Note on KR and Graph Mining

Authors

February 3, 2016

An attempt to model the Graph Mining problem in both IDP as well as ProB makes it clear that neither language allows us to express the problem to its full extent. We try to link the shortcomings of each language to the expressiveness of the underlying logic on which they are built.

0.1 IDP

The IDP language can express *Existential Second Order* problems; problems in which there is an existentially quantified, generally second order, vocabulary of symbols and a first order theory with symbols from that vocabulary.

This restriction to *Existential* Second Order forces us to use a Skolemization based trick when expressing the homomorphic property. We introduce a general function `f` that represents the homomorphisms, and make its dependency on a specific goal graph explicit using an additional argument: `partial f(graph, t_var):node`. In Second Order Logic, this dependency would follow directly from the order of the separate quantifications. We can now use this `f` anywhere we would the regular homomorphic function for a specific graph by fixing the goal graph. Note that this encoding also requires us to make this function `f` partial, as the Graph Mining problem does not require the solution to be homomorphic with *all* goal graphs.

Furthermore, limiting ourselves to existential second order prohibits us from expressing the constraint negative constraint on homomorphism (No more than N_{-} negative examples are homomorphic) in the same model. In fact, the negative constraint asserts a property for all candidate homomorphic functions, which would lead to *universal* quantification. Therefore, our only recourse is to encode the positive constraint and require it to fail when queried.

Of course, other (even uglier) schemes exist to encode this. Should we mention this?

0.2 Eventual encoding

```
//Homomorphism/2 is a higher-order predicate:
//Edge1 and Edge2 are predicates themselves.
homomorphism(Edge1, Edge2) ⟺ ∃ f: (∀ x, y : x ≠ y ⇒ f(x) ≠ f(y)) ∧
```

1 Feature Comparison

1.1 IDP

Pro:

- can model inductive definitions
- allows core formulation in a high-level language (NP)
- handles aggregates
- has support for variety of constraints

Cons:

- cannot handle negative case NP^{NP} complexity
- cannot model subgraph isomorphism independence
- cannot handle dominance, i.e., when one model is preferred over another

ASP Mostly the same but in theory can handle NP^{NP}, in practice however, it would require encoding tricks and unavoidably lead to the same problem as in IDP – indexing homomorphism enumeration.

1.2 proB

Pro:

- can model negative case
- can model subgraph isomorphism independence

Cons:

- cannot handle inductive definitions
- cannot handle different types of aggregates (? needs to be checked again)

the rest of constraints?

2 Code in ProB and IDP

proB encoding

```
MACHINE PositiveAndNegative
INCLUDES PositivePatterns, NegativePatterns, Labels
SETS
  Vertices = {x1,x2,x3,x4,x5,x6,x7,x8}
CONSTANTS
  Label,
```

```
  Template,
  ChosenVertices
DEFINITIONS
  SET_PREF_TIME_OUT == 35000; SET_PREF_MAX_INITIALISATIONS == 1;
  homomorph_with(iso,ToGraph) == (
    iso : ChosenVertices >-> graph_domain(ToGraph) &
    !x.(x:ChosenVertices => Label(x) = node_label(ToGraph,iso(x))) &
    !(x,y).( x|->y : Template
         => (x:ChosenVertices & y:ChosenVertices => iso(x)|->iso(y) : graph_edges(ToGraph))
           ) /* small optimisation: instead of TU ; does not seem to improve runtime */
  );
  homomorph_with_n(iso,ToGraph) == (
    iso : ChosenVertices >-> ngraph_domain(ToGraph) &
    !x.(x:ChosenVertices => Label(x) = nnode_label(ToGraph,iso(x))) &
    !(x,y).( x|->y : Template
         => (x:ChosenVertices & y:ChosenVertices => iso(x)|->iso(y) : ngraph_edges(ToGraph))
           )  /* small optimisation: instead of TU ; does not seem to improve runtime */
  );
 CUSTOM_GRAPH_NODES == { node,col | node : Vertices & col = Label(node)};
 CUSTOM_GRAPH_EDGES == { n1,n2 | n1|->n2:Template & n1:ChosenVertices & n2:ChosenVertices}
PROPERTIES /* for simplicity we assume a global labeling function */
  Label : Vertices --> Labels &
  Label = {(x1,a), (x2,b), (x3,c), (x4,d), (x5,e), (x6,f), (x7,g), (x8,k)} &
  Template = {(x1,x2),(x2,x3),(x3,x4),(x4,x5),(x5,x6),(x6,x7),(x7,x8)} &
  ChosenVertices <: dom(Template) \/ ran(Template) &
  card({p|p:graph & #isop.(homomorph_with(isop,p))}) >= 50 &
  card({p|p:ngraph & #isop.(homomorph_with_n(isop,p))}) <= 50 &
  card(ChosenVertices) = 4 & /* solution found in 8.8 seconds for card = 4*/
  /* additionally request that we have some connectivity of the selected subgraph */
  !v.(v:ChosenVertices => #w.(w:ChosenVertices & (v|->w : Template or w|->v : Template)))
OPERATIONS /* These operations are just there to show some aspects of the solution found */
    /* show which vertices and edges have been selected as the solution pattern */
  Pattern(v,lv,w,lw) = SELECT v:ChosenVertices & w:ChosenVertices & v|->w:Template &
                              lv=Label(v) & lw=Label(w) THEN  skip
  END;
  MatchedPositivePattern(p,isop) = SELECT p:graph & homomorph_with(isop,p) THEN skip END;
  MatchedNegativePattern(p,isop) = SELECT p:ngraph & homomorph_with_n(isop,p) THEN skip END
END
```

IDP encoding

```
vocabulary V{
  type t_var isa nat
  type graph
  invar(t_var)
  pattern_in(graph)
  extern vocabulary Vout
  type label
  type node isa nat
  node_label(graph, node, label)
  t_label(t_var,label)
  edge(graph, node, node)
  threshold:int
  partial f(graph,t_var):node
  t_edge(t_var,t_var)
        path(t_var,t_var)
}
theory T:V{
         // frequency
   #{ graph : pattern_in(graph) } >= threshold.
   // homomorphism definition
   pattern_in(g) & invar(x)  <=> ? y: y=f(g,x).
   //injectivy
   pattern_in(g) & invar(x) & invar(y) & x ~= y      => f(g, x) ~= f(g, y).
   pattern_in(g) & invar(x) & invar(y) & t_edge(x,y) => edge(g,f(g,x), f(g,y)).
   pattern_in(g) & invar(x) & t_label(x,lx)          => node_label(g,f(g,x),lx).
  {
   path(x,y) <- t_edge(x,y) & invar(x) & invar(y).
   path(y,x) <- path(x,y).
   path(x,y) <- ?z: path(x,z) & t_edge(z,y) & invar(y).
  }
  !x y : x ~= y & invar(x) & invar(y) => path(x,y).
  // Cardinality constraint on the size of the query, replace NNN with number
  ?NNNx: invar(x).
  // *Nogoods*
}
```