Generating Random Logic Programs Using Constraint Programming

Abstract

We present a novel approach to generate random logic programs using constraint programming. The constraint programming approach to the problem has a major advantage in that one can easily add additional conditions for the generated programs. Some of these conditions can be expressed using constraints native to the constraint solver Choco (Prud'homme et al., 2017), while novel propagation and entailment checking algorithms have been developed for ensuring independence and/or conditional independence amongst predicates. A constraint programming approach also benefits from many years of research into efficient constraint propagation algorithms which makes it likely that our model can generate larger programs relatively quickly.

1 INTRODUCTION

Motivation:

- Empirical testing of inference algorithms.
- Generating random programs that generate random data.
- Learning: how this can be used for (targeted) learning, when (atomic) probabilities can be assigned based on counting and we can have extra constraints. A more primitive angle: generate structures, learn weights.

We say that a constraint variable is *(fully) determined* if its domain (at the given moment in the execution) has exactly one value.

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 $\mathscr{R} \subseteq \mathscr{P}$.

We will often use $\square$ as a special domain value indicating 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 $\texttt{c}[b]\ \ \texttt{a}$ to denote an array $\texttt{a}$ of length b such that each element of $\texttt{a}$ has type $\texttt{c}$. All constraint variables in the model are integer variables, but, e.g., if the integer i refers to a logical variable X, we will use i and X interchangeably. All indices start at zero.

We also use Choco 4.10.2 (Prud'homme et al., 2017). This works with both Prolog (Bratko, 2012) and ProbLog (Raedt et al., 2007).

1.1 PARAMETERS

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

We also define a measure of how complicated a body of a clause can become. As each body is represented by a tree (see Section 3), we set $\mathcal{M}_{\mathcal{N}} \geq 1$ to be the maximum number of nodes in the tree representation of any clause. We also set $\mathcal{M}_{\mathcal{C}}$ to be the maximum number of clauses in a program. We must have that $\mathcal{M}_{\mathcal{C}} \geq |\mathcal{P}|$ because we require each predicate to have at least one clause that defines it. The model supports eliminating either all cycles or just negative cycles (see Section 7) and enforcing predicate independence (see Section 6), so a list of independent pairs of predicates is another pa-

rameter. 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 (which is 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: `Body`$[\mathcal{M}_\mathcal{C}]$ `bodies` and `Head`$[\mathcal{M}_\mathcal{C}]$ `heads`. Since the order of the clauses does not change the meaning of the program, we can also state our first constraint:

Constraint 1. *Clauses are sorted.*

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

2 HEADS OF CLAUSES

Definition 1. The *head* of a clause is composed of a `predicate` $\in \mathcal{P} \cup \{\square\}$, and `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 `predicate` is less than $\mathcal{M}_\mathcal{A}$. The reason why we need a separate value for the latter (i.e., why it is not enough to fix disabled arguments to a single already-existing value) will become clear in Section 4.

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

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

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

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

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

Constraint 3. *Let*

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

Then

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

Here, $\texttt{nValues}(P)$ counts the number of unique values in P.

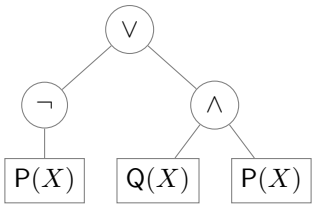

Figure 1: A tree representation of the formula from Example 1

3 BODIES OF CLAUSES

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

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

We can use the `tree` constraint (Fages and Lorca, 2011) to forbid cycles in the `structure` array and simultaneously define `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. `structure`$[0] = 0$.

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

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

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

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

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

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

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

Definition 5. A *node* has a `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 analogously to Definition 2.

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

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

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

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

Constraint 5. `structure` *is sorted.*

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

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

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

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

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

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

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

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*

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

and

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

4 VARIABLE SYMMETRIES

Given any clause, we can permute the variables in it 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 its first occurrence in either the head or the body of the clause is before the first occurrence of Y. Note that the constrains described in this section only make sense if $|\mathcal{V}| > 1$. Also note that all definitions and constraints here are on a per-clause basis.

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

Then we can use the `setsIntsChanneling` 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 7. Let $\texttt{intros}[|\mathcal{V}|] \in \{0, \ldots, N\}$ be such that for $v \in \mathcal{V}$,

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

Here, a value of zero means that the variable does not occur in the clause. We want to use specifically zero for this so that we could use Constraint 12 later. 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.

4.1 REDUNDANT CONSTRAINTS

We add a number of redundant constraints to make search more efficient. First, we can formally 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 a variable not being introduced is so that we could efficiently rephrase Constraint 11 for the `intros` array:

Constraint 12. `allDifferentExcept0(intros)`.

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

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

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

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

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

Using this new variable, we can easily 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* $\texttt{potentials} \cap S = \emptyset$.

5 COUNTING PROGRAMS

In order 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 $\mathcal{C}$. To simplify the task, we only consider clauses without probabilities and disable (negative) cycle elimination. It was experimentally confirmed that the model agrees with the combinatorial formula from this section in 985 different scenarios. The *total arity* of a body of a clause is 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'). 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

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

The first term is the number of ways select n constants. The parameter k is the number of variables used in the clause, and $s_1, \ldots, s_k$ mark the first occurrence of each

variable. For each gap between any two introductions (or before the first introduction, or after the last introduction), we have $s_{i+1} - s_i - 1$ spaces to be filled with any of the $|\mathcal{C}|$ constants or any of the i already-introduced variables.

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

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

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.

6 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.

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

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

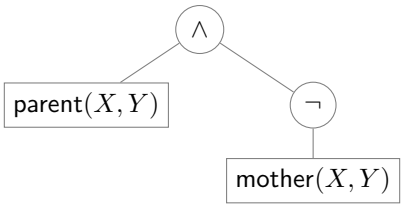

Figure 2: A tree representation of the body of Clause (1)

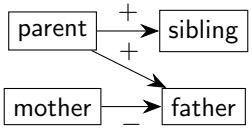

Figure 3: The predicate dependency graph of the program in Example 4. Positive edges are labelled with '+', and negative edges with '−'.

Definition 10. Let P be a predicate in a program $\mathscr{P}$. The set of *dependencies* of P is the smallest set D_P such that:

- $\mathsf{P} \in D_\mathsf{P}$,
- for every $\mathsf{Q} \in D_\mathsf{P}$, all direct predecessors of Q in $G_\mathscr{P}$ are in D_P.

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

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

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

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

We can now list the dependencies of each predicate:

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

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

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

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

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

or

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

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

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

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

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;
return UNDEFINED;

Each entailment algorithm returns one out of three different values: TRUE if the constraint is guaranteed to hold, FALSE if the constraint is violated, and UNDEFINED if whether the constraint will be satisfied or not depends on the future decisions made by the solver. Algorithm 1 outlines a simple entailment algorithm for the independence of two predicates p_1 and p_2. First, we separately calculate all dependencies of p_1 and p_2, and look at the set D of dependencies that p_1 and p_2 have in common. If there are none, then the predicates are clearly independent. If they have a dependency in common that is already fully determined (Δ) for both predicates, then they cannot be independent. Otherwise, we return UNDEFINED.

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

Algorithm 2: Propagation for independence

Data: predicates p_1, p_2; adjacency matrix $\mathbf{A}$
1 **for** $(d_1, d_2) \in \texttt{deps}\,(p_1, 0) \times \texttt{deps}\,(p_2, 0)$ *such that* $d_1.\mathsf{predicate} = d_2.\mathsf{predicate}$ **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);`

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 take care of that.

Algorithm 3: Dependencies of a predicate

Data: adjacency matrix $\mathbf{A}$
Function `deps(`p, allDependencies`)`**:**
 $D \leftarrow \{\Delta(p)\}$;
 while `true` **do**
  $D' \leftarrow \emptyset$;
  for $d \in D$ **and** $q \in \mathcal{P}$ **do**
   edge $\leftarrow \mathbf{A}[q][d.\mathsf{predicate}] = \{1\}$;
   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 $|\mathbf{A}[q][d.\mathsf{predicate}]| > 1$ **and** d **is** $\Delta(r)$ **then**
    $D' \leftarrow D' \cup \{\Gamma(q, q, r)\}$;
   else if $|\mathbf{A}[q][d.\mathsf{predicate}]| > 1$ **and** allDependencies **then**
    $D' \leftarrow D' \cup \{\Upsilon(q)\}$;
  if $D' = D$ **then return** D;
  $D \leftarrow D'$;

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 dependence 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

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

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

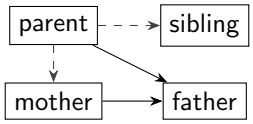

Figure 5: The predicate dependency graph that corresponds to Fig. 4. Dashed edges are undetermined—they may or may not exist.

in collecting all dependencies regardless of their status, then q is a dependency of p as long as the edge from q to d.predicate is possible. Note that if there are multiple paths in the dependency graph from q to p, Algorithm 3 could include q once for each possible type (Δ, Υ, and Γ), but Algorithms 1 and 2 would still work as intended.

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

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

where $\square$ indicates an unknown predicate with domain

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

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

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

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

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

7 NEGATIVE CYCLES

Having no negative cycles in the predicate dependency graph is a requirement for probabilistic logic programming language ProbLog (Kimmig et al., 2009), although it has been shown how the requirement can be alleviated by introducing negative probabilities (Buchman and Poole, 2017). Ideally, we would like to design a constraint for that similar to the constraint for independence in the previous section. However, the difficulty with creating a propagation algorithm for negative cycles is that there seems to be no good way to extend Definition 12 so that the adjacency matrix captures positivity/negativity. Thus, we settle for an entailment algorithm with no propagation.

Algorithm 4: Entailment for negative cycles

Data: a program $\mathscr{P}$
Let $\mathscr{R} \subseteq \mathscr{P}$ be the largest subprogram of $\mathscr{P}$ with its `structure` and predicates in both body and head fully determined[1];
if `hasNegativeCycles` $(G_{\mathscr{R}})$ **then**
  return FALSE;
if $\mathscr{R} = \mathscr{P}$ **then return** TRUE;
return UNDEFINED;

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

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

8 EMPIRICAL PERFORMANCE

Along with constraints, variables, and their domains, two more design decisions are needed to complete the model:

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

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 to fix 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 (Prud'homme et al., 2017) with Java 8 on a personal computer with an Intel Core i5-8250U processor and Arch Linux 5.4.12-arch1-1 operating system. 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 pairs of: $\mathcal{A}$ (such that $\mathcal{M}_\mathcal{A}$ occurs at least once) and the required number of independent pairs (without repetition). We then run the solver with a $60\,\mathrm{s}$ timeout.

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

In Fig. 6, we plot how the mean number of nodes in the binary search tree grows as a function of each parameter (the plot for the median is very similar). The growth of each curve suggest how well/poorly the model scales with higher values of the parameter. From this plot, it

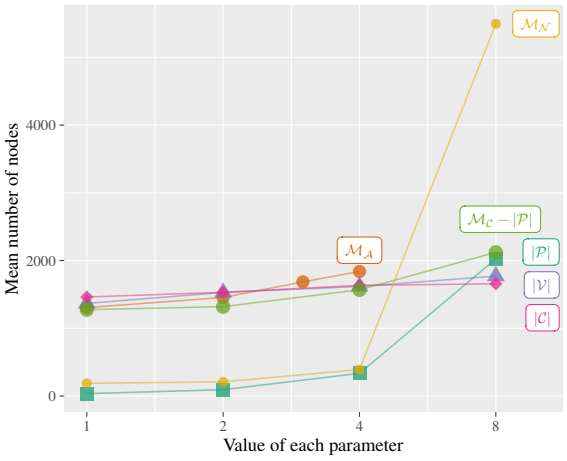

Figure 6: The mean number of nodes in the binary search tree for each value of each experimental parameter. Note that the horizontal axis is on a $\log_2$ scale.

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 dependence, and such trees become more common as the number of nodes increases.

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

9 CONCLUSIONS

One major question that was not addressed in this paper is that of logical equivalence. We eliminated permutation symmetries and constrained variable usage so that the same 'functionality' is always assigned to the same variable, but syntactically different logical formulas are always treated as being different. For example, all three of these formulas are logically equivalent but syntactically different: $\neg(\mathsf{P}(X) \vee \mathsf{Q}(X))$, $\neg\mathsf{P}(X) \wedge \neg\mathsf{Q}(X)$, and $\neg\mathsf{Q}(X) \wedge \neg\mathsf{P}(X)$. While there are many situa-

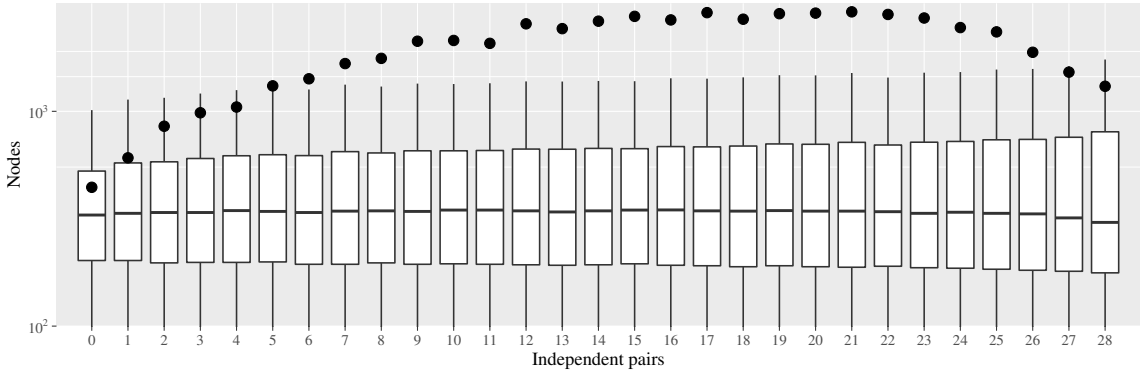

Figure 7: The distribution of the number of nodes in the binary search tree as a function of the number of independent pairs of predicates for $|\mathcal{P}| = 8$. Significant outliers are hidden, the dots denote mean values, and the vertical axis is on a $\log_{10}$ scale.

tions where one would prefer to treat logically equivalent formulas as symmetries that should be eliminated, there are also situations where we do care about enumerating different ways to express the same probability distribution or knowledge base. For instance, one could look for the 'best' logically equivalent formula, or investigate whether some formulas result in faster inference than others.

Another issue worth discussing is that of randomness. Most random models sample from a well-defined probability distribution whereas a constraint model, while potentially sufficiently random, provides no such guarantees. In future work, it would be interesting to examine the probability distribution of logic programs as generated by our model, although it is not obvious how such a probability distribution could be characterised. One example of a clear bias exhibited by our model is that of complexity over simplicity, i.e., if a tree that represents the body of a clause can have up to $\mathcal{M}_\mathcal{N}$ nodes—because the number of possible trees with n nodes grows with n—it is much more likely to have $\mathcal{M}_\mathcal{N}$ or $\mathcal{M}_\mathcal{N} - 1$ nodes than, e.g., one or two.

Acknowledgements

This work was supported by the EPSRC Centre for Doctoral Training in Robotics and Autonomous Systems, funded by the UK Engineering and Physical Sciences Research Council (grant EP/S023208/1).

References

Ivan Bratko. *Prolog Programming for Artificial Intelligence, 4th Edition*. Addison-Wesley, 2012. ISBN 978-0-3214-1746-6.

David Buchman and David Poole. Negative probabilities in probabilistic logic programs. *Int. J. Approx. Reasoning*, 83:43–59, 2017. doi: 10.1016/j.ijar.2016.10.001.

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

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

Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. URL `http://www.choco-solver.org`.

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

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