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.

We will often use $\square$ as a special domain value to indicate some kind of exception. We write `a[`b`]` $\in c$ to mean that `a` is an array of variables of length b such that each element of `a` has domain c. Similarly, we write `c[`b`] a` to denote an array `a` of length b such that each element of `a` has type `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 TODO

– 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.

1.2 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,
– $\mathcal{M}_\mathcal{N} \geq 1$: 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,
– 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 Considering Each Clause Separately

2.1 Heads of Clauses

Our definition is slightly more involved because we want to eliminate some symmetries.

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

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

In both cases, $\square$ *is the disabled value.*

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

$$\mathtt{arity} = \begin{cases} 0 & \text{if } \mathtt{predicate} = \square \\ \text{the arity of } \mathtt{predicate} & \text{otherwise.} \end{cases}$$

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

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

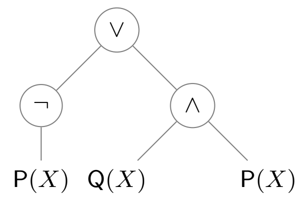

Fig. 1. A tree representation of the formula from Example 1

2.2 Bodies of Clauses

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

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

Auxiliary variables: $\texttt{numNodes}, \texttt{numTrees} \in \{1, \dots, \mathcal{M}_\mathcal{N}\}$. The former counts the number of nodes in the main tree. The latter counts the number of trees in total.

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:

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

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

Nodes

Definition 4. *A* node *has a* $\texttt{name} \in \mathcal{T} \cup \mathcal{P}$ *and* $\texttt{arguments[}\mathcal{M}_\mathcal{A}\texttt{]} \in \mathcal{V} \cup \mathcal{C} \cup \{\square\}$*, where* $\square$ *denotes a disabled argument (the reason why this value is needed will become clear in Section 2.3). The node's* $\texttt{arity} \in [0, \mathcal{M}_\mathcal{A}]$ *is defined by a* `table` *constraint as*

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

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

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

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:

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

Constraints

Constraint 3 $\texttt{treeStructure}$ *represents* $\texttt{numTrees}$ *trees, i.e.,*

$$\texttt{tree}(\texttt{treeStructure}, \texttt{numTrees})^1.$$

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

Constraint 5 $\texttt{numTrees} + \texttt{numNodes} = \mathcal{M}_{\mathcal{N}} + 1$.

Constraint 6 $\texttt{treeStructure}$ *is sorted.*

Constraint 7 *For* $i = 0, \dots, \mathcal{M}_{\mathcal{N}} - 1$, *if* $\texttt{numNodes} \le i$, *then*

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

else

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

Constraint 8 *For* $i = 0, \dots, \mathcal{M}_{\mathcal{N}} - 1$,

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

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

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

Constraint 9 *For* $i = 0, \dots, \mathcal{M}_{\mathcal{N}} - 1$,

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

Constraint 10 *For* $i = 0, \dots, \mathcal{M}_{\mathcal{C}} - 1$, *if* $\texttt{headsOfClauses}[i].\texttt{predicate} = \Box$, *then*

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

and

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

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

2.3 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 5. *Let $N = \mathcal{M}_{\mathcal{A}} \times (\mathcal{M}_{\mathcal{N}}+1)$. Let $\texttt{termsPerClause}[N] \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$ be a matrix, each row of which is a flattened list of all positions for terms in a particular clause.*

Then we can use the `setsIntsChanneling` *constraint to define $\texttt{occurrences}[|\mathcal{C}| + |\mathcal{V}| + 1]$ as a matrix of subsets of $\{0, \ldots, N-1\}$ such that for all $i = 0, \ldots, N-1$, and $t \in \mathcal{C} \cup \mathcal{V} \cup \{\square\}$,*

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

Definition 6. *We define $\texttt{M[}|\mathcal{V}|\texttt{]} \in \{0, \ldots, N\}$ such that for $v \in \mathcal{V}$,*

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

Constraint 11 `M` *is sorted.*

Example 3. Let $\mathcal{M}_{\mathcal{A}} = 4$, $\mathsf{P} \in \mathcal{P}$, $\mathcal{C} = \{a\}$, $\mathcal{V} = \{X, Y, Z\}$. Then $\mathsf{P}(Z, Y, Z)$ would be represented as:

$$\begin{aligned} \texttt{predicate} &= \mathsf{P}, \\ \texttt{arguments} &= [Z, Y, Z, 0], \\ \texttt{arity} &= 3, \\ \texttt{occurrences} &= [\emptyset, \emptyset, \{1\}, \{0, 2\}], \\ \mathbf{M} &= \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 0 \end{bmatrix}. \end{aligned}$$

3 Interactions Between Clauses

Constraint 12 *Each predicate gets at least one clause. Let*

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

Then

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

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

Constraint 13 *Clauses are sorted.*

4 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.

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 combined 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 positions 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 positions with 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:

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

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.

5 Predicate Independence

Definition 7. *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 8. *Let* P *be a predicate in a program $\mathscr{P}$. The* dependencies *of* P *is the smallest set D_P such that:*

- $\mathsf{P} \in D_\mathsf{P}$,
- *for every $\mathsf{Q} \in D_\mathsf{P}$, the nodes with arrows to* Q *in $G_\mathscr{P}$ are all in D_P.*

Definition 9. *Two predicates* P *and* Q *are* independent *if $D_\mathsf{P} \cap D_\mathsf{Q} = \emptyset$.*

Definition 10 (Adjacency matrix representation). *An $|\mathcal{P}| \times |\mathcal{P}|$ adjacency matrix* $\mathbf{A}$ *defined by*

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

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 p1, p2; adjacency matrix A
for (d1, d2) ∈ getDependencies(p1) × getDependencies(p2) s.t.
 d1.predicate = d2.predicate do
    if d1.isDetermined() and d2.isDetermined() then
        fail();
    if d1.isDetermined() then
        A[d2.source][d2.target].removeValue(1);
    else if d2.isDetermined() then
        A[d1.source][d1.target].removeValue(1);
```

Algorithm 2: Entailment

```
Data: predicates p1, p2
D ← {(d1, d2) ∈ getDependencies(p1) × getDependencies(p2) |
 d1.predicate = d2.predicate};
if {(d1, d2) ∈ D | d1.isDetermined(), d2.isDetermined()} ≠ ∅ then
    return FALSE;
if D = ∅ then
    return TRUE;
return UNDEFINED;
```

6 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.

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;
```

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.

Acknowledgments

The author would like to thank Vaishak Belle for his comments. 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