Improving frequent subgraph mining in the presence of symmetry

Christian Desrosiers, Philippe Galinier, Alain Hertz
Ecole Polytechnique de Montreal
{christian.desrosiers,philippe.galinier,alain.hertz}@polymtl.ca

Pierre Hansen
HEC Montreal
pierre.hansen@gerad.ca

Abstract

While recent algorithms for mining the frequent subgraphs of a database are efficient in the general case, these algorithms tend to do poorly on databases that have a few or no labels. Although little attention has been given to such datasets, there are many existing applications which deal with this type of data. In this paper, we present a novel algorithm, called SyGMA, that improves frequent subgraph mining in such cases by limiting the impact of symmetry on calculations, without the use of memory-expensive structures. Through experimentation on various datasets, we show that our algorithm outperforms, in many cases, one of the leading algorithms for this task.

Keywords: *Data mining, frequent subgraphs, graph isomorphism.*

1 Introduction

Graph mining is a recent discipline which aims to extract useful knowledge from a large amount of structured data modeled as graphs. Already, this discipline plays a key role in important fields like chemoinformatics and bioinformatics, especially in the process of drug discovery. In the next decade, its importance will undoubtedly increase with the emergence of new technologies dealing with a greater amount of structured information, particularly in the Web domain. The discovery of frequent subgraphs is a fundamental task of graph mining which consists in finding statistically significant

sub-structures in a database of graphs. Different variants of this task exist, depending on the type of sub-structures we want to obtain. In this paper, we will consider the task of finding the frequent connected edge induced subgraphs of a database. This problem, known as the frequent subgraph mining problem, can be formulated as follows:

Definition 1 (Frequent subgraph mining)**.** Given a graph database $\mathcal{D}$, the support of a graph G in $\mathcal{D}$, written $sup(G, \mathcal{D})$, is the number of graphs in $\mathcal{D}$ containing G as an edge induced subgraph. Given an minimum support threshold s_{min}, the frequent subgraph mining problem consists in finding the connected graphs frequent in $\mathcal{D}$, i.e. the connected graphs G for which $sup(G, \mathcal{D}) \geq s_{min}$.

Several approaches have been proposed for this problem, which can be separated in two groups: levelwise and depth-first approaches. As their name implies, levelwise mining techniques, such as AGM developed by Inokuchi et al. [2] and FSG proposed by Kuramochi and Karypis [3], explore the graph search space level by level, where each level contains graphs that have one more vertex or edge than the previous one. The frequent graphs of the next level are found by first generating candidate graphs with pairs of graphs of the current level, and then filtering out infrequent ones. The main advantage of such techniques comes from the *a-priori* principle by which a graph is frequent only if all its subgraphs are. Since a graph is explored after its subgraphs, it is possible to eliminate infrequent graphs without having to compute their support, by testing if their immediate subgraphs are frequent. However, levelwise approaches suffer from two problems: the generation of many redundant candidate graphs, and the requirement to store the frequent graphs at each level. Depth-first mining approaches, such as gSpan proposed by Han and Yan [9], FFSM by Huan et al. [1], and Gaston by Nijssen and Kok [6], overcome these problems by exploring the graph search space depth-first. Starting with a graph containing a single frequent vertex or edge, these techniques recursively extend a graph by adding a new edge between two existing vertices, or a new vertex connected to an existing vertex. Since a graph is no more frequent than its subgraphs, there is no need to extend infrequent graphs. Infrequent graphs can thus be pruned implicitly, without the risk of pruning frequent ones. Various experimental studies, see [9] for example, have shown depth-first mining approaches to be superior, in most cases, to levelwise ones, both in terms of computation times and memory requirements.

The difficulty of the frequent subgraph mining problem arises from two tasks: enumerating all the possible subgraphs of database graphs, and cal-

culating the support of these subgraphs in the database. Since the vertices of a graph can be ordered in many ways, a graph can have a great number of topologically equivalent copies, called isomorphic graphs. To enumerate all subgraphs without redundancy, one must compute the canonical representation of a graph, which amounts to solving the graph isomorphism problem. Furthermore, testing if a graph is contained in a database graph is a well known *NP*-hard problem called subgraph isomorphism problem. In nearly all cases, support computation is the most costly operation of finding the frequent subgraphs of a database. Yet, the complexity of these tasks is somewhat reduced when the database graphs have added information in the form of vertex or edge labels. For instance, one can use labels to limit the vertices that can be paired while testing for subgraph isomorphism. However, if the database graphs are unlabeled or only have a few labels, then the complexity of these problems greatly reduces the size of manageable datasets. Thus far, little attention has been given to such datasets, and current algorithms tend to do very poorly on them. Still, there are many existing applications which deal with this type of data, mainly in the fields of computer vision, where the information is represented as 2D or 3D meshes, and communication/transportation networks, where the information is mostly topological. Moreover, mining unlabeled subgraphs could yield relations in the database that are both more general and frequent.

In this paper, we present a novel algorithm called SyGMA (Symmetry-free Graph Mining Algorithm) that improves the task of finding the frequent edge induced subgraphs of a database containing graphs that have a few or no labels. This algorithm uses various strategies that reduce the impact of symmetry, caused by the limited number of labels, on the tasks of enumerating subgraphs and computing their support in the database. Unlike most algorithms for the same task, ours does not rely on memory-expensive structures that store graph embeddings, since such a strategy is highly inefficient in these cases. To illustrate this, consider the embeddings of an unlabeled complete graph H (i.e., a graph for which all pairs of vertices are connected by an edge) of m vertices into an unlabeled complete graph G of n vertices. For $m = 6$ and $n = 12$, which are realistic values for this problem, there are as much as $\binom{n}{m}m! = 665280$ embeddings of H in G. Also, for the purpose of simplicity, we have limited our algorithm to deal only with vertex labels. Yet, the techniques presented in this paper could easily be extended to mine other types of subgraphs, such as subgraphs with edge labels, or vertex induced subgraphs.

The rest of this paper is structured as follows. In section 2, we present the details of our algorithm. In section 3, we give some experimental results

that compare, on various instances, SyGMA to one of the most popular frequent subgraph mining algorithms, gSpan. Finally, we conclude this paper with a brief summary of contributions and results.

2 The SyGMA Algorithm

2.1 Preliminary concepts

A labeled graph is a tuple $G = (V, E, L, l)$, where V is a set of vertices, $E \subset V^2$ a set of edges, L a set of labels, and $l : V \rightarrow L$ is a function that gives a unique label to each vertex of G. Given two labeled graphs $G = (V, E, L, l)$ and $G' = (V', E', L', l')$, we say that G is isomorphic to G', written $G \simeq G'$, iff there exists a bijection $\varphi : V \rightarrow V'$, called isomorphism, such that

1. $(u, v) \in E \Leftrightarrow (\varphi(u), \varphi(v)) \in E'$,
2. $\forall v \in V,\ l(v) = l\left(\varphi(v)\right)$.

An automorphism is an isomorphism from a graph to itself. Furthermore, a subgraph isomorphism from G to G' is an isomorphism from G to a subgraph of G'. If such an isomorphism exists, we say that G' contains G and write $G \subseteq G'$.

Let Γ be the set of all permutations of V, and let φ be a permutation of Γ. We write G^{φ} the graph with vertex set $V^{\varphi} = V$ and edge set $E^{\varphi} = \{(u, v) \mid \exists (x, y) \in E \text{ s.t. } u = \varphi(x) \text{ and } v = \varphi(x)\}$. The automorphism group of G is the set containing the automorphisms of G, i.e. the set $Aut(G) = \{\varphi \in \Gamma \mid G^{\varphi} = G\}$. The orbits of a vertex $v \in V$, written $Orb(v)$ is the set of vertices u such that there exists an automorphism mapping v to u, i.e., $Orb(v) = \{u \in V \mid \exists \varphi \in Aut(G) \text{ s.t. } u = \varphi(v)\}$. Similarly, the orbit of a pair of vertices (u, v), written $Orb(u, v)$ is the set of vertex pairs (x, y) such that there exists an automorphism mapping u to x and v to y, i.e., the set $Orb(u, v) = \{(x, y) \in V^2 \mid \exists \varphi \in Aut(G) \text{ s.t. } \varphi(u) = x \text{ and } \varphi(v) = y\}$. A vertex partition of G is an ordered sequence of pairwise disjoint non-empty sets called cells, the union of which is V. Let π be a vertex partition of G, we write $\pi(v)$ the unique cell containing a vertex v. Given two vertex partitions π_1 and π_2, we say that π_1 is finer than π_2 if each cell of π_1 is a subset of a cell in π_2.

Automorphisms can be used to solve the graph isomorphism problem. Thus, we can determine if two graphs G and G' are isomorphic by finding the canonical representation of these graphs and verifying if these representations are identical.

Definition 2 (Canonical representation)**.** Let G be a graph such that $|V| = n$ and A be the symmetrical adjacency matrix of G. We define a function *code* that uniquely maps G to the string produced by concatenating the elements of the upper half of A:

$$code(G) = (a_{1,2}\ a_{1,3}\ a_{2,3}\ \ldots\ a_{i,j}\ a_{i,j+1}\ \ldots\ a_{n-1,n})\,.$$

The canonical representation of G is thus the lexicographically smallest code produced by any permutation of G, i.e., $\min_{\varphi}\ code(G^{\varphi})$, and we call canonical permutation of G any permutation leading to this representation.

2.2 Subgraph Enumeration

The subgraph enumeration strategy used by SyGMA is similar to the one proposed by Kuramochi and Karypis for their algorithm vSiGraM [4], although their algorithm is not made for the frequent subgraph mining problem. Like vSiGraM, our algorithm uses a partial edge ordering that orders the edges of a graph G following the rank of their vertices in a canonical permutation of G:

Definition 3 (Canonical edge ordering)**.** Let G be a graph, φ be a canonical permutation of G and $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$ be two edges of G such that $\varphi(u_1) \leq \varphi(v_1)$ and $\varphi(u_2) \leq \varphi(v_2)$. The canonical edge ordering, defined by precedence operator $\prec_E$, is such that $e_1 \prec_E e_2$ iff $Orb(e_1) \neq Orb(e_2)$ and either one of the following is true

1. $\varphi(u_1) < \varphi(u_2)$
2. $\varphi(u_1) = \varphi(u_2)$ and $\varphi(v_1) < \varphi(v_2)$.

This ordering allows us to transform the search space into a rooted tree by mapping to each graph G a parent graph $p(G)$ produced by removing from G any non-disconnecting edge that is minimum according to $\prec_E$, as well as any vertex isolated by the edge removal. This tree is then explored depth-first, as shown in Figure 1. Starting with a graph G containing a single edge, G is recursively extended until it becomes infrequent. Let e be the last edge added to G, a canonical permutation φ of G is first found using MacKay's Nauty algorithm [5]. In the process, the vertex and vertex pair orbits of G are also obtained, with little added cost. Then, using φ, we find a minimum non-disconnecting edge e^*. If e is not topologically equivalent to e^*, i.e. if $Orb(e) \neq Orb(e^*)$, then we can prune G since another graph isomorphic to G will be explored at a different point of the traversal.

Note that this is different from the strategy used by vSiGraM, where G is pruned if $G - \{e\} \simeq G - \{e^*\}$, thus requiring one more graph isomorphism test. Otherwise, we compute the support of G and extend this graph if it is frequent. For the vertex extensions, we consider for each vertex orbit O_V of G a single vertex v, and a possible label λ. We then extend G into a graph G' obtained by adding to G a new vertex of label λ and connect this vertex to v. Similarly, for edge extensions, we consider all orbits of non-connected vertices O_E and a single vertex pair (u, v) in this orbit. We then create a graph G' by adding to G an edge connecting u and v.

Proposition 1. By traversing depth-first the rooted tree defined by function p, we can explore every graph without redundancy.

Proof. To prove that every connected graph G is explored by the traversal, we must show that there exists a path in the tree from G to the root of this tree. Since all possible vertex and edge extensions are considered in the traversal, G will be explored if its parent is explored. Furthermore, since the parent of a connected graph is also connected, by recursion, G has for ancestor the root of the tree, and is therefore reachable. Next, consider two isomorphic graphs G and G'. Since the canonical edge ordering is insensitive to vertex permutations, a minimum non-disconnecting edge in G is topologically equivalent to one in G'. Thus $p(G) \simeq p(G')$, and since we only consider one extension per vertex or vertex pair orbit, only one of G or G' will be explored. Therefore, the exploration is not redundant. □

2.2.1 Redundant graph detection

While the main lines of our subgraph enumeration strategy are similar to those used in vSiGraM, our algorithm stands out with its efficient technique to prune redundant graphs. This pruning technique uses a procedure that partitions the vertices of a graph G, as shown in Figure 2. The vertices are first ordered by increasing label values and grouped into cells of equal values, forming a partition π_0. Then, at each iteration t, the current partition π_t is refined by considering pairs of cells $V_i, V_j \in \pi$ and by splitting V_j using V_i. Denote $\delta(v, V_i)$ and $\hat{\delta}(v, V_i)$, respectively, the number of non-disconnecting and disconnecting edges incident to a vertex v and any vertex in V_i. The vertices v of V_j are ordered by decreasing values of $\delta(v, V_i)$ and after by decreasing values of $\hat{\delta}(v, V_i)$. These vertices are then split into groups of equal values, forming a subpartition π'. If π' refines V_j, i.e. if $|\pi'| > 1$, V_j is replaced by π' in the partition. This process is repeated until no further refinement is possible. Finally, the partition π_T, returned by this refinement

```
Algorithm SyGMA
  Input: A graph database 𝒟 and a support threshold s_min.
  Output: The frequent subgraphs ℱ of 𝒟.
  ℱ := ∅ ;
  foreach vertex label λ1 in 𝒟 do
      foreach vertex label λ2 in 𝒟, λ1 ≤ λ2 do
          Let G be the graph with two vertices v1 and v2 of label λ1 and λ2,
          and edge (v1, v2) ;
          ℱ := ℱ ∪ explore(𝒟, s_min, G) ;
  return ℱ ;
```

```
Procedure explore(𝒟, s_min, G)
  Input: A graph database 𝒟, a support threshold s_min and a graph G.
  Output: The frequent extensions ℱ of G.
  ℱ := ∅ ;
  Let e be the last edge added to G ;
  Compute the orbits of G and a canonical permutation φ of G ;
  Using φ, find a minimum non-disconnecting edge e* of G ;
  if Orb(e) ≠ Orb(e*) or sup(G, 𝒟) < s_min then return ℱ ;
  % Vertex extensions
  foreach vertex orbit O_V and label λ in 𝒟 do
      Let v be a vertex in O_V ;
      Let G' be the graph obtained by connecting a new vertex of label λ to v ;
      ℱ := ℱ ∪ explore(𝒟, s_min, G') ;
  % Edge extensions
  foreach non-connected vertex pair orbit O_E do
      Let (u, v) be a vertex pair in O_E ;
      Let G' be the graph obtained by connecting vertices u and v ;
      ℱ := ℱ ∪ explore(𝒟, s_min, G') ;
  return ℱ ;
```

Figure 1: Algorithm SYGMA and its recursive procedure *explore*.

```
Refinement procedure
Input: A graph G.
Output: A refined partition of V.
Let π_0 be the initial partition s.t. ∀u,v ∈ V, π(u) < π(v) iff l(u) < l(v);
t := 0 ;
repeat
    π_{t+1} := π_t ;
    t := t + 1 ;
    foreach cell V_i ∈ π_t do
        foreach cell V_j ∈ π_t s.t. |V_j| > 1 do
            Let π' be the subpartition of V_j s.t. ∀u,v ∈ V_j, π'(u) < π'(v) iff
            δ(u,V_i) > δ(v,V_i) or (δ(u,V_i) = δ(v,V_i) and δ̂(u,V_i) > δ̂(v,V_i));
            if |π'| > 1 then replace cell V_j by π' ;
until π_t = π_{t-1} or |π_t| = |V|;
return π_t ;
```

Figure 2: A procedure to find a refined vertex partition.

procedure, is used to obtain a canonical permutation of G. Thus, when looking for a canonical permutation of G, we only consider the permutation of vertices within the cells of π_T. A direct consequence of this is the following proposition:

Proposition 2. Let π_t be the partition of the vertices of a graph G, at any step t of the refinement procedure, and let φ be a canonical permutation of G. For any two vertices $u, v \in V$, if $\pi_t(u) < \pi_t(v)$ then $\varphi(u) < \varphi(v)$.

The pruning technique used by SyGMA detects non-minimum extensions while refining the vertex partition, as described in the following proposition.

Proposition 3. Let G be a graph, let π_t be the partition of the vertices of G at any step t of the refinement procedure, and consider any edge $e_1 = (u_1, v_1)$ of G, such that $\pi_t(u_1) < \pi_t(v_1)$. Edge e_1 is non-minimum in G, following $\prec_E$, if there exists a non-disconnecting edge $e_2 = (u_2, v_2)$ such that $\pi_t(u_2) \leq \pi_t(v_2)$, and if either one of the following applies

1. $\pi_t(u_2) < \pi_t(u_1)$
2. $\pi_t(u_2) = \pi_t(u_1)$ and $\pi_t(v_2) < \pi_t(v_1)$.

Proof. We prove cases (1) and (2) separately.

1. Following Proposition 2, we have that $\varphi(u_2) < \varphi(u_1)$. Moreover, following Definition 3, we have $e_2 \prec_E e_1$ and thus e_1 is not minimum.

2. (a) If there is a vertex $w \in V$ such that (u_1, w) is a non-disconnecting edge and $\pi_t(w) < \pi_t(v_1)$, then, following Proposition 2, we have that $\varphi(w) < \varphi(v_1)$. Moreover, following Definition 3, we have $(u_1, w) \prec_E e_1$ and thus e_1 is not minimum. (b) Else, assume there is no vertex $x \in V$ such that (u_2, x) is a non-disconnecting edge and such that $\pi_t(x) < \pi_t(v_2)$, otherwise use x instead of v_2 in what follows. Let V_i, $i = \pi_t(v_2)$, be the cell containing v_2, and let V_j, $j = \pi_t(u_1) = \pi_t(u_2)$, be the cell containing vertices u_1 and u_2. We have that

$$\begin{array}{ll} \delta(u_2, V_k) = \delta(u_1, V_k) = 0 & ,\text{ for } k < i \\ \delta(u_2, V_k) \geq 1 > 0 = \delta(u_1, V_k) & ,\text{ for } k = i \end{array}$$

Thus, the partition π' produced by splitting V_j with V_i will be such that $\pi'(u_2) < \pi'(u_1)$ and, following Proposition 2, we have that $\varphi(u_2) < \varphi(u_1)$. Moreover, following Definition 3, we have $e_2 \prec_E e_1$ and thus e_1 is not minimum.

□

2.2.2 Non-redundant graph detection

Like in most graph mining algorithms, the techniques used by SYGMA to detect redundant graphs help avoiding many costly isomorphism tests. These techniques, however, are of no help when dealing with graphs that are not redundant. Unlike other graph mining algorithms, SYGMA can also detect non-redundant graphs without any isomorphism test, as described in the next proposition. The proof of this proposition, related to the fact that all the vertices within two separate cells are either connected or not, can be found in [5].

Proposition 4. Let π_T be the vertex partition returned by the refinement procedure for a graph G, and let m be the number of cells of π_T that are trivial, i.e. that contain a single vertex. If G is not pruned by Proposition 3, then G is not redundant if the following conditions are satisfied:

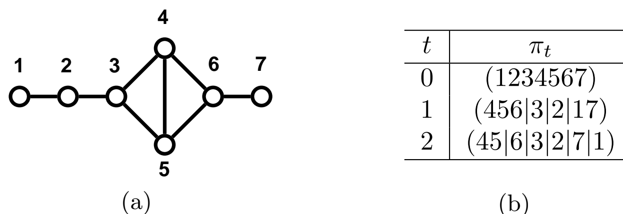
1. $|\pi_T| - m \leq 2$

2. $|V| - m \leq 5$,

i.e. π_T should have at most 2 non-trivial cells and G should have at most 5 non-trivial vertices. If these two conditions are met, then the vertex orbits of G are simply the cells of π_T. Similarly, the vertex pair orbits can also be obtained from π_T. Consider any two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ of G. Suppose, without loss of generality, that $\pi_T(u_1) \leq \pi_T(v_1)$ and $\pi_T(u_2) \leq \pi_T(v_2)$. The orbits of non-connected vertex pairs are such that $Orb(e_1) = Orb(e_2)$ iff $\pi_T(u_1) = \pi_T(u_2)$ and $\pi_T(v_1) = \pi_T(v_2)$.

Although it seems that the conditions of Proposition 4 only apply to very specific cases, the reality is that most graphs satisfy these conditions, especially labeled graphs. In fact, as we will see in the experimental section, no isomorphism test is needed for graphs of five or less vertices, regardless the number of vertex labels of these graphs.

2.2.3 An illustrative example



(a)

t	π_t
0	(1234567)
1	(456|3|2|17)
2	(45|6|3|2|7|1)

(b)

Figure 3: (a) A graph and (b) its vertex partition at step t of the refinement procedure.

In this section, we illustrate the subgraph enumeration strategy of SYGMA using a small example. Consider the graph shown in Figure 3(a), and its vertex partition at each step t of the refinement procedure, shown in (b). This graph, that we denote by G, has only one disconnecting edge: $(2, 3)$. Since G is unlabeled, the first partition π_0 groups all its vertices into a single cell. These vertices are then sorted by decreasing number of non-disconnecting edges, and the result sorted by decreasing number of disconnecting edges. Then, the vertices are grouped into cells of equal value, giving the partition $\pi_1 = (456|3|2|17)$. At step $t = 2$, cell (17) is first split using cell (456) into the subpartition (7|1). Cell (456) is then split using cell (3) into subpartition (45|6), yielding $\pi_2 = (45|6|3|2|7|1)$. Finally, this partition can not be refined any further, and the refinement procedure returns π_2. Since a canonical permutation φ only permutes the vertices within the

cells of this partition, the minimum non-disconnecting edge, i.e. the first non-disconnecting edge encountered while following φ, will necessarily be $(4, 5)$. Suppose that the last edge added to G is $(3, 4)$. At step $t = 1$, we have $\pi_1(4) = \pi_1(5) = 1 < 2 = \pi_1(3)$ and, following case (2) of Proposition 3, $(4, 5)$ is a smaller non-disconnecting edge than $(3, 4)$, in any canonical permutation. Therefore, G is redundant and can be pruned. However, if the last edge added to G is $(4, 5)$, the refinement procedure will then go on without G being pruned. In this case, the partition π_2, returned by the refinement procedure, has one non-trivial cell containing two vertices, cell (45). Thus, following Proposition 4, G is not redundant. Furthermore, the first vertices of each cell of π_2 can be used as the representants of the vertex orbits of G, i.e. the set $\{4, 6, 3, 2, 7, 1\}$. Finally, we obtain the representants of the non-connected vertex pair orbits by taking, for each pair of cells $V_i, V_j \in \pi_2$, a pair of non-connected vertices (u, v) where $u \in V_i$ and $v \in V_j$: $\{(4, 2), (4, 7), (4, 1), (6, 3), (6, 2), (6, 1), (3, 7), (3, 1), (2, 7), (7, 1)\}$.

2.3 Support calculation

As mentioned previously, the important symmetry caused by the reduced number of labels prohibits the use of complex structures to store subgraph embeddings. Instead of relying on such structures, our algorithm solves the subgraph isomorphism problem directly, using a simple subgraph matching method. However, since finding a subgraph isomorphism is a rather complex task, and since our algorithm has to complete this task quite often, we employ some further strategies to calculate the support of a subgraph as efficiently as possible.

2.3.1 Matching constraints

The first strategy is used within the subgraph matching to prune the search space. Suppose we need to determine if a graph $G = (V, E, L, l)$ is a subgraph of $G' = (V', E', L', l')$ and let γ be a possibly partial mapping of V to V', called matching. Let $v \in V$ be any vertex, we define $N(v)$ (resp. $N'(v)$) as the set of vertices adjacent to v in G (resp. G'). Moreover, we define $L(\lambda)$ (resp. $L'(\lambda)$) as the vertices of G (resp. G') which have label λ. We also define $M(\gamma)$ and $\overline{M}(\gamma)$ (resp. $M'(\gamma)$ and $\overline{M}'(\gamma)$) as the vertices of G (resp. G') matched and unmatched under γ. The following proposition gives necessary conditions for two vertices to be matched.

Proposition 5. Let $v \in V$, $v' \in V'$ be two vertices. The pair (v, v') is a candidate to extend a matching γ if the following conditions are respected:

1. $v \in \overline{M}(\gamma)$ and $v' \in \overline{M}'(\gamma)$.
2. $l(v) = l'(v')$.
3. $\forall u \in N(v) \cap M(\gamma)$, $\gamma(u) \in N'(v')$.
4. $\forall \lambda \in L,\ |N(v) \cap L(\lambda) \cap \overline{M}(\gamma)| \leq |N'(v') \cap L'(\lambda) \cap \overline{M}'(\gamma)|$.

The first two conditions are rather trivial, stating that vertices v and v' should not already be matched under γ, and that they should have the same label. The third condition imposes γ to be a subgraph isomorphism, i.e., for all vertices of G adjacent to v and matched under γ, the corresponding vertex in G' should be adjacent to v'. Finally, the last condition verifies that the matching can be extended, i.e., that for every vertex label λ, there are at least the same number of unmatched vertices of label λ adjacent to v' than adjacent to v.

2.3.2 Avoiding redundant calculations

The next strategy exploits previous calculations to limit the search of a new subgraph isomorphism, and is based on the fact that vertices are matched in a static order. Let $\gamma = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ and $\gamma' = \{(v'_1, v'_1), \ldots, (u'_n, v'_n)\}$ be two matchings such that $m \leq n$, we define a lexicographic order on matchings $\prec_\gamma$, such that $\gamma \prec_\gamma \gamma'$ iff either one of the following applies

1. $\exists k,\ 1 \leq k \leq m,\ \text{ s.t. } \begin{cases} u_i = u'_i \text{ and } v_i = v'_i, & i < k \\ u_i < u'_i \text{ or } (u_i = u'_i \text{ and } v_i < v'_i)\,, & i = k \end{cases}.$
2. $u_i = u'_i$ and $v_i = v'_i$, $1 \leq i \leq m$, and $m < n$.

Proposition 6. Let γ be the minimum subgraph matching of a graph G into a graph H according to $\prec_\gamma$, and let G' be the extension of G with edge e. Any matching γ' of G' into H is such that $\gamma \preceq_\gamma \gamma'$.

Proof. We prove this by contradiction. Suppose that $\gamma' \prec_\gamma \gamma$. If e is a vertex extension, let θ be the matching such that

$$\theta = \{(u'_1, v'_1), (u'_2, v'_2), \ldots, (u'_{n-1}, v'_{n-1})\},$$

i.e., γ' without the last pair. Otherwise, if e is an edge extension, then consider $\theta = \gamma'$. Following the definition of an isomorphism, we know that θ is also a matching of G into H. Furthermore, according to $\prec_\gamma$, we have that $\theta \preceq_\gamma \gamma' \prec_\gamma \gamma$. However this contradicts the minimality of γ and, consequently, $\gamma \preceq_\gamma \gamma'$. □

Proposition 6 is used in the following way. Let G be any subgraph visited during the exploration. We store the minimum matchings of G into all the database graphs containing G. Then, when G is extended, we only search for matchings superior or equal to the previous ones, according to $\prec_\gamma$. Let M be the maximum number of vertices of a database graph, and N be the maximum number of edges of a database graph, the total memory requirement of this strategy is in $\mathcal{O}(|\mathcal{D}|MN)$, which is much lower than the memory required to store all the embeddings of G in the database.

2.3.3 Infrequent graph detection

The last strategy allows to detect extensions leading to infrequent graphs, based on the following proposition.

Proposition 7. Let G' be the extension of a graph G with edge e, and consider any graph H such that $G \subset H$. If G' is not frequent then the extension H' of H with edge e is not frequent.

Proof. Since $G \subset H \subset H'$ and because H' contains e, we have that $G' \subset H'$. Moreover, since the support of a graph is no greater than the support of its subgraphs, we have $sup(G', \mathcal{D}) \geq sup(H', \mathcal{D})$. Thus, if G' is not frequent, neither is H'. □

When the extension of a graph G with edge e is found infrequent, we store e and all equivalent edges, i.e. the edges with the same vertex pair orbit, as invalid extensions. Then, while exploring the descendants of G in the search tree, we do not consider these invalid extensions since, by Proposition 7, they lead to infrequent graphs.

3 Experimentation

To evaluate the performance and validity of our algorithm SyGMA, we have conducted two numerical experiments. In the first one, we validate the subgraph enumeration strategy of our algorithm by generating all graphs with a limited number of vertices and labels. In the second one, we benchmark our algorithm on synthetic and real-life datasets. In both experiments, we compare the results we obtained with those obtained with one of the most popular frequent subgraph mining algorithms, gSpan, developed by Yan and Han [9]. We have selected this algorithm for two reasons. First, like our algorithm, gSpan does not use any memory-expensive structure to store the embeddings of a graph in the database. Second, a recent investigation

by Wörlein et al. [8], comparing the principal algorithms for this problem, has shown that algorithms storing embeddings offer no real advantage over gSpan for large instances. All experiments were carried out on a 2.0GHz Intel Pentium IV PC with 512Kb cache and 1Gb RAM, running Linux CentOS release 4.2.

3.1 Subgraph enumeration

In the first experiment, we consider the task of exhaustively generating a large set of graphs. More precisely, given integers N and L, we want to generate all connected graphs that have at least one edge, at most N vertices, and at most L vertex labels. This experiment serves two purposes: validating that the subgraph enumeration strategy is sound and complete, and evaluating how well this strategy deals with graph isomorphism. As reference, we compare our algorithm with the subgraph enumeration employed by gSpan. However, since the available version of gSpan does not allow to simply enumerate graphs, we had to implement our own version of gSpan, optimizing as much as possible the algorithm. For the other experiment, though, we used the original version of gSpan.

Figure 4 summarizes the result of this experiment: (a) gives the average CPU time in microseconds per non-redundant graph generated (*the Y axis has a logarithmic scale*), and (b) the average number of full isomorphism tests per non-redundant graph. Since gSpan has no strategy to detect non-redundant graphs, without carrying out an isomorphism test, its average number of full isomorphism tests per non-redundant graph is 1.0, for all values of L and N. From this figure, we make the following observations. While gSpan shows exponential scaling to the decrease of L and increase of N, our algorithm is little affected by these changes. Thus, the average CPU time per non-redundant graph found by gSpan ranges from 3.6 μsec, for $L = 5$ and $N = 5$, to 761.3 μsec, for $L = 1$ and $N = 10$, which corresponds to a 210-fold increase. By contrast, the average CPU time of our algorithm ranges from 2.4 μsec to 9.9 μsec, for the same values of L and N, which corresponds to a 3-fold increase. Furthermore, SyGMA outperforms gSpan for all values of L and N. In the most extreme case, for $L = 1$ and $N = 10$, the subgraph enumeration strategy used by SyGMA is almost more than 75 times faster than gSpan's. Finally, we can see that only a small fraction of non-redundant nodes required SyGMA to perform an isomorphism test, and that this fraction decreases as L and N increase. For cases where $N \leq 5$, no isomorphism tests were needed, regardless of the value of L.

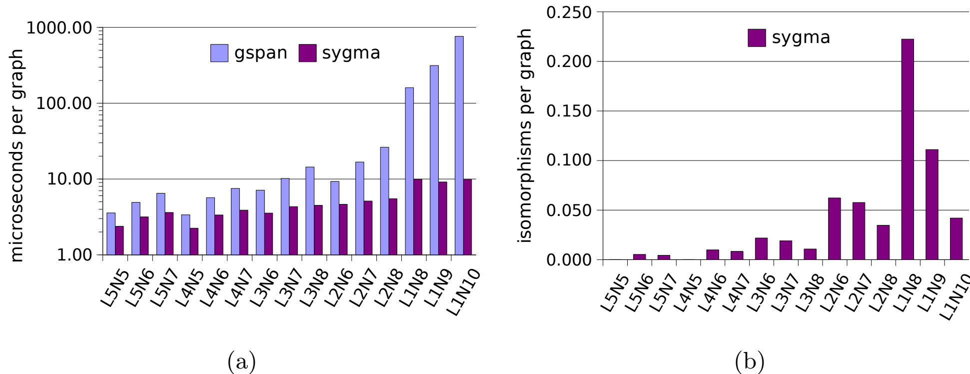


Figure 4: Subgraph enumeration: (a) average CPU time (in microseconds) per non-redundant graph generated, and (b) the number of full isomorphism tests per non-redundant graph.

3.2 Frequent subgraph mining

In the second experiment, we compare the performance of our algorithm to the latest version of gSpan, on the task of finding the frequent subgraphs of synthetic and real-life datasets.

3.2.1 Synthetic data

The synthetic datasets were generated with the random graph generator, implemented by Karypis and Kuramochi for their work in [3], using combinations of values of 5 parameters, which description and values are given in the following table:

	Description	Values
D	Nb. of graphs in the database	1000
T	Avg. size of the database graph	$\{5, 10, 15\}$
F	Avg. nb. of frequent subgraphs	25
I	Avg. size of the frequent subgraphs	15
L	Nb. of vertex labels in the database	$\{1, 2, 3\}$

Figure 5 summarizes the results. It shows, for each dataset, the CPU time in second required by gSpan and SyGMA to find the frequent subgraphs, for decreasing support thresholds (*the Y axis has a logarithmic scale*). As expected, the CPU time increases exponentially as we lower the support threshold, because of the hard subgraph isomorphism task. Furthermore, for identical values of T and support threshold, the CPU time

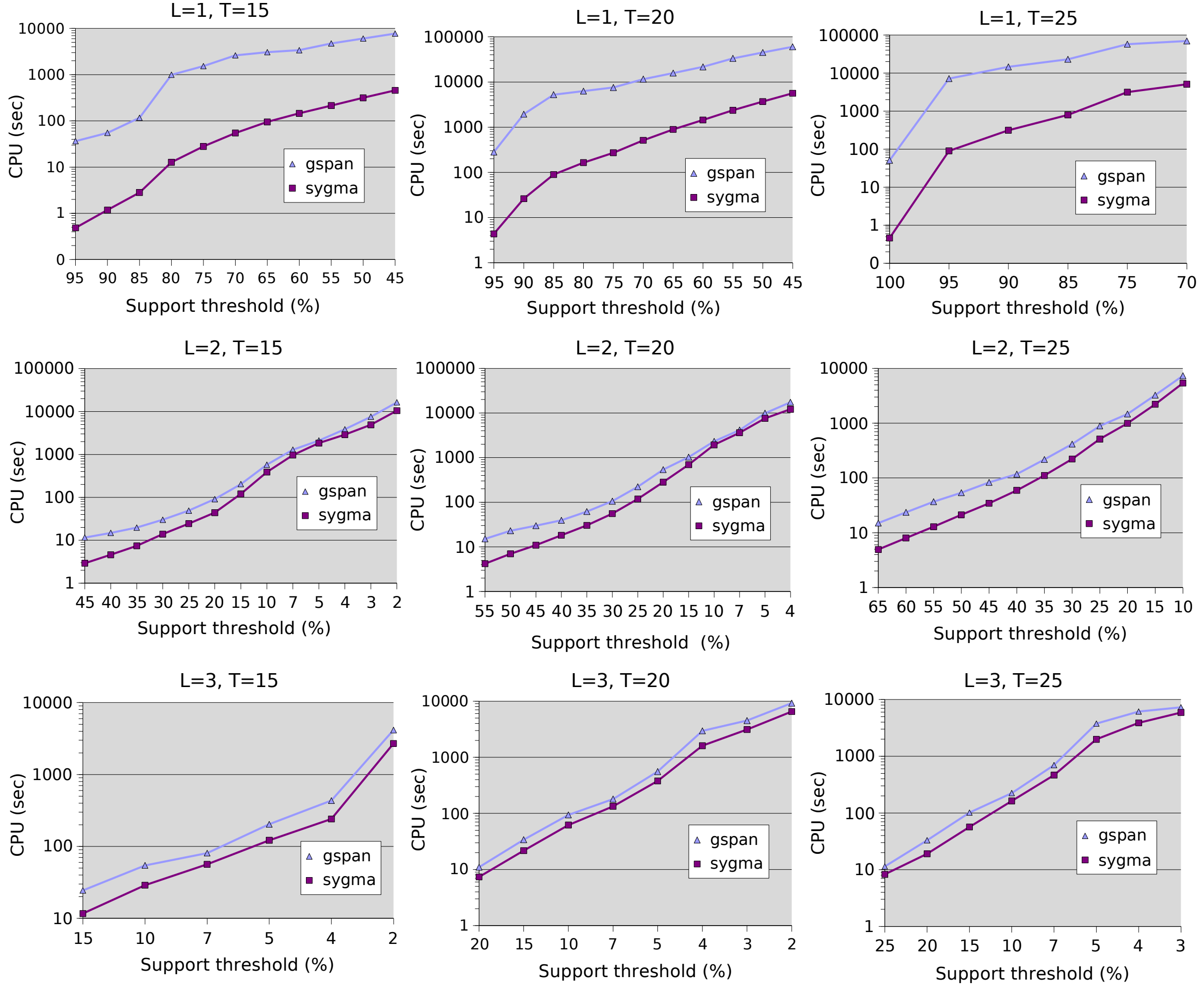
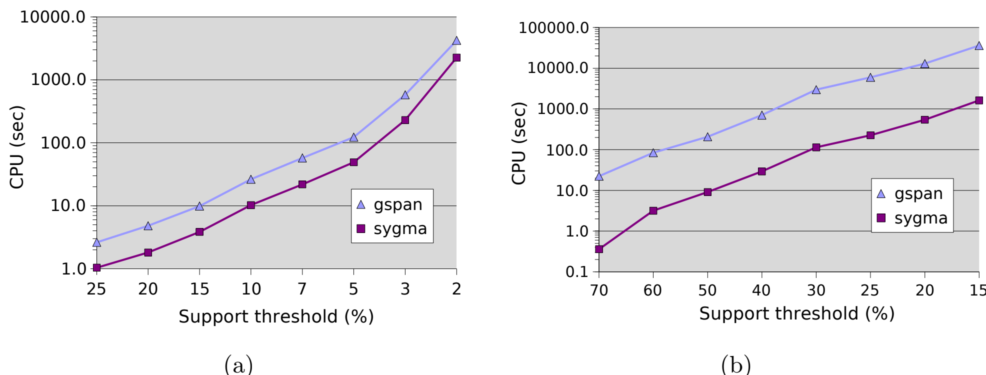


Figure 5: Runtime of gSpan and SyGMA on synthetic datasets.

increases as the number of vertex labels decreases, since there are more frequent subgraphs to discover. From these results, we see that SyGMA is faster than gSpan by up to two orders of magnitude for unlabeled graphs. In the most extreme case, i.e. when $L = 1$, $T = 25$ and the support threshold is 100%, SyGMA is 110 times faster than gSpan. Our algorithm also outperforms gSpan for datasets with 2 and 3 labels, although this improvement is not as substantial.

3.2.2 Chemical compound data

The performance of SyGMA and gSpan was also measured on a real-life dataset of chemical compounds, devised for the Predictive Toxicology Evaluation (PTE) challenge [7]. Two experiments were carried out: in the first experiment, the edge labels of the dataset were discarded, and in the second one, both vertex and edge labels were removed. The results of these experiments are presented in Figure 6. Once more, we notice that the runtimes increase exponentially as the support is reduced. Also, for identical support values, the runtime is much greater on the dataset containing no labels. Comparing both algorithms, we see that SyGMA is 2 to 3 times faster than gSpan on the dataset with labels, and about 25 times faster the gSpan on the unlabeled dataset, showing once more the advantage of SyGMA for mining unlabeled graphs.



(a) (b)

Figure 6: Runtime of gSpan and SyGMA on the PTE dataset for which (a) edge labels were discarded, and (b) both vertex and edge labels were removed.

4 Conclusion

We have presented in this paper a novel algorithm that improves mining the frequent subgraphs of a database that has a few or no labels. This improvement is achieved through original strategies that reduce the number of costly graph and subgraph isomorphism tests, without using memory-expensive structures to store embeddings. We have shown experimentally that our algorithm significantly outperforms one of the most popular algorithm for this task, gSpan, on various synthetic and real-life datasets.

References

[1] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, pages 549–552, 2003.

[2] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer-Verlag, 2000.

[3] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the First IEEE Conference on Data Mining*, pages 313–320, 2001.

[4] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.

[5] B. McKay. Practical graph isomorphism. *Congressus Numeratium*, 30:45–87, 1981.

[6] S. Nijssen and J. N. Kok. The gaston tool for frequent subgraph mining. In *Proceedings of the International Workshop on Graph-Based Tools (Grabats 2004)*, pages 281–285. Elsevier, October 2004.

[7] A. Srinivasan, R. D. King, S. H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1–6. Morgan-Kaufmann, 1997.

[8] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In *PKDD*, pages 392–403, 2005.

[9] X. Yan and H. Jiawei. gspan: Graph-based substructure pattern mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 721–724, Washington, DC, USA, 2002. IEEE Computer Society.