Note on KR and Graph Mining

Authors

February 17, 2016

1 Introduction

Machine learning techniques often contain a process of learning certain patterns from a set of graphs. This problem, which exists in many forms, is called the *Graph Mining* problem.

Wat verantwoording en extra info vragen aan Sergey over waarom dit probleem interessant is. Ook is dit 'structured' graph mining, er bestaan andere vormen geloof ik, hoe bespreken we die?

Definition 1.1. Given a pair $\langle E_+, E_- \rangle$ consisting of a set of *positive* and *negative* examples of *labeled graphs*, respectively, *Graph mining* is the problem of finding one or more *connected labeled graphs* P, called *patterns*, that are *homomorphic* with at least N_+ positive, and at most N_- negative examples.

Definition 1.2. A graph homomorphism f between two labeled graphs $G = (V, E, L)$ and $G' = (V', E', L')$ is a mapping $V \to V'$ from vertices of G to vertices of G' s.t.

- $\forall u, v \in V, \{u, v\} \in E \implies \{f(u), f(v)\} \in E'$ (the mapping preserves edges), and
- $\forall v \in V : L(v) = L(f(v))$ (the mapping respects labelings).

If there exists such a graph homomorphism between graphs G and G' we say G is *homomorphic* with G'.

Furthermore, when looking for more than one pattern, one can impose restrictions on the different patterns that are found. For example, it stands to reason that one wants only *canonical* solutions, meaning that no two patterns found are *isomorphic*.

Definition 1.3. A graph isomorphism f between two labeled graphs $G = (V, E, L)$ and $G' = (V', E', L')$ is a *one-to-one* mapping $V \to V'$ such that f represents a homomorphism from G to G', and its inverse f^{-1} represents a homomorphism from G' to G. If there exists such a graph isomorphism between G and G' we say G (G') is *isomorphic* with G' (G)

Definition 1.4. Let $\mathcal{G}$ be a class of graphs, closed under isomorphism. A function c for which $\forall G, H \in \mathcal{G} : G \simeq H \iff c(G) = c(H)$ and $\forall G \in \mathcal{G} : G \simeq c(G)$ hold, is called a *canonization*. The graph $c(G)$ is called the *canonical form* w.r.t c, and is denoted by $canon(G)$.

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 now try to link the shortcomings of each language to the expressiveness of the underlying logic on which they are built.

First we introduce a new definition of the graph mining problem, equivalent to **Def.** 1.2. We'll assume a sufficiently large supply of vertices, and represent example graphs directly as a triple $\langle Edge, Label, Class \rangle$, consisting of an edge relation and a labeling function over the vertices, as well as a classification (positive/negative).

Definition 1.5. Graph Mining (redefined) Given a supply of vertices V and a set of $\langle E, L, C \rangle$ triples, where E and L represent the edge relation and labeling function over a supply of vertices respectively, we look for a graph $\langle E_p, L_p \rangle$ such that for at least N_+ of the triples $\langle E, L, C \rangle$ with $C = Pos$, and for at most N_- of such triples with $C = Neg$, there exists a function f s.t. $\forall u, v \in V, \{u, v\} \in E_p \implies \{f(u), f(v)\} \in E'$ and $\forall v \in V : L_p(v) = L(f(v))$.

This definition shows great *local coherence* with respect to the different graphs: all characteristics of a graph are represented by separate entities or concepts, which are grouped together for each graph G in the triple that describes the it. A natural choice to represent the set of triples would be a (ternary) predicate. As the domains of this predicate range over predicates and functions, this predicate would be a higher-order predicate.

1.1 IDP

1.1.1 ESO

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 means we cannot express this set as a higher-order predicate. One possible solution is to replicate for each graph the different characteristic predicates and functions, as well as the knowledge (theory) about them. It is clear that this solution is undesirable due to the way it scales and the editing needed with growing problem instances. It retains the local coherence of graph characteristics when it comes to data representation, but prohibits the abstraction (generalization) of knowledge about these properties, as evidenced by our obligation to duplicate the theory for each graph.

Another solution would be to use a trick where we represent each characteristic property by a single general entity for all graphs that behaves the way it should for a specific graph instance based on an additional argument serving as an identifier for the graph of interest. It is clear that this trick forces us to give up the local coherence of graph characteristics that was present in **Def.** 1.5.

But, can we now express the abstraction (generalization) of knowledge about these properties, such as the positive homomorphic property. As evidenced in **Def.** 1.5, normally one would express the positive homomorphic property by quantifying (counting) over all graphs, requiring the existence of a function with the correct properties. Using this trick, does not influence our ability to express this restriction.

However, the restriction to ESO forbids us to quantify over higher-order entities such as functions outside of the vocabulary. Thus, we are required to promote the homomorphic mapping functions to a global property, even though we are only interested in the existence of a mapping, and not in a concrete valid mapping itself. We prevent the same explosion of mapping functions as with the graph characteristics above, using the same trick as above (which in this case corresponds to Skolemization): We introduce a general function `f` that represents all homomorphisms, and make its dependency on a specific example 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.

Much in the same way, 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 its dual positive constraint and require it to fail when queried.

1.1.2 Inductive Definitions

Beyond the Existential Second Order restriction, the IDP language is also extended with inductive definitions. These definitions, evaluated under the well-founded semantics, allows the derivation of negative knowledge that otherwise would be underivable.

A section about inductive definitions, and their use. (Being able to derive negative knowledge)

1.2 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)) ∧
    (∀x, y : Edge1(x, y) ⟹ Edge2(f (x), f (y)))

{
    reachable(x,y,Edge) ← Edge(x,y) ∨ Edge(y,x).
    reachable(x,y,Edge) ← ∃ : reachable(x,z,Edge) ∧ reachable(z,y,Edge).
}

isomorph(Edge1,Edge2) ⟺ ∃f : (∀ x,y:Edge1(x,y) ⟺ Edge2(f(x),f(y))) ∧
    (∀x,y:x≠y ⟹ f(x)≠f(y)).

//∀Pat represents quantification over a predicate Pat/2.
```

```
//A pattern is represented by its Edge relation.
∀P : pattern(P)  ⟹  P(x,y)  ⟺  Template(x,y) .
∀P : pattern(P)  ⟹  #{ Pos : positive(Pos) ∧ homomorphism(P, Pos) } ≥ N+.
∀P : pattern(P)  ⟹  #{ Neg : negative(Neg) ∧ homomorphism(P, Neg) } ≤ N_.
∀P,P2 :pattern(P)∧pattern(P2)∧P≠P2  ⟺  ¬isomorph(P,P2).
```

Tekstuele uitleg hierbij

1.3 ProB

2 Feature Comparison

2.1 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.

Listing 1: ASP positive matching

```
positive_match(G) | not_positive_match(G) :- positive(G).

1 { map(G,X,V) : node(G,V) } 1 :- positive(G), invar(X).

:- positive_match(G), map(G,X,V1), map(G,Y,V2), t_edge(X,Y),
                  not edge(G,V1,V2), invar(X), invar(Y).

positive_count(N) :- N = #count{G:positive_match(G)}.

:- positive_count(N), N < 2.
```

Listing 2: ASP negative matching

```
//Saturated Representation

//negative constraints to check not matching negative graphs

map(G,X,v1) | map(G,X,v2) | map(G,X,v3) | map(G,X,v4) :- invar(X),
    negative(G).

map(G,X,V) :- saturated(G), t_node(X), node(G,V).

saturated(G) :- t_edge(X,Y), map(G,X,V1), map(G,Y,V2), not edge(G,V1,V2
    ), negative(G), invar(X), invar(Y).
saturated(G) :- map(G,X,V),  map(G,Y,V), X != Y, invar(X), invar(Y). //
     we cannot map two different template nodes to the same

negative_match(G) :- not saturated(G), negative(G).

negative_count(N) :- N = #count{G:negative_match(G)}.

:- negative_count(N), N > 1.
```

Listing 3: Canonicity template-based check

```
iso(X,x1) | iso(X,x2) | iso(X,x3) | iso(X,x4) :- invar(X).

candidate_var(X) :- iso(_,X).

%not iso!
iso_saturated :- invar(X1), invar(X2), iso(X1,V1), iso(X2,V2),
    t_edge(V1,V2), not t_edge(X1,X2).
iso_saturated :- invar(X1), invar(X2), iso(X1,V1), iso(X2,V2), not
    t_edge(V1,V2),     t_edge(X1,X2).

iso(X,V) :- invar(X), t_node(V), iso_saturated.

d1(X) :-     invar(X), not candidate_var(X).
d2(X) :- not invar(X),     candidate_var(X).

not_equal :- d1(X). % check that in fact candidate is different from
    the pattern itself
not_equal :- d2(X). % check that in fact candidate is different from
    the pattern itself

iso_saturated :- not not_equal. % should not be completely equal

min_d1(N) :- N = #min{ X: d1(X) }, not iso_saturated.
min_d2(N) :- N = #min{ X: d2(X) }, not iso_saturated.

iso_saturated :- min_d1(N1), min_d2(N2), N1 > N2.
```

Listing 4: Auxilary predicates – probably should be moved to appendix

```
// selects subpattern

t_path(X,Y) :- t_edge(X,Y), invar(X), invar(Y).
t_path(X,Y) :- t_edge(X,Z), t_path(Z,Y), invar(X).
```

```
:- invar(X), invar(Y), not t_path(X,Y).

0 { invar(X) } 1 :- t_node(X).
// auxilary constraints


edge(G,Y,X) :- edge(G,X,Y).
t_edge(Y,X) :- t_edge(X,Y).
node(G,Y)   :- edge(G,Y,_).
t_node(X)   :- t_edge(X,_).
```

Listing 5: Canonicity previous solution isomorphism check

```
iso(s1,X,x1) | iso(s1,X,x2) :- invar(X).
iso(s2,X,x2) | iso(s2,X,x3) :- invar(X).

candidate_var(G,X) :- iso(G,_,X).

iso_saturated(G) :- invar(X1), invar(X2), iso(G,X1,V1), iso(G,X2,V2),
        t_edge(V1,V2), not t_edge(X1,X2).
iso_saturated(G) :- invar(X1), invar(X2), iso(G,X1,V1), iso(G,X2,V2),
    not t_edge(V1,V2),     t_edge(X1,X2).
iso_saturatea(G) :- not equal(G), iso(G,_,_).

iso(G,X,V) :- invar(X), t_node(V), iso_saturated(G).

:- not iso_saturated(G), iso(G,_,_).

d1(G,X) :-     invar(X), not candidate_var(G,X), iso(G,_,_).
d2(G,X) :- not invar(X),     candidate_var(G,X).

not_equal(G) :- d1(G,X). % check that in fact candidate is different
    from the pattern itself
not_equal(G) :- d2(G,X). % check that in fact candidate is different
    from the pattern itself

equal(G) :- not not_equal(G), iso(G,_,_).
```

2.2 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?

3 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*
}
```