Noname manuscript No.
(will be inserted by the editor)

Knowledge Representation Analysis of Graph Mining

Matthias van der Hallen† · Sergey Paramonov† · Gerda Janssens† · Marc Denecker†

Received: date / Accepted: date

Abstract This paper analyses the graph mining problem, and the frequent pattern mining task associated with it. In general, frequent pattern mining looks for a graph which occurs frequently within a network or, in the transactional setting, within a dataset of graphs. We discuss this task in the transactional setting, which is a problem of interest in many fields such as bioinformatics, chemoinformatics and social networks.

We look at the graph mining problem from a Knowledge Representation point of view, hoping to learn something about support for higher-order logics in declarative languages and solvers. Graph mining is studied as a prototypical problem; it is easily expressible mathematically and exists in many variations. As such, it appears to be a prime candidate for a declarative approach; one would expect this allows for a clear, structured, statement of the problem combined with easy adaptation to changing requirements and variations.

Current state-of-the-art KR languages such as IDP and ASP aspire to be practical solvers for such problems [9]. Nevertheless, expressing the graph mining problem in these languages requires unexpectedly complicated and unintuitive encoding techniques. These techniques are in contrast to the ease with which one can transform the mathematical definition of graph mining to a higher-order logic specification, and distract from the problem essentials, complicating possible future adaptation.

In this paper, we argue that efforts should be made towards supporting higher-order logic specifications in modern specification languages, without unintuitive and complicated encoding techniques. We argue that this not only makes representation clearer and more susceptible to future adaptation, but might also allow for faster, more competitive solver techniques to be implemented.

Matthias van der Hallen is supported by a Ph.D. fellowship from the Research Foundation - Flanders (FWO - Vlaanderen).

†
Department Computer Science - KU Leuven,
Celestijnenlaan 200A,
3000 Leuven,
E-mail: firstname.lastname@kuleuven.be

Keywords Knowledge Representation · Higher Order · Graph Mining · Answer Set Programming · Imperative Declarative Programming

1 Introduction

Many real-world families of problems can be formalised as the combination of some smaller problems, and consequently lend themselves to a declarative modelling approach, as studied in Knowledge Representation (KR). Specifically, knowledge representation offers a natural framework for declarative modelling satisfying 'The Principle of Elaboration Tolerance' [33]. In short, this principle states that declarative specifications are easily adapted to new requirements or changed circumstances.

Not only do problems of a composite nature benefit from an elaboration tolerant approach, they are also often easily expressed using higher-order logic. However, current state-of-the-art knowledge representation systems such as IDP and Answer Set Programming (ASP) are elaboration tolerant but provide little or no support for higher-order expressions. This lack of support leads to contorted models employing many encoding techniques. Thus, it is an open challenge for KR systems to provide support for abstract, higher-order modelling while remaining elaboration tolerant. We study the *Graph Mining* problem as an example of such a composite problem, easily expressible in higher-order logic, with the aim of gathering insight in and deriving techniques for such problems in general. As a first step towards this goal, we will:

- Propose a higher-order encoding of graph mining that closely follows its mathematical model (Section 3).
- Explore how the current state-of-the-art KR systems model graph mining using modelling techniques (Section 4).
- Propose and experiment with additional solver techniques derived from these modelling techniques that can support higher-order encodings; as the modelling language is modified only with well-known generalisations of existing language constructs, elaboration tolerance is not affected (Section 5).

Graph mining One of the most fundamental tasks within the realm of data mining is *frequent pattern mining*: the task of enumerating patterns which occur frequently in a dataset. *Graph mining* is a variant of *frequent pattern mining* in which the patterns are structured as (labeled) graphs. The dataset in which patterns must occur is either a single large-scale network or a vast set of separate, smaller graphs. The latter option is often referred to as the *transactional setting*. In the context of graph mining, for a pattern to 'occur' in a graph $\mathcal{G}$, it must be homomorphic to a subset of the graph $\mathcal{G}$.

Owing to the research background of one of the authors, and because it is computationally more feasible, this work will only consider the transactional setting. This transactional setting is relevant as it can be used for *knowledge discovery* from graph structured data in many domains, such as chemoinformatics, natural language processing and bioinformatics. For example, within bioinformatics, graph mining can be used to find molecular substructures (such as benzene rings) that possibly predict or cause certain properties such as lumocity or the mutagenicity

of diseases such as Salmonella. Within natural language processing, graph mining can identify key concepts in a transcript from a graph representation of the natural language sentences.

As real-world problems are computationally challenging, the field of data mining has developed numerous specialised imperative algorithms. These different imperative algorithms correspond to the many variants of pattern mining tasks described in the literature, from various types of item set mining, where data is propositional, to tasks involving more structured data and patterns, such as trees and graphs. Well known examples of algorithms for frequent pattern mining in databases of graphs are gspan [42] and gaston [35].

However, the need for many different algorithms for only slightly different variants within pattern mining tasks has motivated the exploration of more declarative approaches. For example, it has been shown that Constraint Programming (CP) [15] and Answer Set Programming (ASP) [25] can express *item set mining*, which is a setting of frequent pattern mining where data is propositional and can be represented in a table. Their results demonstrate that such tasks can be accomplished in a declarative way with an acceptable performance penalty. Furthermore, its different variations can be supported with only minimal changes.

When mining more complex and structured data than item sets, such as graphs or sequences, predicate logic has been used for representation and inductive logic programming [34] has emerged as a way to mine such data. While we know of no earlier declarative approaches to graph mining, recent work on sequence mining [21] applies ASP to the basic constrained case of frequent sequence mining. Their solution performs pattern generation, the frequency check, and uses an extension of ASP called asprin [8] to prefer patterns which satisfy more involved properties such as maximality (no larger patterns exist) or coverage (no other pattern occurs in the same examples). However, because graphs are more complex structures than sequences, extending their solution to graphs is non-trivial. In the context of graphs, for example, checking *occurrence* corresponds with a homomorphism check (beyond P), instead of a (polynomial) subset check.

Owing to its descriptive complexity, the homomorphism check could easily be expressed using higher-order logic, but this is not supported by state-of-the-art declarative languages such as IDP and ASP. We show that graph mining can nevertheless be supported using various encoding techniques, making it an ideal candidate for our case study into combining support for higher-order logic with elaboration tolerance.

Higher-order logic As mentioned earlier, the composite nature of the graph mining problem lends itself for a higher-order logic specification, in which quantified variables can range not only over individuals but also over sets, or sets of sets. For example, a high-level view of the graph mining problem consists of generating connected labeled graphs (patterns), checking whether they occur frequently within a dataset, and filtering out patterns which are too similar to others (e.g. *isomorphic*), leaving only those patterns which we will call *canonical*. In this view, it makes sense to describe the mechanisms behind checking for canonicity and occurrence separately, which, as we will show, translates nicely to higher-order specifications.

Specification languages offer varying levels of support for higher-order logic. On the one hand, meta-programming, as known from Logic Programming [1], has

inspired the introduction of higher-order atoms in Hex [16] and the higher-order syntax in HiLog [11]. Predicate symbols can be either constants as in Prolog (first-order case) or variables (second-order case). The latter range over predicate names, and not the predicate space itself, essentially combining second-order syntax with first-order semantics. On the other hand, formal specification languages such as Z [7], B [2], Event-B [3] and TLA [28] extend predicate logic with set theory and offer higher-order datastructures. ProB [29] is a constraint solver, animator and model checker for such languages, implemented in SICStus Prolog.

While it is possible to express the graph mining problem in such languages directly using higher-order logic, earlier work [23] has shown that these systems miss flexibility with respect to inferences other than model expansion, and that their performance does not rival that of systems based on revolutionary techniques such as CDCL [39]. However, these systems, examples of which are the ones for the IDP [14] and the ASP [17,18] languages, currently do not allow higher-order syntax. Nevertheless, several techniques exist for these languages that allow the user to simulate higher-order logic to model problems such as graph mining. This observation leads us to inquire whether these techniques can be generalised and be used to provide these languages with built-in support for higher-order.

2 Formalization of graph mining

In this section we mathematically formalize the transactional graph mining problem. As we will only consider the transactional setting in this work, we will simply refer to it as 'graph mining'. First, we define graphs, graph homomorphism, and the concept of a pattern. We then express what it means for a pattern to be canonical, which is needed when we want to mine more than one pattern.

2.1 Patterns

We start with a comprehensive formal definition of the graph mining problem. First, we will assume the existence of two finite, sufficiently large sets: a set V consisting of vertices, and a set L of labels for those vertices.

Definition 1 (Labeled Graph) A labeled graph $\mathcal{G}$ is a tuple $\langle N, E, l\rangle$ where N is a subset of the vertices V, called the nodes of the graph $\mathcal{G}$, E is a binary predicate on N that represents the set of (directed) edges and l is a unary function from N to L.

Definition 2 (Connectedness) A graph $\mathcal{G} = \langle N, E, l\rangle$ is *connected* iff for each pair of nodes v and v' in N, there exists an edge $\left(v, v'\right) \in E$ or there exists a sequence $v, v_1 \dots v_n, v'$ such that there exist edges (v, v_1), (v_i, v_{i+1}) and $\left(v_n, v'\right) \in E$, where $1 \leq i \leq n-1$.

Definition 3 (Graph Homomorphism) A *(injective) graph homomorphism* f from a labeled graph $\mathcal{P} = \langle N, E, l\rangle$ to a labeled graph $\mathcal{G}' = \langle N', E', l'\rangle$ is an (injective) mapping $f : N \to N'$ from nodes of $\mathcal{P}$ to nodes of $\mathcal{G}'$ such that:

- $\forall u, v \in N : (u, v) \in E \implies (f(u), f(v)) \in E'$ (the mapping preserves edges), and

– $\forall v \in N : l(v) = l'(f(v))$ (the mapping respects labelings).

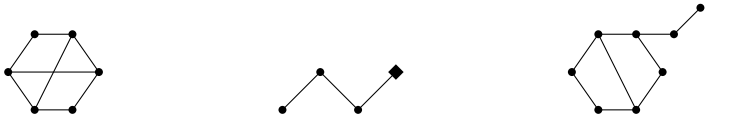
If an (injective) graph homomorphism from graph $\mathcal{P}$ to $\mathcal{G}'$ exists, we say $\mathcal{P}$ is (injectively) *homomorphic*[1] with $\mathcal{G}'$.

Definition 4 (Graph Mining) Given a sufficiently large set of vertices V and labels L, two sets $\mathbb{G}_+$ and $\mathbb{G}_-$ of positive, respectively negative example graphs over V and L, two thresholds N_- and N_+, and a graph $\mathcal{T}$ over V and L called the template, we look for a graph $\mathcal{P}$ represented by tuple $\langle N_\mathcal{P}, E_\mathcal{P}, l_\mathcal{P} \rangle$ such that:

– $\mathcal{P}$ is a vertex-induced subgraph of $\mathcal{T}$, meaning that the edges of $\mathcal{P}$ are exactly those edges of $\mathcal{T}$ such that both endpoints are chosen to be nodes of $\mathcal{P}$,
– $\mathcal{P}$ is fully *connected*,
– $\mathcal{P}$ is injectively homomorphic with at least N_+ positive examples $\mathcal{G}_+ \in \mathbb{G}_+$,
– $\mathcal{P}$ is injectively homomorphic with at most N_- negative examples $\mathcal{G}_- \in \mathbb{G}_-$.

We call these injective homomorphisms the positive (negative) homomorphisms, and the restriction on their number the positive (negative) homomorphic property, respectively.

Note that we choose *injective* homomorphisms as the matching operator in this definition, and include the concept of a template graph to guide the search as well as to limit the search space. These choices are inspired by their appropriateness for many use cases in the realm of bioinformatics, chemoinformatics and social networks. However, both of these choices can be changed effortlessly, in the mathematical definition as well as in any specifications of the problem: We can easily drop the injectivity constraint in any logic specification, and can choose the fully connected graph as template without loss of generality. For the remainder of the work, we will use 'homomorph' to mean 'injectively homomorph' unless specifically stated otherwise.



(a) Positive Example (b) Negative Example (c) Template Graph

Figure 1: A graph mining instance ($N_+ = 1, N_- = 0$) with pattern candidates. Node labels are differentiated by the shape of their indicator (circle, diamond).

(a) Candidate a (b) Candidate b

Figure 2: Pattern candidates for the graph mining instance shown in Figure 1

As an example of a graph mining problem instance, take the problem set shown in **Figure** 1. Node labels are differentiated by the shape of their indicator: circle or diamond. All nodes have the same (circle) label, except for the rightmost node

[1] Note that within the data mining community, injectively homomorphic is also commonly known as subgraph isomorphic.

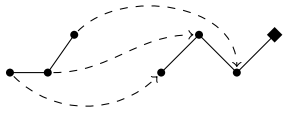


Figure 3: A mapping of candidate 2b to the negative example 1b.

in the negative example 1b. Furthermore, when interpreting these graphs, all (visualised) edges are bidirection, i.e. they would represent two opposingly directed edges when translated to textual data. The angles and lengths of edges are irrelevant, only the connections are relevant. We take the positive and negative thresholds to be $N_+ = 1, N_- = 0$, meaning we require at least one homomorphism with a positive example and allow no homomorphisms with negative examples. There is one positive example (**Figure** 1a), and one negative example (**Figure** 1b), while **Figure** 1c shows the template graph.

Figures 2a, 2b show a pattern and an invalid pattern candidate respectively: They are both connected subgraphs of the template. However, because we require at least one homomorphism with a positive example, and allow no homomorphisms with negative examples (i.e., problem parameters $N_+ = 1$ and $N_- = 0$), **Figure** 2a represents a 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** 2b, this graph is clearly homomorphic with both the positive as well as the negative example: A possible mapping from 2b to the negative example 1b is shown in **Figure** 3. Therefore, 2b is an invalid pattern candidate, not a pattern.

2.2 Canonical patterns

To extend on the graph mining task described above, 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 5 (Graph Isomorphism) A graph isomorphism f between two labeled graphs $\mathcal{G} = \langle N, E, l \rangle$ and $\mathcal{G}' = \langle N', E', l' \rangle$ is a *one-to-one* mapping $N \to N'$ such that f represents an injective homomorphism from $\mathcal{G}$ to $\mathcal{G}'$, and its inverse f^{-1} represents an injective homomorphism from $\mathcal{G}'$ to $\mathcal{G}$. If there exist graph isomorphisms between $\mathcal{G}$ and $\mathcal{G}'$ we say $\mathcal{G}$ and $\mathcal{G}'$ are *isomorphic*.

(a) First candidate pattern

(b) Second candidate pattern

Figure 4: Possible patterns

Given the graph mining problem instance specified in **Figure** 1, we have already established that **Figure** 4a is a pattern. When we try to mine a second pattern,

we might suggest a pattern as shown in **Figure** 4b. 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, both patterns candidates are isomorphic, and thus only one should be accepted as a pattern.

Definition 6 (Canonical Patterns) A set of *canonical patterns* is a set $\mathbb{P}$ of patterns $\mathcal{P}_1, ..., \mathcal{P}_n$, such that for each pair of different elements (of $\mathbb{P}$) $\mathcal{P}_i, \mathcal{P}_j$ holds that there does not exist an isomorphism between $\mathcal{P}_i$ and $\mathcal{P}_j$.

When we mine multiple patterns, we will pose the additional requirement that the mined patterns must be canonical. Of course, with the above definition of canonicity, many solutions will exist: any pattern can be interchanged for any of its isomorphic counterparts to generate a new solution. If this is unwanted, this can be prevented by introducing an ordering on the isomorphic patterns, and requiring that each pattern in the solution is the minimal pattern among its isomorphic counterparts.

3 A higher-order specification of Graph Mining

In this section, we explore how the mathematical formalization of the graph mining problem can be translated towards a higher-order specification. In the first section, we will discuss how graphs introduce higher-order objects when modeled closely to the mathematical definition. Next, we will discuss the higher-order specification of the complete graph mining problem. We conclude by identifying a set of desired properties for graph mining specifications and their solvers, which the proposed higher-order specification satisfies.

3.1 Representation of graphs

Graphs are the main concept in the graph mining problem, and, when represented using tuples $\langle N, E, l \rangle$, they take the form of *composite objects*: these graphs are a collection of first-order objects, namely two predicates and a function. As such, a set of graphs is equivalent to a set of tuples: the most straightforward representation of such a set would be a ternary predicate, with the node and edge predicate and the labeling function as arguments:

```
Patterns = {({1,2,3}, {(1,2),(2,3),(3,1)},
            {(1↦a), (2↦b), (3↦a)})}
```

It is very natural to consider and represent 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 $\mathcal{G}$ in the tuple that describes $\mathcal{G}$. We refer to this as the *local coherence* of the graph representation.

Alternative representations could, for example, introduce an edge predicate for each graph separately (e.g. called `edge_g1/2`[2], `edge_g2/2`), or, as we will be forced to do later on, they could introduce an edge predicate for *all* graphs at once. This obscures the relationship between the different characteristics of the same graph:

[2] We use `predicate_name/n` to mean the predicate with name `predicate_name` and arity `n`.

- In the first alternative this relationship is only present in the *name* of each predicate (which prohibits us to reason about it in a general way). Furthermore, without any additional constraints, it is possible to specify a graph only partially, e.g., only provide an edge relationship.
- In the second alternative the relationship can be expressed using an identifier. However, in contrast to the higher-order tupling approach, it is still possible to specify a graph only partially.

Representing graphs instead as a tuple of its components is not only a very natural representation, it also very explicitly shows that all example graphs are *independent*, and that the searches for homomorphisms between a pattern and example graphs are independent too. This motivates us to reason about graphs as locally coherent objects in our logical models as well. The next section explores a higher-order specification that achieves this goal.

3.2 A higher-order specification

In **Listing** 1, we propose a specification for the graph mining problem, using features such as higher-order logic and inductive definitions. Regarding syntax and style, we devise a syntax for illustration purposes inspired by IDP [14], which closely corresponds to FO logic with $\leftarrow$ for inductive definitions, as opposed to $\Rightarrow$ for classical implication, and [*type*] for type annotations. We identify four major syntactical additions w.r.t. regular IDP syntax:

- We introduce the keyword `so-type`. The `so-type` keyword can be used to define a second-order type. As such, the type does not represent a set of domain elements from the Herbrand universe, but instead represents a set of (tuples of) predicates or functions. These predicates and functions themselves must be typed using first-order types.
- When presented with an object of a second-order type consisting of a tuple, one will often want to access one specific part of the tuple. To this end, the different parts of a tuple are named. These names provide a way to project a second-order object to one of its parts using .-syntax familiar from object-oriented programming. For example, if a tuple `result` representing test scores contains two elements, the `name` and `score`, then `result.score` accesses the score.
- We introduce the possibility to quantify over predicates or functions, using the special quantifiers $\forall_{SO}$ and $\exists_{SO}$. These quantifications must be typed: $\exists_{SO}$ `F [I:O]` means that there exists a function `F` which takes elements of type `I` as input and returns elements of `O` as output. Likewise, $\exists_{SO}$ `P [(I,O)]` means that there exists a predicate `P`, taking elements from `I` and `O` as its first and second argument, respectively.
- We allow higher-order predicates with arguments of a second-order type. These predicates can be defined using an inductive definition. These are predicates which take (tuples of) predicates and functions as an argument. The semantics of these higher-order inductive definitions is defined in Dasseville et al. [13], where they were called *template definitions*. For an example, we refer to **Lines** 32-40 of **Listing** 1, where the predicate `is_pattern/1`, which takes a second-order argument `graph` as an argument, is defined using an inductive definition.

As with IDP, we first define a vocabulary `V`, and define a theory `T` over this vocabulary `V`. Then, when presented with a specific graph mining instance, we can encode this into a structure `S` and perform the model expansion inference to find a solution. We will further explore the contents of these three language blocks in the sections below.

3.2.1 Vocabulary

As mentioned above, the first thing we define is the vocabulary `V`. First, we define the types `vertex` and `label` to represent the set of vertices V and set of labels L from the mathematical formalization. Next, we define the second-order type `graph`, which is declared as a tuple of a predicate `node/1`, a predicate `edge/2`, and a function `labeling`. These predicates and functions represent the exact subset of vertices which are the nodes, the edges between these nodes and the labeling of these nodes. As such, we have defined all the necessary types for the graph mining problem.

Now, we can define the necessary predicates: We define the higher-order predicates `homomorph/2` and `isomorph/2`, which are binary relations between graphs. Next, we define some simple sets of graphs as unary higher-order predicates over graphs: the positive example set `positive/1` and its negative counterpart `negative/1`, the set of canonical patterns `canonical_pattern/1`, and the set of patterns `is_pattern/1`. Lastly, we define:

- a ternary predicate `connected/3`, which is true if the two nodes represented by the first two arguments are in fact connected within the graph given as a third argument,
- a higher-order function `template` which refers to the chosen template graph, and
- the two thresholds from the problem statement in Definition 4, N_- and N_+ as simple integers.

3.2.2 Theory

In the theory, we define a number of the higher-order predicates using the concept of template definitions, as described by [13]. Whenever a defined predicate accepts a second-order type as argument, it can be decomposed using matching (e.g., **Line** 20). Quantification over second-order objects uses annotated quantifiers ($\exists_{SO}$ and $\forall_{SO}$) and must be typed (any unary predicate represents a type), e.g., **Line** 21. We will adhere to the convention that variables referring to higher-order objects are upper case, whereas variables referring to propositional objects are lower case.

First, we define the concepts of `homomorph/2` and `isomorph/2`: We express the constraint that two graphs are only homomorph if it is possible to find a function `F` from nodes of the first graph to nodes of the second graph, as can be derived from the existential second-order quantification $\exists_{SO}$ in combination with the typing statement `[N1:N2]` (**Line** 21). In line with Definition 3, we first express that this function must be injective (**Line** 21). Next, we specify that it must preserve edges (**Line** 22), and we conclude by specifying that it must preserve labels as well (**Line** 23). For the definition of `isomorph/2`, we follow the mathematical definition in the same way, except for the usage of f^{-1}: as computing the inverse of a function is not an operation in higher-order logic, we specifically state how the function `F`

must be bijective (**Line** 25 and 26), and how we can use `F` to express that f^{-1} must preserve edges as well (**Line** 28).

Next, we define the concept of connectedness within a given graph: This can be defined rather straightforwardly using an inductive definition by nothing that either the two nodes are connected directly, or there exists a third node connected with both argument nodes.

We continue by defining the concept of a pattern, following the requirements of Definition 4:

- a pattern is a vertex-induced subgraph of the template (using dot notation to access the separate components of a variable of second-order type, **Line** 35),
- which is also fully connected, and
- if we count the number of graphs from the positive (resp. negative) example set such that the proposed pattern graph is homomorphic with the chosen graph, the result should exceed (resp. should not exceed) the positive (resp. negative) threshold, as evidenced by the count aggregates (**Line** 37-38).

Lastly, we provide two constraints saying that for a graph `P` to be a canonical pattern, it must first be a pattern, and secondly, no other canonical pattern `P2` must exist such that `P` and `P2` are isomorphic.

This encoding compactly specifies the graph mining problem, in a way that closely corresponds to its mathematical definition, providing several general graph properties as templates.

3.3 Desired properties of graph mining specifications

Using the graph mining problem as a case study, we derived a set of desirable properties that a good KR specification and its associated solver should satisfy. First, we discuss properties of the KR specification itself:

1. Labeled graphs are the main concept in the mathematical definition of the graph mining problem. In this definition, labeled graphs are seen as a mathematical object consisting of a vertex relation, an edge relation and a labeling function. Thus, a good KR specification should treat labeled graphs as (higher-order) objects.
   It is clear from the second-order type `graph` *in* ***Listing*** *1 that this higher-order specification satisfies this property.*
2. All example graphs are independent, so the search for a homomorphism between a pattern and a given example graph can be performed independently. A good KR specification should allow one to write the necessary quantifications locally to make this evident, as opposed to quantifying globally using the vocabulary.
   The universal quantification over example graphs hidden in the count aggregates of ***Line*** *37 in* ***Listing*** *1, combined with the existential quantification of* `F` *on* ***Line*** *21, clearly identifies the independence: for every single example graph, a* ***separate*** *function* `f` *proving the homomorphism can be chosen.*
3. The search for a homomorphism between pattern and example graph is always the same, regardless of the sign of the example graph (negative or positive). The only difference is the at most/at least constraint on the number of homomorphisms. A good KR specification preserves the similarity of these constraints.

Listing 1: Higher-order encoding for the general graph mining problem

```
1  vocabulary V {
2    type vertex
3    type label
4    so-type graph of (node(vertex), edge(vertex,vertex), labeling(vertex):label)
5
6    homomorph(graph, graph)
7    isomorph(graph, graph)
8    positive(graph) % a set of positive graphs
9    negative(graph)
10   canonical_pattern(graph)
11   is_pattern(graph)
12   connected(vertex,vertex, graph)
13   template:graph % a given template
14   N_: int
15   N+: int
16 }
17
18 theory T {
19  {
20  homomorph((N1, E1, L1), (N2, E2, L2)) ←
21     (∃SO F [N1:N2]: (∀ x [N1] y [N1]: x ≠ y ⟹ F(x) ≠ F(y)) ∧
22      (∀ x [N1] y [N1]: E1(x, y) ⟹ E2(F(x), F(y))) ∧
23      (∀ x [N1]: L1(x) = L2(F(x)))).
24  isomorph((N1, E1, L1),(N2, E2, L2)) ←
25     (∃SO F [N1:N2]: (∀ y [N2]: ∃ x [N1]: F(x)=y) ∧
26      (∀ x [N1] y [N1]: x ≠ y ⟹ F(x) ≠ F(y)) ∧
27      (∀ x [N1] y [N1]: E1(x, y) ⟹ E2(F(x), F(y))) ∧
28      (∀ x [N2] y [N2]: E2(x, y) ⟹ ∃ fx [N1] fy [N1]: E1(fx, fy) ∧ x = F(fx) ∧ y = F(fy)
          )∧
29      (∀ x [N1]: L1(x) = L2(F(x)))).
30  connected(x, y, (N, E, L)) ← E(x, y) ∨ E(y, x).
31  connected(x, y, (N, E, L)) ← ∃ z [N]: connected(x, z, (N, E, L)) ∧ connected(z, y, (N, E,
        L)).
32  is_pattern((N, E, L)) ←
33    (
34      (∀ x [N]: (template.node(x) ∧ ∀ y [N] : E(x,y) ⟺ template.edge(x,y)
35          ∧ L(x) = template.labeling(x))) ∧
36      (∀ x [N] y [N]: x ≠ y ⟹ connected(x, y, (N,E,L)))∧
37      (#{ Pos : positive(Pos) ∧ homomorph((N,E,L), Pos) } ≥ N+) ∧
38      (#{ Neg : negative(Neg) ∧ homomorph((N,E,L), Neg) } ≤ N_)
39    )
40  }
41  ∀P [graph] : canonical_pattern(P) ⟹ is_pattern(P).
42  ∀P [graph] P2 [graph] : canonical_pattern(P)∧canonical_pattern(P2)∧P≠P2 ⟹ ¬isomorph(P
        , P2).
43 }
```

*The great similarity between **Lines** 37 and 38 shows that our proposed higher-order specification satisfies this property.*

4. We want to be able to find multiple, non-isomorphic, patterns.
   The definition of `canonical_pattern/1`, and the definition of `is_pattern/1` as a set of pattern graphs allows us to express the problem independent of the number of patterns we want to mine.
5. We want to express constraints such as connectedness of the different nodes in the pattern.
   *The concept of inductive definitions, as used in **Lines** 30-31 of **Listing** 1 shows that we can express constraints such as connectedness in an easy way.*

We also identify some desirable properties for the systems solving a good KR specification of the graph mining problem:

6. We want to perform multiple inferences on the problem, with only minimal changes to the model. In other words, the system should be elaboration tolerant with respect to other inferences, as well as new constraints.
   For example, we might not be interested in just any set of patterns, instead we might want a set of 5 patterns such that they share the highest number of nodes.
7. We prefer specification(s) which can (together) be solved in a single solver call. While specifications are preferably modular to make it easier to reuse them, ideally the composition of specifications would be solvable by a single solver call, requiring no procedural code to tie them together.

The higher-order encoding above satisfies the different properties we identify for a modeling; as such we view it as a *preferred way of encoding* the graph mining problem. Nevertheless, state-of-the-art specification systems either do not accept such specifications, or, e.g., ProB, miss the flexibility of multiple inferences or the performance [23] of techniques such Conflict Driven Clause Learning (CDCL) which can (up-to exponentially) reduce the search space by learning new clauses when encountering conflicts and backjumping.

In the next section, we explore which encoding techniques enable us to write a working specification for the graph mining problem in state-of-the-art specification systems: IDP and ASP in particular.

4 First-order encodings of Graph Mining

In the previous section, we have shown how graph mining could be specified in a system which supports higher-order logic. In this section, we investigate how state-of-the-art KR systems without support for higher-order logic, such as IDP and ASP, can model the graph mining problem, paying special attention to the use of encoding techniques, which might be used in the future to support higher-order logic in general.

4.1 IDP

First, we will explore how we can encode the graph mining problem in a state-of-the-art first-order solver such as IDP. We base ourselves on the mathematical specification of graph mining introduced in Section 2, as well as the higher-order specification explored in Section 3.

4.1.1 Existential Second Order

The IDP language allows problem specifications written in *first-order* (FO) theories T, extended with types, arithmetic, aggregates, and inductive definitions. **Listing** 2 shows an example. The symbols in these theories can be quantified locally, or quantified implicitly in the vocabulary V. Symbols quantified locally can only be propositional, whereas the vocabulary can contain first-order symbols such as functions or predicates (making the vocabulary a *second-order object*).

In the graph mining problem, we are looking for an interpretation I of the symbols in vocabulary V such that I satisfies T, called a model. This corresponds

Listing 2: IDP example using inductive definitions

```
vocabulary V{
   type node
   edge(node, node)
   connected(node, node)
}
theory T : V {
   ∀n[node] : ∃n2[node] : edge(n, n2) ∨ edge(n2, n).
   {
      connected(x, y) ← edge(x, y) ∨ edge(y, x).
      connected(x, y) ← ∃z [node] : connected(x, z) ∧ connected(z,y).
   }
}
structure S : V{ node = {1;2;3} }
structure Result : V{
   node = {1; 2; 3}, edge = {1,1; 1,2; 2,3}
   connected = {1,1; 1,2; 1,3; 2,1; 2,2; 2,3; 3,1; 3,2; 3,3}
}
```

to existential quantification of all (including FO) symbols in V. **Listing** 2 shows an example of how IDP extends a given interpretation `S` into a model `Result`. Due to the existential quantification of symbols in V, and the lack of locally quantifiable FO symbols, IDP is limited to model expansion for *existential second-order* problems, which does not include graph mining. We now expand on the underlying shortcomings, and how to sidestep them.

Inferences One of the main philosophies of IDP is its underlying *Knowledge Base* paradigm [14]. Essentially, this paradigm states that a modeller should model the knowledge within a problem domain, without thinking of how data will flow when solving specific queries, or wondering which inference will be performed. Instead, it should be possible to perform various inferences on a single, unchanging specification of the graph mining problem. For example, the most straightforward inference in the case of graph mining would likely be *model expansion*. **Listing** 2 shows how model expansion would expand the structure `S` into the structure `Result`. Other inferences of interest for the graph mining problem are, for example, *optimization*. Optimization would allow us to, e.g., minimize or maximize over the number of nodes in the pattern graph, or the number of nodes in the pattern with a certain label, with only minimal changes to the specification.

Definitions versus Constraints A main feature of the IDP language is that it supports FO formulas as well as a rule-based definition constructs (between curly braces). The FO formulas express open world knowledge while definitions express closed world knowledge and can be used to express inductive or recursive definitions such the definition of `connected/2` in **Listing** 2.

This follows from the use of the well-founded semantics underlying the definition construct. In IDP, the theory of two atomic FO axioms `e1(1,2). e1(2,1).` expresses the open world knowledge that `(1,2)` and `(2,1)` belong to the predicate `e1/2`, while the definition `{e1(1,2). e2(2,1).}`, written with brackets, expresses a definition by exhaustive enumeration, hence the closed world knowledge that `e1` *is* the set `(1,2)`, `(2,1)`. E.g., the first does not entail that `(1,1)` does not belong to `e1/2`, while the definition does. The theory in **Listing** 2 expresses that connected

is the transitive closure of the edge relation, and that the connected relation is the total relation. The combination of definition and axiom induces a strong constraint on the value of the edge relation.

4.1.2 Modelling the graph mining problem in IDP

In this subsection we identify three main issues encountered when modelling the graph mining problem:

- the representation of graphs,
- local ∃ quantification over functions, and
- local ∀ quantification over functions.

The paragraphs below discuss each of the issues and provide an overview of the ways we can currently solve them.

Issue 1: representing graphs First, we must represent the sets of graphs, as specified in **Def.** 4. **Listing** 3 shows how this was done in higher-order logic, defining a higher-order predicate `positive/3` with the node predicate, edge predicate and labeling function as arguments [3]. The first graph consists of nodes 1 (labeled a), 2 (labeled b) and is fully connected. This locally coherent representation preserves a graph as an independent tuple of predicates and functions. However, IDP's vocabulary `V` cannot contain such a second-order symbol.

One possible solution is to replicate for each graph the different characteristic predicates and functions, as shown in **Listing** 4, which uses different predicate names for every part of every graph. Using this solution, encoding a property such as "In every graph, all nodes have at least two outgoing edges" must be stated for every graph and its edge predicate explicitly, as no relation exists between the different edge predicates and label functions:

```
∀ n[node] : ∃ n1 [node] n2[node] : e1(n, n1) ∧ e1(n, n2) ∧ n1 ≠ n2.
∀ n[node] : ∃ n1 [node] n2[node] : e2(n, n1) ∧ e2(n, n2) ∧ n1 ≠ n2.
```

It is clear that this solution is not a good KR approach. Furthermore, it is undesirable due to the way it scales with growing problem instances: it prohibits the abstraction (generalization) of knowledge in the theory.

[3] Note the use of inductive definitions, in contrast to constraints, as this allows the derivation of negative knowledge i.e. `positive/3` *only* contains these two graphs and no others.

Listing 3: Higher-order predicate modeling the set $\mathbb{G}_+$ of **Def.** 4.

```
{
  positive({1,2}, {1,2; 2,1}, {1↦a; 2↦b}).
  positive({1,2,3}, {1,3; 2,1}, {1↦c; 2↦b; 3↦a}).
}
```

Listing 4: Multiple individual global relations

```
{
  e1(1, 2). lb1(1)=a.
  e1(2, 1). lb1(2)=b.
  e2(1, 3). lb2(1)=c.
  e2(2, 1). lb2(2)=b.
            lb2(3)=a.
}
```

Listing 5: Disjoint union using indexed global relations

```
{
  e(g1, 1, 2). lb(g1, 1)=a.
  e(g1, 2, 1). lb(g1, 2)=b.
  e(g2, 1, 3). lb(g2, 1)=c.
  e(g2, 2, 1). lb(g2, 2)=b.
               lb(g2, 3)=a.
}
```

A more workable solution is to represent each characteristic property, such as the edge relation, by a single global relation for all graphs, as shown in **Listing** 5. This relation behaves the way it should for a specific graph instance based on an additional argument serving as an identifier for the graph of interest. This global edge relation now corresponds to the *disjoint* or *tagged union* of the graphs' edge relations, with tags drawn from a set `g` of graph identifiers. Generalizing over the different graphs, we can now encode the property stated above as:

```
∀ gid[g] : ∀ n[node] : ∃ n1 [node] n2[node] : e(gid, n, n1) ∧ e(gid, n, n2) ∧ n1 ≠ n2.
```

Although this representation is the de facto standard way of representing complex objects such as graphs, it is clear that this representation forces us to give up the local coherence of graph characteristics that was present in **Def.** 4. For example, without additional constraints, it is still possible to specify a graph $\mathcal{G}$ only partially, e.g., by providing only an entry in the global `edge/3` relation. Note that the higher-order model from Section 3 allows elegant expression, as it introduces specific terms (tuples and sets) which could elegantly express these graph characteristics in a way that preserves local coherence.

Issue 2: local ∃ quantification over functions The positive homomorphic property can be expressed using a count aggregate, as shown in **Listing** 6. First we quantify over all example graphs `G`, or per *issue 1*, their identifiers, and subsequently express that there must exist a function `F` that represents a homomorphism from our pattern graph $\mathcal{P}$ to `G`.

Listing 6: Quantifying over functions locally.

```
#{G | G ∈ 𝔾+ ∧∃ F : F is a homomorphism from 𝒫 to G} ≥ N+
```

However, IDP forbids us from locally quantifying over first-order symbols such as the function `F` from **Listing** 6. We must promote the homomorphic functions to a symbol in the vocabulary, even though we are only interested in the existence of a mapping, not its identity. Reusing the disjoint union technique proposed above avoids the need to introduce a homomorphic function for each example graph

separately. Note, we introduce a function `f/2` representing all homomorphisms, and explicate its dependency on a specific example graph using an additional argument `gId`. In second-order logic, this dependency would follow directly from the syntactic order of the quantifications.

Listing 7: Globalized existential functions

```
Vocabulary V {
...
partial f(graphid, vertex):vertex
...
}
...
#{gId | gId ∈ 𝔾₊ : f(gId) is a homomorphism from 𝒫 to gId} ≥ N₊
```

While all example graphs have an `edge`, `label`, ... relation, not all example graphs have a homomorphic function. Therefore, `f` is not defined for graph identifiers that correspond to such graphs, meaning `f` has become a partial function.

By adopting this proposed solution, we can now write an IDP specification for the graph mining problem handling only the positive constraint, as shown in **Listing** 7. Note that without the negative constraint, the problem is of a simpler nature (it is an NP decision problem). The next issue discusses how we can add the negative constraint into our IDPmodel.

Issue 3: local $\forall$ *quantification over functions* It is possible to restate the negative homomorphic constraint to *deciding that no homomorphism exists* for enough of the negative examples. However, deciding that no homomorphism from one graph to another exists is a coNP decision problem. As an NP (or Σ_1^p) solver, IDP cannot solve this problem directly. One might be tempted to simply specify the negative homomorphic property simply as:

`#{g | g ∈ 𝔾₋: f(g) is a homomorphism from 𝒫 to g}≤ N₋.`

However, the IDP solver has no obligation to maximize the number of homomorphisms it finds for `f`, only to satisfy the constraints. Thus, it can choose `f` such that it does not represent a homomorphism for a graph `g` $\in \mathbb{G}_-$. As our constraints are satisfied, we are led to believe that our pattern candidate is a valid pattern.

Immerman [24] has shown that this is inherently linked to IDPs limit to Existential Second Order. Indeed, checking that our pattern $\mathcal{P}$ is homomorphic with no more than N_- negative graphs is equivalent with checking that enough negative examples `G` exist for which no homomorphism exists (**Listing** 9). This clearly leads to a universal quantification over a function variable, which IDP cannot express.

Listing 9: Quantifying over functions locally.

```
#{g | g ∈ 𝔾₋ ∧ ∀ f : f is not a homomorphism from 𝒫 to g}
```

A way to work around this is by encoding the dual (i.e., negated) problem, and conclude that the problem is satisfied if and only if no model exists for the dual problem. This can be checked using an NP solver. However, this technique can only be implemented in IDP by writing two theories:

- one (positive) theory $\mathcal{T}^+$ (see **Listing** 8), which expresses the positive homomorphic property and generates pattern candidates, and

Listing 8: IDP specification handling the positive constraint of the graph mining problem

```
vocabulary V_pos{
  type vertex isa nat
  type label
  type graphid

  //Predicates determining the template graph.
  template_node(vertex)
  template_edge(vertex, vertex)
  template_label(vertex):label

  //Predicates describing the pattern graph
  pattern_node(vertex)
  pattern_edge(vertex, vertex)
  pattern_label(vertex):label

  //Predicates describing the positive example graphs
  example_edge(graphid, vertex, vertex)
  example_label(graphid, vertex):label
  N+: int

  partial f(graphid, vertex):vertex //Represents the homomorphisms with the example graphs
  homo_with(graphid) //True for graphs for which f represents a correct homomorphism
  connected(vertex, vertex) //connected(a, b) is true if there exists a path
  //from a to b in the pattern
}

theory Positive:V_pos{
   //The pattern is a vertex-induced subgraph of the template:
   ∀x [vertex] : (pattern_node(x) ⟹ template_node(x))
       ∧ (∀ y [vertex] : pattern_edge(x, y) ⟺ (template_edge(x, y) ∧ pattern_node(x) ∧
            pattern_node(y))) ∧
       ∧ pattern_label(x) = template_label(x).
   //The pattern is a connected subgraph of the template: From every node in the pattern,
   //There exists a path to every other node in the pattern.
   ∀x [vertex] y[vertex] : x ≠ y ∧ pattern_node(x) ∧ pattern_node(y) ⟹ connected(x, y).
   {
     connected(x, y) ← pattern_edge(x, y) ∨ pattern_edge(y, x).
     connected(x, y) ← ∃z[vertex] : connected(x, z) ∧ connected(z, y).
   }

   //Existence of a homomorphic f from the pattern to example graph with graphid gid.
   ∀gid[graphid] : ∀x[vertex] : homo_with(gid) ∧ pattern_node(x) ⟺ ∃ y[vertex] : y=f(
        gid,x).
   ∀gid[graphid] : ∀x [vertex] y[vertex] : homo_with(gid) ∧ pattern_node(x) ∧ pattern_node(
        y) ∧ x≠y ⟹ f(gid, x) ≠ f(gid,y).
   ∀gid[graphid] : ∀x [vertex] y[vertex] : homo_with(gid) ∧ pattern_node(x) ∧ pattern_node(
        y) ∧ pattern_edge(x,y) ⟹ example_edge(gid, f(gid,x), f(gid,y)).
   ∀gid[graphid] : ∀x[vertex] : homo_with(gid) ∧ pattern_node(x) ⟹
      pattern_label(x) = example_label(gid, f(gid,x)).

   //At least N homomorphisms must be found
   #{ gid [graphid] : homo_with(gid) } >= N+.
}
```

– one negative theory $\mathcal{T}^-$, shown in **Listing** 10, which expresses the (dual of) negative homomorphic property and rejects pattern candidates that do not satisfy this constraint.

In IDP, one must provide procedural code that ties these two theories and their inferences together, allowing pattern candidates to be communicated between them.

Canonicity As graph isomorphism is known to be in NP (recent research suggests it is in the Quasi-Polynomial complexity class QP [5]), the isomorphism restriction when looking for multiple patterns is no more complex than coNP. Therefore, we can use the same technique of encoding the dual and performing a satisfiability check that must fail for the canonicity requirement.

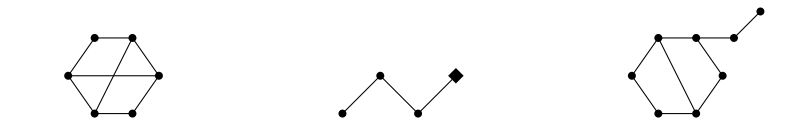
However, it is at this point we take into account the evaluation strategy. As mentioned above, having two separate theories means that we must tie the inferences together using procedural code. This can be done using the Lua interface made available by IDP. However, using this interface, we cannot prevent having to reground the theory every time an inference is performed.

Consequently, to minimise the number of times that we must reground the theories, we choose to introduce a separate theory $\mathcal{T}^{iso}$ (shown in **Listing** 11) for the canonicity constraint, which *generates* all isomorphic patterns by finding values for a unary predicate `pattern/1` representing a pattern isomorphic with `pattern_node/1`, such that two functions `f/1` and `g/1` can be found that are each others inverse and that satisfy the conditions of homomorphisms; i.e. they preserve edges and labels.

This way, after finding a pattern candidate and checking the positive and negative homomorphism restriction, we generate all isomorphic patterns and subsequently introduce additional clauses in the candidate generation process which prohibit these patterns from becoming candidates.

4.1.3 Visualising the Constraints

Retaking the example graph mining instance from Section 2 (See **Figure** 5), which consisted of 1 positive and 1 negative example, together with a template graph, and setting the graph mining parameters $N_+ = 1, N_- = 0$, we illustrate how the different constraints affect the possible patterns. We consider a subset of the candidate pattern space in **Figure** 6.



(a) Positive Example (b) Negative Example (c) Template Graph

Figure 5: (Repeat) A graph mining instance with ($N_+ = 1, N_- = 0$).

First, **Lines** 28 to 31 of $\mathcal{T}^+$ (**Listing** 8) ensure that patterns are vertex-induced subgraphs of the template. This consists of three subconditions: that (**Line** 29) the nodes of a candidate pattern are a subset of the nodes the template graph, that (**Line** 30) if the template features an edge between two selected nodes, the candidates must feature this edge as well, and that (**Line** 31) a node's label remains unchanged.

With the pattern candidates visualised in **Figure** 6, this constraint prunes candidate 6a, as it misses the diagonal edge in the hexagon present in template graph 5c, failing the second subcondition. It also prunes candidate 6d, because the

Listing 10: IDP specification handling the negative constraint of the graph mining problem

```
1  vocabulary V_neg{
2    type vertex isa nat
3    type label
4    type graphid
5
6    //Predicates describing the pattern graph
7    pattern_node(vertex)
8    pattern_edge(vertex, vertex)
9    pattern_label(vertex):label
10
11   //Predicates describing the negative example graphs
12   example_edge(graphid, vertex, vertex)
13   example_label(graphid, vertex):label
14   N_ : int
15
16   partial f(graphid, vertex):vertex //Represents the homomorphisms with the example graphs
17   homo_with(graphid) //True for graphs for which f represents a correct homomorphism
18 }
19
20 theory Negative:V_neg{
21   //Existence of a homomorphic f from the pattern to example graph with graphid gid.
22   ∀gid[graphid] : ∀x[vertex] : homo_with(gid) ∧ pattern_node(x) ⟺ ∃ y[vertex] : y=f(
         gid,x).
23   ∀gid[graphid] : ∀x[vertex] y[vertex] : homo_with(gid) ∧ pattern_node(x) ∧ pattern_node(y
         ) ∧ x≠y ⟹ f(gid, x) ≠ f(gid,y).
24   ∀gid[graphid] : ∀x[vertex] y[vertex] : homo_with(gid) ∧ pattern_node(x) ∧ pattern_node(y
         ) ∧ pattern_edge(x,y) ⟹ example_edge(gid, f(gid,x), f(gid,y)).
25   ∀gid[graphid] : ∀x[vertex] : homo_with(gid) ∧ pattern_node(x) ⟹
26      pattern_label(x) = example_label(gid, f(gid,x)).
27
28   //Can we find more than N_ homomorphisms
29   #{ gid [graphid] : homo_with(gid) } >= N_ .
30 }
```

rightmost node's label has changed into a diamond, and candidate 6e as it has too many diagonal edges, again failing the second subcondition.

Lines 34-38 prune any candidates that are not connected, such as 6c.

Lines 41-48 prune candidates that do not occur often enough in the positive examples (for this toy instance, at least once). The constraints of **Lines** 41-45 ensure that in case `homo_with(gid)` is true, the following holds: a mapping (1) must exist, (2) it must preserve inequality, (3) it must preserve the patterns edges, and (4) it must preserve labels, respectively. **Line** 48 specifies that enough homomorphisms must exist (at least one). As a result, candidates 6d and 6e are pruned, as no mapping exists for 6e to positive example 5a that preserves the diamond label and no mapping for 6d preserves all diagonal edges.

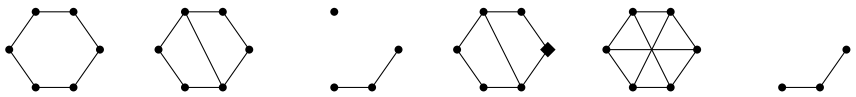
Lastly, the negative theory $\mathcal{T}^-$ (**Listing** 10) uses the same constraints as $\mathcal{T}^+$ **Lines** 41-48 to find patterns that are homomorph with too many negative examples. These constraints prune 6f, as a possible mapping was shown earlier, in **Section** 2, **Figure** 3.

4.1.4 Solving the graph mining problem using IDP

Now that we have modelled the graph mining problem in IDP, we also want to *use* this model to solve graph mining problems. As our model consists of multiple

Listing 11: IDP specification handling the canonicity constraint of the graph mining problem

```
vocabulary V{
  type vertex isa nat
  type label

  //Predicates describing the pattern graph
  pattern_node(vertex)
  pattern_edge(vertex, vertex)
  pattern_label(vertex):label

  //Predicates describing the template
  template_node(vertex)
  template_edge(vertex, vertex)
  template_label(vertex):label

  //Predicate describing an isomorphic pattern
  pattern(vertex)

  partial f(vertex):vertex //The homomorphism from pattern_node to pattern.
  partial g(vertex):vertex //The homomorphism from pattern to pattern_node.
}

theory iso:V{
  //f and g only have an image for vertices in pattern_node / pattern respectively.
  ∀x [vertex] pattern_node(x) <=> ∃ y: y = g(x).
  ∀x [vertex] pattern(x) <=> ∃ y: y = f(x).

  ∀x [vertex]: pattern(x) ⟹ pattern_node(f(x)).
  ∀x [vertex]: pattern_node(x) ⟹ pattern(g(x)).

  //f and g preserve edges
  ∀x [vertex] y [vertex]: pattern_node(x) ∧ pattern_node(y) ∧ template_edge(x, y) ⟹
       template_edge(g(x), g(y)).
  ∀x [vertex] y [vertex]: pattern(x) ∧ pattern(y) ∧ template_edge(x,y) ⟹ template_edge(
       f(x), f(y)).
  //f and g preserve labels
  ∀x [vertex]: pattern_node(x) ⟹ template_label(x) = template_label(g(x)).
  ∀x [vertex]: pattern(x) ⟹ template_label(x) = template_label(f(x)).
  //f and g are injective
  ∀x [vertex] y [vertex]: x < y ∧ pattern(x) ∧ pattern(y) ⟹ f(x) ≠ f(y).
  ∀x [vertex] y [vertex]: x < y ∧ pattern_node(x) ∧ pattern_node(y) ⟹ g(x) ≠ g(y).
  //f and g are each others inverse
  ∀x [vertex] : pattern(x) ⟹ g(f(x)) = x.
}
```



(a) (b) (c) (d) (e) (f)

Figure 6: A subset of the pattern space for **Figure** 5.

theories, we must use procedural code to tie together different inferences on the different theories, to eventually produce the correct answers. From the discussion above, we identify three main theories:

– $\mathcal{T}^+$ that generates a pattern, satisfying the positive homomorphic constraint,
– $\mathcal{T}^-$ that checks the negative homomorphic constraint, and

– $\mathcal{T}^{iso}$, that generates all isomorphic patterns.

The entire procedural loop can then be described as follows:

1. We first ask IDP for a model of $\mathcal{T}^{+}$.
2. We extract from this model the pattern candidate (i.e. the value of `pattern_node/1`, `pattern_edge/2` and `pattern_label/1`) and, using the satisfiability inference, check whether it satisfies $\mathcal{T}^{-}$; note that as $\mathcal{T}^{-}$ encodes the dual of the negative homomorphic constraint, failing the satisfiability check means the negative homomorphic constraint is satisfied and vice versa.
3. Regardless of whether the generated candidate satisfied $\mathcal{T}^{-}$, we let $\mathcal{T}^{iso}$ generate all its isomorphic patterns. These can be transformed to clauses that, when added to $\mathcal{T}^{+}$, prevent generation of isomorphic pattern candidates: such clauses state that it must not be true that a future generated pattern consists of exactly those nodes.
4. We repeat this process until the necessary number of patterns was found or the search space was exhausted.

4.2 ASP

The **ASP** language is closely related to IDP. An ASP encoding consists of a set of rules, which allow us to derive the head of a rule whenever its body is true. The head and body of a rule can contain variables, as long as every variable is *safe*, meaning it occurs positively in the body. In this case the head can be derived for any assignment to the variables that makes the body true.

One of the main differences between ASP and IDP is the choice of semantics: ASP looks for the answer set models, whereas IDP looks for well-founded models. Leveraging the minimality property of answer sets, ASP can prevent the invalid models of the example discussed in *Issue 3: local* $\forall$ *quantification over functions*, without creating two separate theories or writing procedural code. Instead, it relies on an encoding technique called the *saturation* technique [17], which we will discuss in Section 4.2.1 when we discuss how to encode the negative homomorphic property.

Another difference between ASP and IDP is that the former generally only allows *uninterpreted* functions, which can be viewed as constructors that bring structure in data. However, our specification of the graph mining problem features many *interpreted* functions, i.e. those representing a homomorphism between two graphs. Luckily, we can represent n-ary functions such as `pattern_label/1` using an $n+1$-ary predicate, and express functionality constraints explicitly.

4.2.1 Modeling the graph mining problem in ASP

Listing 12 shows the ASP specification of the graph mining problem. Note that we use the same naming scheme `pattern_node/1`, `pattern_edge/2` and `pattern_label/2`, and introduce the constants `np` and `nm` to correspond to the problem parameters N_+ and N_-, respectively.

We identify the same three issues for ASP as we had for IDP namely *representing graphs*, *local* $\exists$ *quantification over functions* and *local* $\forall$ *quantification over functions*. Due to the close relation between IDP and ASP, it is not surprising

that the first two issues are once again solved using the same global disjoint union technique as explained in the sections discussing the *representation of graphs* and the *local* $\exists$ *quantification over functions* in IDP.

However, as ASP does not allow functions, we represent n-ary functions such as `pattern_label/1` using an $n+1$-ary predicate, and express functionality constraints explicitly.

Generating pattern candidates As in **Listing** 8, we will first specify that the pattern is a vertex-induced subgraph of the template (**Lines** 5-7):

- First, we open up the `pattern_node/1` predicate using a choice rule. (**Line** 5)
- we state that every edge in the template between two pattern nodes implies a corresponding edge in the pattern.
- we specify that every pattern node preserves the unique label it had in the template.

As patterns must be connected, we include a set of rules and constraints expressing that every node of the pattern must be connected to every other node of the pattern (**Lines** 10-14). As with IDP, for the earlier introduced toy example from **Figure** 5, these constraints filter candidates 6a, 6d, and 6e because they do not represent vertex-induced subgraphs, and 6c as it violates connectedness.

Positive homomorphisms First, we will look at the positive homomorphic constraint, specifying the necessary number of homomorphisms with positive examples. First, we guess for every positive example graph whether a homomorphism exists using a choice rule (**Line** 17), and represent these graphs using `homo_with(G)`. For every positive graph `G` with a homomorphism, we create a mapping `f(G,X,V)` relating a graph id `G` and pattern node `X` with *exactly one* example node `V` (**Line** 18). We introduce constraints such that, for each positive example graph with a homomorphism, the mapping must be injective and must preserve edges as well as labels (**Lines** 20-22). We conclude the positive constraint by specifying that the number of mappings that correspond with a homomorphism should be higher or equal to our threshold N_+ (**Line** 24). Again, this constraint filters candidates 6d and 6e.

Negative homomorphisms We now look at how to encode the negative homomorphic constraint, which specifies that the number of homomorphisms with negative graphs does not exceed N_-. To encode this in the same model, we use the *saturation* technique as mentioned earlier.

First, we again specify that each pattern node `X` is mapped to at least one example node `V` (**Line** 27). Essentially, **Line** 27 serves to guess a possible mapping `f/1`: this is often called $\mathtt{P}_{guess}$ in discussions of the saturation technique. Note that we do not impose an upper limit on the number of example nodes `V` which match with `X`, as we did for the positive homomorphism (**Line** 18). This is essential for the saturation technique: while using this rule (**Line** 27) can derive a one-to-one mapping from pattern nodes to template nodes, the rule *does not prohibit* a larger mapping.

Our next rule will allow the derivation of such a larger mapping: we introduce a predicate called `saturated/1` which will be true for those negative graphs `G` for which no homomorphism exists. When no homomorphism for a graph `G` exists, we specify that the matching function `f/3` should match every pattern node `X`

with *every* example node `V` (**Line** 28). This way, `f/3` becomes so large that it is impossible that `f/3` belongs to the minimal answer set unless there does *not exist* a homomorphism from the pattern to this (negative) example graph. This is commonly referred to as $\texttt{P}_{sat}$: it describes how the guessed predicate will 'blow up' when no correct guess exists. Due to this 'blowup', the minimality property of answer sets means the solver will look for an f that represents a homomorphism for as many of the negative example graphs as possible.

We continue the encoding of the negative constraint by specifying the possible reasons a mapping does not represent a homomorphism (**Lines** 32-36). Either the mapping is not injective, or it does not preserve edges or labels: These lines are commonly referred to as $\texttt{P}_{check}$.

Lastly, we specify that the pattern is allowed to be homomorphic with at most N_- negative examples (**Lines** 38-39). In our toy example, this prunes candidate 6f.

Canonicity The same saturation technique can be applied to the isomorphism restriction; making it possible to model the entire graph mining problem in a single model. We also introduce the notion of a *lexicographical ordering* of graphs, based on the natural order of the nodes: we presume the `#max` and `#min` aggregates on nodes are defined, and a successor predicate `succ/2` is available that holds for any two nodes `a`, `b` s.t. `b` immediately follows `a` in this natural order. A graph $\mathcal{G}$ is lexicographically smaller than a graph $\mathcal{G}$' if the smallest node not shared between $\mathcal{G}$ and $\mathcal{G}$' is a node of $\mathcal{G}$. We can now say that a pattern $\mathcal{P}$ is canonical if it is the *lexicographically smallest* graph among all its isomorphic graphs. The main idea is that for every choice for `pattern_node/1`, we want ASP to find an isomorphism with another subset of template nodes that is lexicographically *smaller* (i.e. a counterexample for the statement that `pattern_node/1` is canonical). If ASP cannot find such an isomorphism, we saturate the answer set. Thus, saturated answer sets correspond to choices for `pattern_node/1` s.t. *no lexicographically smaller* isomorphic graph exists, which are exactly the canonical patterns.

Looking at our model (**Lines** 42-45), to enforce canonicity we again guess a relation, in this case `iso/2`. Semantically, `iso/2` is the predicate representation of a function between nodes of the pattern and nodes of a hypothetical, different and *canonical* form of that same pattern. Such a function does not exist if the pattern itself is, in fact, canonical: In that case `iso/2` will be saturated.

Because we only want answer sets that correspond to canonical patterns, after including the saturation rule $\texttt{P}_{sat}$ (28) for `iso/2`, we add a constraint saying that all answer sets must be saturated. Furthermore, we create two helper predicates:

- `isoNode/1` which is true for those nodes in the image of `iso/2`, and
- `compl/1` which is true for those nodes *not* in the image of `iso/2`.

Because in some situations we will saturate `iso/2`, we cannot define `compl/1` as `compl(X):-not isoNode(X)`, as this would make the resulting saturated answer set unstable (The rules deriving `compl` would disappear from the Gelfond-Lifschitz reduct).

Therefore, we define by induction a helper predicate `codCT(X,Y)` which is true iff `X` is *not* the *unique image* of `Y` or any smaller node. Note that we do this *without negation of any saturated symbols* (`iso/2`, `isoNode/1`).

1. This trivially holds for the smallest node `F` if it is not a pattern node, as then `F` is not in the domain of `iso/2` (**Line** 51).

2. This holds for the smallest node `F` if `iso/2` maps `F` to a node differing from `X` (**Line** 52).
3. This holds by induction for the tuple (`X`,`Y`) if it holds for the node preceding `Y` and either `Y` is not a pattern node (**Line** 53) or `Y` is mapped to a node differing from `X` (**Line** 54).

We can now define `compl` as the nodes `X` s.t. `codCT` holds for the largest node, i.e. it is not the image of the highest node or any below it.

Next, we must specify when we saturate ($\texttt{P}_{check}$). This occurs whenever `iso/2` does not represent an isomorphism (because it does not preserve edges, labels, or is not injective), or when `iso/2` represents an isomorphism with a graph that is *not lexicographically smaller*. To encode this last condition, we define by induction (**Lines** 71-75) a predicate `identity_below/1` which indicates that up to, but not including a certain node, `pattern_node/1` and `isoNode/1` are identical. Now, whenever there exists a node such that `pattern_node/1` and `isoNode/1` are identical up to that node, and that node itself is part of `pattern_node/1` but not of `isoNode/1` (expressed by the complement `compl/1`), we must saturate. Likewise, we must saturate whenever `pattern_node/1` and `isoNode/1` are identical, which we handle in **Lines** 66 and 74-75.

Saturation technique Saturation as a technique is a powerful way of including constraints that are expressed using formulas with second-order universal quantification (i.e. corresponding to Σ_2^p decision problems). Examples of such constraints are the introduction of negative example graphs or the canonicity of patterns as discussed above, but also other constraints that impose a preference order on patterns, e.g. maximality (prevents patterns that are subsets of other patterns) or coverage (orders patterns by comparing the set of matched positive patterns using subset ordering). While the *saturation technique* successfully prevents the need of a procedural loop for such constraints, it is clear that this technique is not derived from a natural KR translation of the Graph Mining definition.

4.2.2 Solving the graph mining problem using ASP

As ASP is able to represent the graph mining problem using a single model, solving the graph mining problem using ASP is pretty straight-forward. However, it is important to note that by default, finding a different homomorphism between a canonical pattern and an example would lead to a different answer set. However, as the pattern itself is the same, this is in fact not a new solution. As such, we must limit the answer sets to those that differ for their choice of pattern nodes. Using a solver such as clingo, this is possible by enabling *answer set projection* [20], which limits the different answer sets to those that differ on a specific set of facts. By specifically projecting to the facts representing the pattern nodes, we obtain the desired behavior.

4.3 Comparative Summary

In Section 3.3, we have identified a set of desirable properties that a good KR specification should satisfy. Table 1 summarizes how IDP and ASP score with respect to these properties.

Listing 12: ASP using the saturation technique

```
#const nm = N_-.
#const np = N_+.

% Patterns are vertex-induced subgraphs of the template
0 { pattern_node(X) } 1 :- template_node(X).
pattern_edge(X, Y) :- pattern_node(X), pattern_node(Y), template_edge(X, Y).
pattern_label(X, V) :- pattern_node(X), template_label(X, V).

% Patterns are connected
connected(X) :- #min{Y : pattern_node(Y)}=X.
connected(Y) :- connected(X), pattern_edge(X, Y), X != Y.
connected(Y) :- connected(X), pattern_edge(Y, X), X != Y.

:- pattern_node(X), not connected(X).

% Positive homomorphic constraint:
homo_with(G) | not homo_with(G) :- positive(G).
1 { f(G, X, V) : example_node(G, V) } 1 :- homo_with(G), pattern_node(X).

:- homo_with(G), pattern_node(X), pattern_node(Y), X != Y, example_node(G, V), f(G, X, V),
     f(G, Y, V).
:- homo_with(G), f(G, X, V1), f(G, Y, V2), template_edge(X, Y), not example_edge(G, V1, V2)
     , pattern_node(X), pattern_node(Y).
:- homo_with(G), pattern_node(X), f(G, X, V), pattern_label(X, L), example_label(G, V, L2),
      L != L2.

:- #count{G:homo_with(G)} < np.

% Negative homomorphic constraint:
f(G, X, V) : example_node(G, V) :- pattern_node(X), negative(G). % P_guess
f(G, X, V) :- saturated(G), pattern_node(X), example_node(G, V). % P_sat

% The following lines describe the reasons for a graph to be saturated (P_check):
% We cannot map two different pattern nodes to the same example node.
saturated(G) :- negative(G), f(G, X, V), f(G, Y, V), X != Y, pattern_node(X), pattern_node(
     Y).
% The mapping must preserve edges.
saturated(G) :- negative(G), template_edge(X, Y), f(G, X, V1), f(G, Y, V2), not
     example_edge(G, V1, V2), pattern_node(X), pattern_node(Y).
% The mapping must preserve labels.
saturated(G) :- negative(G), template_node(X), f(G, X, V), template_label(X, L),
     example_label(G, V, L2), L != L2.

neg_homo_with(G) :- not saturated(G), negative(G).
:- #count{G:neg_homo_with(G)} > nm.

% Canonicity constraint:
iso(X, V) : template_node(V) :- pattern_node(X). % P_guess
iso(X, V) :- pattern_node(X), template_node(V), sat. %P_sat
:- not sat.


isoNode(V) :- iso(X, V).
compl(X) :- template_node(X), codCT(X, M), M=#max{Z:template_node(Z)}.

%codCT(X,Y): X is not the image of iso for Y or any node below it
codCT(X, F) :- template_node(X), not pattern_node(F), F=#min{Z:template_node(Z)}.
codCT(X, F) :- template_node(X), iso(F, Y), Y!=X, F=#min{Z:template_node(Z)}.
codCT(X, B) :- template_node(B), succ(A, B), codCT(X, A), not pattern_node(B).
codCT(X, B) :- template_node(B), succ(A, B), codCT(X, A), iso(B, Y), Y!= X.

% iso must preserve edges
sat :- iso(X, W), iso(Y, Z), template_edge(X, Y), not template_edge(W, Z).
sat :- iso(X, W), iso(Y, Z), not template_edge(X, Y), template_edge(W, Z).
% iso must preserve labels
sat :- iso(X, Y), template_label(X, L1), template_label(Y, L2), L1!=L2.
% iso must be injective
sat :- iso(X, Y), iso(X, Z), Y!=Z.
% The inverse of iso must be injective
sat :- iso(X1, Y), iso(X2, Y), X2!=X1.
sat :- identity_below(X), pattern_node(X), compl(X).
sat :- identity_below(sup).

% identity_below(X) iff every node below (not including) X is either
% - in the iso candidate and the pattern, or
% - not in the iso candidate nor in the pattern.
identity_below(M) :- #min{X:template_node(X)}=M.
identity_below(X) :- template_node(X), succ(Y, X), identity_below(Y), pattern_node(Y),
     isoNode(Y).
identity_below(X) :- template_node(X), succ(Y, X), identity_below(Y), not pattern_node(Y),
     compl(Y).
identity_below(sup) :- identity_below(M), #max{X:template_node(X)}=M, pattern_node(M),
     isoNode(M).
identity_below(sup) :- identity_below(M), #max{X:template_node(X)}=M, not pattern_node(M),
     compl(M).

#show pattern_node/1.
```

Property	IDP	ASP
1. Graph as a single object	No: Global disjoint union technique	No: Global disjoint union technique
2. Independence of homomorphisms	No: Global disj. union & partial function	No: Global disj. union & partial function
3. Similarity of $\geq$ and $\leq$ constraint	Partial: Similar but theory splitting required	No: Requires saturation technique
4. Multiple patterns (isomorphism)	No: theory splitting required	Yes: Using saturation technique
5. Connectedness	Yes: Using inductive definitions	Yes: Using ASP rules
6. Multiple inferences	Yes: Model checking, expansion, minimization	Yes: Model checking, expansion, minimization
7. Single solver call	No: Two calls, one model per pattern	Partial: One answer set per pattern

Table 1: Summary of the desirable properties in IDP and ASP

4.4 Performance experiments

In this section, we will investigate the performance of both the IDP as well as the ASP model. These experiments were performed on a Ubuntu 16.04 LTS system with an Intel i7-4770 CPU @ 3.40GHz, with 8GB RAM, on which IDP (version 3.7.0) and Clingo (version 5.2.2) were installed. Every experiment was run with an 8 GB memory limit and a 20 hour time limit.

We created graph mining problem instances from two well-known machine learning datasets [38]: mutagenesis and yoshida. These datasets consist of a set of labeled graphs representing labeled molecules;

- in *yoshida* 265 molecules are ranked according to their bioavailability, and
- in *mutagenesis*, 230 molecules are trialed for their mutagenicity on Salmonella.

Most discussions of state-of-the-art specialized algorithms do not use any negative examples. However, the mutagenesis dataset allows us to characterize 92 molecules as 'negative', specifically those who inhibit the mutability of Salmonella (i.e. a mutagenicity of ≤ 1). As our specifications easily accommodate a dataset with negative examples, we create graph mining instances from the *yoshida* and *mutagenesis* datasets by randomly selecting one positively labeled graph from the dataset to serve as the *template* required for our solution. Next, we chose values for N_+ and N_-: For *yoshida*, which only has positive examples, we chose an N_+ value of 26 (10% of the dataset). For *mutagenesis* we chose an N_+ of 90 (66% of the positive examples) and 30 (33% of the negative examples). We used both the ASP and the IDP solution to compute 120 *canonical* (i.e. non-isomorphic) patterns.

4.4.1 Results

Yoshida: The results for the *yoshida* dataset are visualised in **Figure** 7 (Note the need for different scales). Since IDP as a byproduct, can list all isomorphic solutions for each canonical pattern, we could and have cross-validated the results of both solvers.

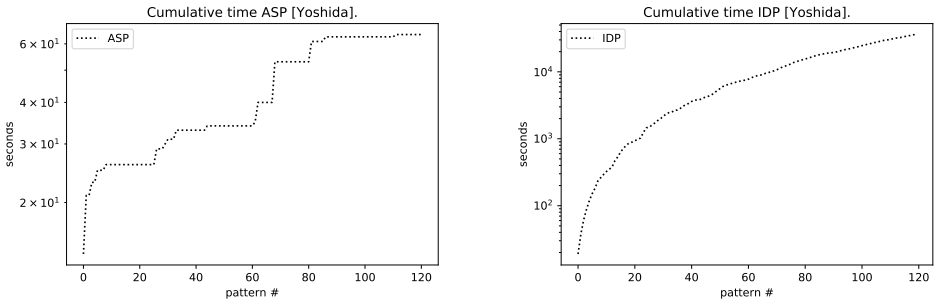


Figure 7: Cumulative run times for the Yoshida dataset.

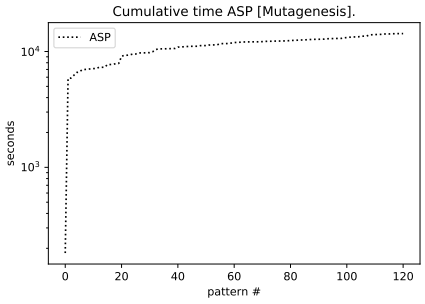


Figure 8: Cumulative ASP run time for the mutagenesis dataset.

It is clear that the ASP solution, which utilizes a single solver process, outperforms the IDP system, which must repeatedly ground the same problem due to the interaction of generating pattern candidates satisfying the positive homomorphic constraint and checking canonicity of pattern candidates. As the *yoshida* dataset only contains positive examples, we can also illustrate the performance of state-of-the-art specialized algorithms. However, specialized algorithms such as gSpan generally mine *all* patterns, and do this without requiring a specific template graph: it can use any graph in the database as a template. As such, we have mined all patterns for our *yoshida* instance; this takes about 1.39 seconds for gSpan [36]. Comparatively, when Clingo (ASP) and IDP are asked to mine all patterns, ASP takes about 282.5 seconds, whereas IDP does not finish within the time limit of 20 hours.

Mutagenesis: When mining the mutagenesis dataset for patterns with $N_+ = 90$ and $N_- = 30$, IDP does not find any patterns before the time limit of 20 hours has passed. The results for ASP are shown in **Figure** 8, which mines 120 patterns in a little under four hours.

4.4.2 Discussion

It is clear from these experiments that state-of-the-art specialised algorithms outperform our declarative solutions by several orders of magnitude. However, our declarative solutions can easily be extended to support negative examples, such as those in the mutagenesis dataset, whereas specialised algorithms require an extensive overhaul. This is an example of the level of Elaboration Tolerance [33] that declarative languages exhibit. This gives declarative approaches a great benefit in use cases where performance is not the primordial factor, for example while prototyping or when requirements frequently change.

These experiments also show a clear divide between the ASP language and the IDP language. While ASP solvers can natively support constraints with universal second-order quantification (i.e. Σ_2^p), IDP must resort to multiple solver instances tied together using procedural code. As a result, IDP must ground the problem repeatedly and, as communication between the solver instances is limited, is not able to learn valuable information about when a candidate will or will not pass the checks performed by different solver instances. On the other hand, constraints featuring second-order universal quantification are only supported by ASP using the advanced *saturation* technique, reducing the 'naturality' of the encoding. In the next section, we reconsider the higher-order model of Section 3, and look how declarative systems, in particular IDP, can introduce support for such higher-order models.

5 Solver Techniques

In the previous sections, we have established that (1) an intuitive encoding of the graph mining problem exists using higher-order logic, and that (2) encoding techniques are required to express the problem in a specification language based on first-order logic. Regrettably, these encoding techniques decrease the graph mining model's intuitiveness and can be a significant hurdle for the modeller. This section investigates the encoding techniques encountered in Section 4 so as to identify ways to generalise these techniques and integrate them in the solver, effectively shifting the burden of these techniques from modeller to solver by supporting higher-order specifications. Specifically, integration techniques are suggested for the state-of-the-art IDP-system, which currently uses a typical ground-and-solve technique [26]. In a ground-and-solve system, two distinct phases can be identified: in the first phase, all quantifications are instantiated such that the encoding does not contain any variables, while in the second a SAT-solver finds a model for the resulting ground instance. In general, techniques that support higher-order logic will interleave these two phases in various degrees.

One concern typically raised in the context of solver for higher-order logic is that rising language expressivity will go hand in hand with decreasing performance. From a theoretical point of view, this is clearly a valid concern, however our hypothesis is that, in practice, expressing real-world problems using higher-order logic does not necessarily include a performance loss with respect to their previous first-order encodings. Furthermore, the additional structure expressed in higher-order encodings might even allow for a performance gain. Specifically, an important aspect through which we think a higher-order encoding, when supported with

Listing 13: Excerpt of the HO specification of graph mining

```
1 homomorphism((N1, E1, L1), (N2, E2, L2)) ←
2    (∃SO F [N1:N2]: (∀ x, y [N1]: x ≠ y ⟹ F(x) ≠ F(y)) ∧
3     (∀ x [N1] y [N1]: E1(x, y) ⟹ E2(F(x), F(y))) ∧
4     (∀ x [N1]: L1(x) = L2(F(x)))).
5  ...
6 (#{ Pos : positive(Pos) ∧ homomorphism((N,E,L), Pos) } ≥ N+)
7  ...
```

the right solver techniques, can in increase performance is *independence analysis*, i.e. the discovery of independent subproblems. Some support for the claim that better independence analysis will lead to better performance can be found on the propositional level, in recent work [31,37] from the Quantified Boolean Formulas (QBF) research community. This work shows the benefit of estimating or learning the dependencies between quantifications of propositional variables. Writing down knowledge in a more expressive language such as higher-order logic leads to the availability of additional structure and latent constraints within the knowledge specification. Often, using the additional structure available, we already express many interesting (in)dependencies. Consider the following two examples: First, when we existentially quantify over a (set of) higher-order object(s) satisfying a constraint, we can look for this (set of) object(s) independently from the larger problem. Second, in the case of a universal quantification of a higher-order object, we can check the relevant constraints for every possible instantiation of this higher-order quantification separately.

Returning to the higher-order modelling of graph mining, for example, it is clear that the question of whether two specific graphs match is a subproblem which can be solved *independently* of other matchings. This independence is signalled by the quantification over graphs `(N,E,L)` (**Line** 6 of **Listing** 13), even though it is hidden in the aggregate expression counting `homomorphisms`. Further evidence of the independence can be found in the definition of `homomorphism/2`, as it uses only two types of symbols: locally quantified symbols and predicate arguments. A smart solver should analyse the higher-order specification to detect and exploit these (in)dependencies, and, when discussing solver techniques, we will pay specific attention to how this can be achieved.

5.1 Nested Solvers

As was pointed out in **Section** 4, two of the main issues with higher-order encodings for systems such as IDP are (1) the ∀ quantification over higher-order objects such as functions and predicates, and (2) the occurrence of local quantification, both existential as well as universal. The technique of *nested solvers* addresses both these issues: when presented with a universal quantification or any local quantification over a function or predicate, the solver can spawn another, secondary instance of itself which is identical except for the specification being solved. Indeed, this second instance, also called an *oracle* is initialised with that part of the original specification where the quantified symbol is in scope, possibly transformed to an existential quantification.

At the propositional level, Bogaerts et al. [6] recently explored the idea of solving QBF instances using nested SAT solvers supporting CDCL, with favourable results. Their underlying idea is to use the identity $\forall \overline{x} : \phi_1 \Leftrightarrow \neg\exists \overline{x} : \neg\phi_1$ to transform arbitrary formulas to the form $\exists \overline{x} : (phi \wedge \neg\exists \overline{y} : \psi)$. They show how the top solver can perform standard SAT-solver propagation on ϕ, and how the second solver (the oracle) can check whether there exists an assignment $\overline{y}$ satisfying ψ. It is important to note that as the transformation above introduces negations in front of the quantification $\exists \overline{y}$, the oracle call is performed within a negative context: models $\mathcal{M}$ of $\mathcal{O}_\psi$ are transformed into conflicts and learned clauses $\mathcal{C}$ for the top solver. This approach effectively construes a QBF solver, and as predicted, this technique can exploit (in)depencies. While QCDCL [32] traditionally limits the order in which variables can be decided to the order in which they are specified in the quantifier prefix, this restriction is not necessary when working with nested solvers, opening up research tracks on the effects of eager versus lazy calling of the nested solver, or the effect of splitting up variables within a single quantifier prefix level if they are independent.

This technique, taken from the propositional level, can be recreated at the predicate level by rewriting and splitting theories and handing them off to separate, nested solvers functioning as oracles. As on the propositional level, we are faced with the same trade off between eager and lazy calling of nested solvers. Another research challenge is the transformation of models $\mathcal{M}$ to learned clauses $\mathcal{C}$: as multiple models for the oracle's theory can exist, one can follow different approaches for transforming one or more models of the theory into a learned clause.

On the level of predicate level, the nested solver technique is closely related to "modulo-theory" frameworks such as *SAT-modulo-theory* or *ASP-modulo-theory* [19]. These frameworks offer a way of injecting *procedural code* for complex problems using *global constraints*. One example is the injection of prefix-projection [27] in recent declarative approaches to sequential pattern mining [4]. By contrast, the nested solvers technique does not inject *procedural code*, instead it injects another solver instance (an oracle). Thus, in the nested solver approach, even the injected knowledge is specified declaratively, and, in a full implementation, the split points between the levels of solvers are introduced without involvement of the user.

Nested solvers in Graph Mining: Turning back to graph mining, and looking only at the positive homomorphism requirement and the non isomorphism requirement, we identify three different strategies for introducing oracle calls to the graph mining problem. These three strategies correspond to different options for splitting the graph mining specification, and we will call these strategies the **monolithic**, the **semi decomposed** and the **fully decomposed** strategies. All three are visualised in **Figure** 9, where every (sub)solver or oracle call is represented as an IDP block, and the different positive example graphs are labeled as Ex1, Ex2 and Ex3. In each case, the main or top-level solver is the leftmost IDP block, responsible for generating candidates.

– The **monolithic** strategy is the default strategy as explained in Section 4.2.2. As such, this strategy could only gain from a tighter integration between solver instances, which would allow reuse of grounding and efficient communication of learned clauses.

This strategy splits off the *generation* of a pattern candidate and *checking the positive homomorphism constraint*, from the *non isomorphism check*. It thus consists of only two solver instances: one that takes the template and all example graphs, subsequently producing a pattern candidate satisfying the positive constraint; and one that, using the other patterns, checks whether a pattern candidate is isomorphic to an already discovered pattern. This second solver then reports back to the first, and the necessary clauses are generated to prevent generating the same pattern candidate again.
- The **semi decomposed** strategy further splits off the *generation* of a pattern candidate from the *test* phase where the solver checks whether the pattern candidate is homomorphic with sufficiently many positive example graphs. Based on the outcome of this check, the solver either reports back to the first solver which can register that the pattern candidate was not a valid pattern and generates the necessary clauses to prevent regeneration, or it passes the pattern candidate on to a third solver performing the non isomorphism check as in the case of the **monolithic** strategy.
- the **fully decomposed** strategy exploits the independence between the different positive example graphs; it introduces a separate oracle call for each example graph, reporting the results of the checks to an aggregation unit. This aggregation unit then reports back to either the first solver or to another solver performing the non isomorphism check as in the case of the **monolithic** strategy.

Note that all three strategies split the theory on points where quantifications are used within the theory; in fact, they each correspond to a *splitting strategy*:
- The **monolithic** approach splits *only* when encountering a *second-order universal* quantification, which would put the problem outside of the expressive power of a conventional SAT solver.
- The **fully decomposed** approach splits the theory when encountering *any* second-order quantification; this includes the existential second order quantification present in the definition of `homomorphism/2` in **Listing** 13.
- The **semi decomposed** approach splits the theory on the outermost quantification for any rule containing *any* second-order quantification.

As it is possible that after splitting, the theory being split off still contains formulas with second-order quantifications, the splitting rules must be performed recursively.

Experiments: To get an idea of the performance of the nested solvers technique, and whether it might lead to some performance gains, we mimicked the implementation of a nested solver, without implementing a fully functional nested solver. By introducing a 'pipeline' of separate IDP3 call instances, our implementation replicates the different oracle calls in the nested solver approach, while being specifically tailored to graph mining. It uses an imperative language to manage the different calls and modify the models $\mathcal{M}_\psi$ of oracle calls to new clauses $\mathcal{C}_\psi$ added to subsequent calls.

For each strategy proposed above, we have introduced a corresponding 'pipeline' in the experiments. The experiment is set up such that every pipeline mines a certain amount of patterns from a dataset. Like most imperative solutions, they do this in a fixed order: first patterns of length n are mined, starting with

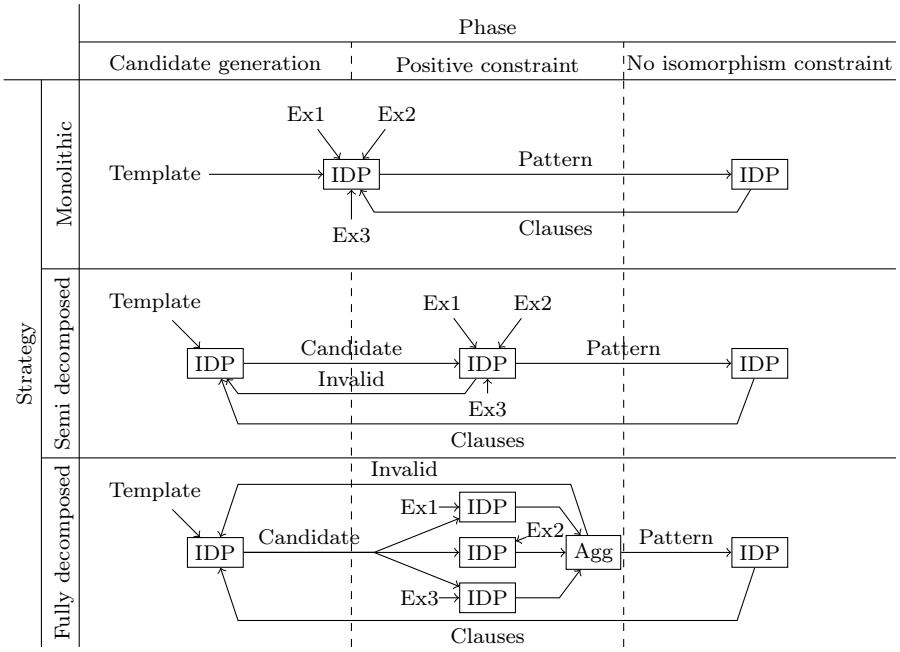


Figure 9: The three different strategies proposed for subsolvers.

$n = 2$, raising the length of the mined patterns to $n + 1$ whenever all patterns of length n are mined. This fixed order allows exploitation of an *anti-monotonicity* property often used by imperative solutions: *Whenever a pattern candidate fails the positive homomorphism check, every extension of this pattern candidate will also fail the positive homomorphism constraint.*

This property can easily be encoded as additional knowledge in a higher-order specification of the graph mining problem, as it defines a predicate `isPattern/1` representing whether or not a graph $\mathcal{G}$ is a pattern. When we look at the proposed strategies, both the **semi** and **fully decomposed** pipeline capture the necessary information for exploiting the *anti-monotonicity* property: they signal both the valid as well as the invalid pattern candidates through the imperative interface managing the different oracle calls. However, the **monolithic** pipeline does not; invalid pattern candidates are discarded internally in the solver instance handling candidate generation and the positive constraint. As a result, the invalid pattern candidates of length n cannot be used by the **monolithic** pipeline to additionally filter the candidate generation when it starts searching for patterns of length $n+1$. As a result, the **monolithic** pipeline does not exploit the *anti-monotonicity* property, but also does not impose a search direction, which can become an advantage for certain datasets.

Dataset generation & specifications: To test the performance of all three pipelines, we have reused the *yoshida* dataset from Section 4.4, and have modified the *mutagenesis* dataset by labeling all molecules as *positive*. This modification is motivated by two key insights:

1. Specialized algorithms do not feature negative examples,
2. When splitting the model into multiple theories solved by separate oracles, the theory for negative examples is the same as that for the positive examples. The only difference is how the satisfiability results are handled: for negative examples, an UNSAT is handled as a SAT for the positive examples and vice versa.

Lastly, we have also created a graph mining problem from the well known *bloodbarr* dataset [30], where 413 molecules are ranked according to the degree to which the molecule can cross the blood-barrier stream.

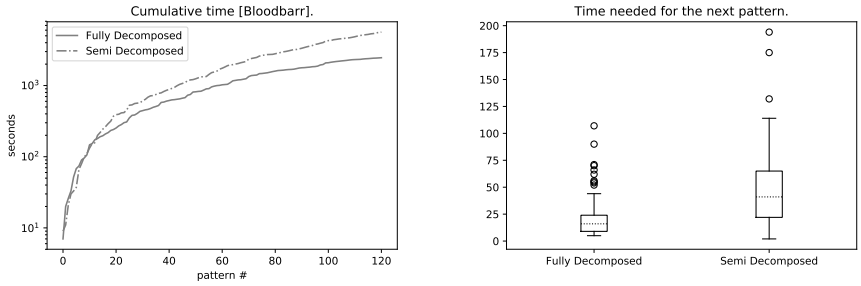
We have reused the approach described in Section 4.4, and ran experiments using the same machine and time/memory limits.

Results: **Figure** 10 shows the resulting cumulative runtimes for each of the pipelines on a log-scale y-axis, and a boxplot of the time spent mining each pattern for the **fully decomposed** and the **semi decomposed** pipelines. Our boxes cover the data from the first quartile (Q1) to the third quartile (Q3), while the whiskers extend to the last datum less than $Q3$ plus 1.5 times the interquartile range (IQR). All other data points are considered outliers, and are plotted as individual dots. A horizontal dotted line indicates the median. For the Bloodbarr datasets, no results for the **monolithic** pipeline could be given, as it exceeded the memory limit of 8GB.

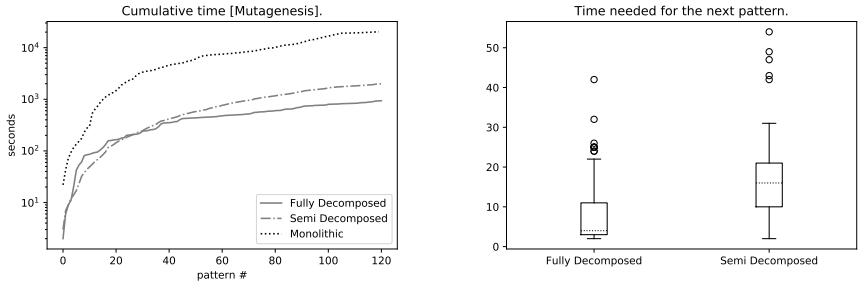
Focussing on the difference between the **semi** and **fully decomposed** pipelines, all three datasets (**Figure** 10b) show a similar factor of two difference in favor of the **fully decomposed** pipeline. The difference between the pipelines that use oracles for the positive constraint on the one hand (the **semi** and **fully decomposed** pipelines), and the **monolithic** pipeline on the other hand, suggests that a large benefit can be achieved from using a separate oracle for the checking phase.

Furthermore, the difference between the **semi** and **fully decomposed** pipelines shows that the benefit of introducing oracles, at least in graph mining, increases when we further introduce an oracle call for each independent graph. Recall that the possibilities for decomposition in the graph mining case are found by syntactical analysis; they correspond to second-order quantifications, and their position in the hierarchy relative to each other and other, first-order quantifications. In fact, this is why we advocate the use of local quantifications, as opposed to having to quantify all second-order symbols in the vocabulary. This syntactical argument suggests that finding good decompositions for other problems based on the presence of second-order quantifications is feasible.

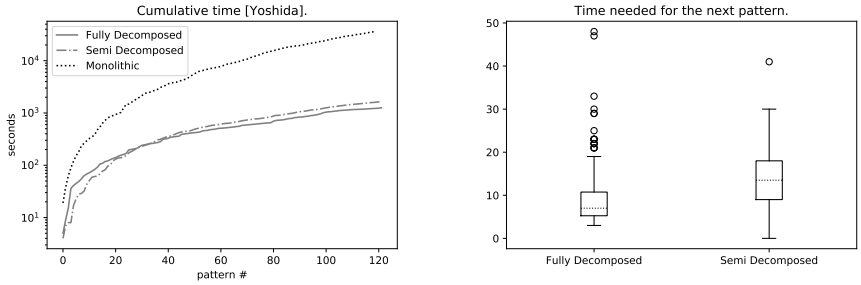
In fact, the **semi** and **fully decomposed** strategies can mine 120 patterns from the yoshida dataset in 1633 and 1255 seconds respectively. Likewise, for the mutagenesis dataset, these strategies mine the 120 requested patterns in 1996 and 939 seconds respectively. While this is still an order of magnitude larger than ASP, we note that in these experiments we focussed on how many oracles should be introduced and where, and as a result, IDP must still repeatedly ground each theory.



(a) IDP results for the bloodbarr dataset, with N_+=41.



(b) IDP results for the mutagenesis dataset, with N_+=23.



(c) IDP results for the yoshida dataset, with N_+=26.

Figure 10: Cumulative runtimes and time spent per pattern by IDP for the three datasets with only positive examples.

5.2 Lazy Grounding

The third issue with higher-order encodings, as pointed out in **Section** 4, concerns the data representation of sets of higher-order objects, such as graphs in the *graph mining* problem. While the *disjoint union* technique proposed in **Section** 4 can be used, even automating the rewrite so it is no hinderance for modellers, one problem is that it tends to produce very large groundings. Furthermore, the earlier introduced nested solvers technique results in a system that has to ground not just the main theory, but also has to ground theories for every nested solver, while

retaining as many grounding optimization techniques as possible. This problem could be mitigated by using the *lazy grounding* technique.

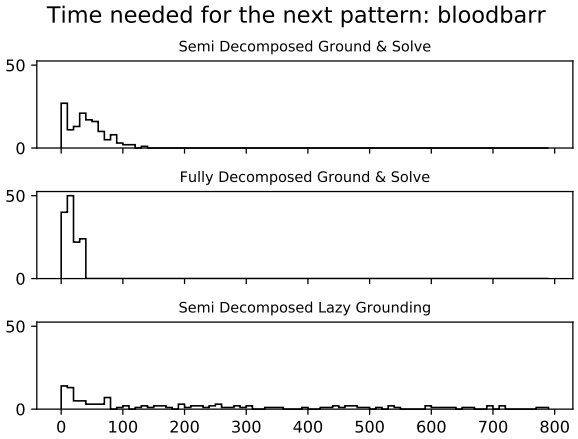
Lazy grounding is a technique where theories are only grounded partially: only those parts of the theory that are relevant to finding a satisfying assignment are grounded. The framework of de Cat et al. [10], for example, uses the concept of justifications to denote a way of generating a complete assignment for non-grounded parts of the theory. It then suffices to ground parts only if it is not possible to construct a justification for them anymore. An experimental algorithm for model expansion with lazy grounding based on this framework has been implemented within IDP.

Experiments: To get an indication of the impact of lazy grounding, we repeated the experiments explained above while enabling lazy grounding for every IDP call covering the 'positive constraint checking' phase. Note that in this case, the disjoint union technique has already been applied manually. This would allow the solver to defer the grounding of example graphs until they are necessary to satisfy the positive constraint. In the 'Ground and solve' setup, the **fully decomposed** pipeline is able to prevent grounding and solving some example graphs by eagerly evaluating the aggregation and stopping as soon as the threshold is reached, giving it a clear advantage over the **semi decomposed** pipeline, which has to ground the entire problem first. When using lazy grounding, we expect the **semi decomposed** pipeline to behave more like the **fully decomposed** pipeline, as it should be able to bypass any grounding for unnecessary graphs.

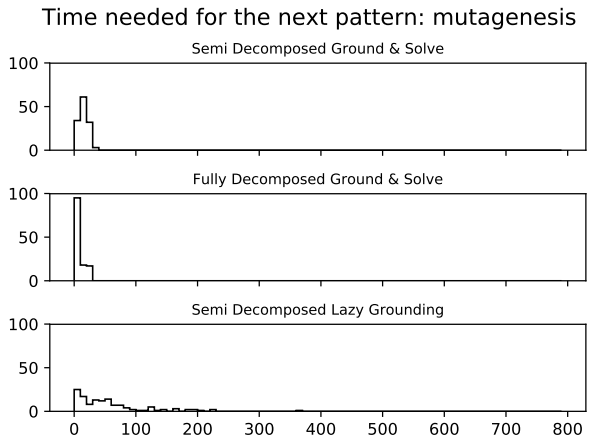
Results: **Figure** 11 shows for each dataset a histogram of the time needed to mine the next pattern for the **semi decomposed** and **fully decomposed** pipelines with ground and search, and the **semi decomposed** pipeline with lazy grounding. These figures show that the lazy grounding option actually causes a slowdown for the **semi decomposed** pipeline, quite frequently needing significantly more time to check a pattern, as evidenced by the long tail of the **semi decomposed** pipeline with lazy grounding. **Figure** 12 shows, for the mutagenesis dataset, the size of grounding as the number of literals (12a) and the memory usage of the **semi decomposed** pipeline in kilobytes with and without lazy grounding (12b). While **Figure** 12a shows that lazy grounding produces smaller groundings, **Figure** 12b shows that the effective memory usage using lazy grounding, while in general smaller, sometimes exceeds that of the default ground & solve option.

One possible cause for the apparent slowdown caused by the lazy grounding option is the setup cost of lazy grounding, which can be high: When using lazy grounding, additional data structures are required. Another possible factor is the 'penalty' incurred in the experimental implementation when lazy grounding has to ground an additional graph, w.r.t eagerly grounding all patterns at the same time. Further evidence for this factor can be found by noting that the slowdown with lazy grounding is less dramatic for the mutagenesis and yoshida datasets. From **Figure** 13, which shows a boxplot of the number of example graphs that had to be inspected before accepting or refuting a pattern candidate for each dataset[4], we can conclude that these datasets are in some sense 'easier', as the **fully decomposed** pipeline on average has to inspect fewer graphs per pattern candidate for these datasets than for the bloodbarr dataset.

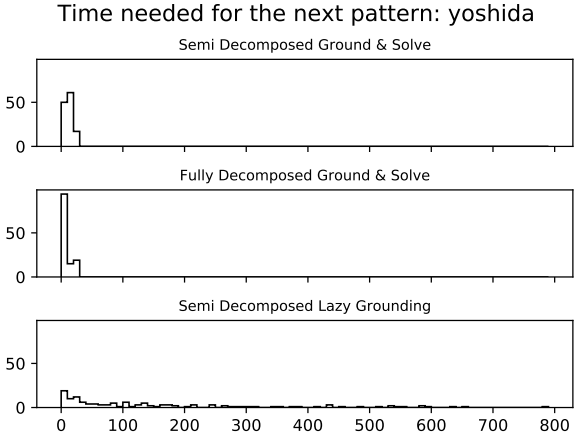
[4] Numbers taken from runs of the **fully decomposed** pipeline.



(a) Lazy grounding effects for IDP on the bloodbarr dataset, with N_+=41.

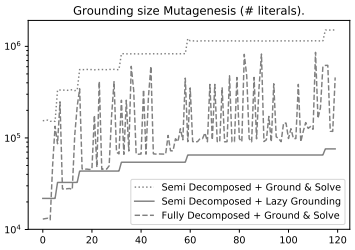


(b) Lazy grounding effects for IDP on the mutagenesis dataset, made strictly positive, with N_+=23.

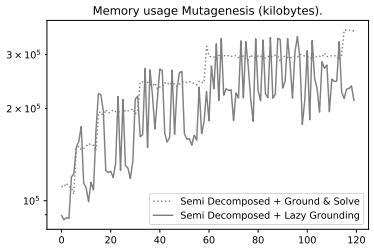


(c) Lazy grounding effects for IDP on the yoshida dataset, with N_+=26.

Figure 11: Histograms of time needed to mine the next pattern by IDP. Only strictly positive datasets were used.



(a) Grounding size for pattern check in Mutagenesis dataset, as number of literals.



(b) Memory usage for pattern check in Mutagenesis dataset, in kilobytes.

Figure 12: Grounding size (#lits) and memory usage (kilobytes) of Ground & Solve and Lazy Grounding approaches in Mutagenesis dataset.

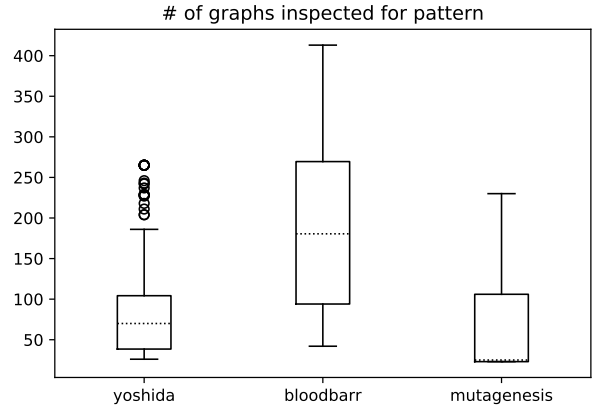


Figure 13: Boxplot: # of example graphs inspected before accepting or refuting a pattern candidate.

6 Conclusion

Graph mining is an increasingly important task within the realm of data mining. It is being used in bioinformatics, chemoinformatics [40], circuit analysis and many other fields. As shown in Section 2, it is easily defined mathematically. As Section 3 has shown, it is also possible to easily transform this mathematical definition into a higher-order logic specification with great clarity. However, support for higher-order logic specifications is limited in current state-of-the-art KR systems.

We specifically (Section 4) looked at how IDP and ASP, two state-of-the-art KR languages, allow one to model the graph mining problem. The resulting first-order encodings are not straightforwardly derived from the mathematical definition of graph mining, and, relatedly, do not reach the same level of clarity or ease of adaptation. Instead, several encoding techniques such as the *disjoint union*

technique and *saturation* technique are required to make the graph mining problem expressible in these languages. Furthermore, it is not always possible to solve a graph mining problem with a single solver call, limiting its compatibility with the *knowledge base paradigm*. Instead, we must write procedural code which fixes the flow of information and the inference(s) being used while writing the model.

Looking at these limitations, we have identified opportunities for adding support for higher-order logic to state-of-the-art KR systems such as IDP or ASP. However, the expressiveness of higher-order logic in general raises concerns about the performance of systems supporting higher-order logic. Nevertheless, we suggested that in real-world applications, higher-order logic might open up new ways for solvers to benefit from structure in problems, for example, through independence analysis. Thus, in Section 5, we investigated two techniques for adding higher-order logic support to KR systems, while paying attention to how they might aid such independence analysis: The first technique, *nested solvers*, was concerned with supporting universal quantification and local quantifications of higher-order objects, while the second, *lazy grounding*, was mainly concerned with issues surrounding data representation.

For *nested solvers* we experimented with an implementation in imperative code that should mimic nested solvers in two different settings (**fully decomposed** and **semi decomposed**) and have found that both settings hold a clear advantage over the first-order **monolithic** setting. For **lazy grounding** we experimented with its existing experimental implementation in IDP, turning it on for the **semi decomposed** setting. While we had thought that *lazy grounding* would bring the **semi decomposed** setting closer to the **fully decomposed** setting, we actually found that, in its current implementation, *lazy grounding* actually causes a slowdown. It is clear that more research with respect to existing techniques and systems is needed.

7 Future Work

The conclusions from the previous section motivate us to further explore ways of introducing higher-order logic in existing state-of-the-art KR systems. In this section, we identify three potential paths for going forward: (1) we build upon the idea of nested solvers, implementing syntax based decomposition techniques introducing stacks of subsolvers automatically, not only for graph mining but for other higher-order logic specifications as well, (2) we ground to *quantified Boolean formulas* (QBF) as an alternative approach to nested solvers, and/or (3) we further explore *lazy grounding*. These last two paths are discussed in more detail.

7.1 Grounding to QBF

Existing state-of-the-art KR systems are commonly based on SAT solvers. One other option to introduce higher-order support is by using Quantified Boolean Formula or QBF solvers instead.

These solvers accept formulas of the form

$$\forall x_1 \exists x_2 \forall x_3 \dots Q_n x_n : \phi(x_1, x_2, x_3, \dots, x_n)$$

where $\forall x_1 \exists x_2 \forall x_3 \ldots Q_n x_n$ is called the *quantifier prefix* and $\phi(x_1, x_2, x_3, \ldots, x_n)$ represents an unquantified Boolean formula. To test the performance of a KR-system based on a QBF solver, we would first implement a grounding system from higher-order expressions to QBF formulas, making use of the fact that we can ground quantification over a predicate by quantifying over the ground atoms representing the predicate. Currently, work towards a grounding system from higher-order expressions to QBF formulas has started, and recently, results of a prototype system supporting second order quantification, arithmetic and constraints have been published [22].

We can then use existing QBF solvers to solve the resulting ground formulas. By employing a solver which uses a dependency learning technique [37], it is possible to solve these formulas by assigning atoms a value without taking into account their order in the quantifier prefix.

Such a system can derive a set of independencies using the same syntactical analysis of the higher-order specification proposed earlier. Then, instead of starting with the empty set, these derived dependencies can be used to bootstrap a QBF solver supporting dependency learning [37] (e.g. DepQBF).This way, we can again leverage additional independencies evident in the higher-order specification while possibly deriving more, perhaps otherwise unidentified dependencies on the propositional level.

7.2 Lazy Grounding

As mentioned in Section 5.2, we expected that enabling lazy grounding for the **semi decomposed** pipeline would prevent the grounding and solving of some of the example graphs, and that, as a result, the **semi decomposed** pipeline would achieve results very close to the **fully decomposed** pipeline. Instead, our experiments showed an overall slowdown of the **semi decomposed** pipeline when using lazy grounding. Possible explanations are a high setup cost for the general method of lazy grounding or a high penalty being incurred every time an additional graph has to be grounded.

Metrics such as the 'hardness' of each dataset, which expresses how many example graphs on average are needed to accept or refute a pattern candidate, can give some indication towards the cause of the overall slowdown. However, for a detailed analysis changes to both the experiments as well as the lazy grounding implementation are needed.

It is important to note that other approaches exist to implement lazy grounding [12,41] in ASP, for example lazy constraints, where a set of constraints $\mathcal{C}$ is identified which causes large grounding. This set $\mathcal{C}$ is *not grounded*, instead, a solution candidate is generated without it and this solution candidate is subsequently checked w.r.t. the constraints in $\mathcal{C}$. If any of the constraints in $\mathcal{C}$ is violated, a (ground) conflict is learned. We remark that this is similar to the way that the **semi** and **fully decomposed** pipelines split candidate generation from checking the positive (and negative) constraints. As such, we would expect that a good implementation of this technique would indeed show similar performance gains as the introduction of these pipelines, without the need of procedural code, and this can be the target of future research.

Acknowledgements

We would like to thank the reviewers for their insightful and helpful comments, thanks to which many improvements could be made.

References

1. Abramson, H., Rogers, H.: Meta-programming in Logic Programming. MIT Press (1989)
2. Abrial, J.R.: The B-Book. Cambridge University Press (1996). DOI 10.1017/CBO9780511624162
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
4. Aoga, J.O.R., Guns, T., Schaus, P.: An efficient algorithm for mining frequent sequence with constraint programming. In: P. Frasconi, N. Landwehr, G. Manco, J. Vreeken (eds.) Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part II, *Lecture Notes in Computer Science*, vol. 9852, pp. 315–330. Springer (2016). DOI 10.1007/978-3-319-46227-1_20. URL https://doi.org/10.1007/978-3-319-46227-1_20
5. Babai, L.: Graph isomorphism in quasipolynomial time. CoRR **abs/1512.03547** (2015). URL http://arxiv.org/abs/1512.03547
6. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Solving QBF instances with nested SAT solvers. In: A. Darwiche (ed.) Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016., *AAAI Workshops*, vol. WS-16-05. AAAI Press (2016). URL http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12603
7. Bowen, J.P.: Formal Specification and Documentation using Z. International Thomson Computer Press (1996)
8. Brewka, G., Delgrande, J.P., Romero, J., Schaub, T.: asprin: Customizing answer set preferences without a headache. In: AAAI, pp. 1467–1474. AAAI Press (2015)
9. Bruynooghe, M., Blockeel, H., Bogaerts, B., de Cat, B., Pooter, S.D., Jansen, J., Labarre, A., Ramon, J., Denecker, M., Verwer, S.: Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with *IDP3*. Theory and Practice of Logic Programming (TPLP) **15**(6), 783–817 (2015). DOI 10.1017/S147106841400009X. URL https://doi.org/10.1017/S147106841400009X
10. de Cat, B., Denecker, M., Bruynooghe, M., Stuckey, P.J.: Lazy model expansion: Interleaving grounding with search. J. Artif. Intell. Res. **52**, 235–286 (2015). DOI 10.1613/jair.4591. URL https://doi.org/10.1613/jair.4591
11. Chen, W., Kifer, M., Warren, D.S.: Hilog: A foundation for higher-order logic programming. The Journal of Logic Programming **15**(3), 187–230 (1993)
12. Cuteri, B., Dodaro, C., Ricca, F., Schüller, P.: Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. Theory and Practice of Logic Programming (TPLP) **17**(5-6), 780–799 (2017)
13. Dasseville, I., van der Hallen, M., Janssens, G., Denecker, M.: Semantics of templates in a compositional framework for building logics. Theory and Practice of Logic Programming (TPLP) **15**(4-5), 681–695 (2015). DOI 10.1017/S1471068415000319. URL https://doi.org/10.1017/S1471068415000319
14. De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modelling language: The IDP system. CoRR **abs/1401.6312v2** (2016). URL http://arxiv.org/abs/1401.6312v2
15. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: ACM SIGKDD, pp. 204–212 (2008)
16. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Redl, C., Schüller, P.: A model building framework for answer set programming with external computations. Theory and Practice of Logic Programming (TPLP) **16**(4), 418–464 (2016)
17. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: A primer. In: Reasoning Web, *Lecture Notes in Computer Science*, vol. 5689, pp. 40–110. Springer (2009)
18. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers (2012)

19. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + control: Preliminary report. CoRR **abs/1405.3694** (2014)
20. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: Constraint Programming, Artificial Intelligence and Operations Research (CPAIOR), *Lecture Notes in Computer Science*, vol. 5547, pp. 71–86. Springer (2009)
21. Guyet, T., Moinard, Y., Quiniou, R., Schaub, T.: Efficiency Analysis of ASP Encodings for Sequential Pattern Mining Tasks, pp. 41–81. Springer International Publishing, Cham (2018). DOI 10.1007/978-3-319-65406-5_3. URL https://doi.org/10.1007/978-3-319-65406-5_3
22. van der Hallen, M., Janssens, G.: A grounder from second-order logic to qbf. In: Quantified Boolean Formulas, Papers from the 2018 FLoC Quantified Boolean Formulas and Beyond Workshop, Oxford, England, July 8, 2018 (accepted), Federated Logic Conference (FLoC): workshop proceedings (2018)
23. van der Hallen, M., Paramonov, S., Leuschel, M., Janssens, G.: Knowledge representation analysis of graph mining. CoRR **abs/1608.08956** (2016). URL http://arxiv.org/abs/1608.08956
24. Immerman, N.: Descriptive complexity and model checking. In: V. Arvind, R. Ramanujam (eds.) Foundations of Software Technology and Theoretical Computer Science, 18th Conference, Chennai, India, December 17-19, 1998, Proceedings, *Lecture Notes in Computer Science*, vol. 1530, pp. 1–5. Springer (1998). DOI 10.1007/978-3-540-49382-2_1. URL https://doi.org/10.1007/978-3-540-49382-2_1
25. Järvisalo, M.: Itemset mining as a challenge application for answer set enumeration. Logic Programming and Nonmonotonic Reasoning (LPNMR), pp. 304–310 (2011)
26. Kaufmann, B., Leone, N., Perri, S., Schaub, T.: Grounding and solving in answer set programming. AI Magazine **37**(3), 25–32 (2016). URL http://www.aaai.org/ojs/index.php/aimagazine/article/view/2672
27. Kemmar, A., Lebbah, Y., Loudni, S., Boizumault, P., Charnois, T.: Prefix-projection global constraint and top-k approach for sequential pattern mining. Constraints **22**(2), 265–306 (2017). DOI 10.1007/s10601-016-9252-z. URL https://doi.org/10.1007/s10601-016-9252-z
28. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
29. Leuschel, M., Butler, M.J.: ProB: An automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008)
30. Li, H., Yap, C.W., Ung, C.Y., Xue, Y., Cao, Z.W., Chen, Y.Z.: Effect of selection of molecular descriptors on the prediction of bloodbrain barrier penetrating and nonpenetrating agents by statistical learning methods. Journal of Chemical Information and Modeling **45**(5), 1376–1384 (2005). DOI 10.1021/ci050135u. URL https://doi.org/10.1021/ci050135u. PMID: 16180914
31. Lonsing, F., Biere, A.: Depqbf: A dependency-aware QBF solver. JSAT **7**(2-3), 71–76 (2010). URL http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_6_Lonsing.pdf
32. Lonsing, F., Egly, U., Gelder, A.V.: Efficient clause learning for quantified boolean formulas via QBF pseudo unit propagation. In: M. Järvisalo, A.V. Gelder (eds.) Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7962, pp. 100–115. Springer (2013). DOI 10.1007/978-3-642-39071-5_9. URL https://doi.org/10.1007/978-3-642-39071-5_9
33. McCarthy, J.: Elaboration tolerance. In: Working Papers of the Fourth International Symposium on Logical formalizations of Commonsense Reasoning, Commonsense-1998 (1998)
34. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. J. Log. Program. **19/20**, 629–679 (1994). DOI 10.1016/0743-1066(94)90035-3. URL https://doi.org/10.1016/0743-1066(94)90035-3
35. Nijssen, S., Kok, J.N.: Frequent graph mining and its application to molecular databases. In: Proceedings of the IEEE International Conference on Systems, Man & Cybernetics: The Hague, Netherlands, 10-13 October 2004, pp. 4571–4577. IEEE (2004). DOI 10.1109/ICSMC.2004.1401252. URL https://doi.org/10.1109/ICSMC.2004.1401252
36. Paramonov, S., Chen, T., Guns, T.: Generic mining of condensed pattern representations under constraints. In: CEUR: Young Scientist's Second International Workshop on Trends in Information Processing Proceedings (YSIP), vol. 1837, pp. 138–177 (2017)

37. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. In: S. Gaspers, T. Walsh (eds.) Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10491, pp. 298–313. Springer (2017). DOI 10.1007/978-3-319-66263-3_19
38. Rückert, U., Kramer, S.: Optimizing feature sets for structured data. In: J.N. Kok, J. Koronacki, R.L. de Mántaras, S. Matwin, D. Mladenic, A. Skowron (eds.) Machine Learning: ECML 2007, 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4701, pp. 716–723. Springer (2007). DOI 10.1007/978-3-540-74958-5_72. URL https://doi.org/10.1007/978-3-540-74958-5_72
39. Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: International Conference on Computer-Aided Design (ICCAD), San Jose, California, USA, November 10-14 1996, pp. 220–227 (1996)
40. Takigawa, I., Mamitsuka, H.: Graph mining: procedure, application to drug discovery and recent advances. Drug Discovery Today **18**(1), 50 – 57 (2013). DOI https://doi.org/10.1016/j.drudis.2012.07.016. URL http://www.sciencedirect.com/science/article/pii/S1359644612002759
41. Weinzierl, A.: Blending lazy-grounding and CDNL search for answer-set solving. In: Logic Programming and Nonmonotonic Reasoning (LPNMR), *Lecture Notes in Computer Science*, vol. 10377, pp. 191–204. Springer (2017)
42. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: Proceedings of the 2002 IEEE International Conference on Data Mining, ICDM '02, pp. 721–. IEEE Computer Society, Washington, DC, USA (2002)