Tackling scalability issues in mining path patterns from knowledge graphs: a preliminary study
Pierre Monnin, Emmanuel Bresso, Miguel Couceiro, Malika Smaïl-Tabbone, Amedeo Napoli, Adrien Coulet

To cite this version:

Pierre Monnin, Emmanuel Bresso, Miguel Couceiro, Malika Smaïl-Tabbone, Amedeo Napoli, et al.. Tackling scalability issues in mining path patterns from knowledge graphs: a preliminary study. ALGOS 2020 - 1st International Conference on Algebras, Graphs and Ordered Sets, Aug 2020, Nancy, France. hal-02913224

HAL Id: hal-02913224
https://inria.hal.science/hal-02913224
Submitted on 7 Aug 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract

Features mined from knowledge graphs are widely used within multiple knowledge discovery tasks such as classification or fact-checking. Here, we consider a given set of vertices, called seed vertices, and focus on mining their associated neighboring vertices, paths, and, more generally, path patterns that involve classes of ontologies linked with knowledge graphs. Due to the combinatorial nature and the increasing size of real-world knowledge graphs, the task of mining these patterns immediately entails scalability issues. In this paper, we address these issues by proposing a pattern mining approach that relies on a set of constraints (e.g., support or degree thresholds) and the monotonicity property. As our motivation comes from the mining of real-world knowledge graphs, we illustrate our approach with PGxLOD, a biomedical knowledge graph.

Keywords Path · Path Pattern · Ontology · Knowledge Graph · Scalability

1 Introduction

Knowledge graphs [1] have a central role in knowledge discovery tasks. For example, Linked Open Data [2] have been used in all steps of the knowledge discovery process [3]. In particular, features mined from knowledge graphs have been used in multiple applications such as knowledge base completion [4], explanations [5], or fact-checking [6]. Here, we focus on knowledge graphs expressed using Semantic Web standards [7]. In this context, vertices are either individuals that represent entities of a world (e.g., places, drugs, etc.), literals (e.g., integers, dates, etc.), or classes of individuals (e.g., Person, Drug, etc.). Arcs are defined by triples (subject, predicate, object) in the Resource Description Format language. Such a triple states that the subject is linked to the object by a relationship qualified by the predicate (e.g., has-side-effect, has-name, etc.). Classes and predicates are defined in ontologies, i.e., formal representations of a domain [8], and organized into two hierarchies ordered by the subsumption relation. In Semantic Web standards, individuals, classes, and predicates are identified by a Uniform Resource Identifier (URI). We view such a knowledge graph as a directed labeled multigraph \( \mathcal{K} = (\Sigma_V, \Sigma_A, V, A, s, t, \ell_V, \ell_A) \), where

*Supported by the PractiKPharma project, founded by the French National Research Agency (ANR) under Grant ANR15-CE23-0028, and by the Snowball Inria Associate Team.
We propose to mine from \( K \) sequences of pairs \( p \rightarrow e \) that represent an arc labeled by the predicate \( p \) incident to an individual \( e \). A path is associated with all seed vertices that root it in \( K \). The support of a path counts such seed vertices. For example, the support of \( (i) \) is 1 since only \( n_1^I \) root it, i.e., \( n_1^I \rightarrow v_2 \rightarrow v_3 \) exists in \( K \).

More generally, paths may share several characteristics. For instance, intermediate vertices in paths may instantiate the same ontology classes. We propose to capture these characteristics by considering path patterns in addition to paths. Path patterns are sequences of pairs \( p \rightarrow E \), where \( p \) is a predicate and \( E \) is either an individual or a class. Such a pair indicates that an arc labeled by \( p \) is incident to (i) \( E \) if \( E \) is an individual or (ii) an individual that instantiates \( E \) if \( E \) is a class. A path pattern is associated with all seed vertices that root a path captured by the path pattern. Its support counts such seed vertices. For example, the support of \( (ii) \) is 2 since \( n_1^I \rightarrow v_2 \rightarrow v_3 \rightarrow v_5 \) exists in \( K \).

Consequently, the support of \( (iii) \) is 2. This illustrates the fact that path patterns may capture additional common characteristics of seed vertices, and thus interestingly complete paths. Mining these patterns constitutes a challenging task due to the combinatorial nature and the size of real-world knowledge graphs, which naturally entail scalability issues. For example, \( p \rightarrow v_2 \rightarrow v_3 \rightarrow v_5 \) can be generalized by up to 11 path patterns.

This mining task and its inherent scalability issues constitute the main concerns of the present work. To the best of our knowledge, works available in the literature do not address such issues in knowledge graphs with the adopted granular modeling of path patterns. However, inspired by existing graph mining works \(^{[9]}\), we propose an Apriori-based approach that alleviates these scalability issues by relying on (i) a set of constraints (e.g., support or degree thresholds), (ii) the hierarchy of ontology classes, (iii) an incremental expansion of paths and patterns, and (iv) the monotonic character of the support of paths and patterns. We provide a reusable implementation on GitHub \(^{[4]}\).

The remainder of this paper is organized as follows. In Section \(^{2}\) we outline related works that motivated our proposed approach. In Section \(^{3}\) we present in details our approach to mine paths and path patterns, and discuss how it tackles scalability issues. We illustrate our framework in Section \(^{4}\) on PGLOD, a real-world biomedical knowledge graph \(^{[10]}\). We comment on our results as well as indicate directions of future work in Sections \(^{5}\) and \(^{6}\).

2 Related work

Path patterns have been widely studied in different settings, for example graph rewriting \(^{[11]}\) and query answering \(^{[12]}\). Here, we recall some works that tackle the problem of mining path patterns from knowledge graphs in different application contexts.

\(^{2}\) Here, we discard literals from \( V \) and arcs that are incident to literals from \( A \).

\(^{3}\) Hence, \(|\Sigma_V| = |V|\).

\(^{4}\) https://github.com/pmonnin/kgpm
Figure 1: Example of a canonical graph $K^C$. $n^C_1$ and $n^C_2$ are canonical seed vertices, all $v_i$ are canonical individuals, and all $T_s$ are canonical ontology classes. Prefixes of URIs were omitted for readability purposes. The definition of “canonical” is given in Subsection 3.1.

One major work dealing with feature mining from RDF graphs focuses on graph kernels that count common substructures (i.e., walks, subtrees) [13, 14]. To avoid an explosion in the number of features, the authors remove patterns with a low or high frequency [14]. Graph features can be used in various tasks such as knowledge base completion. For example, AMIE [15] mines Horn clauses, i.e., conjunction of triples, to predict another triple. Similarly, in Context Path Model [16], the authors model paths as sequences of predicates between the source and target entities, i.e., $s \xrightarrow{p_1} p_2 \ldots \xrightarrow{p_k} t$. Here, a triple is predicted based on the paths existing between the involved entities. Shi and Weninger [6] also model paths as sequences of predicates but they use them from a fact checking perspective. They check whether a triple $s \xrightarrow{p} t$ is true by predicting it from a set of learned discriminative paths $o_s \xrightarrow{p_1} p_2 \ldots \xrightarrow{p_{k-1}} p_k \xrightarrow{p} o_t$, where $o_s$ and $o_t$ are respectively the set of classes instantiated by $s$ and $t$. Our framework differs from the previous two as we aim at mining features by exploring the graph from the given seed vertices and we model path patterns using the intermediate entities and the ontology classes that they instantiate.

Our motivation also comes from explainable approaches that rely on the descriptive power of features mined from knowledge graphs. For example, Explain-a-LOD [5] enriches statistical data sets with features from DBpedia. When correlations can be established between statistics and DBpedia features, these features can be used as explanations for the original statistics. For example, the quality of living in cities has been correlated with whether these cities are European capitals. Explain-a-LOD leverages the different outputs of FeGeLOD [17], some of which corresponding to our approach. For instance, the so called relations $\xrightarrow{e}$ are paths, whereas the so called qualified relations $\xrightarrow{e}$, where $e$ is replaced by a class $t$ instantiated by $e$, are path patterns. Alternatively, Vandewiele et al. [18] propose to learn a decision tree to classify entities based on paths of a knowledge graph. The authors suggest that the predictions of their model are explainable as they are obtained by a “white-box” model (i.e., the decision tree) that combines interpretable features (i.e., paths from a knowledge graph). Interestingly, this system considers paths with their intermediate predicates and entities, $i.e., root \xrightarrow{p_1} e_1 \cdots \xrightarrow{p_k} e_k$. They allow a generalization of both predicates and entities by the use of a wildcard (*). In our context, this would correspond to generalizing entities by the top level ontology class $\top$. However, unlike their framework, we do not generalize predicates. In their study, they focus on paths of the form root $\xrightarrow{*} \cdots \xrightarrow{*} e$, which somewhat corresponds to extracting neighbors and their distance from seed vertices.

Finally, path patterns are somewhat similar to generalized association rules [19] and the concept of raising [20]. Indeed, both works replace entities in rules by ontology classes to increase the support while preserving a high confidence. Inspired by works that prune redundant generalized rules [21], our approach mines paths and path patterns that are non-redundant and comply with some given constraints.
Figure 2: Main steps to mine a set $F$ of features (i.e., neighbors, paths, and path patterns) associated with a set $N$ of seed vertices from a knowledge graph $K$. Step 4 is optional and depends on the application domain.

Table 1: Parameters that configure the mining of interesting neighbors, paths, and path patterns in a knowledge graph $K$. Each parameter is associated with a domain and is used in specific steps (see Figure 2 for step numbers). Parameter $m$ is specific to the considered application. Here, we illustrate the role of $m$ with the biomedical domain.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Domain</th>
<th>Steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>$\mathbb{N}^+$</td>
<td>2, 3</td>
<td>Maximum length of paths and path patterns</td>
</tr>
<tr>
<td>$t$</td>
<td>$\mathbb{N}$</td>
<td>3</td>
<td>Maximum level for generalization in class hierarchies</td>
</tr>
<tr>
<td>$d$</td>
<td>$\mathbb{N}$</td>
<td>2, 3</td>
<td>Maximum degree ($u = \text{true}$) or out degree ($u = \text{false}$) to allow expansion</td>
</tr>
<tr>
<td>$l_{\text{min}}$</td>
<td>$\mathbb{N}$</td>
<td>2, 3</td>
<td>Minimum support for features</td>
</tr>
<tr>
<td>$l_{\text{max}}$</td>
<td>$\mathbb{N}$</td>
<td>3</td>
<td>Maximum support for features</td>
</tr>
<tr>
<td>$u$</td>
<td>$\mathbb{B}$</td>
<td>2, 3</td>
<td>Whether only out arcs ($u = \text{false}$) or all arcs ($u = \text{true}$) are traversed</td>
</tr>
<tr>
<td>$\beta_{\text{predicates}}$</td>
<td>List of URIs</td>
<td>2, 3</td>
<td>Blacklist of predicates not to traverse</td>
</tr>
<tr>
<td>$\beta_{\text{exp-types}}$</td>
<td>List of URIs</td>
<td>2, 3</td>
<td>Blacklist of classes whose instances are not to reach</td>
</tr>
<tr>
<td>$\beta_{\text{gen-types}}$</td>
<td>List of URIs</td>
<td>2, 3</td>
<td>Blacklist of classes not to use in generalization</td>
</tr>
</tbody>
</table>
| $m$       | \{none, p, g, m, pg, pgm\} | 4     | Optional and domain-dependent filtering strategy 
Illustrated here with the biomedical domain |

3 Towards a scalable approach to mine interesting paths and path patterns

In this paper, we consider a knowledge graph $K$ and a set of seed vertices $N = \{n_1, n_2, \ldots, n_p\} \subseteq V$. The task is to mine neighbors, paths, and path patterns from $K$ that are associated with these seed vertices. For example, given a set of drugs that cause or not a side effect, we aim to mine features that can later be used to classify these drugs.

In the following subsections, we propose algorithms to build a binary matrix $M$ of size $|N| \times |F|$ from the knowledge graph $K$ and the set of seed vertices $N$. The set $F$ consists of interesting neighbors, paths, and path patterns mined from $K$, i.e., neighbors, paths, and path patterns that satisfy the constraints defined in terms of the parameters summarized in Table 1. These parameters will be detailed in the following subsections. $M$ associates a seed vertex $n \in N$ with its features $f \in F$, i.e., if $M_{n,f} = \text{true}$, then $n$ has feature $f$. We outline our approach in Figure 2, where steps 1, 2, and 3 are mandatory, while step 4 is optional and depends on the application domain.

3.1 Canonicalizing $K$

The first step of our approach consists in canonicalizing the knowledge graph $K$, i.e., unifying vertices that represent the same real-world entity. We use the canonicalization word by analogy with the canonicalization of knowledge bases, which consists in unifying equivalent individuals into one [4]. Indeed, in knowledge bases under the Open Information Extraction paradigm, facts and entities can be represented by synonymous terms, which leads to co-existing and equivalent individuals. For example, in such knowledge bases, two individuals Obama and Barack Obama can co-exist. Similarly, in $K$, vertices can be connected through arcs labeled by the $\text{owl:sameAs}$ predicate, indicating that these vertices are actually representing the same real-world entity.

Such a situation typically arises when $K$ comprises several data sets. For example, a drug can be represented by two vertices linked by an $\text{owl:sameAs}$ arc, resulting from the information extraction of two independent drug-related databases. Therefore, their merging allows an easy access to the full extent of the knowledge in $K$ about the drug they
represent. Such a canonicalization process corresponds to edge contraction in graph theory (i.e., taking graph quotient). In our framework, it reduces to contracting arcs whose label is the `owl:sameAs` predicate.

To perform this canonicalization, we must respect the semantics associated with the `owl:sameAs` predicate, and thus take into account its symmetry and transitivity. Indeed, an `owl:sameAs` arc between two vertices either is explicitly stated in $\mathcal{K}$ or follows from existing arcs and these two properties. Let us consider a vertex $v$ in $\mathcal{K}$. The canonicalization step merges $v$ with all its identical vertices based on `owl:sameAs` arcs. To compute this set of vertices, it suffices to compute the connected component of $v$ in the undirected spanning subgraph formed by the `owl:sameAs` arcs of $\mathcal{K}$. Indeed, undirected edges comply with the symmetry of `owl:sameAs` and connected components comply with the transitivity of `owl:sameAs`.

As a result, this step takes $\mathcal{K}$ as input and outputs its canonical graph $\mathcal{K}^C = (\Sigma_V^C, \Sigma_A^C, V^C, A^C, s^C, t^C, \ell_V^C, \ell_A^C)$. Similarly to $\mathcal{K}$, $\mathcal{K}^C$ is a directed labeled multigraph where

- $V^C$ is the set of canonical vertices.
- $A^C$ is the set of canonical arcs connecting canonical vertices through predicates.
- $\Sigma_V^C$ is the set of canonical vertex labels.
- $\Sigma_A^C$ is the set of canonical arc labels.
- $s^C : A^C \rightarrow V^C$ (respectively $t^C : A^C \rightarrow V^C$) associates a canonical arc to its canonical source (respectively target) vertex.
- $\ell_V^C : V^C \rightarrow \Sigma_V^C$ (respectively $\ell_A^C : A^C \rightarrow \Sigma_A^C$) maps a canonical vertex (respectively a canonical arc) to its label.

Each canonical vertex in $\mathcal{K}^C$ represents a vertex from $\mathcal{K}$ and all its identical vertices. It is possible for a canonical vertex in $\mathcal{K}^C$ to only represent one vertex from $\mathcal{K}$ if $v$ has no identical vertices. This corresponds to creating a surjective mapping $\lambda : V \rightarrow V^C$ associating a vertex from $\mathcal{K}$ to its equivalent canonical vertex in $\mathcal{K}^C$. Canonical arcs in $\mathcal{K}^C$ are constructed by using $\lambda$ to map the source and target vertices of arcs in $\mathcal{K}$ to canonical vertices. Similarly, the set of seed vertices $N$ is mapped to the set of canonical seed vertices, denoted by $N^C$.

Remark 1. Note that storing URIs has a high memory footprint. Thus, in $\mathcal{K}^C$, we use indices in $\mathbb{N}$ instead of URIs to label vertices and arcs, i.e., $\Sigma_V^C \subseteq \mathbb{N}$ and $\Sigma_A^C \subseteq \mathbb{N}$. This leads to a reduced memory consumption in subsequent algorithms. Each canonical vertex has one unique label, differing from labels of other canonical vertices, i.e., $|\Sigma_V^C| = |V^C|$. This “relabelling” is inspired by the work of de Vries and de Rooij [13] that use a structure named `pathMap` to represent a path by an integer. We developed our own structure for this relabelling, which we named `CacheManager`.

3.2 Mining interesting neighbors and types

3.2.1 Mining interesting neighbors

Here, we select all vertices that are neighbors of at least one seed vertex in $N^C$ by performing a breadth-first search constrained by parameters $k$, $d$, $u$, $b_{\text{predicates}}$, and $b_{\text{exp-types}}$. Neighbors are selected by traversing at most $k$ arcs from the seed vertices in $N^C$. If $u =$ `false`, then only outgoing arcs are traversed; otherwise, all arcs are traversed regardless of their orientation.

However, not all neighboring vertices are of interest. For example, we want to avoid provenance metadata vertices. Indeed, they may not constitute discriminative features as they are specific to the vertex they describe. As we aim to use ontology classes to generate path patterns, we also need to keep the graph exploration over the individuals of $\mathcal{K}$ and avoid traversing `rdf:type` arcs. To this aim, we do not traverse arcs that are labeled by a predicate whose URI or prefix of URI is blacklisted in $b_{\text{predicates}}$. For example, we blacklist in $b_{\text{predicates}}$ the prefix of the provenance ontology `PROV-` and the URI of the `rdf:type` predicate.

Additionally, we provide a blacklist $b_{\text{exp-types}}$ of URIs or prefixes of classes whose instances must not be reached. Hence, we do not reach individuals that instantiate directly or indirectly a blacklisted class, by following `rdf:type` and `rdfs:subClassOf` arcs. For example, in a use case of classifying drugs that cause or not a side effect, one may want to avoid neighbors that represent the side effect. That is why, the ontology class representing the side effect is blacklisted in $b_{\text{exp-types}}$.

\[\text{http://www.w3.org/ns/prov#} \]
\[\text{http://www.w3.org/1999/02/22-rdf-syntax-ns#type} \]
When mining neighboring vertices, we may encounter vertices with a high degree, hereafter named hubs. If the graph exploration considered their numerous neighbors, then the size of the selected neighborhood would increase exponentially, thus causing a scalability issue. Additionally, hub neighbors may not constitute specific and discriminative features. Indeed, if a hub can be reached from some seed vertices, i.e., appears in their neighborhood, the neighbors of the hub will be reached by the same seed vertices. That is why, in our approach, we propose to stop the graph exploration at vertices whose degree is strictly greater than parameter $d$.

**Remark** 2. If $u = \text{false}$, then the degree of a vertex only counts outgoing arcs, otherwise all arcs are counted. The degree does not count arcs whose predicate is blacklisted in $b_{\text{predicates}}$. The degree counts arcs incident to a vertex that instantiates a blacklisted class in $b_{\text{exp-types}}$.

As a result, with $k$, $d$, $u$, $b_{\text{predicates}}$, and $b_{\text{exp-types}}$ fixed, we obtain a set of neighboring vertices, denoted by $\mathcal{N}(N^C) \subseteq V^C$. Each neighboring vertex $v \in \mathcal{N}(N^C)$ may only appear in the neighborhood of some seed vertices from $N^C$ w.r.t. the parameters. Thus, $v$ is associated with these seed vertices, which we indicate by defining the support set of $v$.

**Definition 1** (Support set of a neighbor). For a given choice of these parameters, the support set of a neighboring vertex $v \in \mathcal{N}(N^C)$ is denoted by $\text{SupportSet}(v) \subseteq N^C$ and defined as the set of seed vertices from $N^C$ having $v$ as neighbor. The support of a neighbor is defined as the cardinal of its support set.

Note that some vertices in $\mathcal{N}(N^C)$ are not very discriminative: when they are associated with very few vertices from $N^C$ or nearly all of them. This motivates the use of parameters $l_{\text{min}}$ and $l_{\text{max}}$ that define the minimum and maximum support for a neighbor to appear in the set $F$ of features. Hence, a neighbor $v \in \mathcal{N}(N^C)$ constitutes a feature in the output matrix $M$ if and only if $l_{\text{min}} \leq |\text{SupportSet}(v)| \leq l_{\text{max}}$. We denote the set of interesting neighbors to appear in $F$ by

$$\mathcal{N}_i(N^C) = \left\{ v \mid v \in \mathcal{N}(N^C) \text{ and } l_{\text{min}} \leq |\text{SupportSet}(v)| \leq l_{\text{max}} \right\}.$$  

In $M$, for $n^C \in N^C$ and $v \in \mathcal{N}_i(N^C)$, we have $M_{n^C,v} = \text{true}$ if and only if $n^C \in \text{SupportSet}(v)$.

**Example 1.** From $K^C$ in Figure 1 and with $k = 3$, $d = 4$, $l_{\text{min}} = 2$, $l_{\text{max}} = 3$, $u = \text{false}$, $b_{\text{predicates}} = \{ \text{type}, \text{subClassOf} \}$, and $b_{\text{exp-types}} = \emptyset$, we obtain:

$$\mathcal{N}(N^C) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_8, v_9\};$$

$$\text{SupportSet}(v_1) = \text{SupportSet}(v_6) = \{n^C_1, n^C_2\};$$

$$\text{SupportSet}(v_2) = \text{SupportSet}(v_3) = \text{SupportSet}(v_8) = \{n^C_1\};$$

$$\text{SupportSet}(v_4) = \text{SupportSet}(v_5) = \text{SupportSet}(v_9) = \{n^C_2\};$$

$$\mathcal{N}_i(N^C) = \{v_1, v_9\}.$$

The graph exploration stops at $v_1$. Indeed, it is considered a hub as its degree is greater than $d$. Thus, vertices on the left of Figure 1 are not explored. Because $u = \text{false}$, the graph exploration cannot reach $v_7$. If $b_{\text{exp-types}} = \{T_3\}$, then $v_3$ and $v_5$ cannot be traversed, resulting in $\mathcal{N}(N^C) = \{v_1, v_2, v_4, v_8, v_9\}$.

### 3.2.2 Mining interesting types

Observe that we can use $\mathcal{N}(N^C)$ to compute interesting types over the considered neighborhood. These interesting types will alleviate a scalability issue arising when building path patterns in Subsection 3.3. Interesting types must be computed over $\mathcal{N}(N^C)$ and not $\mathcal{N}_i(N^C)$ as vertices whose support is below $l_{\text{min}}$ can instantiate interesting types. As an intuitive example, in Figure 1 $T_3$ is associated with both $n^C_1$ (because of $v_3$) and $n^C_2$ (because of $v_5$). For $l_{\text{min}} = 2$, $v_3$ and $v_5$ will not be selected as features, however $T_3$ can be used in path patterns.

Parameters $t$ and $b_{\text{gen-types}}$ constrain the ontology classes considered in the construction of path patterns, and they are integrated in the computation of interesting types. Parameter $t$ specifies the maximum level of considered classes in ontology hierarchies. This level is computed by starting at vertices to generalize and following $\text{rdfs:subClassOf}$ arcs. $t = 0$ only allows to generalize vertices with $\top$, which is considered to be instantiated by all vertices. For example, $v_3$ can be generalized by $T_2$ and $\top$ if $t = 1$, and by $T_2$, $T_3$, and $\top$ if $t = 2$. Additionally, types used for generalization must not be blacklisted in $b_{\text{gen-types}}$. This blacklist consists of URIs or prefixes of ontology classes not to be used during the construction of path patterns. For example, we refrain from considering general classes such as $\text{pgxo:Drug}$.

---

7From a similar assessment, de Vries and de Rooij [14] tackle the hub issue by removing edges based on frequency of pairs (source, predicate) and (predicate, target).

8[http://pgxo.loria.fr/Drug](http://pgxo.loria.fr/Drug)
To compute interesting types, we must first compute their support set. This motivates the following predicate:

\[
\text{inst}(v, T, t, b_{\text{gen-types}}) = \begin{cases} 
\text{true} & \text{if } v \text{ instantiates } T \text{ under parameters } t \text{ and } b_{\text{gen-types}} \\
\text{false} & \text{otherwise}
\end{cases}
\]

We can then define the support set of an ontology class \( T \) as follows:

**Definition 2** (Support set of an ontology class). The support set of an ontology class \( T \) is the union of support sets of vertices \( v \in \mathcal{N}(\mathcal{N}^C) \) that can be generalized by \( T \) under parameters \( t \) and \( b_{\text{gen-types}} \). Formally:

\[
\text{SupportSet}(T) = \bigcup_{v \in \mathcal{N}(\mathcal{N}^C) \text{ such that } \text{inst}(v, T, t, b_{\text{gen-types}})} \text{SupportSet}(v)
\]

Finally, the set of interesting types used to build path patterns is defined as \( T_{\geq l_{\min}} = \{ T \mid l_{\min} \leq |\text{SupportSet}(T)| \} \).

**Example 2.** With the same parameters as in Example 1 we obtain

\[
\text{SupportSet}(T_2) = \{ n_1^C \} ; \quad \text{SupportSet}(T_1) = \{ n_1^C, n_2^C \} ; \quad T_{\geq l_{\min}} = \{ T_1, T_3, T_5, T_6, \top \} .
\]

### 3.3 Mining interesting paths and path patterns

This step focuses on mining interesting paths and path patterns rooted by seed vertices \( n_i^C \in \mathcal{N}^C \). We use the term *path feature* (PF) to indicate a path or a path pattern.

**Definition 3** (Path feature). A path feature is a sequence of atomic elements that are pairs \( p \rightarrow E \) where \( p \) is a predicate and \( E \) is either (i) an individual (for paths), or (ii) an individual or an ontology class (for path patterns). The length of a path feature counts the number of its atomic elements.

**Example 3.** In Figure 1 the path \( p_1 \rightarrow v_2 \rightarrow p_2 \rightarrow v_3 \rightarrow p_3 \rightarrow v_6 \) can be rooted by \( n_1^C \) and is of length 3. The path pattern \( p_1 \rightarrow T_1 \rightarrow p_2 \rightarrow T_3 \) can be rooted by \( n_1^C \) and \( n_2^C \) and is of length 2.

Interesting path features are built by a breadth-first expansion starting at vertices in \( \mathcal{N}^C \). As previously, \( k \) defines the maximum number of arcs traversed. Hence, path features are of length 1 to \( k \). Observe that a scalability issue arises when mining interesting path features. Indeed, there may be several paths between two vertices and each vertex in a path can be generalized by several ontology classes. For instance, for \( t = 2 \), \( p_1 \rightarrow v_2 \rightarrow p_2 \rightarrow v_3 \rightarrow p_3 \rightarrow v_6 \) can be generalized by up to 23 path patterns. We propose a mining procedure that alleviates the scalability issues associated with the mining of path patterns. This mining procedure relies on the *monotonicity* of the support set of path features, which is defined as follows:

**Definition 4** (Support set of a path feature). The support set of a path consists of all vertices from \( \mathcal{N}^C \) that root it in \( \mathcal{K}^C \). Formally,

\[
\text{SupportSet}(p_1 \rightarrow v_a \ldots p_h \rightarrow v_b) = \left\{ n^C \in \mathcal{N}^C \mid n^C \text{ } p_1 \rightarrow v_a \ldots p_h \rightarrow v_b \text{ exists in } \mathcal{K}^C \right\}.
\]

The support set of a path pattern consists of all vertices from \( \mathcal{N}^C \) that root it in \( \mathcal{K}^C \) that is captured by the path pattern. Formally,

\[
\text{SupportSet}(p_a \rightarrow E_a \ldots p_b \rightarrow E_b) = \left\{ n^C \in \mathcal{N}^C \mid n^C \text{ } p_a \rightarrow v_a \ldots p_b \rightarrow v_b \text{ exists in } \mathcal{K}^C \text{ and } \forall v_i, \ i = E_i \text{ or } \text{inst}(v_i, E_i, t, b_{\text{gen-types}}) \right\}.
\]

The support of a path feature is defined as the cardinal of its support set.

**Example 4.** From Figure 1 we have \( \text{SupportSet}(p_1 \rightarrow v_2 \rightarrow p_2 \rightarrow v_3 \rightarrow p_3 \rightarrow v_6) = \{ n_1^C \} \).

Our approach of mining interesting path features is guided by the *dependency structure* as illustrated in Figure 3. At first, this structure is empty and it is then augmented at each iteration of Algorithm 1 whose operations are described and illustrated below.

**Remark 3.** As introduced in Remark 1 there are some hacks to help mitigating some of the scalability drawbacks. Storing and manipulating a path feature as a list of elements has a high memory footprint. Thus, such a list is only stored once in our CacheManager structure and the returned index (from \( \mathbb{N} \)) is used in mining algorithms.
Figure 3: Dependency structure used when mining interesting path features from the canonical graph in Figure 1. Parameters are $k = 3$, $t = 2$, $d = 4$, $u = false$, $b_{predicates} = \{type, subClassOf\}$, $b_{exp-types} = \emptyset$, $b_{gen-types} = \emptyset$, $l_{min} = 2$, and $l_{max} = 3$. Path features are displayed with their support set. Solid arrows represent expansion, dashed arrows represent generalization, rectangles represent paths, and rounded rectangles represent path patterns. Hatched rectangles represent discarded path features because of support limits or more specific path features with identical support set. Green rectangles represent path features ultimately added to $F$. Blank rectangles represent path features that respect specificity and support constraints but are not in $F$ because of other features (in green). For readability purposes, path features are displayed as the list of their elements instead of indices from $\mathbb{N}$ actually used to save memory.
We only generalize paths with types $T$ which is expanded into Algorithm 1. In the first iteration, the neighborhood of seed vertices in $N_C$ is used. If no path in $P_h$ can be expanded, i.e., the neighborhood of their last vertex does not contain reachable vertices under the constraints, then Algorithm 1 ends.

Example 5. From the graph in Figure 1, the first expansion generates the following paths: $p_1 \rightarrow v_1$, $p_3 \rightarrow v_2$, $p_1 \rightarrow v_4$, $p_6 \rightarrow v_8$, and $p_6 \rightarrow v_9$. In the second expansion, $p_4 \rightarrow v_1$ is not expanded as $v_1$ is a hub under $d = 4$. Since their respective neighborhood does not contain reachable vertices, $p_6 \rightarrow v_8$ and $p_6 \rightarrow v_9$ are also not expanded. The expansion of $p_1 \rightarrow v_2$ generates $p_1 \rightarrow v_2$, $p_2 \rightarrow v_3$ whereas the expansion of $p_1 \rightarrow v_4$ generates $p_1 \rightarrow v_4$, $p_2 \rightarrow v_5$.

3.3.1 Expand paths in $P_h$

Each path $P \in P_h$ is expanded with pairs $P \rightarrow v_e$ that are chosen in the neighborhood of the last individual of $P$. This choice is constrained by parameters $k, d, u, b_{\text{predicates}}$, and $b_{\text{exp-types}}$ as in Subsection 3.3.19. In the first iteration, the neighborhood of seed vertices in $N_C$ is used. If no path in $P_h$ can be expanded, i.e., the neighborhood of their last vertex does not contain reachable vertices under the constraints, then Algorithm 1 ends.

3.3.2 Generalize expanded paths into path patterns

Let $P_e$ be the expansion of $P \in P_h$ as previously described, i.e., $P_e = P \rightarrow v_e$. We generalize $P_e$ by:

- Generating patterns $P \rightarrow v_e$ with types $T \in T_{l_{\text{min}}}$ for which the predicate $\text{inst}(v_e, T, t, b_{\text{gen-types}})$ is verified.
- Retrieving from the dependency structure the path patterns that generalize $F(\ldots)$ and expanding them with $P \rightarrow v_e$, and $P \rightarrow T$ for all $T \in T_{l_{\text{min}}}$ for which the predicate $\text{inst}(v_e, T, t, b_{\text{gen-types}})$ is verified.

Intuitively, this generalization operation allows to expand path patterns.

Example 6. In the first iteration, $p_1 \rightarrow v_2$ is generalized by $p_1 \rightarrow T_1$ and $p_1 \rightarrow \top$. As we will see, $p_1 \rightarrow \top$ is not kept in the dependency structure at the end of the first iteration (see Subsections 3.3.3 and 3.3.4). In the second iteration, $p_1 \rightarrow v_2$ expands into $p_1 \rightarrow v_2$, $p_1 \rightarrow v_3$. In the dependency structure, we retrieve $p_1 \rightarrow T_1$ as the path pattern generalizing $p_1 \rightarrow v_2$, which is expanded into $p_1 \rightarrow T_1$, $p_2 \rightarrow v_3$, $p_1 \rightarrow T_1$, $p_2 \rightarrow T_3$, and $p_1 \rightarrow T_1$, $p_2 \rightarrow \top$.

We only generalize paths with types $T \in T_{l_{\text{min}}}$ to avoid the generation an important number of uninteresting path patterns that would then be discarded, thus reducing the memory footprint. Indeed, by definition, if $T \notin T_{l_{\text{min}}}$, then $|\text{SUPPORT}(T)| < l_{\text{min}}$. Additionally, given a path feature $P$, $|\text{SUPPORT}(P)| \leq \min_{E \in P} |\text{SUPPORT}(E)|$, where $E$ can be a class or an individual involved in $P$. Thus, if $T \notin T_{l_{\text{min}}}$ is used in a path pattern $P$, we would have $|\text{SUPPORT}(P)| < l_{\text{min}}$, therefore generating an uninteresting path pattern that would be discarded later.

3.3.3 Keep most specific path features

 Inspired by works that prune redundant generalized rules during their generation [21], we keep only the “most specific” path patterns among those that have the same support set.

---

9 $P_h$ is explained in Subsection 3.3.19
10 Additionally, to avoid loops, $F$ can only be expanded at iteration $h$ with individuals $v_e$ such that there exists at least one seed vertex in $\text{SUPPORT}(P)$ whose shortest distance to $v_0$ is $h$.
11 Not all generated path patterns remain in the dependency structure at the end of an iteration, see Subsections 3.3.3 and 3.3.4.
Definition 5 (More specific path pattern). A path pattern $P_1$ is more specific than another path pattern $P_2$ if every atomic element of $P_1$ is more specific than the atomic element of $P_2$ at the same position. An atomic element $p_1 \rightarrow E_1$ is more specific than another atomic element $p_2 \rightarrow E_2$ if and only if:

(i) $p_1 = p_2$, i.e., both atomic elements involve the same predicate and

(ii) $E_1$ is more specific than $E_2$.

When path patterns have the same support set, keeping the most specific ones remove redundant generalizations, thus reducing their number and the computational burden. Additionally, we ensure a high descriptive power because the most specific paths are the most descriptive. Intuitively, a path pattern involving a class $T$ is less descriptive than another pattern involving a subclass or an instance of the class. However, keeping the most specific path patterns is computationally expensive, which led us to propose the following computational procedure.

We notice that the support set of a path pattern is the union of the support sets of the paths it generalizes. Therefore, we discard path patterns that generalize only one path. Indeed, such path patterns have the same support set as their original path and are more general, by definition.

Example 7. For $h = 3$, we discard the following path patterns: $p_1 \rightarrow v_2 \rightarrow p_2 \rightarrow v_3 \rightarrow \top$ and $p_1 \rightarrow v_4 \rightarrow p_2 \rightarrow v_5 \rightarrow p_3 \rightarrow \top$.

However, there may exist path patterns that generalize several more specific path features while having the same support set. Such path patterns should also be discarded.

Example 8. For $h = 1$, $p_2 \rightarrow \top$ shares the same support set as the more specific path pattern $p_1 \rightarrow T_1$ and thus should be discarded.

To efficiently discard path patterns, we avoid computing their whole hierarchy. Instead, we focus on retaining only the most specific ones in the prefix tree depicted in Figure 4. This prefix tree is incrementally augmented and stores the most specific path patterns for a specific iteration and support set. In this tree, individuals/classes and predicates involved in path patterns are indexed separately. Thus, its depth is twice the length of path patterns of the current iteration.

The prefix tree enables an efficient storage and selection of the most specific path patterns. Indeed, let $P$ be a path pattern to be compared with those already stored. A breadth-first traversal is performed to detect more specific patterns than $P$: we only traverse identical or more specific elements according to Definition 5. At any depth, if such elements cannot be found, then $P$ is one of the most specific patterns and the traversal stops. On the contrary, if the traversal reaches a leaf containing more specific elements, then there is a pattern more specific than $P$ with the same support set. Consequently, $P$ is discarded and removed from the dependency structure.

If $P$ is to be stored, another breadth-first traversal is performed by considering identical or more general elements than the ones in $P$. When the traversal reaches a leaf, it means that more general patterns than $P$ are currently stored and have the same support set. These are removed from the prefix tree before storing $P$. They are also removed from the dependency structure.

Remark 4. As the prefix tree relies on associative arrays and sets, the computational cost of traversal, insertion, and removal is reduced. Additionally, resetting the tree at each expansion and each support set reduces the number of patterns to traverse and thus the global computational cost.

3.3.4 Select generated path features to add to $F$, and paths to add to $P_{h+1}$

In this subsection, we determine (i) which paths and path patterns should be added as features in $F$, and consequently, (ii) which paths to add to $P_{h+1}$, i.e., to expand during the next iteration.

A path feature $P$ can be added in $F$ if:

(C1) $l_{\text{min}} \leq \text{SupportSet}(P) \leq l_{\text{max}}$.

(C2) Prefixes of $P$ with the same support set do not already exist in $F$.

(C3) If $P$ is a path pattern, it does not generalize a path with the same support set.

When $P$ is in $F$, the output binary matrix $M$ verifies $M_{n^C, P} = \text{true}$ for all $n^C \in \text{SupportSet}(P)$.

---

[12] It is noteworthy that we do not consider the hierarchy of predicates in this work.

[13] A class is more specific than all its super-classes and an individual is more specific than all classes it instantiates.
We also select the paths and path patterns to expand during the next iteration. To reduce their number, we rely on the domain-dependent filtering. After the previous steps, we obtain a feature set when considered, some specified ontology classes [21, 22].

Remark 5. Note that (C2) allows to focus on shorter paths in \( \mathcal{F} \). However, (C2) is not applied if \( P \) ends with an individual and there exist a prefix of \( P \) in \( \mathcal{F} \) that ends with a class. \( P \) is considered more descriptive than its prefix, because of the individual in the last position. Hence, we add \( P \) in \( \mathcal{F} \) and remove its prefix. For example, in Figure 3, \( P_1 \rightarrow T_1 \rightarrow P_2 \rightarrow \top \rightarrow \top \) is replaced by \( P_1 \rightarrow T_1 \rightarrow P_2 \rightarrow T_3 \rightarrow v_6 \) in \( \mathcal{F} \) for \( h = 3 \).

Remark 6. (C3) is motivated as the path is more specific and thus more descriptive than \( P \) and should be added instead of \( P \).

We also select the paths and path patterns to expand during the next iteration. To reduce their number, we rely on the \( l_{\min} \) constraint and the monotonicity of the support set. It is clear that, for a path feature \( P \), we have \( |\text{SupportSet}(P)| \leq \min_{E \in P} |\text{SupportSet}(E)| \), where \( E \) can be a class or an individual involved in \( P \). Thus, when expanding a path feature, its support set remains identical (for paths and path patterns) or decreases (for path patterns).

Consequently, we add to \( \mathcal{P}_{h+1} \) paths whose expansion may generate path features complying with the \( l_{\min} \) constraint, i.e., paths with a support greater than \( l_{\min} \) or paths that are generalized by a pattern with a support greater than \( l_{\min} \). This monotonicity property also lets us remove from the dependency structure patterns whose support is lower than \( l_{\min} \). Indeed, such patterns cannot be used during the generalization step of the next iteration since they will inevitably generate a pattern whose support is smaller than \( l_{\min} \). As a result, the monotonicity property does entail a reduction in the number of paths and path patterns considered in the next iteration, thus reducing the computational cost.

Example 9. For example, in the first iteration, the path \( P_1 \rightarrow v_2 \) is added to \( \mathcal{P}_2 \) as it is generalized by \( P_1 \rightarrow T_1 \) whose support is greater than \( l_{\min} = 2 \). At the end of the second iteration, we remove \( P_1 \rightarrow v_2 \rightarrow P_2 \rightarrow T_3 \) because its support is lower than \( l_{\min} = 2 \), and thus its expansion cannot generate an interesting pattern.

3.4 Optional and domain-dependent filtering

After the previous steps, we obtain a feature set \( \mathcal{F} \) containing interesting neighbors, paths, and path patterns. These features have been mined without taking into account domain constraints known to experts. We propose to apply domain-dependent filtering on \( \mathcal{F} \) with parameter \( m \). Such filters reduce the size of \( \mathcal{F} \) and integrate interestingness constraints based on expert knowledge.

Example 10. To classify drugs causing or not a side effect, experts may want to focus on features containing a biological pathway, a gene or a GO class, or a MeSH class. Therefore, we propose three atomic filters, only keeping neighbors, paths, and path patterns containing at least a pathway \( (m = p) \), a gene or a GO class \( (m = g) \), or a MeSH class \( (m = m) \). Such atomic filters can be combined to form disjunctive filters. For example, the \( m = pg \) filter keeps features from \( \mathcal{F} \) containing at least a pathway or a gene or a GO class. When a filter is applied to a neighbor, this neighbor must be, e.g., a pathway for the \( p \) filter. When a filter is applied to a path or a path pattern, it means that one of its individuals / ontology classes must be, e.g., a pathway for the \( p \) filter.

This domain-dependent filtering is similar to approaches that generalize association rules and prune those that involve some specified ontology classes [21,22].
4 Experimental setup

To illustrate our approach, we will address the following task: from a knowledge graph, mine a set of features to classify drugs depending on whether they cause a specific side effect.

We explore PGxLOD\(^1\), a knowledge graph that aggregates several sets of Linked Open Data (LOD) describing drugs, phenotypes, and genetic factors: PharmGKB, ClinVar, DrugBank, SIDER, DisGeNET, and CTD. This aggregation may lead to features combining units from several LOD sets. Indeed, LOD sets may contain different and incomplete knowledge. Their combined use then enables leveraging a greater amount of knowledge, where some LOD sets complete information provided by others. This asks for a canonical knowledge graph as described in Subsection 3.1. For instance, it is possible to complete the knowledge related to a drug described in PharmGKB if it is linked with an owl:sameAs arc to the same drug described in DrugBank. This constitutes the key interest in combining LOD sets in knowledge discovery and data mining tasks, as discussed by Ristoski and Paulheim\(^2\).

We will use the following data sets that comprise positive (⊕) and negative (⊖) drug examples:

**Data set 1** (Drug Induced Liver Injury (DILI)\(^2\)). It is formed by 1,036 drugs in 4 classes: “most DILI concern” (192 drugs), “ambiguous DILI concern” (254 drugs), “less DILI concern” (278 drugs), and “no DILI concern” (312 drugs). We mapped these drugs from their PubChem identifiers to identifiers from PharmGKB, otherwise DrugBank, otherwise KEGG, resulting in the set of seed vertices \(N_D^{DILI} = N_B^{DILI} \cup N_T^{DILI}\) such that:

- \(|N_B^{DILI}| = 146\) drugs (118 from PharmGKB, 17 from DrugBank, and 11 from KEGG). The positive drug examples are from the “most DILI concern” class.
- \(|N_T^{DILI}| = 224\) drugs (206 from PharmGKB, 9 from DrugBank, and 9 from KEGG). The negative drug examples are from the “no DILI concern” class.

**Data set 2** (Severe Cutaneous Adverse Reactions (SCAR)\(^2\)). It is formed by 874 drugs in 5 classes: “very probable” (18 drugs), “probable” (19 drugs), “possible” (94 drugs), “unlikely” (697 drugs), and “very unlikely” (46 drugs). We mapped these drugs from their PubChem identifiers to identifiers from PharmGKB, otherwise DrugBank, otherwise KEGG, resulting in the set of seed vertices \(N^{SCAR} = N_B^{SCAR} \cup N_T^{SCAR}\) such that:

- \(|N_B^{SCAR}| = 102\) drugs (100 from PharmGKB and 2 from DrugBank). The positive drug examples are from the “very probable”, “probable”, and “possible” classes.
- \(|N_T^{SCAR}| = 290\) drugs (286 from PharmGKB and 4 from DrugBank). The negative drug examples are from the “unlikely” and “very unlikely” classes.

We implemented our approach in Python\(^3\). We used a server with 700 GB of RAM and the following parameter values \(k \in \{1, 2, 3, 4\}, t \in \{1, 2, 3\}, d = 500, u = false, l_{min} = 5, l_{max} = +\infty, m \in \{p, g, m, pg, pgm\}\). It should be noted that \(k = 4\) was only tested with \(t = 1\) because of memory issues caused by the high number of generated features.

Statistics about the features are detailed for \(k = 3, t = 3\) and \(k = 4, t = 1\) in Table 2 and discussed in the next section. We obtained the features associated with the DILI data set under \(k = 3, t = 3\) in approximately 1 hour. However, computing the features with \(k = 4, t = 1\) on the same data set required 4 days and 380 GB of RAM.

5 Results and discussion

The first two lines of Table 2 show the number of neighbors and types reachable before applying support limits. Enforcing these limits constitutes a first reduction of these numbers, thus reducing the memory and computational footprints. Here, numbers are approximately divided by 2. The mining of path features always relies on \(l_{min}\), and thus it is not possible to count the number of all possible paths and path patterns. However, we show the number of path features generated during the mining, which already illustrates the combinatorial explosion. Enforcing support constraints and removing redundant generalizations allow to reduce their number in \(F\) (here, approximately by 20). Finally, the domain-dependent filtering defined by \(m\) also radically scales down the number of features ultimately output. However, this filtering only happens as post-processing and does not alleviate the scalability issues arising during the mining of patterns.

We observe a drastic increase in the number of neighbors and path features alongside \(k\), which highlights the scalability issues of mining large knowledge graphs. Considering additional levels in ontology hierarchies by increasing \(t\) also
For manually reviewing the output features, we noticed multiple path features across the aggregated LOD sets. This illustrates one of the fundamental aspects of Linked Open Data: the combination of different data sets enables to go beyond their original purposes and coverage. However, it is clear that combining LOD sets leads to bigger knowledge graphs, exacerbating the scalability issues.

The values of parameters depend on the objectives and domain knowledge of the analyst guiding the mining process, especially for blacklists and support thresholds. However, metrics about the knowledge graph may provide guidance. Indeed, statistics about node degrees can help to find a trade-off between exploration and combinatorial explosion with parameter \(d\). Similarly, the depth of class hierarchies influences the value of \(t\). For example, general classes may not be of interest to the analyst, thus reducing \(t\). The parameter \(k\) can be set by considering the graph diameter. As it is common in mining processes, iterations may be required to find the best configuration. Regarding \(m\), it is for now hard-coded and only suitable to some biomedical applications. Inspired by ontologies that allow to interactively define mining workflows \[3\], we could adapt this parameter to other applications by proposing such an interactive definition.

When manually reviewing the output features, we noticed multiple path features across the aggregated LOD sets. This multiplies the number of path features. To illustrate, with 700 GB of RAM, we could not set \(t\) to values greater than 1 for \(k = 4\). Consequently, there is still room for improvements in terms of memory consumption, e.g., through an efficient storing of paths and path patterns. These future improvements could enable to consider the full neighborhood of seed vertices, which is reached for greater values of \(k\) and \(t\) and involves a far greater amount of vertices. For example, for the DILI data set, the full neighborhood is reached for \(k = 23\) and \(t = 21\) (for \(d = 500\)), or \(k = 19\) and \(t = 21\) (when the degree constraint is disabled). The full amount of reachable neighbors is 4 times \((d = 500)\) or 9 times greater \((d = +\infty)\) than with \(k = 4\).

The values of parameters depend on the objectives and domain knowledge of the analyst guiding the mining process, especially for blacklists and support thresholds. However, metrics about the knowledge graph may provide guidance. Indeed, statistics about node degrees can help to find a trade-off between exploration and combinatorial explosion with parameter \(d\). Similarly, the depth of class hierarchies influences the value of \(t\). For example, general classes may not be of interest to the analyst, thus reducing \(t\). The parameter \(k\) can be set by considering the graph diameter. As it is common in mining processes, iterations may be required to find the best configuration. Regarding \(m\), it is for now hard-coded and only suitable to some biomedical applications. Inspired by ontologies that allow to interactively define mining workflows \[3\], we could adapt this parameter to other applications by proposing such an interactive definition.

When manually reviewing the output features, we noticed multiple path features across the aggregated LOD sets. This is made possible by aggregating and canonicalizing multiple LOD sets in the knowledge graph. This result particularly illustrates one of the fundamental aspects of Linked Open Data: the combination of different data sets enables to go beyond their original purposes and coverage. However, it is clear that combining LOD sets leads to bigger knowledge graphs, exacerbating the scalability issues.

Regarding our approach, we only canonicalize vertices, \(i.e.,\) individuals and ontology classes. Nevertheless, predicates used on arcs can also be identified as identical, leading to a canonicalization of arcs. In this context, we could benefit from matching approaches, such as PARIS \[25\]. By identifying identical classes, predicates, and individuals, these matching approaches could further improve the canonicalization and, therefore, increase the number of common features between seed vertices from different data sets. Similarly, we could consider literals and arcs incident to literals that were purposely discarded here. However, the canonicalization of literals raises several challenging issues due to their heterogeneity in terms of syntactic variations, unit measures, and the precision of numerical values. Other reasoning mechanisms and semantics associated with Semantic Web standards could be taken into account. For example, predicates can be defined as transitive, and thus the canonical knowledge graph could also result from their transitive closure.

<table>
<thead>
<tr>
<th></th>
<th>DILI</th>
<th>SCAR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(k = 3, t = 3)</td>
<td>(k = 4, t = 1)</td>
</tr>
<tr>
<td><strong>Before (l_{min}) and (l_{max})</strong></td>
<td><strong>Neighbors</strong></td>
<td><strong>Neighbors</strong></td>
</tr>
<tr>
<td></td>
<td>175,652</td>
<td>179,694</td>
</tr>
<tr>
<td></td>
<td>13,580</td>
<td>13,677</td>
</tr>
<tr>
<td></td>
<td><strong>Types</strong></td>
<td><strong>Types</strong></td>
</tr>
<tr>
<td></td>
<td>628,681</td>
<td>639,050</td>
</tr>
<tr>
<td></td>
<td>18,940</td>
<td>18,526</td>
</tr>
<tr>
<td><strong>After (l_{min}) and (l_{max})</strong></td>
<td><strong>Neighbors</strong></td>
<td><strong>Neighbors</strong></td>
</tr>
<tr>
<td></td>
<td>71,560</td>
<td>90,690</td>
</tr>
<tr>
<td></td>
<td>281,657</td>
<td>312,029</td>
</tr>
<tr>
<td></td>
<td>7,372</td>
<td>8,800</td>
</tr>
<tr>
<td></td>
<td>12,421</td>
<td>13,177</td>
</tr>
<tr>
<td></td>
<td><strong>Path features in (F)</strong></td>
<td><strong>Path features in (F)</strong></td>
</tr>
<tr>
<td></td>
<td>790,605</td>
<td>10,169,975</td>
</tr>
<tr>
<td></td>
<td>10,169,975</td>
<td>11,465,585</td>
</tr>
<tr>
<td></td>
<td><strong>Path features gen.</strong></td>
<td><strong>Path features gen.</strong></td>
</tr>
<tr>
<td></td>
<td>20,145,635</td>
<td>29,011,996</td>
</tr>
<tr>
<td></td>
<td>251,791,519</td>
<td>255,672,772</td>
</tr>
<tr>
<td><strong>After (m)</strong></td>
<td><strong>Neighbors</strong></td>
<td><strong>Neighbors</strong></td>
</tr>
<tr>
<td></td>
<td>4,069</td>
<td>4,214</td>
</tr>
<tr>
<td></td>
<td>31,804</td>
<td>33,361</td>
</tr>
<tr>
<td></td>
<td><strong>Path features in (F)</strong></td>
<td><strong>Path features in (F)</strong></td>
</tr>
<tr>
<td></td>
<td>102,674</td>
<td>147,936</td>
</tr>
<tr>
<td></td>
<td>2,291,846</td>
<td>2,400,642</td>
</tr>
<tr>
<td><strong>Full neighborhood</strong> ((d = 500))</td>
<td><strong>Neighbors</strong></td>
<td><strong>Neighbors</strong></td>
</tr>
<tr>
<td></td>
<td>2,419,957</td>
<td>2,419,920</td>
</tr>
<tr>
<td></td>
<td>51,477</td>
<td>51,472</td>
</tr>
<tr>
<td></td>
<td><strong>Reached at (k, t)</strong></td>
<td><strong>Reached at (k, t)</strong></td>
</tr>
<tr>
<td></td>
<td>(k = 23, t = 21)</td>
<td>(k = 23, t = 21)</td>
</tr>
<tr>
<td><strong>Full neighborhood</strong> ((d = +\infty))</td>
<td><strong>Neighbors</strong></td>
<td><strong>Neighbors</strong></td>
</tr>
<tr>
<td></td>
<td>5,488,531</td>
<td>5,488,510</td>
</tr>
<tr>
<td></td>
<td>53,486</td>
<td>53,484</td>
</tr>
<tr>
<td></td>
<td><strong>Reached at (k, t)</strong></td>
<td><strong>Reached at (k, t)</strong></td>
</tr>
<tr>
<td></td>
<td>(k = 19, t = 21)</td>
<td>(k = 20, t = 21)</td>
</tr>
</tbody>
</table>

Table 2: Number of features mined for the two considered data sets for parameters \(d = 500\), \(u = false\) before and after applying \(l_{min} = 5\), \(l_{max} = +\infty\), and \(m = pgm\). Path features are always computed considering at least \(l_{min}\) to avoid combinatorial explosion. We display the number of path features generated (gen.) during the exploration and the number of path features ultimately in \(F\) after keeping the most specific and applying limit constraints. Types are not considered as features but are displayed to illustrate the possible combinatorial explosion in path patterns. Statistics about the full neighborhood are given as comparison.
Regarding the modeling of path patterns, we could generalize paths with both the hierarchy of classes and the hierarchy of predicates. In addition to keeping the most specific patterns, we could use other metrics to further reduce their redundancy (e.g., approaches relying on hierarchies [26,27] and extents of ontological classes [27]). This could also reduce the number of generated patterns, therefore improving the scalability of the mining approach. Neighbors could be enriched with the distance between them and seed vertices, which would correspond to the generalized paths of KGPTree [18]. We could also use other approaches than binary features (e.g., counting [13,14], relative counting [28]). More importantly, it remains to test our mined features within a complete classification task to measure the influence of $k$, $t$, and the three kinds of features (neighbors, paths, and path patterns).

6 Conclusion

In this preliminary study, we addressed the scalability issues associated with the task of mining neighbors, paths, and path patterns from a knowledge graph and a set of seed vertices. We proposed a method for tackling these issues, which we illustrated by mining a real-world knowledge graph. Our results highlight the importance of considering the scalability of approaches when mining features from ever-growing knowledge graphs. Our work alleviates part of the computational cost (time and memory) of mining paths and path patterns but also reveals the need for a further reduction. Such future research works could enable the modeling of more complex path patterns, for example, considering the hierarchy of predicates.

References


