Note on KR and Graph Mining

Authors

February 23, 2016

1 Introduction

feel free to rephrase and correct me here, that's an early attempt to set up a story SERGEY
Frequent Pattern Mining, one of the "super-problems" in Data Mining, has received significant attention over the last years [Aggarwal and Han, 2014]. Especially useful in applications has turned out to be structured mining such as graph mining, where one is looking for sub-structures that often occur in the data, e.g., a graph that is a subgraph of many graphs in the dataset. Recently, there is a line of work towards declarative pattern mining [Guns et al., 2011] and today we look into declarative structured pattern mining that just started receiving attention in both data mining and knowledge representation communities [Négrevergne and Guns, 2015, Paramonov et al., 2015]. We still need to mention the different types of homomorphism/isomorphisms. MATTHIAS

Our main goal is to investigate the problem of graph mining from the knowledge representation perspective and more concretely we are going to look in the following questions:

$\boldsymbol{Q_1}$: What is the logical model of the frequent graph mining problem?

$\boldsymbol{Q_2}$: How this model can be implemented in the existing logic programming languages?

$\boldsymbol{Q_3}$: Do all necessary concepts and primitives already present in the existing logic programming languages?

we need a smooth transition here or should we just start a new section like "formalization"? SERGEY

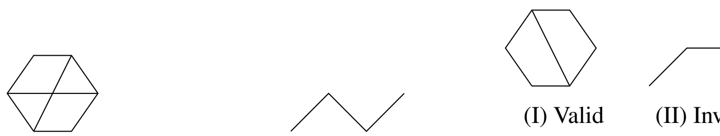
Definition 1.1. Given a pair (E_+, E_-) consisting of a set of *positive* and *negative* examples of *labeled graphs*, respectively, *Graph mining* is the problem of finding one *connected labeled graph* P, called a *pattern*, that is *homomorphic* with at least N_+ positive, while homomorphic with at most N_- negative examples.

Definition 1.2. A graph $\mathcal{G} = \langle V, E, L \rangle$, where V is the set of vertices, E the set of edges and L the labeling function, is *connected* iff for each pair of vertices v and v', there exists an edge $(v, v') \in E$ or there exists a sequence $v \dots v_1 \dots v_n \dots v'$ such that there exist edges (v, v_1), (v_i, v_{i+1}) and $(v_n, v') \in E$.

Definition 1.3. A graph homomorphism f from labeled graph $G = (V, E, L)$ to $G' = (V', E', L')$, where V is the set of vertices, E the set of edges and L the labeling function, is a mapping f : $V \to V'$ from vertices of $\mathcal{G}$ to vertices of $\mathcal{G}'$ 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 $\mathcal{G}$ and $\mathcal{G}'$ we say $\mathcal{G}$ is *homomorphic* with $\mathcal{G}'$.



(I) Valid (II) Invalid

(a) Positive Example (b) Negative Example (c) Pattern Candidates

Figure 1: Example 1:

Take, for example, the problem set shown in Figure 1. There is one positive example (Fig 1a), and one negative example (Fig 1b) We assume all nodes have the same label. Figure 1c shows a valid and an invalid pattern. Requiring at least one homomorphism with a positive example, and allowing no homomorphisms with negative examples (i.e. problem parameters $N_+ = 1$ and $N_- = 0$), only Figure 1I represents a valid pattern. It is clear that there exists a mapping from each node from the valid pattern to a node of the positive example, while no such mapping exists for the negative example. Looking at Figure 1I, this graph is clearly homomorphic with both the positive as well as the negative example. Therefore, it is not a pattern.

1.1 Multiple patterns

To extend on this task, we can look for multiple patterns, instead of just one. In this case, 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 1.4. A graph isomorphism f between two labeled graph $\mathcal{G} = \langle V, E, L\rangle$ and $\mathcal{G}' = \langle V', E', L'\rangle$ is a *one-to-one* mapping $V \to V'$ such that f represents a homomorphism from $\mathcal{G}$ to $\mathcal{G}'$, and its inverse f^{-1} represents a homomorphism from $\mathcal{G}'$ to $\mathcal{G}$. If there exists such a graph isomorphism between $\mathcal{G}$ and $\mathcal{G}'$ we say $\mathcal{G}$ and $\mathcal{G}'$ are *isomorphic*.

Definition 1.5. Let $\mathbb{G}$ be a set of graphs, closed under isomorphism. A function c for which $\forall \mathcal{G}, \mathcal{H} \in \mathbb{G} : \mathcal{G} \simeq \mathcal{H} \iff c(\mathcal{G}) = c(\mathcal{H})$ and $\forall \mathcal{G} \in \mathbb{G} : \mathcal{G} \simeq c(\mathcal{G})$ hold, is called a *canonization*. The graph $c(\mathcal{G})$ is called the *canonical form* w.r.t c, and is denoted by $canon(\mathcal{G})$.

(a) First candidate pattern (b) Second candidate pattern

Figure 2: Possible patterns

Given the graph mining problem as specified in Figure 1, we have already established that Figure 2a is a valid pattern. When we try to mine a second pattern, we might suggest a pattern as shown in Figure 2b. A quick check, however, will show that there is a one-to-one mapping f such that both f as well as its inverse f^{-1} preserve edges. As a result, the first candidate pattern and the second are isomorphic. By Definition 1.5 only one of these two patterns can be the canonical form, and therefore only one of these two candidate patterns should be accepted as a valid pattern.

1.2 Rewording

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.** 1.1. 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 between the vertices and a labeling function over the vertices, as well as a classification (positive/negative).

Definition 1.6. Graph Mining (redefined) Given a set of vertices V and a set $\mathbb{G}$ of $\langle E, L, C\rangle$ triples, where E and L represent the edge relation and labeling function over the supply of vertices respectively, we look for a (set $\mathbb{P}$ of) graph(s) represented by tuple(s) (E_p, L_p) 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 $\forall v \in V : L_p(v) = L(f(v))$.

cleanly extend this definition to multiple graphs, or separate the single and multiple pattern case again

- $\#\Big\{\langle E, L, Pos\rangle \in \mathbb{G} \mid \exists f :$ f is an homomorphism from P to $\langle E, L, Pos\rangle\Big\} \geq N_+$
- $\#\Big\{\langle E, L, Neg\rangle \in \mathbb{G} \mid \exists f :$ f is an homomorphism from P to $\langle E, L, Neg\rangle\Big\} \leq N_-$

draft: possible different wording

When reasoning about graphs, we see *each graph* as a *coherent* ensemble of its own components: all characteristics (edges, labeling ...) of a graph are represented by separate entities or concepts, which are grouped together for each graph G in the triple that describes it. We refer to this as the *local coherence* of the graph representation. As graphs are the main concepts we talk about in the graph mining problem, keeping this concept together leads to a more natural KR representation. Furthermore, this locally coherent representation makes it very explicit that all example graphs are *independent*, and that the search for a homomorphism between pattern and example graph is independent as well. A good solver can then discover this independence and leverage it to achieve better performance.

A set of graphs, such as the argument of the cardinality operator above or the set of example graphs $\mathbb{G}$, is equivalent to a set of triples. The most straightforward representation would therefore be a ternary predicate. As the domains of this predicate range over predicates and functions, this predicate would be a higher-order predicate.

Illustrate local coherence with table

	E	L	C
$\mathcal{G}_1$	E_1	L_1	Pos
$\mathcal{G}_2$	E_2	L_2	Pos
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathcal{G}_n$	E_n	L_n	Pos

Figure 3: Local coherence

2 Related Work

to be expanded SERGEY

3 Modellings

3.1 IDP

3.1.1 Existential Second Order

The IDP language can express problems that consist of a set of symbols, called the vocabulary V, and a theory, called T, that uses symbols from this vocabulary. The symbols in the vocabulary can be propositions, but they can also represent predicates and functions. These last two types of symbols make the vocabulary, in general, *second order*. The theory T is restricted to a *first order* theory, extended with arithmetic, aggregates, and inductive definitions. Our inference of choice in the graph mining problem is model expansion; we search for an interpretation I of symbols in the vocabulary V such that this interpretation I satisfies the theory T. This corresponds to the implicit *existential quantification* of all symbols in the vocabulary, both the first order as well as the second order symbols. In conclusion, we say IDP can express model expansion for *Existential Second Order* problems.

The restriction to *Existential* Second Order means we cannot express the set of example graphs, as specified in Definition 1.6, as a higher-order predicate. One possible solution is to replace 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.** 1.6.

But, can we now express the abstraction (generalization) of knowledge about these properties, such as the positive homomorphic property. As evidenced in **Def.** 1.6, 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.

3.1.2 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)

3.2 Eventual encoding

```
//Homomorphism/2 is a higher-order predicate:
```

```
//Edge1 and Edge2 are predicates themselves.
homomorphism(<Edge1,Label1>, <Edge2,Label2>)
 ⟺  ∃ f: (∀ x, y : x ≠ y ⇒ f(x) ≠ f(y)) ∧
     (∀x, y : Edge1(x, y)  ⟹  Edge2(f (x), f (y))) ∧
     (∀ x : Label1(x) = Label2(f(x)))
```

```
{
    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 : Label1(x) = Label2(f(x))) ∧
     (∀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)  ⟹  #{ 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

3.3 ProB

4 Feature Comparison

4.1 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,_,_).
```

4.2 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?

5 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*
}
```

References

C. C. Aggarwal and J. Han, editors. *Frequent Pattern Mining*. Springer, 2014. ISBN 978-3-319-07820-5.

T. Guns, S. Nijssen, and L. D. Raedt. Itemset mining: A constraint programming perspective. *Artif. Intell.*, 175(12-13):1951–1983, 2011.

B. Négrevergne and T. Guns. Constraint-based sequence mining using constraint programming. In *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, pages 288–305, 2015.

S. Paramonov, M. van Leeuwen, M. Denecker, and L. De Raedt. An exercise in declarative modeling for relational query mining. In *International Conference on Inductive Logic Programming, Inductive Logic Programming, Kyoto, 20-22 August 2015*. Springer, Dec. 2015.