Generating Random Logic Programs Using Constraint Programming

Paulius Dilkas[0000−1111−2222−3333]

University of Edinburgh, Edinburgh, United Kingdom
`p.dilkas@sms.ed.ac.uk`

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

Keywords: Constraint Programming · Logic Programming · Probabilistic Logic Programming.

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. 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 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 [3]. This works with both Prolog [1] and ProbLog [4]. Tested with SWI-Prolog [5].

1.1 Parameters

Parameters:

- a list of predicates $\mathcal{P}$,
- a list of their arities $\mathcal{A}$ (including zero),
  - maximum arity $\mathcal{M}_\mathcal{A} \coloneqq \max \mathcal{A}$.
- 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,
- $\mathcal{M}_{\mathcal{N}} \ge 1$: maximum number of nodes in the tree representation of a clause,
- $\mathcal{M}_{\mathcal{C}} \ge |\mathcal{P}|$: maximum number of clauses in a program,
- maximum number of solutions,

We also define $\mathcal{T} = \{\neg, \land, \lor, \top\}$. All decision variables of the model are contained in two arrays:

- `Body[`$\mathcal{M}_{\mathcal{C}}$`] bodiesOfClauses`,
- `Head[`$\mathcal{M}_{\mathcal{C}}$`] headsOfClauses`

2 Heads of Clauses

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

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

The reason why $\square$ must be a separate 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*

$$\texttt{arity} = \begin{cases} \textit{the arity of } \texttt{predicate} & \textit{if } \texttt{predicate} \in \mathcal{P} \\ 0 & \textit{otherwise.} \end{cases}$$

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

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

3 Bodies of Clauses

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

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

Example 1. Let $\mathcal{M}_{\mathcal{N}} = 8$. Then $\neg \mathsf{P}(X) \lor (\mathsf{Q}(X) \land \mathsf{P}(X))$ corresponds to the tree in Fig. 1 and can be encoded as:

$$\begin{aligned} \texttt{treeStructure} &= [0, 0, 0, \quad 1, \quad\quad 2, \quad\quad 2, \;\; 6, 7], \\ \texttt{treeValues} &= [\lor, \neg, \land, \mathsf{P}(X), \mathsf{Q}(X), \mathsf{P}(X), \top, \top], \\ \texttt{numNodes} &= 6, \\ \texttt{numTrees} &= 3. \end{aligned}$$

In the rest of this section, we will describe how the elements of `treeValues` are encoded and list a series of constraints that make this representation unique.

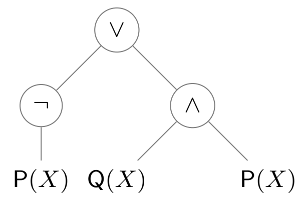

Fig. 1. A tree representation of the formula from Example 1

3.1 Nodes

Definition 4. *A* node *has a* $\texttt{name} \in \mathcal{T} \cup \mathcal{P}$ *and* $\texttt{arguments}[\mathcal{M}_{\mathcal{A}}] \in \mathcal{V} \cup \mathcal{C} \cup \{\square\}$*. The node's* $\texttt{arity}$ *can then be defined analogously to Definition 2.*

We can use Constraint 1 again to disable extra arguments.

Example 2. Let $\mathcal{M}_{\mathcal{A}} = 2$, $\mathcal{P} = [\mathsf{P}, \dots]$, $\mathcal{A} = [1, \dots]$, 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. *We define* $\texttt{numTrees} \in \{1, \dots, \mathcal{M}_{\mathcal{N}}\}$ *to count the number of trees in our representation of a clause using the* $\texttt{tree}(\texttt{treeStructure}, \texttt{numTrees})$[1] *constraint.*

Definition 6. *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*

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

Constraint 2 $\texttt{treeStructure}[0] = 0$*.*

Constraint 3 $\texttt{treeStructure}$ *is sorted.*

Constraint 4 *For* $i = 0, \dots, \mathcal{M}_{\mathcal{N}} - 1$*, if* $\texttt{numNodes} \leq i$*, then*

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

else

$$\texttt{treeStructure}[i] < \texttt{numNodes}.$$

[1] This constraint uses dominator-based filtering by Fages and Lorca [2].

Constraint 5 *For* $i = 0, \dots, \mathcal{M}_\mathcal{N} - 1$,

$$\begin{aligned}
\texttt{count}(i, \texttt{treeStructure}_{-i}) = 0 &\iff \texttt{treeValues}[i].\texttt{name} \in \mathcal{P} \cup \{\top\},\\
\texttt{count}(i, \texttt{treeStructure}_{-i}) = 1 &\iff \texttt{treeValues}[i].\texttt{name} = \neg,\\
\texttt{count}(i, \texttt{treeStructure}_{-i}) > 1 &\iff \texttt{treeValues}[i].\texttt{name} \in \{\land, \lor\},
\end{aligned}$$

where $\texttt{treeStructure}_{-i}$ *denotes array* `treeStructure` *with position* i *skipped.*

Each constraint corresponds to node i having no children, one child, and multiple children, respectively.

Constraint 6 *For* $i = 0, \dots, \mathcal{M}_\mathcal{N} - 1$,

$$\texttt{treeStructure}[i] \neq i \implies \texttt{treeValues}[i].\texttt{name} \neq \top.$$

Constraint 7 *For* $i = 0, \dots, \mathcal{M}_\mathcal{C} - 1$, *if* $\texttt{headsOfClauses}[i].\texttt{predicate} = \square$, *then*

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

and

$$\texttt{bodiesOfClauses}[i].\texttt{treeValues}[0].\texttt{name} = \top.$$

4 Eliminating 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.

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* $\texttt{occurrences}[|\mathcal{C}| + |\mathcal{V}| + 1]$ *as an array of subsets of* $\{0, \dots, N-1\}$ *such that for all* $i = 0, \dots, N-1$, *and* $t \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$,

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

Definition 8. *We define* $\texttt{introductions[}|\mathcal{V}|\texttt{]} \in \{0, \dots, N\}$ *such that for* $v \in \mathcal{V}$,

$$\texttt{introductions}[v] = \begin{cases} \min \texttt{occurrences}[v] & \textit{if } \texttt{occurrences}[v] \neq \emptyset \\ N & \textit{otherwise.} \end{cases}$$

Constraint 8 `introductions` *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

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

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

5 Interactions Between Clauses

Constraint 9 *Each predicate gets at least one clause. Let*

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

Then

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

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

Constraint 10 *Clauses are sorted.*

6 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.

7 Predicate Independence

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

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 $\mathscr{P}$. We add an edge from predicate P to predicate Q if there is a clause in $\mathscr{P}$ with Q as the head and P mentioned in the body.*

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

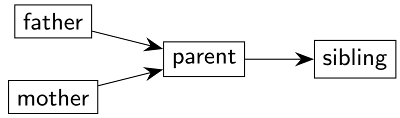

Fig. 2. The predicate dependency graph of the program in Example 4

– $\mathsf{P} \in D_{\mathsf{P}}$,
– *for every* $\mathsf{Q} \in D_{\mathsf{P}}$*, the nodes with arrows to* Q *in* $G_{\mathscr{P}}$ *are all 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 program:

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

Its predicate dependency graph is in Fig. 2. We can then list the dependencies of each predicate:

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

Hence, the only pair of independent predicates in this program is father and mother.

Definition 12 (Adjacency matrix representation). *An* $|\mathcal{P}| \times |\mathcal{P}|$ *adjacency matrix* $\mathbf{A}$ *defined by*

$$\begin{aligned} A_{i,j} = 0 \iff & \nexists k : \mathtt{headsOfClauses}[k].\mathtt{predicate} = j \text{ and}\\ & i \in \{a.\mathtt{name} \mid a \in \mathtt{bodiesOfClauses}[k].\mathtt{treeValues}\}.\end{aligned}$$

A dependency is an algebraic data type that is either determined (in which case it holds only the index of the predicate) or undetermined (in which case it also holds the indices of the source and target vertices, corresponding to the edge responsible for making the dependency undetermined).

Propagation for independence:

– Two types of dependencies: determined and one-undetermined-edge-away-from-being-determined.
– Look up the dependencies of both predicates. For each pair of matching dependencies:
  - If both are determined, fail.
  - If one is determined, the selected edge of the other must not exist.

Algorithm 1: Propagation

```
Data: predicates p_1, p_2; adjacency matrix A
for (d_1, d_2) ∈ getDependencies(p_1) × getDependencies(p_2) s.t.
 d_1.predicate = d_2.predicate do
    if d_1.isDetermined() and d_2.isDetermined() then
        fail();
    if d_1.isDetermined() then
        A[d_2.source][d_2.target].removeValue(1);
    else if d_2.isDetermined() then
        A[d_1.source][d_1.target].removeValue(1);
```

Algorithm 2: Entailment

```
Data: predicates p_1, p_2
D ← {(d_1, d_2) ∈ getDependencies(p_1) × getDependencies(p_2) |
 d_1.predicate = d_2.predicate};
if {(d_1, d_2) ∈ D | d_1.isDetermined(), d_2.isDetermined()} ≠ ∅ then
    return FALSE;
if D = ∅ then
    return TRUE;
return UNDEFINED;
```

Algorithm 3: Computing the dependencies of a predicate

```
Data: an n × n adjacency matrix A
Function getDependencies(p):
    D ← {p};
    repeat
        D' ← D;
        for d ∈ D do
            for i ← 1 to n do
                edgeExists ← A[i][d.predicate] = {1};
                if edgeExists and d.isDetermined() then
                    D' ← D' ∪ {i};
                else if edgeExists and not d.isDetermined() then
                    D' ← D' ∪ {(i, d.source, d.target)};
                else if |A[i][d.predicate]| > 1 and d.isDetermined() then
                    D' ← D' ∪ {(i, i, d.predicate)};
    until D' = D;
    return D;
```

8 Entailment Checking for Negative/All Cycles

1. Let C be a set of clauses such that their bodies and predicates in their heads are fully determined.
2. If $C = \emptyset$, return UNDEFINED.
3. Construct an adjacency list representation of a graph where vertices represent predicates. Each edge is either *positive* or *negative*. There is an edge from p to q if q appears in the body of a predicate with p as its head. The edge is negative if, when traversing the tree to reach some instance of q, we pass through a $\neg$ node. Otherwise, it's positive.
4. Run a modified cycle detection algorithm that detects all cycles that have at least one negative edge.
5. If we found a cycle, return FALSE.
6. If C encompasses all clauses, return TRUE.
7. Return UNDEFINED.

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.
- Perhaps negative cycle detection could use the same graph as the independence propagator? If we extend each domain to -1, 0, 1, but that might make propagation weaker or slower.
- 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.
- Inference options to explore. Logspace vs normal space. Symbolic vs non-symbolic. Propagate evidence (might be irrelevant)? Propagate weights? Supported knowledge compilation techniques: sdd, sddx, bdd, nnf, ddnnf, kbest, fsdd, fbdd.
- Mention the random heuristic. Mention that restarting gives better randomness, but duplicates become possible. Restarting after each run is expensive. Periodic restarts could be an option.

Acknowledgments

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

1. Bratko, I.: Prolog Programming for Artificial Intelligence, 4th Edition. Addison-Wesley (2012)

2. 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
3. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017), http://www.choco-solver.org
4. Raedt, L.D., 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), http://ijcai.org/Proceedings/07/Papers/396.pdf
5. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP **12**(1-2), 67–96 (2012). https://doi.org/10.1017/S1471068411000494