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

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.

2 TODO

- Given fixed parameters, use combinatorial arguments to calculate how many different programs there are and check that I'm generating the same number.
- Formal definition (here and in the predicate invention paper): two predicates are independent if all of their groundings are independent.
- A constraint for logical equivalence.
- Show that the set of all ProbLog programs is equal to the set of programs I can generate (alternatively, show that, given any ProbLog program, there are parameter values high enough to generate it).
- 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.

3 Parameters

Parameters:

- maximum number of solutions,
- $\mathcal{M}_{\mathcal{N}}$: maximum number of nodes in the tree representation of a clause,
- $\mathcal{M}_{\mathcal{C}}$: maximum number of clauses in a program,
- option to forbid all cycles or just negative cycles,
- a list of probabilities that are randomly assigned to clauses,
- a list of predicates $\mathcal{P}$,
- a list of their arities $\mathcal{A}$,
  - maximum arity $\mathcal{M}_{\mathcal{A}} \coloneqq \max \mathcal{A}$.
- a list of variables $\mathcal{V}$,
- and a list of constants $\mathcal{C}$.

We also define $\mathcal{T} = \{\neg, \wedge, \vee, \top\}$. All decision variables of the model are contained in two arrays of length $\mathcal{M}_{\mathcal{C}}$:

- `Body[] bodiesOfClauses`
- `Head[] headsOfClauses`

4 General Constraints

Constraint 1 *For $i = 0, \ldots, \mathcal{M}_{\mathcal{C}} - 1$, let $p_i = \texttt{headsOfClauses}[i].\texttt{predicate}$. Then*

$$\texttt{sort}((p_i)_{i=0}^{\mathcal{M}_{\mathcal{C}}-1}, (p_i)_{i=0}^{\mathcal{M}_{\mathcal{C}}-1}),$$

i.e., all clauses describing the same predicate are listed together and sorted.

Constraint 2 *Each predicate gets at least one clause. Let $P = \{h.\texttt{predicate} \mid h \in \texttt{clauseHeads}\}$. Then*

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

Constraint 3 *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.

5 Atoms

Definition 1. *An* atom *is a* `predicate` $\in \mathcal{T} \cup \mathcal{P}$ *and a list of* `arguments` *of length* $\mathcal{M}_\mathcal{A}$ *in* $\mathcal{V} \cup \mathcal{C} \cup \{\square\}$*, where* $\square$ *means the position is either reserved for a variable, or disabled. The atom's* `arity` *is a number in* $[0, \mathcal{M}_\mathcal{A}]$ *defined by a* `table` *constraint, according to the* `predicate`*.*

Constraint 4 *For* $i = 0, \ldots, \mathcal{M}_\mathcal{A} - 1$*,*

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

6 Bodies of Clauses

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

- `treeStructure`*: list of length* $\mathcal{M}_\mathcal{N}$ *with domain* $[0, \mathcal{M}_\mathcal{N} - 1]$*.*
  - `treeStructure`$[i] = i$*: the* i*-th node is a root.*
  - `treeStructure`$[i] = j$*: the* i*-th node's parent is node* j*.*
- `treeValues`*:* $\mathcal{M}_\mathcal{N}$ *atoms.*

Auxiliary variables: `numNodes`, `numTrees` $\in \{1, \ldots, \mathcal{M}_\mathcal{N}\}$.

6.1 Constraints

Constraint 5 `tree(treeStructure, numTrees)`*, i.e.,* `treeStructure` *represents* `numTrees` *trees (dominator-based filtering [2]).*

Constraint 6 `treeStructure`$[0] = 0$.

Constraint 7 `numTrees` $+$ `numNodes` $= \mathcal{M}_\mathcal{N} + 1$.

Constraint 8 `sort(treeStructure, treeStructure)`*, i.e.,* `treeStructure` *is sorted.*

Constraint 9 *For* $i = 0, \ldots, \mathcal{M}_\mathcal{N} - 1$*, if* `numNodes` $\leq i$*, then*

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

else

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

Constraint 10 *For* $i = 0, \ldots, \mathcal{M}_\mathcal{N} - 1$*,*

- *has 0 children* $\iff$ `treeValues`$[i]$`.predicate` $\in \mathcal{P}$*;*
- *has 1 child* $\iff$ `treeValues`$[i]$`.predicate` $= \neg$*;*
- *has* > 1 *child* $\iff$ `treeValues`$[i]$`.predicate` $\in \{\wedge, \vee\}$*.*

Constraint 11 *For $i = 0, \dots, \mathcal{M}_\mathcal{N} - 1$,*

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

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

Constraint 12 *Adjacency matrix representation:*

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

7 Head of a Clause

Our definition of a head of a clause is more restrictive than Definition 1.

Definition 3. *The* head *of a clause is defined by two lists:*

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

We also define the predicate's `arity` *using the same* `table` *constraint.*

Constraint 13 *For each* $v \in \texttt{variables}$*,* $v \leq \texttt{arity}$*.*

Constraint 14 *For $i = 0, \dots, \mathcal{M}_\mathcal{A} - 1$,*

$$i \geq \texttt{arity} \implies \texttt{constants}[i] = 0.$$

Constraint 15 *Connecting the two lists:*

$$\sum_{v \in \texttt{variables}} v = \texttt{count}(\Box, \texttt{constants}) + \texttt{arity} - \mathcal{M}_\mathcal{A}.$$

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

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

8 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 p_1, p_2; adjacency matrix $\mathbf{A}$
for $(d_1, d_2) \in$ `getDependencies(`p_1`)` $\times$ `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**
    $\mathbf{A}[d_2.\mathsf{source}][d_2.\mathsf{target}]$`.removeValue(1);`
  else if d_2`.isDetermined()` **then**
    $\mathbf{A}[d_1.\mathsf{source}][d_1.\mathsf{target}]$`.removeValue(1);`

Algorithm 2: Entailment

Data: predicates p_1, p_2
$D \leftarrow \{(d_1, d_2) \in$ `getDependencies(`p_1`)` $\times$ `getDependencies(`p_2`)` $\mid$ d_1.predicate $= d_2$.predicate$\}$;
if $\{(d_1, d_2) \in D \mid d_1$`.isDetermined()`$, d_2$`.isDetermined()`$\} \neq \emptyset$ **then**
  return *FALSE*;
if $D = \emptyset$ **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;
```

9 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