Generating Random Logic Programs Using Constraint Programming

Paulius Dilkas[0000−1111−2222−3333]

University of Edinburgh, Edinburgh, UK
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.

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

- Heads are too restrictive: `P`(X, Y, X) should be allowed.
- A constraint for logical equivalence.
- 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 $\mathcal{T} = \{\neg, \wedge, \vee, \top\}$. 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

Our definition is slightly more involved because we want to impose some additional constraints, namely, that the variables in the atom are consecutive

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

- `predicate` $\in \mathcal{P} \cup \{\square\}$, *where* $\square$ *denotes a disabled clause.*
- `variables[`$|\mathcal{V}|$`]` $\in [0, \mathcal{M}_{\mathcal{A}}]$*: how many times each variable appears in the head atom.*
- `constants[`$\mathcal{M}_{\mathcal{A}}$`]` $\in \mathcal{C} \cup \{\square\}$, *where* $\square$ *denotes that the position is reserved for a variable.*

We define the `predicate`*'s* `arity` *using the* `table` *constraint as*

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

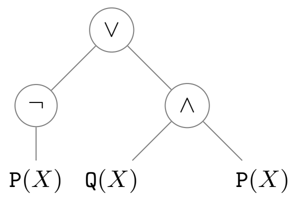

Fig. 1. A tree representation of the formula from Example 1

Constraint 1 *For each* $v \in \texttt{variables}$*,* $v \leq \texttt{arity}$*.*

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

$$i \geq \texttt{arity} \implies \texttt{constants}[i] = \square.$$

Constraint 3 *Connecting the two lists:*

$$\mathcal{M}_{\mathcal{A}} - \texttt{arity} + \sum_{v \in \texttt{variables}} v = \begin{cases} \mathcal{M}_{\mathcal{A}} & \textit{if } \texttt{predicate} = \square \\ \texttt{count}(\square, \texttt{constants}) & \textit{otherwise.} \end{cases}$$

In `variables`, all zeros must go after all non-zeros. For example, if we have to pick one variable out of two, we must pick the first one.

Constraint 4 *For* $i = 0, \dots, |\mathcal{V}| - 2$*, and* $j = i + 1, \dots, |\mathcal{V}| - 1$*,*

$$\texttt{variables}[i] \neq 0 \quad \textit{or} \quad \texttt{variables}[j] = 0.$$

3 Bodies of Clauses

Definition 2. *The body of a clause is defined by:*

- $\texttt{treeStructure}[\mathcal{M}_{\mathcal{N}}] \in [0, \mathcal{M}_{\mathcal{N}} - 1]$ *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*.*
- $\texttt{Node}[\mathcal{M}_{\mathcal{N}}]$ `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 $\neg\texttt{P}(X) \vee (\texttt{Q}(X) \wedge \texttt{P}(X))$ corresponds to the tree in Fig. 1 and can be encoded as:

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

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.

3.1 Nodes

Definition 3. *A* node *has a* `name` $\in \mathcal{T} \cup \mathcal{P}$ *and* `arguments`$[\mathcal{M}_\mathcal{A}] \in \mathcal{V} \cup \mathcal{C}$*. The node's* `arity` $\in [0, \mathcal{M}_\mathcal{A}]$ *is defined by a* `table` *constraint as*

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

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

$$i \geq \texttt{arity} \implies \texttt{arguments}[i] = 0^1.$$

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

$$\begin{aligned} \texttt{name} &= \texttt{P}, \\ \texttt{arguments} &= [X, 0], \\ \texttt{arity} &= 1. \end{aligned}$$

3.2 Constraints

Constraint 6 `treeStructure` *represents* `numTrees` *trees, i.e.,*

$$\texttt{tree}(\texttt{treeStructure}, \texttt{numTrees})^2.$$

Constraint 7 $\texttt{treeStructure}[0] = 0$.

Constraint 8 $\texttt{numTrees} + \texttt{numNodes} = \mathcal{M}_\mathcal{N} + 1$.

Constraint 9 `treeStructure` *is sorted.*

Constraint 10 *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

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

Constraint 11 *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* `treeStructure` *with position* i *skipped.*

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

Constraint 12 *For* $i = 0, \dots, \mathcal{M}_\mathcal{N} - 1$,

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

If the clause should be disabled, $\texttt{numNodes} = 1$ and $\texttt{treeValues}[0].\texttt{name} = \top$.

[1] Zero here represents the first variable or constant, regardless of what it is.
[2] This constraint uses dominator-based filtering by Fages and Lorca [2].

4 Managing Clauses

Constraint 13 *For* $i = 0, \dots, \mathcal{M}_{\mathcal{C}} - 1$*, let* $p_i = \texttt{headsOfClauses}[i].\texttt{predicate}$*. Then* $(p_i)_{i=0}^{\mathcal{M}_{\mathcal{C}}-1}$ *is sorted.*

Constraint 14 *Each predicate gets at least one clause. Let* $P = \{h.\texttt{predicate} \mid h \in \texttt{headsOfClauses}\}$*. Then*

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

Constraint 15 *Let* $\prec$ *be any total order defined over bodies of clauses, and let* $\preceq$ *be its extension with equality (in the same way as* $\leq$ *extends* $<$*). If*

$$\texttt{headsOfClauses}[i-1].\texttt{predicate} = \texttt{headsOfClauses}[i].\texttt{predicate},$$

then $\texttt{bodiesOfClauses}[i-1] \preceq \texttt{bodiesOfClauses}[i]$*.*

For example, $\preceq$ can be implemented as `lexLessEq` over the decision variables of each body.

Constraint 16 *Adjacency matrix representation:*

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

5 Counting Programs

Let p_a be the number of predicates in $\mathcal{P}$ with arity $a \in \mathcal{A}$.

Number of atoms (leaves of the tree):

$$A = \sum_{a \in \mathcal{A}} p_a (|\mathcal{V}| + |\mathcal{C}|)^a$$

Number of clauses:

$$C = 1 + \sum_{n=1}^{\mathcal{M}_{\mathcal{N}}} T(n),$$

where $T(n)$ is defined recursively as:

$$T(1) = A$$

and

$$T(n) = T(n-1) + 2 \sum_{\substack{c_1 + \cdots + c_k = n-1, \\ k \geq 2, \\ c_i \geq 1 \text{ for all } i}} \prod_{i=1}^{k} T(c_i).$$

Example of ordered partitions:

$$3 = 2 + 1 = 2 + 1 = 1 + 1 + 1,$$

so for $n = 4$, the sum would have three terms.

Number of heads for a specific predicate with arity $a \in \mathcal{A}$:

$$H_a = |\mathcal{C}|^a + \sum_{v=1}^{a} \binom{a}{v} |\mathcal{C}|^{a-v} \sum_{k=0}^{|\mathcal{V}|-1} \binom{v-1}{k}.$$

First, select the v positions dedicated for variables. The remaining $a-v$ constants can then be filled in $|\mathcal{C}|^{a-v}$ ways. Filling v positions with $k+1$ variables in a non-decreasing manner (without skipping any variables) can be seen as putting k 'bars' in the $v-1$ spaces between v positions. Each bar represents switching to the next variable.

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:

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

where

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

This is without any kind of cycle detection and without probabilities.

6 The Independence Constraint

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;
```

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;
```

7 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.

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