Using Constraint Programming to Generate Random Logic Programs

Paulius Dilkas

24th 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.

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.

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 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).

4 Conditional Independence

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;

Algorithm 4: Potential root nodes of the required expression, assuming that the node at index i is part of the expression

Data: connective c, a set of predicates P
Function `potentialRoots(`clause, i`):`

```
R ← ∅;
V ← clause.getTreeValues(i);
for v ∈ V do
    if v = c then
        R ← R ∪ {(i, |V| = 1)};
    else if v ∈ P then
        R′ ← clause.getTreeStructureDomainValues(i);
        R ← R ∪ {(r, |R′| = 1) | r ∈ R′};
return R;
```