Generating Random Logic Programs Using Constraint Programming

Paulius Dilkas

28th December 2019

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.

2 TODO

- Make negative cycle detection use the graph representation.
- 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).
- 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.
- Describe: entailment checking for cycles and negative cycles.
- A constraint for logical equivalence.
- Rename everything into something more appropriate.

Both determined $\implies$ fail(). Both determined but at least is one masked by a (probable/determined) mask $\implies$ nothing. One determined $\implies$ the other one cannot exist.

3 Parameters

Parameters:

- maximum number of solutions
- MAX_NUM_NODES (in the tree representation of a clause)

- MAX_NUM_CLAUSES
- option to forbid all cycles or just negative cycles
- a list of probabilities that are randomly assigned to clauses: $\{0.1, 0.2, \ldots, 0.9, 1, 1, 1, 1, 1, 1\}$.
- a list of predicates $\mathcal{P}$,
- a list of their arities $\mathcal{A}$,
  - MAX_ARITY $= \max \mathcal{A}$.
- a list of variables $\mathcal{V}$,
- and a list of constants $\mathcal{C}$.

We also define $\mathcal{T} = \{\neg, \wedge, \vee, \top\}$.
Decision variables:

- `Body[] bodiesOfClauses`
- `Head[] headsOfClauses`: a list of length MAX_NUM_CLAUSES

4 General Constraints

Constraint 1 (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 2. *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 $<$).*

$$\texttt{headsOfClauses}[i-1].\texttt{predicate} = \texttt{headsOfClauses}[i].\texttt{predicate} \implies \texttt{bodiesOfClauses}[i-1] \preceq \texttt{bodiesOfClauses}[i].$$

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 MAX_ARITY 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, MAX_ARITY] defined by a `table` constraint, according to the `predicate`.

Constraint 3. *For $i = 0, \ldots,$ MAX_ARITY $- 1$,*

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

6 Bodies of Clauses

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

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

Auxiliary variables: $\texttt{numNodes}, \texttt{numTrees} \in \{1, \ldots,$ MAX_NUM_NODES$\}$.

6.1 Constraints

Constraint 4. `treeStructure` *represents* `numTrees` *trees.*

Constraint 5. $\texttt{treeStructure}[0] = 0$.

Constraint 6. $\texttt{numTrees} + \texttt{numNodes} = \text{MAX\_NUM\_NODES} + 1$.

Constraint 7. `treeStructure` *is sorted.*

Constraint 8. *For* $i = 0, \dots, \text{MAX\_NUM\_NODES} - 1$*, if* $\texttt{numNodes} \leq i$*, then*

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

else

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

Constraint 9. *For* $i = 0, \dots, \text{MAX\_NUM\_NODES} - 1$*,*

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

Constraint 10. *For* $i = 0, \dots, \text{MAX\_NUM\_NODES} - 1$*,*

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

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

Constraint 11. *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

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

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

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

Constraint 12. *For each variable* $v \in \texttt{variables}$*,* $v \leq \texttt{arity}$.

Constraint 13. *For* $i = 0, \dots, \text{MAX\_ARITY} - 1$*,*

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

Constraint 14 (Connecting the two lists)**.** $\sum_{v \in \texttt{variables}} v = \texttt{count}(\square, \texttt{constants}) + \texttt{arity} - \text{MAX\_ARITY}$.

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 15. *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**
        $\mathbf{A}[d_2.\text{source}][d_2.\text{target}]$`.removeValue(`$1$`);`
    else if d_2`.isDetermined()` **then**
        $\mathbf{A}[d_1.\text{source}][d_1.\text{target}]$`.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 $n \times n$ adjacency matrix $\mathbf{A}$

Function `getDependencies`(p)**:**

- $D \leftarrow \{p\}$;
- **repeat**
  - $D' \leftarrow D$;
  - **for** $d \in D$ **do**
    - **for** $i \leftarrow 1$ **to** n **do**
      - edgeExists $\leftarrow \mathbf{A}[i][d.\mathsf{predicate}] = \{1\}$;
      - **if** edgeExists **and** d.`isDetermined()` **then**
        - $D' \leftarrow D' \cup \{i\}$;
      - **else if** edgeExists **and not** d.`isDetermined()` **then**
        - $D' \leftarrow D' \cup \{(i, d.\mathsf{source}, d.\mathsf{target})\}$;
      - **else if** $|\mathbf{A}[i][d.\mathsf{predicate}]| > 1$ **and** d.`isDetermined()` **then**
        - $D' \leftarrow D' \cup \{(i, i, d.\mathsf{predicate})\}$;
- **until** $D' = D$;
- **return** D;