Using Constraint Programming to Generate Random Logic Programs

Paulius Dilkas

26th 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.
- Describe how the trees work with many examples.
- Have a 'Constraint' environment.

2 TODO

- Support for constants and multiple variables.
  - Each clause is defined for each predicate with standard variable names (X, Y, etc.).
  - There is a list of constants and a list of predicates with their arities. Maybe also a list of variables.
- Make negative cycle detection use the graph representation.
- Finish the propagation algorithm for conditional independence. The propagation algorithm for independence is extended with masks that are either potential or definite. Masking happens in two stages: first, we mask expressions within formulas, and then predicates. Masking algorithm uses an algorithm for perfect bipartite matching.
- 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.

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
- `maxNumNodes` (in the tree representation of a clause)
- list of predicates with their variables
- maximum number of clauses
- option to forbid all cycles or just negative cycles
- list of probabilities that are randomly assigned to clauses: $\{0.1, 0.2, \dots, 0.9, 1, 1, 1, 1, 1, 1\}$

Decision variables:

- `IntVar[] clauseAssignments`: a predicate or disabled
- `Clause[] clauses`
- `Head[] clauseHeads`

4 General Constraints

Constraint 1 (Each predicate gets at least one clause)**.** `numDisabledClauses` *is defined by a* `count` *constraint.*

$$\mathtt{numDistinctValues} = \begin{cases} \mathtt{numPredicates} + 1 & \textit{if}\ \mathtt{numDisabledValues} > 0 \\ \mathtt{numPredicates} & \textit{otherwise} \end{cases}$$

(also constrained using the `nValues` *constraint).*

Constraint 2 (`clauseAssignments` are sorted)**.**

$$\mathtt{clauseAssignments}[i-1] = \mathtt{clauseAssignments}[i] \implies \mathtt{clause}[i-1] \preceq \mathtt{clause}[i].$$

5 Bodies of Clauses

Definition 1. The body of a clause is defined by:

- `IntVar[] treeStructure`
  - `treeStructure`$[i] = i$: the i-th node is a root.
  - `treeStructure`$[i] = j$: the i-th node's parent is node j.
- `IntVar[] treeValues`: $\neg$, $\wedge$, $\vee$, $\top$, and any predefined predicates with variables.

Auxiliary variables: $\mathtt{numNodes}, \mathtt{numTrees} \in \{1, \dots, \mathtt{maxNumNodes}\}$.

5.1 Constraints

Constraint 3. `treeStructure` *represents* `numTrees` *trees.*

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

Constraint 5. $\texttt{numTrees} + \texttt{numNodes} = \texttt{maxNumNodes} + 1$.

Constraint 6. `treeStructure` *is sorted.*

Constraint 7. *For* $i = 0, \dots, \texttt{maxNumNodes} - 1$, *if* $\texttt{numNodes} \leq i$, *then*

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

else

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

Constraint 8. *For* $i = 0, \dots, \texttt{maxNumNodes} - 1$,

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

Constraint 9. *For* $i = 0, \dots, \texttt{maxNumNodes} - 1$,

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

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

Constraint 10. *Adjacency matrix representation:*

$$A[i][j] = 0 \iff \nexists k : \texttt{clauseAssignments}[k] = j \textit{ and } i \in \texttt{clauses}[k].\texttt{treeValues}$$

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 $p_1$, $p_2$; 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($p_2$) $\mid d_1.\text{predicate} = d_2.\text{predicate}\}$;
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 $\leftarrow \mathbf{A}[i][d.\text{predicate}] = \{1\}$;
                if edgeExists and $d$.isDetermined() then
                    $D' \leftarrow D' \cup \{i\}$;
                else if edgeExists and not $d$.isDetermined() then
                    $D' \leftarrow D' \cup \{(i, d.\text{source}, d.\text{target})\}$;
                else if $|\mathbf{A}[i][d.\text{predicate}]| > 1$ and $d$.isDetermined() then
                    $D' \leftarrow D' \cup \{(i, i, d.\text{predicate})\}$;
    until $D' = D$;
    return $D$;
```