Mining Generalized Graph Patterns With Pattern Joins And No Canonicity Tests

Tomas Martin[1], Petko Valtchev[1], and Abdoulaye Baniré Diallo[1]

Université du Québec à Montréal, Dept. Informatique

Abstract. Graph pattern mining (GPM) boils down to traversing the space of common sub-graphs while assessing their interestingness whereby a key concern is the non redundancy (each graph tested only once). We aim at generalized graph patterns (GGP), i.e. with hierarchies on both vertex and edge labels. While most GP and GGP miners rely on canonical encodings for graphs, those involve costly graph manipulations. Here, we propose an original approach that guarantees canonicity without explicit canonicity tests. To that end, we design an adapted encoding and a novel pattern refinement operator mixing edge-wise extensions with pattern joins. The resulting *jCan* miner largely outperforms its competitors.

1 Introduction

Graph patterns (GP) [1] are a versatile pattern format reaching over a wide range of compatible datasets, from molecular models to social networks to arbitrary linked data triple stores. Yet GP mining (GPM) is known for its high computing costs, primarily stemming from expensive duplicate pattern avoidance/detection and support computing primitives, both heavily relying to (sub-)graph isomorphism [9]. Generalized patterns [11], in turn, involve abstract items –as opposed to individual ones in plain pattern– whereby these are drawn from a taxonomy or a class hierarchy. They bring higher abstraction and intelligibility where plain patterns would drown a user into less frequent and overly detailed patterns. Generalized graph patterns (GGP) combine the flexibility of the former with the abstraction power of the latter [5].

GPM methods and generalized variants thereof rely –to varying degrees– on canonical encoding schemes [10, 8], i.e. unique representations. Each encoding is tested to ensure it corresponds to the canonical form of a graph. By introducing an order over encodings and only keeping track of canonical ones it is possible to enumerate every sub-graph only once. Yet, most approaches bump into large numbers of non-canonical encodings which are expensive to detect/identify.

From there, we can characterize existing approaches over two axes representing (1) ability to distinguish canonical encodings and/or duplicate candidates (i.e., *spreading*), (2) use of hierarchical information for pruning purposes (i.e., *flatness*). First, on one end of the spreading spectrum, methods fully rely on computationally expensive encoding schemes [15, 10] while at the other extreme, they forgo completely their usage during enumeration and opt for a particularly expensive duplicate detection at a later stage [3]. Secondly, on the flatness axis a number of methods renounce to exploit any hierarchical information [15] while

other adopt a divide-and-conquer (D&C) approach [3, 8] where topologies are captured followed by a label-refinement step. As a general trend, the higher reliance on canonical encoding the less a method exploits hierarchical information.

As trade-off, we propose to combine the two-step approach with a suitable canonical form for patterns, namely the DFS-encoding from *gSpan*. To ensure it can seamlessly work on GGPs, our pattern language is first extended to emulate `is-a` hops via edge-wise extension: A dedicated edge type is added labelled by `is-a`. Also, a proper ranking of the classes from the taxonomy ensures the encoding complies to those edges' semantics. While these two tools might –combined with the canonicity tests *à la gSpan*– seem enough for listing all canonical DFS-encodings of FGGP, such a strategy faces some serious issues. Indeed, it risks an excessive use of those tests, up to a prohibitive total cost. More dramatically, the subspace of canonical DFS-encodings of all label refinements of the generic topologies from step 1), proved disconnected w.r.t. to pure `is-a` edge-wise extensions. Spelled differently, some canonical DFS-encodings have no parent DFS-encoding that is both canonical and diverges by a single `is-a` edge.

We designed a more general DFS encoding-based approach with a *join* operation on GGP encodings exploiting specialization links between GGPs from different topology-induced subspaces of the overall pattern space and sets of automorphic graph vertices (*orbits*). The former means it breaks the D&C nature of the two-step approach as dependencies arise between subspaces. Yet, its merit is the join primitive preserves the canonicity of its arguments, i.e., the encompassing miner *jCan* (for Join Canonical) skips a large number of canonicity tests w.r.t. *Taxogram*, and two versions of *gSpan* (experimentally observed). Here we discuss vertex label specializations yet *jCan* works also on edge label ones.

In what follows, Sec. 2 presents surveys GGP miners. Then, Sec. 3 provides properties and algorithmic details. Next, Sec. 4 compares jCan to its competitors and points to some limitations. Finally, Sec. 5 concludes.

2 Related work

Our data are directed labelled (multi-)graphs, e.g. RDF or property graphs. In a graph $g = \langle V, E \rangle$, V is a set of vertices, and $E \subseteq (V \times V)$ a (multi-)set of edges. We use $\nu(g)$ and $\epsilon(g)$ to denote the sets of vertices and edges of g, respectively, while $\lambda : (V \cup E) \to H\rangle$ maps vertices and edges onto a label hierarchy H.

Canonical encodings. Let $\mathbb{G}$ the set of all graphs ($g \in \mathbb{G}$), $\cong$ denote labelled graph isomorphism, and $[\ ]_{\cong}$ an equivalence classes of isomorphic graphs in $\mathbb{G}$. The related *subgraph* isomorphism is denoted by $\dashv_{\cong}$. It induces the generality relationship $\sqsubseteq$ between patterns in our pattern space $\mathcal{L}_p$ (set of all patterns). The order induced by sub-graph isomorphism between graphs in $\mathbb{G}$ is readily extendable between $[\ ]_{\cong}$. An equivalence class-based partitioning of graph representations helps formulating the FGPM task as a traversal of each $[\ ]_{\cong}$ class exactly once, navigating between classes exploiting the $\dashv_{\cong}$ relation.

Instead of explicitly manipulating $[\ ]_{\cong}$-classes or members thereof, *encodings* can be used: They are sets of *ordered* fixed-size tuples from a universe of tu-

ple items. An order over encodings enables a (complete) enumeration thereof, smoother than directly working on graphs.

Encoding a graph g to one of its encodings $\Gamma(g)$ amounts to code each edge e with a tuple t to form $\Gamma(g) = \{t = \text{encode}(e) \mid \forall\ e \in \epsilon(g)\}$. The exact shape of edge-tuples may vary yet follows the template $\langle \pi_\Gamma(n_1), \pi_\Gamma(n_2), \ldots \in \mathbb{N},\ l_0 = \lambda(e), l_1 = \lambda(n_1), l_2 = \lambda(n_2), \ldots \in \Lambda \rangle$ where n_x represents unique identifiers for vertices, l_x represents label information, and π_Γ is a ranking for vertices (vertex *subscript* in [14]). With each encoding scheme a total order $<_T$ between tuples exists (usually by comparing position and labels) inducing π_Γ.

To introduce *canonical* encodings, let $[\ ]_\Gamma$ be equivalence classes of equivalent encodings, i.e., the sets of encodings $\Gamma(g)$, $\Gamma(g')$ s.t. $g \cong g'$. Within each $[g]_\Gamma$ or $[\Gamma(g)]_\Gamma$ class, only a unique member –canonical– represents the set of all equivalent encodings (same $[\ ]_\Gamma$), as well as all graphs within $[g]_\cong$. The criteria for an encoding's canonicity is method-specific yet usually exploits an extremum criteria (e.g. min or max within $[\ ]_\Gamma$). Using encodings an FPGM method exploits $\subseteq$ relationship between members of $[\ ]_\Gamma$-classes instead of a largely more complex $\dashv_\cong$ one between members of $[\ ]_\cong$-classes. Therefore, a non-redundant graph pattern enumeration enters every $[\ ]_\cong$ **exactly once** to determine the frequency of the pattern (in $\mathcal{L}_p$) by exploring every $[\ ]_\Gamma$ for canonical encodings.

FGPM methods. Main differences between methods lie in the design of their graph encoding scheme and their traversal strategy over the boolean lattice formed by the $\dashv_\cong$ relationship[1]. Historically, AGM [6] comes first closely followed by FSG [7] and #-Path [12]. All three exploit an *apriori*-like exploration strategy (i.e., level-wise) and propose similar graph encoding spaces by exploiting serializations of adjacency matrices.

In short, all three suffer from a high cost candidate generation strategy. Indeed, they need to maintain/generate non-canonical patterns to reach completeness, in addition to having to explicitly manipulate $[\ ]_\Gamma$ subsets to ensure no duplicates are produced. AGM distinguishes a unique canonical member of $[\ ]_\Gamma$ as the lexicographically smallest serialization of a vertex-sorted adjacency matrix or $min(\{\Gamma(g) \in [g]_\Gamma\})$. Alternatively, FSG re-uses the same canonicity criteria as AGM and also follows a join-based strategy. Yet it exploits a common immediate sub-graph (designated as *core* [7]) and join operations can output more than one candidate, including potential encodings within the same $[\ ]_\Gamma$.

Soon after, gSpan [14], FFSM [4] and MoFa [2] propose more refined encoding spaces, where all –and only– canonical encodings are covered by a unique subtree[2]. Each canonical encoding must have a unique parent which is also canonical: every prefix of a canonical $\Gamma(g)$ is also a canonical one. Conversely, any (prefix-preserving) extension of a non-canonical $\Gamma(g)$ is also not a canonical $\Gamma(g)$. From there, *anti-monotony* of the canonicity status with regards to the precedence relationship can be established [14].

[1] Aspects like support computation or internal data structures are assumed irrelevant here since orthogonal to our designs and interchangeable between methods.

[2] MoFa has two iterations, canonicity status is only introduced in the second.

Moreover, all three share an appealing property for canonical $\Gamma(g)$: building and/or verifying canonical encodings is a purely local operation, i.e., only g and $\Gamma(g)$ are required. Basically, they propose a pattern decomposition into a unique ordered spanning tree complemented by a set of cycles (implicit in FFSM). In short, each pattern is incrementally (edge-by-edge) constructed using a minimal coverage strategy: BFS for FFSM and MoFa, and DFS for *gSpan*. Edges covered by the spanning tree are denoted as *non-cyclic* while those not covered are denoted as *cyclic* edges (resp. *forward* and *backward* edges in [14]). To extend an encoding one must add a new tuple larger than any other w.r.t. $<_T$. Yet, successfully extending an encoding is not sufficient to produce a **canonical** one: $\Gamma(g')$ a canonical extension of $\Gamma(g)$ must have $\Gamma(g)$ as immediate prefix. As a result, for *gSpan* a new forward edge is appended to a path from the root vertex and the one added last (due to DFS). That path-shaped induced sub-graph is known as the *rightmost path* and we denote it as $\overrightarrow{\rho}(\Gamma(g))$.

As an example, Fig. 2 's pattern p contains two $(l, \_, q)$ triples and one $(c, \_, l)$ around a common "l"-labeled vertex. First look for the smallest triple using only label information: assuming $q < c < l$, any of the two (l, _, q) triples can form the tuple $\langle 0, 1, q, \_, l\rangle$. Next, incrementally cover p's graph looking for the smallest tuple from any edge neighbouring $\overrightarrow{\rho}(\Gamma(g))$ (requires **all** embeddings for partial cover) and update it. The final result is $\{\langle 0, 1, q, \_, l\rangle, \langle 1, 2, l, \_, q\rangle, \langle 1, 3, l, \_, c\rangle\}$.

Later, *Gaston* [9] proposes to break down the pattern search space into three: a space for paths (no vertex with degree > 2), one for trees (no paths, no cycles) and one for graphs where only patterns with cycles admitted. Most notably, its *ad hoc* enumeration strategy allows to skip expensive canonicity checks on paths and trees but remains somewhat lacking for patterns with cycles. Here, it requires an explicit canonization (i.e., build the code) and a check into a repository of already explored cyclic graph patterns.

In summary, existing FGPM methods which rely on joins need –to some degree– to maintain non-canonical encodings and/or output more than one unique candidate. Here, we re-use *gSpan*'s encoding spaces with join operations yet show that we can bypass the above limitations and offer guarantees for canonicity status.

GGPM methods. GGPM was introduced in [5] adapting *AGM* [6] to taxonomies of vertex/edge labels. While the intended refinement operator(s) is not fully specified, it is understood the canonical encoding scheme of *AGM* is reused, i.e. duplicates are removed level-wise, which is expensive.

Taxogram [3] was, arguably, the first dedicated FGGP miner, albeit with vertex-only label taxonomies. It applies a D&C strategy over re-labelled data: At step 1), it runs *gSpan* on graphs where each label is replaced by its top-level super-class. It thus discovers all FGPs (topologies). Next, in a D&C mode, each pattern is label refined by successive `is-a` hops on its vertices, up till yielding an infrequent specialization thereof. With no canonicity considerations, isomorphism checks are performed eventually to spot duplicates.

Zhang et al. [15] studied mining a limited flavor of FGGP, called *link patterns*, from RDF data provided with a domain ontology. Similar to *Taxogram*, they re-

label their data graphs and then use *gSpan* in a pure GPM mode. Yet, in a key departure from the above approach, they only use the *most specific* compatible type in the ontology to replace a resources ID in the respective graph vertex.

Two FGGP miners were studied in [10]: A first one emulates the two-step approach of *Taxogram* while, reportedly, reflecting the DFS encoding from 1) (also done by *gSpan*) in 2). However, the crucial impact of label refinement on canonicity isn't examined. The second method avoid relies on data augmentation to taxonomy traversals: Beside re-labelling a vertex with the respective top-level class, it appends to that vertex the respective taxonomy path going down to the original label node. Thus, `is-a` hops are simulated by edge-wise extensions, i.e. FGGPM is reduced to FGPM. Yet, as admitted in the paper, this increases the number of costly isomorphism tests.

To sum up, pure data-augmentation methods [15, 10] re-use GP encoding spaces (no adaptation). On the flip side, the hierarchy being lost on them, they can't use the `is-a` links to prune the highly combinatorial pattern space (due to the larger label set). Search-wise spelled, these spaces are wider yet shallower. Conversely, methods performing explicit `is-a` hops [3, 5] search through narrower yet deeper spaces. They benefit from hierarchy-based pruning, i.e. avoid nodes corresponding to `is-a` hops on infrequent patterns. Yet, such methods pay higher price for a) spotting the canonical member of $[\ ]_\Gamma$ encoding classes or b) remove duplicate FGGPs in a post-processing step. As a trade-off, we look here at label hierarchy-aware canonical encoding of patterns that reduces the canonicity test costs and thus increases scalability.

3 *jCan*: a novel FGGP miner

Toward richer and more expressive patterns. Our FGGPs capture complex and implicit dependencies in the raw data, as exemplified in Fig. 1 with an example from a dairy production dataset federating data about milk yield and composition, animal well-being, herd improvement, etc. Here the complex dynamics between life-sciences, production management and quality testing makes it hard to extract meaningful insights.

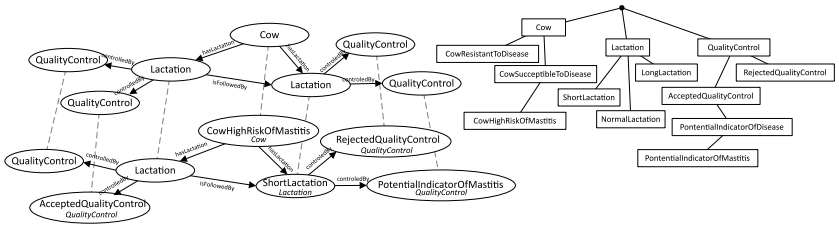


Fig. 1: Two FGGPs and a hierarchy (dairy production data).

Both patterns p and p' represent the same trend at different abstraction levels: A proportion ($\sigma = X$) of the cows with above-average risk of mastitis (udder

inflammatory disease) indicators have indeed experienced shorter and problematic lactations, yet not as their first, strongly suggesting that same disease. On the right, with provide with an excerpt of a label hierarchy H to clarify the relationship between vertex labels in the patters. Clearly, p is too generic to convey any useful information while p' might be too specific.

We adopt the two-step approach from Sec. 2: (1) mining the frequent generic topologies from data relabelled with top-level labels from H, (2) refining those `is-a` hops. At the second step, given a pattern p, a refinement operator yields the pattern set $\{p' \mid p \sqsubseteq p', h(p') = h(p) + 1\} \cap \mathcal{L}_p$ or, alternatively, the set of encoding equivalence classes $\{[g']_\Gamma \mid g' \dashv\cong p.g, |\epsilon(g')| = |\epsilon(p.g)| + 1\}$, i.e., the sets of encoding classes corresponding to all immediate specializations of p.

jCan focuses on the specialization step using a hybrid scheme of join and pattern-growth operations. It exploits the handy properties of *gSpan*'s encoding space (see Sec. 2) to propose an original DFS encoding-based specialization.

DFS-encoding of specializations. A specialization operation corresponds to a label replacement: Given a vertex, its label is replaced by one of its immediate descendants in H. Yet, with a label replacement-based operation, given the canonical enumeration mechanism in *gSpan*, a GGP encoding can hardly be a canonical predecessor of its specializations [8]. In the best case scenario, encodings for a pattern and its specialization could be siblings, i.e., differing only on the highest DFS-tuple. In a less favourable case might have no common ancestor within *gSpan*'s pattern space. Only topology extensions are admissible operations in *gSpan*. Thus, we opt for representing each downward shift in H by a dedicated DFS tuple. We call that operation *DFS-specialization*.

Definition 1. *A DFS-specialization operation, analogous to [14]'s DFS-extension, adds a (forward) edge to a DFS-encoding $\Gamma(g)$ yet is not restricted to $\overrightarrow{\ }(\Gamma(g))$.*

As an example, in Fig. 1 instead of replacing a label h (e.g., `QualityControl`) by its immediate descendant h' (e.g., `AcceptedQualityControl`) we add a new DFS-tuple $\langle x, y, h, \texttt{is-a}, h' \rangle$. From there, a *canonical* DFS-specialization corresponds to a DFS-specialization operation producing a canonical encoding.

A precedence relation between canonical encodings is necessary in order to form an exploitable spanning tree over the pattern space. For *gSpan*, $\prec_{\text{dfs}}$ between **canonical** encodings (Sec. 2) imposes (1) extending an encoding with only larger DFS-tuples, (2) ensuring that the resulting encoding $\Gamma(g)$ is minimal within its $[g]_\Gamma$, and (3) preserves its parent's encoding as prefix. To comply to the above constraints the `is-a` edge label must have the largest rank in π_λ among all edge labels. Moreover, all descendants $h'$of a given vertex label h must satisfy $\pi_\Gamma(h') > \pi_\Gamma(h)$, for the same reason[3].

Specializations through joins. We expand upon results from [14]. Prop. 1 characterizes canonicity status for **both** prefix and non-prefix extensions (we must consider all vertices, outside $\overrightarrow{\ }$ becomes allowed) of a canonical $\Gamma(g)$. We exploit automorphism orbits which have largely ignored by the FGPM literature.

[3] For a class to rank lower than subclasses we use a depth-first prefix traversal of H.

In short, automorphism orbits $\Omega(g)$ are sets of equivalent vertices w.r.t a graph g's topology and labeling. If two vertices v and v' share the same orbit, then there exists a self-isomorphism over g where v and v' are mapped to each other[4].

Property 1. Given a **canonical** DFS-encoding $\Gamma(g)$ for a graph g, its set of automorphism orbits $\Omega(g)$, extending $\Gamma(g)$ with a forward tuple $t = \langle \pi_\Gamma(v), \pi_\Gamma(v'), , , \rangle$ cannot produce a canonical encoding of a super-graph of g **if**:

1. $v \in \overrightarrow{\rho}(\Gamma(g))$ and $v, v' \in o, \pi_\Gamma(v') > \pi_\Gamma(v)$ yet $v' \in \nu(\overrightarrow{\rho}(\Gamma(g)))$;
2. $v \in \overrightarrow{\rho}(\Gamma(g))$ and at least one other member of its orbit is outside $\overrightarrow{\rho}(\Gamma(g))$;
3. $v \notin \overrightarrow{\rho}(\Gamma(g))$ and $\pi_\Gamma(v)$ is not the smallest in its orbit outside $\overrightarrow{\rho}(\Gamma(g))$.

Prop. 1 helps *a priori* prune non-canonical encodings but is not sufficient to guarantee a canonical one (tuple labels can influence canonicity status): using only vertex ranking information makes possible to skip many expensive tests.

For a graph representation g, one of its orbits $o \in \Omega(g)$ and a fixed forward DFS-tuple t, extending any vertex v within o with t, will result in a different $\Gamma(g')$ with $g \dashv\!\cong g'$ yet within the same $[g']_\Gamma$. Among those only a unique one is $[g']_\Gamma$'s canonical $\Gamma(g)$. After performing that canonical extension operation, o does not contain v anymore and by iteratively repeating the same process, one can **induce** an **order** on vertices of the same orbit where lower ranking vertices must be extended first to result in a canonical $\Gamma(g)$. Invariably, vertices outside $\overrightarrow{\rho}$ will rank lower than those within while for vertices on $\overrightarrow{\rho}$, deeper vertices will rank lower. Hence, Property 1 provides the overall enumeration scheme: get extensions outside $\overrightarrow{\rho}$ first, then in a decreasing order, explore $\overrightarrow{\rho}$-extensions[5].

Property 2. Given $\Gamma(g)$ a (canonical) DFS-encoding, $\forall x \in \nu(g)$ s. t. $x \notin \overrightarrow{\rho}(\Gamma(g))$, $\exists\ \Gamma(g') \prec_{\text{dfs}} \Gamma(g)$ and *exists* $x' \in \overrightarrow{\rho}(\Gamma(g'))$, s.t. $\pi_{\Gamma(g')}(x') = \pi_{\Gamma(g)}(x)$.

Prop. 2 spans over $\prec_{\text{dfs}}$ stating for any vertex outside $\overrightarrow{\rho}(\Gamma(g))$, there is at least one (possibly non direct) ancestor $g' \prec_{\text{dfs}} g$, for which that vertex is in $\overrightarrow{\rho}(g')$. Consequently, the $\pi_{\Gamma(g)}(x)$-th vertex in $\Gamma(g')$ can be DFS-specialized, yielding a canonical encoding. With $\Gamma(g')$ direct prefix of $\Gamma(g)$, it is immediate that any DFS-specialization on $\overrightarrow{\rho}(g')$, say g'', shares a common $|\Gamma(g)|$-subgraph with g. Let $\Gamma(g'') - \Gamma(g') = t_1$ and $\Gamma(g) - \Gamma(g') = t_2$. Simply put, $g''' = \Gamma(g) \cup \{t_1\} \cup \{t_2\}$ is a valid DFS encoding for one of g's DFS-specializations.

Property 3. Given a DFS-encoding $\Gamma(g)$ of a graph g, $\forall \Gamma(g') \prec_{\text{dfs}} \Gamma(g)$, any non-$\overrightarrow{\rho}$ extension of $\Gamma(g)$ leading to a canonical $\Gamma(g'')$ can be obtained by a join operation between $\Gamma(g)$ and a DFS-specialization descendant of $\Gamma(g')$.

Prop. 3 expands on Prop. 2 by allowing for join operations between DFS-encodings sharing an identical prefix (yet not necessary immediate). *Join* operations (denoted $\bowtie$) are well-known in frequent pattern mining [4]. Instead of extending a pattern, $\bowtie$ combines two patterns having a common parent in the pattern space: it corresponds to extending the common parent with the differences between both siblings (joined pattern is a common descendant).

[4] Also, $\Omega(g)$ constitutes a complete and disjoint partitioning of $\nu(g)$.
[5] Prop. 1 helps prune *gSpan* candidate encodings; for backward a similar one exists.

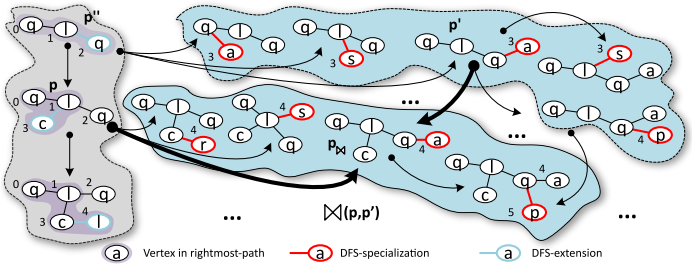


Fig. 2: Overview of *jCan* pattern space traversal.

As an example, consider pattern p in Fig. 2. It contains two (l, _, q) triples and one (c, _, l) around a common "l"-labeled vertex. First look for the smallest triple using only label information: assuming $q < c < l$, any of the two (l, _, q) triples can form the tuple $\langle 0, 1, q, \_, l\rangle$. Next, incrementally cover p's graph by looking for the smallest possible tuple from any edge neighbouring $\overrightarrow{\rho}(\Gamma(g))$. The final result is $\{\langle 0, 1, q, \_, l\rangle, \langle 1, 2, l, \_, q\rangle, \langle 1, 3, l, \_, c\rangle\}$ corresponding the vertex rankings annotating each vertex in Fig 2.

Definition 2. *For a generic topology pattern p, an immediate ancestor topology p'' and one of its specializations p', we define $p \bowtie p'$ as,*

$$\underbrace{\{t \mid t \in \Gamma(p'.g)\}}_{\textit{All DFS-tuples in } \Gamma(p'.g)} \cup \underbrace{\{shift(t_1 = max(\{t \in \Gamma(p.g)\}), |\nu(p'.g)| - |\nu(p'.g)|\})}_{\textit{Highest tuple in } \Gamma(p.g)\textit{, shifted } \pi_\Gamma(v\_) \textit{ accordingly}}.$$

Our join operation combines a canonical $\Gamma(p.g)$ from **generic topology** p and a canonical $\Gamma(p'.g)$ for a **DFS-specialization** descending from p's ancestor in $\mathcal{L}_p$, p''. Conceptually, $\bowtie$ extends the DFS-encoding a common core between $p.g$ and $p'.g$ with one DFS-tuple representing a specialization and another DFS-tuple representing a topology extension, respectively from $\Gamma(p'.g)$ and $\Gamma(p.g)$. Every $\Gamma(p_\bowtie.g)$ has two "canonical" parents: a generic topology p and a DFS-specialization p' of an immediate ancestor of p. Moreover, $p \bowtie p'$ amounts to $\Gamma(p_\bowtie.g) = \Gamma(p'.g) \cup \{t_1\}$ where t_1's vertex rankings must be increased[6] to reflect the number of out-of-$\overrightarrow{\rho}(\Gamma(p.g))$ DFS-specializations in $\Gamma(p'.g)$. Here, t_1 ranks highest thus $\Gamma(p'.g)$ is a direct prefix of $\Gamma(p_\bowtie.g)$. The reason is $p.g$'s newest vertex w.r.t $p''.g$ stems from $\overrightarrow{\rho}(\Gamma(p'.g))$, $\overrightarrow{\rho}(\Gamma(p''.g))$ and $\overrightarrow{\rho}(\Gamma(p_\bowtie.g))$.

In Fig. 2 consider p: its vertex v s.t. $\pi_\Gamma(v) = 2$ is outside $\overrightarrow{\rho}(\Gamma(p.g))$. It cannot be directly extended to produce a canonical encoding. Yet, $\Gamma(p''.g) \prec_{\text{dfs}} \Gamma(p.g)$ and $\nu(\overrightarrow{\rho}(\Gamma(p''.g)))$ contains v. Here $p_\bowtie = p \bowtie p'$.

For an example using encodings, in Fig. 3 the highest DFS-tuple in $\Gamma(p_0)$ is $\langle 3, 6, l, \_, q\rangle$ and $\Gamma(p_1)$ contains one DFS-specialization tuple $\langle 2, 3, q, \texttt{is-a}, q'\rangle$ w.r.t its generic topology. Thus, the shift function will increment vertex rankings π_Γ in $\Gamma(p_0)$'s highest tuple by one to output $\langle 4, 7, l, \_, q\rangle$. The canonical

[6] *shift* increments tuple π rankings by the number of vertices added by $\bowtie$, i.e., vertex difference between $p'.g$ (generic parent topology) and its specialization $p''.g$.

encoding resulting of the join operation is $\Gamma(p_2.g) = \{\langle 0, 1, q, \_, l\rangle, \langle 1, 2, l, \_, q\rangle, \langle 2, 3, q, \texttt{is-a}, q'\rangle, \langle 1, 4, l, \_, l\rangle, \langle 4, 5, l, \_, c\langle, \_, 5, 1, c, \_, l\rangle, \langle 4, 6, l, \_, q\rangle, \langle 4, 7, l, \_, q\rangle\}$.

It can be shown that a large part of such join operations result in canonical encodings. The result of $p \bowtie p'$ **will be canonical**, unless (a) the last tuple in $\Gamma(p.g)$ is smaller than any of those in $\Gamma(p''.g)$, or (b) there is no canonical $\Gamma(p.g)$ admissible for a join operation producing the canonical member within $[p_{\bowtie}]_\Gamma$ (extremely rare yet can happen). Case (a) is avoided by respecting label ranking constraints proposed in Sec. 3 but is not sufficient. Additionally, let $X = \pi_\Gamma(v)$ with $v \in \nu(p'.g)$ correspond to the highest DFS-specialization tuple in p': if the highest DFS-tuple in $\Gamma(p.g)$ stems from a vertex with $\pi_\Gamma(v') \geq X, v' \in \nu(p.g)$, then $\Gamma(p.g) \bowtie \Gamma(p'.g)$ will not be canonical. Note that if $\pi_\Gamma(v') \geq \pi_\Gamma(v)$ then it is also true that for any p' (immediate) descendant DFS-specialization (monotony). By preventing $\bowtie$ operations in such cases and stopping the exploration of the p''s descendant DFS-specializations, we can efficiently prune non-canonical $\Gamma(g)$.

Case (b) is trickier: To perform a join, it is necessary that $\Gamma(p.g)$ covers an induced sub-graph of $\Gamma(p_{\bowtie})$ with an order preserving π_Γ vertex mapping (using $\dashv_{\cong}$, intuitively similar to a prefix). This relates to the shift function: That mapping guarantees that the highest ranking DFS-tuple corresponds to the topology extension $\langle v = \nu(p_{\bowtie}.g) - \nu(p.g), e = \epsilon(p_{\bowtie}.g) - \epsilon(p.g)\rangle$. Fortunately, there always exists a member within $[p]_\Gamma$ which satisfies the induced sub-graph constraint yet, in some extremely rare cases it is not the canonical one. Recall that for $\Gamma(p_{\bowtie}.g)$ to be canonical, all its prefixes must also be. This is a known issue with join-based FGPM methods where non-canonical encodings have to be discovered, maintained and used in join operations to guarantee completeness [4]. It is not a suitable approach since this adds both storage and runtime overhead.

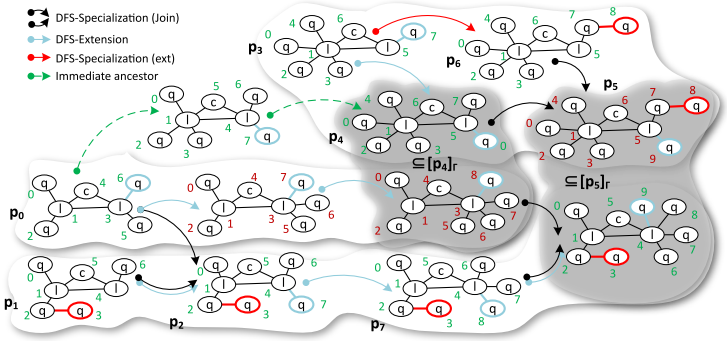


Fig. 3: Advanced examples of DFS-specializations with joins.

Instead of maintaining non-canonical encodings, we identify the case: Some patterns that represent generic topologies in $\mathcal{L}_p$ share a $p \sqsubseteq p'$ relationship yet their canonical $\Gamma(p.g) \perp \Gamma(p'.g)$. This happens when $p.g$ represents a sym-

metric graph: In many cases, any $\overrightarrow{\rho}()$-extension yields a non-canonical encoding (Prop. 1). To avoid them, one must exploit not only the $\prec_{\text{dfs}}$ relationship when performing DFS-specialization joins but also the $\sqsubseteq$ one from $\mathcal{L}_p$. In other words, join operations must be performed with the frequent/canonical DFS-specializations of all immediate parents in $\mathcal{L}_p$. As an illustration, consider the join of p_4 and p_6 in Fig. 3: It yields a non-canonical encoding for p_5. The canonical member of $[p_5]_\Gamma$, drawn just below p_5, can be produced by joining another encoding in $[p_4]_\Gamma$ (below p_4) and p_7 rather than p_4 and p_6. Yet, $\overline{\Gamma(p_4)}$ the alternative encoding for p_4 is not a canonical one. Here, $\overline{\Gamma(p_4)}$ (non-canonical member of $[p_4]_\Gamma$, hence corresponding to p_4 in $\mathcal{L}_p$) will be derived at runtime (from the difference $p_4 - p_0$) prior to computing DFS-specializations for p_4.

In summary, our *jCan* method exploits join operations in two settings: a guaranteed canonicity one corresponding to an overwhelmingly large majority of scenarios, and extremely rare cases when canonicity tests remain necessary.

4 Evaluation

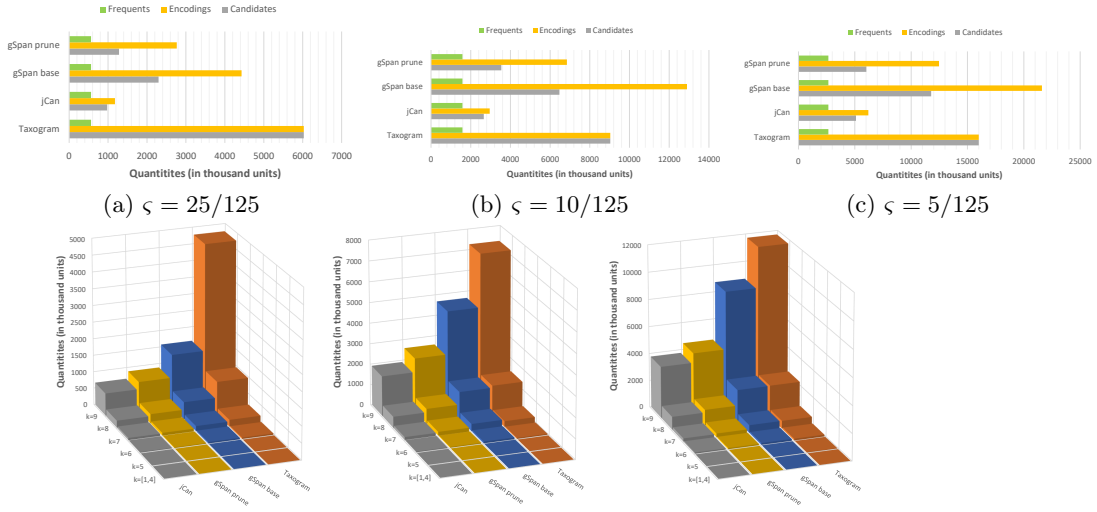
Experimental setup. All our codes were developed, compiled and executed (single-threaded) with Java 11 (HotSpot 64-Bit). Experiments were run on Windows 10 Professional x64 with an Intel i9-9900K processor and 64 GB of RAM. Rather than measuring CPU time or memory, we opt for tracking the number of candidate patterns (including potential duplicates), candidate encodings and frequent (unique) patterns. We look at each method's candidates-to-patterns gap as well as candidate patterns (or *candidates*) and candidate encodings (or *encodings*). In short, candidates inform on pruning capacities of the method in $\mathcal{L}_p$ while encodings can reveal an over-reliance on expensive canonicity tests.

For our evaluation we use a triplestore with milk control data from dairy production. Overall, it logs 10+ years of Canadian dairy production, inclusive data about 1.6M cows, 6.6k herds and around 20M quality tests. Here we use a small sample of the dairy dataset and a limited subset of the hierarchy as abstract items. We designed three different scenarios where we vary the minimum support threshold ς and the hierarchy H size. Basically, those two constitute the most impactful parameters: the number of graphs in the database is a less decisive factor than average graph size (already large enough at 2K+ vertices).

$\lvert g \in D \rvert$	$\mu(\nu(g))$	ς	k_{max}	|F|	|H|	|T|
125	477.4	25	9	561,352	189	65
125	477.4	10	9	1,584,974	506	213
125	477.4	5	9	2,656,650	636	256

Table 1: Information about sample datasets, settings and frequent patterns $|F|$.

Comparison of *jCan*, *gSpan* and *Taxogram*. On the top of Fig. 4, for each of the three scenarios, the numbers of frequent, candidates and encodings per method are drawn (horizontal bars). On the bottom, a detailed view on the numbers of candidates per method is provided, further split by depth in $\mathcal{L}_p$.



(a) $\varsigma = 25/125$ (b) $\varsigma = 10/125$ (c) $\varsigma = 5/125$

Fig. 4: Comparison between *jCan*, *Taxogram*, *gSpan* and an improved *gSpan*.

jCan bests the other three methods on both ratios. Overall, on a number of tests it scores around 30% lower than the improved *gSpan* and around half than *gSpan*. Also, its number of encodings is way lower than both *gSpan* (by a factor of 4-5) and its improved version (by a factor of 2-3). This seems to imply that our DFS-specialization scheme can substantially reduce the canonicity tests.

As a general trend, *Taxogram* underperforms: Since not $[\,]_\Gamma$-aware, it tests every generated encoding against the database. Clearly, its ratio candidates/frequents is the worse among the four. Curiously enough, it spends most of its computing effort on a small number of generic topologies whose subspaces seemingly prove hard to traverse. While not directly visible on our figures, topologies with extremely large $[\,]_{\cong}$-classes have also extremely large $[\,]_\Gamma$-classes. Intriguingly, on both lower ς scenarios, *gSpan* produces even more encodings than *Taxogram* yet it manages to prune them to a lower rate.

Both *gSpan* and our improved *gSpan* see their performance decrease when lowering ς. Indeed, while both ratios remain comparable along three settings, they get closer to *Taxogram* and less competitive with *jCan*. To the best of our knowledge, this is explained by *Taxogram*'s explicit use of H for frequency-based pruning and *gSpan*'s lack thereof.

About candidate distributions by depths per method, previously observed relative efficiency holds. Unsurprisingly, most candidates are located at maximum depth in $\mathcal{L}_p$. In practice, for candidates it is a crucial point since their frequency is to be assessed – through sub-graph isomorphism or extending graph embeddings – with exponential costs depending on k. Yet there is a clear gap between *jCan*/*gSpan* and the other two. Mainly, it comes from using more advanced candidate pruning techniques. More generally, that candidate distribution clearly indicates that each additional $\mathcal{L}_p$ level to be considered for mining adds an ex-

ponential overhead. Moreover, candidate-level pruning techniques, while critical, are not sufficient in practice. Indeed, roughly 50% of improved *gSpan*/jCan candidates turn out to be frequent which is a rather satisfactory result. But one of two encodings produced by (improved) *gSpan* turns out to be non-canonical thus wasting precious resources. Conversely, *jCan* drastically reduces its number of encodings and thus achieves a smother and more efficient traversal of $\mathcal{L}_p$.

5 Conclusion

We presented a FGGPM method splitting the task into topology and label-processing steps whose weak spot is duplicate generation. As a remedy, we adapted the widely-used DFS canonical encoding from *gSpan* to graph edges reflecting `is-a` hops and designed a n operation that combines edge-wise extensions with pattern joins while taking into account node orbits. As a result, our method *jCan* achieves non redundant pattern space traversal which ensures high efficiency. Yet, the D&C *modus operandi* of the original approach is lost.

Next, we'll examine novel canonical encodings based on orbit-aware (partial) vertex orderings such as those yielded by well-known graph invariants [13].

References

1. Aggarwal, C., et al.: Frequent Pattern Mining. Springer, 2014 edn. (Aug 2014)
2. Borgelt, C.: Canonical Forms for Frequent Graph Mining, p. 337–349. Studies in Classification, Data Analysis, and Knowledge Organization, Springer (2007)
3. Cakmak, A., Ozsoyoglu, G.: Taxonomy-superimposed graph mining. In: Proc. of the 11th intl. conf. on EDBT. pp. 217–228. ACM (2008)
4. Huan, J., et al.: Efficient mining of frequent subgraphs in the presence of isomorphism. In: Third IEEE international conference on data mining. IEEE (2003)
5. Inokuchi, A.: Mining generalized substructures from a set of labeled graphs. In: Fourth IEEE ICDM. p. 415–418. IEEE (2004)
6. Inokuchi, A., et al.: An apriori-based algorithm for mining frequent substructures from graph data. In: PKDD. pp. 13–23. Springer (2000)
7. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: IEEE ICDM (2001)
8. Martin, T., et al.: Towards Mining Generalized Patterns From RDF Data And A Domain Ontology. In: Proceedings of GEM@ECML-PKDD2021. Springer (2021)
9. Nijssen, S., Kok, J.: Frequent graph mining and its application to molecular databases. IEEE Trans. on Syst., Man and Cybernetics **5**, 4571–4577 (2004)
10. Petermann, A., et al.: Mining and ranking of generalized multi-dimensional frequent subgraphs. In: IEEE ICDIM. pp. 236–245. IEEE, Fukuoka (2017)
11. Srikant, R., Agrawal, R.: Mining generalized association rules. Future Generation Computer Systems **13**(2–3), 161–180 (1997)
12. Vanetik, N., et al.: Computing frequent graph patterns from semistructured data. In: IEEE ICDM. pp. 458–465 (2002)
13. Weisfeiler, B., Leman, A.: The reduction of a graph to canonical form and the algebra which appears therein. NTI, Series **2**(9), 12–16 (1968)
14. Yan, X., Han, J.: gSpan: Graph-based substructure pattern mining. In: IEEE ICDM. pp. 721–724 (2002)
15. Zhang, X., et al.: Mining link patterns in linked data. In: WAIM. pp. 83–94. Springer (2012)