i.e., the differences in performance are small and consistent across all parameter values (including parameters not shown in the figure). Unsurprisingly, the most important predictor of inference time is the number of facts. However, after fixing the number of facts to a constant value, we can still

[4] Forward SDDs (FSDDs) and forward BDDs (FBDDs) [20, 21] are omitted because the former uses too much memory and the implementation of the latter seems to be broken at the time of writing.

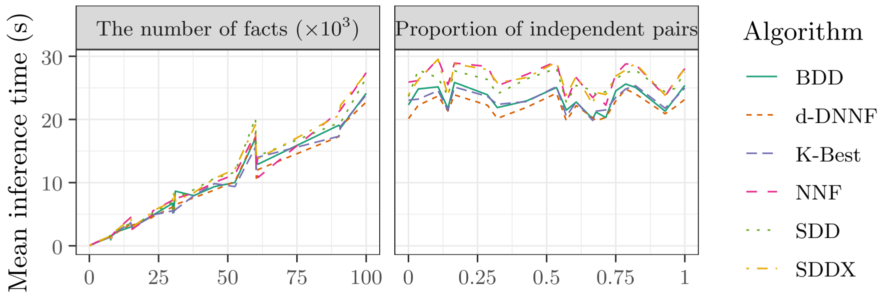

Fig. 6. Mean inference time for a range of ProbLog inference algorithms as a function of the total number of facts in the program and the proportion of independent pairs of predicates. For the second plot, the number of facts is fixed at 10^5.

observe that inference becomes harder with higher arity predicates as well as when facts are mostly probabilistic (see Fig. 5). Finally, according to Fig. 6, the independence structure of a program does not affect inference time, i.e., state-of-the-art inference algorithms—although they are supposed to [10]—do not exploit situations where separate parts of a program can be handled independently.

9 Conclusion

We presented a constraint model for generating both logic programs and probabilistic logic programs. The model avoids unnecessary symmetries, is reasonably efficient and supports additional constraints such as predicate independence. The advantages of this approach lie in customisability and speed, although the main disadvantage is that there are no guarantees about the underlying probability distribution from which programs are sampled.

Also 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.

Finally, our experimental results provide the first comparison of inference algorithms for probabilistic logic programming languages that generalises over programs, i.e., is not restricted to just a few programs and data sets. While the results did not reveal any significant differences among the algorithms, they did reveal a shared weakness, i.e., the inability to ignore the part of a program that is easily seen to be irrelevant to the given query. We hope that our work can facilitate the development of inference (as well as learning) algorithms and provide a better understanding of the strengths and weaknesses of current approaches.

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. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
4. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: Walsh, T. (ed.) IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011. pp. 819–826. IJCAI/AAAI (2011). https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-143, http://ijcai.org/proceedings/2011
5. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002). https://doi.org/10.1613/jair.989
6. 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
7. 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
8. 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)
9. 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
10. Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., De Raedt, L.: Inference in probabilistic logic programs using weighted CNF's. In: Cozman, F.G., Pfeffer, A. (eds.) UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011. pp. 211–220. AUAI Press (2011), https://dslpitt.org/uai/displayArticles.jsp?mmnu=1&smnu=1&proceeding_id=27
11. Gutmann, B., Kimmig, A., Kersting, K., De Raedt, L.: Parameter learning in probabilistic databases: A least squares approach. In: Daelemans, W., Goethals, B., Morik, K. (eds.) Machine Learning and Knowledge Discovery in Databases, European Conference, ECML/PKDD 2008, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part I. Lecture Notes in Computer Science, vol. 5211, pp. 473–488. Springer (2008). https://doi.org/10.1007/978-3-540-87479-9_49
12. 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

13. 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
14. 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
15. 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
16. 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
17. Russell, S.J.: Unifying logic and probability. Commun. ACM **58**(7), 88–97 (2015). https://doi.org/10.1145/2699411
18. 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)
19. 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
20. Tsamoura, E., Gutiérrez-Basulto, V., Kimmig, A.: Beyond the grounding bottleneck: Datalog techniques for inference in probabilistic logic programs (technical report). CoRR **abs/1911.07750** (2019), http://arxiv.org/abs/1911.07750
21. 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)
22. Walsh, T.: General symmetry breaking constraints. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4204, pp. 650–664. Springer (2006). https://doi.org/10.1007/11889205_46
23. 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
24. 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