Generating Random Logic Programs Using Constraint Programming

Abstract

The Abstract paragraph should be indented 0.25 inch (1.5 picas) on both left and right-hand margins. Use 10 point type, with a vertical spacing of 11 points. **Abstract** must be centered, bold, and in point size 12. Two line spaces precede the Abstract. The Abstract must be limited to one paragraph.

1 INTRODUCTION

Motivation:

- 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.

TODO: define all the relevant terminology from logic and constraint programming.

If a predicate has arity n and it is as of yet undecided what n terms will fill those spots, we will say that the predicate has n *gaps*. 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

Parameters:

- a list of predicates $\mathcal{P}$,
- a list of their arities $\mathcal{A}$ (including zero, but assuming that at least one predicate has non-zero arity),
  - *maximum arity* $\mathcal{M}_\mathcal{A} := \max \mathcal{A}$.
- a list of variables $\mathcal{V}$,
- and a list of constants $\mathcal{C}$.
  - Each of them can be empty, but $|\mathcal{C}| + |\mathcal{V}| > 0$.
- a list of probabilities that are randomly assigned to clauses,
- option to forbid all cycles or just negative cycles,
- $\mathcal{M}_\mathcal{N} \geq 1$: maximum number of nodes in the tree representation of a clause,
- $\mathcal{M}_\mathcal{C} \geq |\mathcal{P}|$: maximum number of clauses in a program,

We also define $\mathcal{T} = \{\neg, \land, \lor, \top\}$. All decision variables of the model are contained in two arrays: $\texttt{Body}[\mathcal{M}_\mathcal{C}]\ \texttt{bodies}$ and $\texttt{Head}[\mathcal{M}_\mathcal{C}]\ \texttt{heads}$.

Constraint 1. *Clauses are sorted.*

A random selection of probabilities is added at the end in case a probabilistic program is desired. Regular logic programs can be generated by setting the list of probabilities to $\{1\}$.

2 HEADS OF CLAUSES

Definition 1. The *head* of a clause is composed of

- a $\texttt{predicate} \in \mathcal{P} \cup \{\square\}$.
- and $\texttt{arguments[}\mathcal{M}_\mathcal{A}\texttt{]} \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$

The reason why $\square$ must be a separate value will become clear in Section 4.

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

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

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

Constraint 3. *Each predicate gets at least one clause. 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.

3 BODIES OF CLAUSES

Definition 3. The body of a clause is defined by:

- $\texttt{structure[}\mathcal{M}_\mathcal{N}\texttt{]} \in [0, \mathcal{M}_\mathcal{N} - 1]$ such that:
  - $\texttt{structure}[i] = i$: the i-th node is a root.
  - $\texttt{structure}[i] = j$: the i-th node's parent is node j.
- $\texttt{Node[}\mathcal{M}_\mathcal{N}\texttt{] values}$.

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:

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

In the rest of this section, we will describe how the elements of $\texttt{values}$ are encoded and list a series of constraints that make this representation unique.

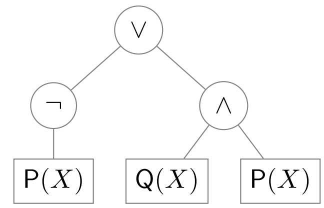

Figure 1: A tree representation of the formula from Example 1

3.1 NODES

Definition 4. 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 $\texttt{arity}$ can then be defined analogously to Definition 2.

We can use Constraint 2 again to disable extra arguments.

Example 2. Let $\mathcal{M}_\mathcal{A} = 2$, $\mathcal{P} = [\mathsf{P}, \ldots]$, $\mathcal{A} = [1, \ldots]$, and $X \in \mathcal{V}$. Then the node representing atom $\mathsf{P}(X)$ has:

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

3.2 CONSTRAINTS

Definition 5. Let $\texttt{numTrees} \in \{1, \ldots, \mathcal{M}_\mathcal{N}\}$ be the number of trees in $\texttt{structure}$, defined using the $\texttt{tree}$ constraint (Fages and Lorca, 2011).

Definition 6. For convenience, we also define $\texttt{numNodes} \in \{1, \ldots, \mathcal{M}_\mathcal{N}\}$ to count the number of nodes in the main tree. We define it as

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

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

Constraint 5. $\texttt{structure}$ *is sorted.*

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

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

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

Each of the three equivalences in Constraint 7 corresponds to node i having no children, one child, and multiple children, respectively.

Constraint 8. *Only root nodes can have $\top$ as the value. For $i = 0, \dots, \mathcal{M}_\mathcal{N} - 1$,*

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

A way to 'disable' a clause:

Constraint 9. *For $i = 0, \dots, \mathcal{M}_\mathcal{C} - 1$, 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 7. Let $N = \mathcal{M}_\mathcal{A} \times (\mathcal{M}_\mathcal{N} + 1)$. Let $\texttt{terms}[N] \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$ be a flattened array of all gaps 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 $\{0, \dots, N-1\}$ such that for all $i = 0, \dots, N-1$, and $t \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$,

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

We introduce an array that, for each variable, holds the position of its first occurrence.

Definition 8. Let $\texttt{intros}[|\mathcal{V}|] \in \{0, \dots, N\}$ 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. 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.

Constraint 10. `intros` *are sorted.*

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

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

Then $\texttt{terms} = [X, Y, \square, \square, X, Z, Y, Z]$ (the boxes represent the conjunction node), $\texttt{occ} = [\{0, 4\}, \{1, 6\}, \{5, 7\}, \{2, 3\}]$, and $\texttt{intros} = [0, 1, 5]$.

4.1 REDUNDANT CONSTRAINTS

We add a number of redundant constraints to make search more efficient.

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

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

Constraint 12. `allDifferentExcept0(intros)`.

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

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

Definition 9. We define an auxiliary set variable

$$\texttt{potentialIntros} \subseteq \{0, \dots, N\}$$

to act as a set of possible values that `intros` can take, i.e., for $v \in \mathcal{V}$, $\texttt{intros}[v] \in \texttt{potentialIntros}$.

Constraint 14. *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

$$\texttt{potentialIntros} \cap S = \emptyset.$$

In simpler terms, Constraint 14 says that if a node in the tree representation of a clause represents something other than a predicate, then a variable cannot be introduced as one of its 'arguments'.

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 $n > 1$:

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

The first term here represents negation, i.e., negating an expression 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 fill n gaps with terms can be expressed as

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

The first term is simply the number of ways to fill all n gaps with 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:

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

where

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

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

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 10. 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.

Definition 11. 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 12. Two predicates P and Q are *independent* if $D_{\mathsf{P}} \cap D_{\mathsf{Q}} = \emptyset$.

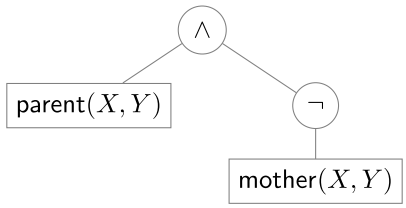

Figure 2: The tree representation of the body of Clause (1).

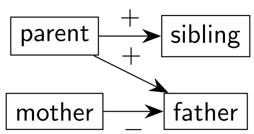

Figure 3: The predicate dependency graph of the program in Example 4. Positive edges are labeled with '+', and negative edges with '−'.

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

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

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

Definition 13 (Adjacency matrix representation)**.** 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:

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

Figure 4: The adjacency matrix defined using Definition 13 for Example 5

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

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

Algorithm 1: Propagation for independence

Data: predicates p_1, p_2; adjacency matrix $\mathbf{A}$
for $(d_1, d_2) \in$ `deps(`p_1`, 0)` $\times$ `deps(`p_2`, 0)` *such that* d_1.predicate $= d_2$.predicate **do**
  if d_1 **is** $\Delta(\_)$ **and** d_2 **is** $\Delta(\_)$ **then** `fail()`;
  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**
    $\mathbf{A}[s][t]$`.removeValue(1)`;

Algorithm 2: Entailment for independence

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

Note that if there are multiple paths in the dependency graph from Q to P, when constructing D_{P}, Algorithm 3 could include Q once for each possible type (Δ, Υ, and Γ). However, Algorithms 1 and 2

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

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

where $\square$ indicates an unknown predicate with domain

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

Algorithm 3: Dependencies of a predicate

Data: adjacency matrix $\mathbf{A}$

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

  $D \leftarrow \{\Delta(p)\}$;
  while `true` **do**
    $D' \leftarrow \emptyset$;
    for $d \in D$ **and** $q \in \mathcal{P}$ **do**
      edge $\leftarrow \mathbf{A}[q][d.\text{predicate}] = \{1\}$;
      if edge **and** d **is** $\Delta(\_)$ **then**
        $D' \leftarrow D' \cup \{\Delta(q)\}$
      else if edge **and** d **is** $\Gamma(\_, s, t)$ **then**
        $D' \leftarrow D' \cup \{\Gamma(q, s, t)\}$;
      else if $|\mathbf{A}[q][d.\text{predicate}]| > 1$ **and** d **is** $\Delta(r)$ **then**
        $D' \leftarrow D' \cup \{\Gamma(q, q, r)\}$;
      else if $|\mathbf{A}[q][d.\text{predicate}]| > 1$ **and** allDependencies **then**
        $D' \leftarrow D' \cup \{\Upsilon(q)\}$;
    if $D' = D$ **then return** D;
    $D \leftarrow D'$;

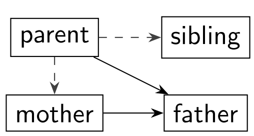

Figure 5: The predicate dependency graph that corresponds to Fig. 4. Dashed edges are undetermined—they may or may not exist.

The predicate dependency graph (without positivity/negativity) defined by Definition 13 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 2 return TRUE.

7 NEGATIVE CYCLES

Algorithm 4: Entailment check 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 $\mathscr{R} = \emptyset$ **then return** UNDEFINED;
if `hasNegativeCycles` ($G_{\mathscr{R}}$) **then**
  return FALSE;
if $\mathscr{R} = \mathscr{P}$ **then return** TRUE;
return UNDEFINED;

The `hasNegativeCycles` function is just a simple extension of the cycle detection algorithm that 'travels' around the graph following edges and checking if each vertex has already been visited or not.

The difficulty with creating a propagation algorithm for negative cycles is that there seems to be no good way to extend Definition 13 so that the adjacency matrix captures positivity/negativity.

8 EMPIRICAL PERFORMANCE

We split the decision variables into the following groups:

1. all head predicates,
2. for each clause:
   (a) `structure`,
   (b) body predicates,

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

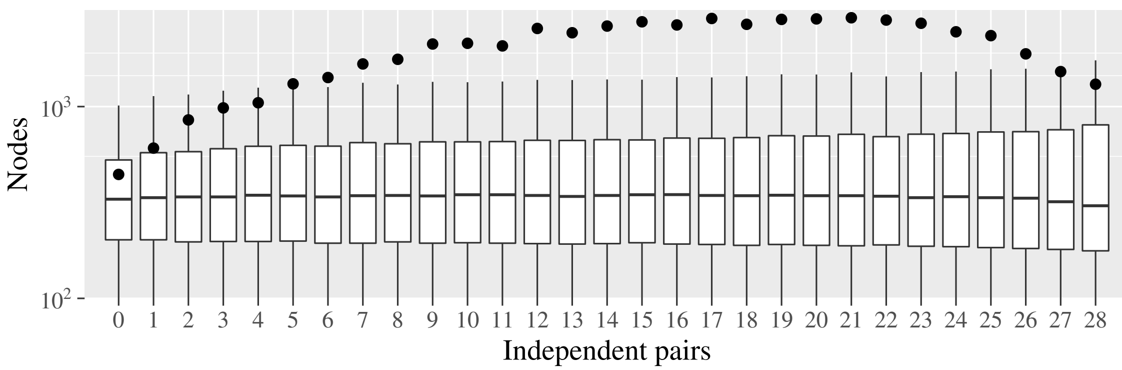

Figure 6: TODO

(c) head arguments,
(d) (if $|\mathcal{V}| > 1$) `intros`,
(e) body arguments.

And use the 'fail first' variable ordering heuristic within each group.

We make no claim that this is optimal, but it does avoid major sources of thrashing.

Value ordering heuristic is random (to make the results random).

We also employ a geometric restart policy, restarting after $10, 20, 40, \dots$ failures (contradictions).

- $|\mathcal{P}|$, $|\mathcal{V}|$, $|\mathcal{C}|$, $\mathcal{M}_{\mathcal{N}}$, the number of additional clauses are in $\{1, 2, 4, 8\}$.
- $\mathcal{M}_{\mathcal{A}} \in \{1, 2, 3, 4\}$.
- all possible numbers of independent pairs: $0, \dots, \binom{|\mathcal{P}|}{2}$.

Repeat 10 times:

- Pick a random combination of $|\mathcal{P}|$ arities such that $\mathcal{M}_{\mathcal{A}}$ is occurs at least once (in an ordered way, since order doesn't matter).
- Pick the required number of independent pairs (without repetition).
- Run the solver with a $60\,\mathrm{s}$ timeout.

Results:

- almost $400\,000$ experiments.
- $97.7\,\%$ of total running times is less than $1\,\mathrm{s}$
- 4 timeouts: all with 8 predicates, 8 additional clauses, and $\mathcal{M}_{\mathcal{N}} = 8$; the rest of the parameters differ.
- Running time (or number of nodes) as a function of various parameters:
  - The number of clauses has a clear impact on time but not number of nodes.
  - $\mathcal{M}_{\mathcal{N}}$ has a huge impact on both.
  - The proportion of independent pairs of predicates is acting weird. Investigate.

9 CONCLUSION & FUTURE WORK

- A constraint for logical equivalence. An algorithm to reduce each tree to some kind of normal form. Not doing this on purpose. Leaving for further work.
- Could investigate how uniform the generated distribution of programs is. Distributions of individual parameters will often favour larger values because, e.g., there are more 5-tuples than 4-tuples.
- Mention the random heuristic. Mention that restarting gives better randomness, but duplicates become possible.
- Could add statistics about what constraints tend to conflict.

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.

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.

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.