Improving frequent subgraph mining in the presence of symmetry

Christian Desrosiers Philippe Galinier Pierre Hansen Alain Hertz

1 Introduction

The difficulty of the frequent subgraph mining problem arises from the tasks of enumerating the subgraphs and calculating their support in the dataset. If the dataset graphs have additional information in the form of labels, these problems can be solved quite easily. However, if the dataset graphs are unlabeled or only have a few labels, then the complexity of these problems greatly reduces the number and sizes of the dataset graphs that can be managed. Thus far, researchers working on the frequent subgraph mining problem have given little attention to such datasets, and current algorithms tend to do poorly on them. Yet, there are many applications which deal with this type of data, mainly in the fields of compute vision where the data is structured as 2D or 3D meshes [8], or communication/transportation networks where the information is mostly topological.

To reduce the cost of support calculation, most frequent subgraph mining algorithms, such as FFSM [2] and GASTON [7], use complex data structures that store the embeddings of subgraphs in the dataset. Yet, when dealing with unlabeled graphs or graphs that have a few labels, which may have an exponential number of embeddings, such structures are highly inefficient. In this paper, we present some strategies that reduce the number of support calculations in a dataset of graphs having a few labels, without the use of memory-expensive structures, and allow to efficiently enumerate graphs without redundancy. Most of these strategies are improvements made to a frequent subgraph mining algorithm we have developed, called SYGMA [1]. Although our algorithm only deals with vertex labels, it is possible to transform any graph with N (labeled) vertices and L edge labels into a graph having no edge label and at most $N \log L$ vertices, as shown in [5].

2 The algorithm

The main lines of our algorithm are similar to Kuramochi and Karypis's VSIGRAM algorithm[1] [4]. Starting with a graph G containing a single frequent edge, we recursively extend G by adding a new edge, or a new vertex connected to G. To determine if G should be extended, we first obtain a vertex permutation φ for which the sequence of elements in the permuted adjacency matrix of G is minimal. Such a permutation, called *canonical labeling*, is found using McKay's program NAUTY [6]. In the process, we also obtain the orbits of vertices and vertex pairs of G. We then use φ to partially order the edges of G as follows. Denote $Orb(u, v)$ the orbit of a pair of vertices u, v, and let $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$ be two edges of G such that $\varphi(u_1) < \varphi(v_1)$ and $\varphi(u_2) < \varphi(v_2)$. The edge ordering is such that $e_1 \prec e_2$ if and only if $Orb(e_1) \neq Orb(e_2)$ and either $\varphi(u_1) < \varphi(u_2)$ or $u_1 = u_2 \ \wedge \ \varphi(v_1) < \varphi(v_2)$.

Using this ordering, we then find the minimum non-disconnecting edge e^* of G. Let e be the last extension edge of G, we prune G if e is not in the same orbit as e. This is more efficient than the technique used by VSIGRAM, which consists in testing that $G - e$ is isomorphic to $G - e^*$, and thus costing one graph isomorphism test. If G is not pruned, we then compute its support in the dataset. If G is frequent we extend it as follows. Let L be the set of labels of the graphs containing G. For each vertex orbit O_v and each label l of L, we extend G by connecting a new vertex of label l to a single vertex v of this orbit. Likewise, for every orbit of unconnected vertex pairs O_p, we connect a single pair of vertices u, v of O_p.

In order to reduce the number of permutations considered by NAUTY we use vertex invariants to partition the vertices of G, and only consider permutations of vertices within the cells of the partition. For reasons that will become clear later, we present two ways of partitioning the vertices. In the first par-

[1] Note that VSIGRAM is not for the frequent subgraph mining problem.

Refinement procedure

Input: An initial partition π_0.
Output: A refined partition π.

$t := 0$;
repeat
  $t := t + 1$;
  $\pi_t := \pi_{t-1}$;
  foreach *cell* $V_i \in \pi_t$ **do**
    foreach *cell* $V_j \in \pi_t$ *such that* $|V_j| > 1$ **do**
      Divide V_j in a partition
      $\pi_{tj} = (V_{j1}, \ldots, V_{jk})$ s.t. $\forall u, v \in V_j$,
      $\pi_{tj}(u) < \pi_{tj}(v) \Leftrightarrow \delta(u, V_i) > \delta(v, V_i)$;
until $\pi_t = \pi_{t-1}$;
return $\pi = \pi_t$;

Figure 1: The refinement procedure.

titioning, we start by dividing the vertices in two cells, the first containing the vertices that are not incident to any disconnecting edge, and the second cell the rest of the vertices. This is done so that the minimum edge according to φ will never be a disconnecting edge. We then subdivide the vertices of these two cells such that vertices with the same degree are in the same subcells, and sort these subcells by increasing degree. We then repeat the same process, this time splitting the vertices of the resulting cells by increasing label. Finally, we further refine this partition using the procedure shown in Figure 1. Denote $\delta(v, W)$ the number of vertices of $W \subseteq V$ adjacent to v, and let $\pi(v)$ be index of the cell of a partition π containing v. The refinement procedure produces a partition π such that for all $u, v \in V$, $\pi(u) = \pi(v) \Leftrightarrow \forall V_i \in \pi,\ \delta(u, V_i) = \delta(v, V_i)$. The second way of partitioning the vertices is identical to the first, except that, in the second step, we partition the vertices by decreasing degree.

3 Improvement strategies

3.1 Redundant graph detection

The first technique, called *pre-extension pruning*, prunes some extensions that cannot be minimum in the resulting graph. To illustrate this technique, suppose that we want to extend a graph G by connecting two existing vertices u, v, and suppose that we partition the vertices by increasing degree. Let W contain the vertices of G which are minimum according to the canonical labeling. If $|W| = 1$ then (u, v) can be minimum in the extended graph only if either $u \in W$ or $v \in W$, otherwise any non-disconnecting edge touching the vertex of W would be lesser than (u, v) in the extended graph, and thus, (u, v) would not be minimal. For the same reason, if $|W| = 2$, then (u, v) can only be minimum if $W = \{u, v\}$. Finally, if $|W| > 2$, then (u, v) cannot be minimum. Using the same idea, we can develop some low-cost tests to prune extensions in the case where vertices are partitioned by decreasing degree.

The second detection technique, called *post-extension pruning*, detects non-minimum extensions while refining the vertex partition. Let G be the graph produced by extension $e_1 = (u_1, v_1)$ and π_t be the partition produced at any step t of the refinement procedure. Without loss of generality, suppose that $\pi(u_1) \leq \pi(v_1)$. We can prune extension e_1 if there exists an edge $e_2 = (u_2, v_2)$ (suppose that $\pi_t(u_2) \leq \pi_t(v_2)$) such that $\pi_t(u_2) < \pi_t(u_1)$, or such that $\pi_t(u_2) = \pi_t(u_1)$ and $\pi_t(v_2) < \pi_t(v_1)$.

3.2 Non-redundant graph detection

Although some frequent subgraph mining algorithms have efficient techniques to detect redundant graphs, none of them have techniques that detect non-redundant graphs without having to compute their canonical representation. For instance, a graph G is canonical in gSpan if a pre-order traversal following the numbering of its vertices yields a lexicographically minimum code. If G is not canonical, then gSpan stops as soon as it finds another pre-order traversal yielding a lesser code. However, if G is canonical, then gSpan needs, in the worst case, to try an exponential number of traversals.

Let G be a graph that was not pruned by the *pre* or *post* extension pruning, and let π be the partition returned by the refinement procedure that has m cells containing more than one vertex. G is non-redundant if $m = 0$, or if $m = 1 \ \wedge \ |\pi| > |V| - 5$, or if $m = 2 \ \wedge \ |\pi| > |V| - 4$ (see [6] for details).

Although it seems that the above conditions only apply to very specific cases, in reality, most graphs actually satisfy these conditions, as we will see in the experimental section. This is especially true for labeled vertices, since these labels greatly reduce the symmetry of the graph.

3.3 Dynamic coding

Consider an infrequent graph G. The edges of G can be divided in two sets E^+ and E^-, such that E^+ contains the edges which, when removed from G, produce

a frequent graphs, and E^- contains the other edges. If E^+ is empty, then G will never be explored. However, if neither these sets are empty, then G may or may not be explored, depending on which of these two sets contain the minimum edge. Although we do not know beforehand which subgraphs are frequent and which are not, we do know which ones are more likely to be. For instance, paths are very likely to be frequent, since every graph of diameter d has at least one path of $d+1$ vertices. Likewise, trees are more likely to be frequent than more complex graphs. Thus, by favouring edges that are likely to produce, when removed, infrequent subgraphs, we reduce the odds of having to compute the support of G.

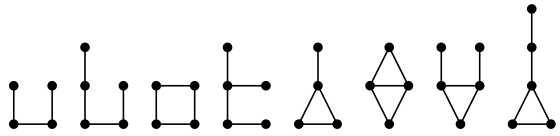


Figure 2: Graphs (b)-(e) are extensions of graph (a) and graphs (f)-(h) are extensions of graph (e).

The two ways of partitioning the vertices, by increasing and decreasing degree, lead to different topologies of the search space. To illustrate this, consider the unlabeled graphs of Figure 2. If we sort the vertices by increasing order, graph (a) will have graphs (b) and (c) as extensions. However, if we sort the vertices by decreasing degree, we then we also have graphs (d) and (e) as valid extension. The situation is reversed for graphs that are not paths or trees. For instance, if we sort the vertices by increasing degree, then graph (e) has graphs (f)-(h) as valid extensions, while the same graph has no valid extension if we sort by decreasing degree.

We use this idea as follows. If a graph G is a path or a tree, i.e., if $|E| = |V| - 1$, we then sort the vertices of G by increasing degree, otherwise we sort these vertices by decreasing degree. This compromise, which limits the number of extensions of path and trees early in the search, and later limits the extensions of small compact graphs, has shown to work well in most cases.

3.4 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 lexicographic order. Let G be the current subgraph. We store the lexicographically minimum matchings of G into all the dataset graphs containing G. Then, when G is extended, we only search for matchings superior or equal to the previous ones, according to the lexicographic order.

3.5 Infrequent graph detection

The last strategy allows to detect extensions leading to infrequent graphs, based on the following idea. Let H_1 be the extension of a graph G_1 with edge e. If H_1 is not frequent then the extension of a graph G_2, such that $G_1 \subset G_2$, with edge e is not frequent. When the extension of a graph G with edge e is infrequent, we store edge e and all equivalent edges, i.e., edges with the same vertex pair orbit, as invalid extensions. Then, when extending with edge e a graph H, itself an extension of G, if e was previously found to be invalid, it is skipped. This strategy allows us to avoid many costly subgraph isomorphism tests.

4 Experimentation

The experiments presented in this section were carried out on a 2.0GHz Intel Pentium IV PC with 512Kb cache and 1Gb RAM, running Linux Cent OS release 4.2. For these experiments, we chose to compare our algorithm with gSpan [10] for two reasons. Firstly, it is the only efficient algorithm that does not store embeddings, which makes it suitable for graphs having few or no labels. Secondly, an investigation carried out by Wörlein et al. [9] showed that, for large problem instances, algorithms storing embeddings offered no significant advantage over gSpan.

In the first experiment, we evaluate the performance and validity of our enumeration strategy by generating all connected graphs that have at least one edge, at most N vertices, and at most L vertex labels. 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 experiments, though, we used the original version of gSpan.

The results of this experiment are presented in Figure 3. (a) gives the average number of generated graphs per non-redundant graph, and evaluates the efficiency of the *pre* and *post* pruning techniques to detect redundant graphs. Since gSpan employs no special techniques for this task, we simply put the number of graphs it generated. We can see that our algorithm is rather insensitive to the number of vertices and labels, while gSpan shows a clear increase

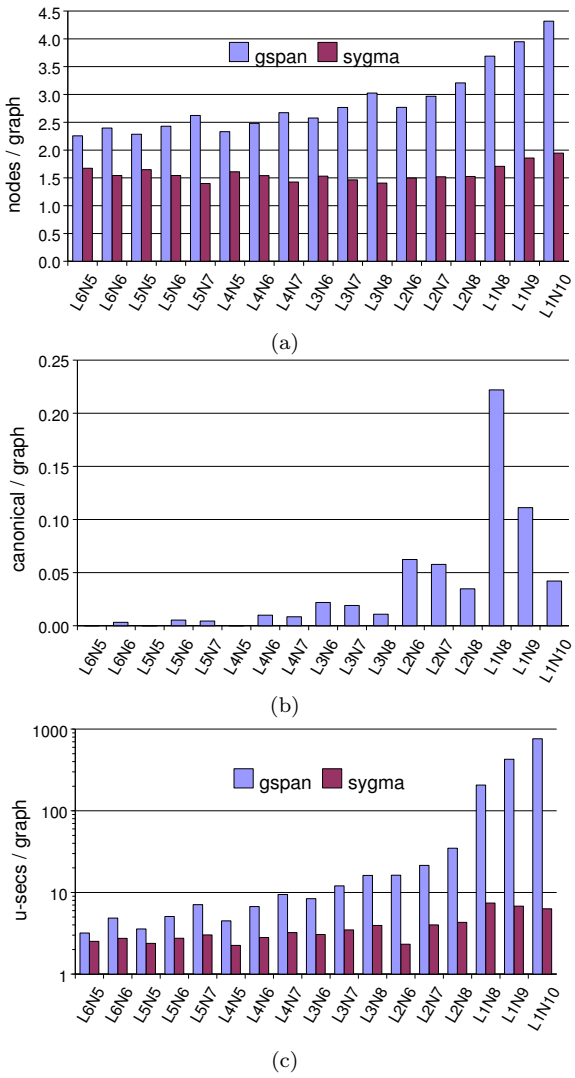


Figure 3: Subgraph enumeration results.

when the number of labels are reduced and the number of vertices are increased. Furthermore, (b) gives the average number of canonical labeling computations per non-redundant graph, and evaluates the efficiency of our techniques to detect redundant graphs. We observe that a very small proportion of graphs actually required the computation of a canonical labeling. Finally, (c) gives the average CPU time (in microseconds) per non-redundant graph. Again, we can see that our enumeration strategy is little affected by the number of vertices and labels, while gSpan shows an exponential increase as the number of labels is reduced, and the number of vertices increased. In the most extreme case, i.e., for $N = 10$ and $L = 1$, our algorithm could enumerate all graphs 120 times faster then gSpan.

In the second experiment, we compare our algorithm to the latest version of gSpan, on the task of finding the frequent subgraphs of a dataset. Since we found no available benchmark dataset having a reduced number of labels, we decided to generate synthetic datasets using the data generator implemented by Karypis and Kuramochi for their work in [3]. We generated 9 datasets using combinations of values of 5 parameters, which description and used values are given in the following table:

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

Figure 4 summarizes the results of this experiment. It shows, for each dataset, the CPU time (in seconds) required by gSpan and SyGMA to find the frequent subgraphs, for decreasing support thresholds. As expected, the CPU time increases exponentially as we lower the support thresholds and increase T, due to the hard subgraph isomorphism problem. We can see from these results that SyGMA is faster then gSpan by up to two orders of magnitude for the unlabeled case, regardless of the value of T. Furthermore, our algorithm also outperforms gSpan for the case where graphs have 2 and 3 labels, although the improvement is not as substantial.

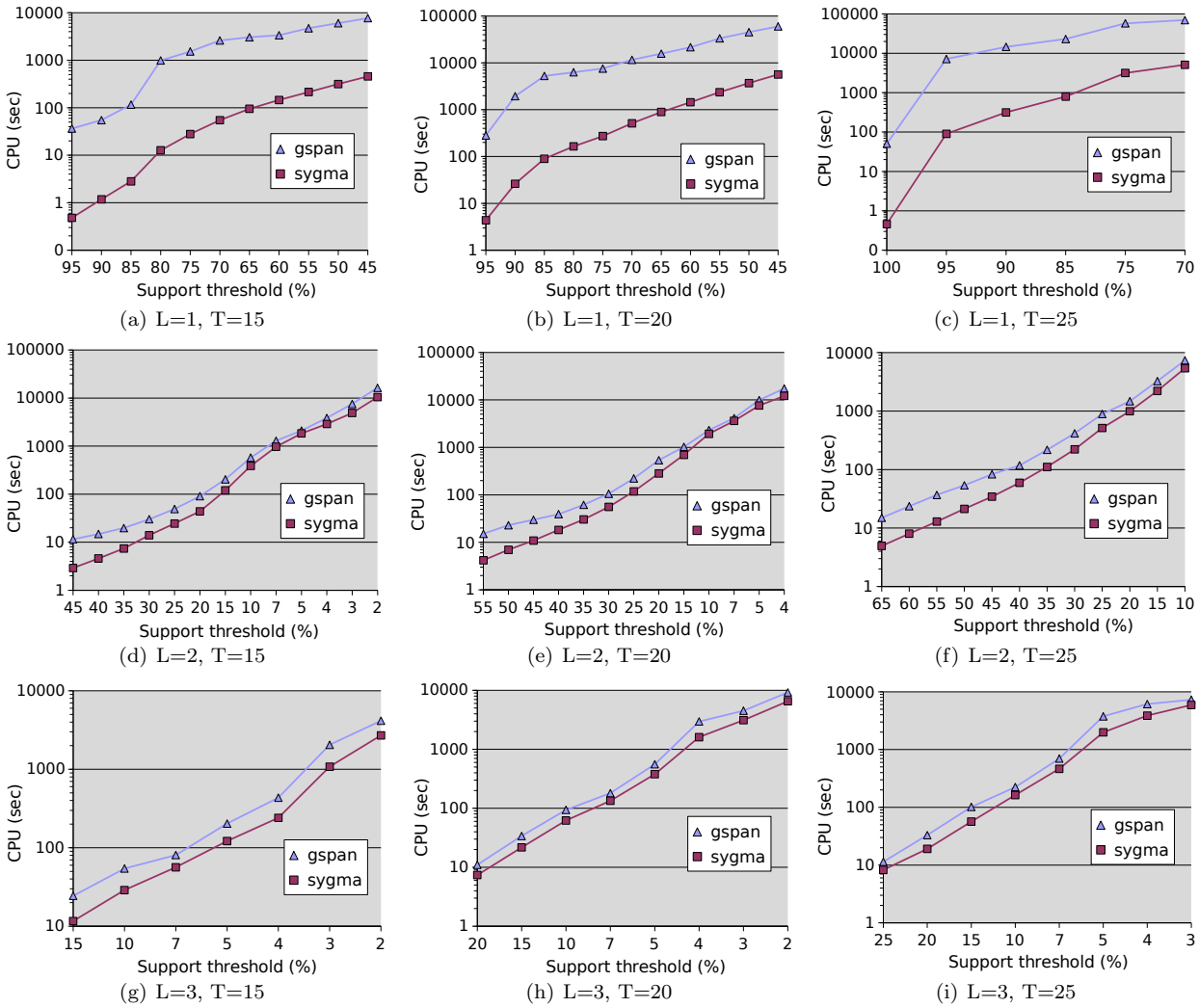
5 Conclusion

We have presented strategies to improve the task of finding the frequent subgraphs in a dataset with a few labels. These strategies reduce the number of costly graph and subgraph isomorphism tests, without using memory-expensive structures to store embeddings. Finally, we have shown experimentally that our algorithm significantly outperforms an algorithm for the same task, gSpan, on synthetic datasets.

References

[1] C. Desrosiers, P. Galinier, P. Hansen, and A. Hertz. Sygma: Reducing symmerty in graph mining. Technical Report G-2007-12, Les cahiers du GERAD, 2007.

[2] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism.



(a) L=1, T=15 (b) L=1, T=20 (c) L=1, T=25

(d) L=2, T=15 (e) L=2, T=20 (f) L=2, T=25

(g) L=3, T=15 (h) L=3, T=20 (i) L=3, T=25

Figure 4: Runtimes of gSpan and SyGMA on synthetic datasets.

In *Proc. of the 3rd IEEE Int. Conf. on Data Mining (ICDM)*, pages 549–552, 2003.

[3] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. First IEEE Conf. 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. nauty user's guide (version 2.2), http://cs.anu.edu.au/bdm/nauty/nug.pdf. Technical report.

[6] B. McKay. Practical graph isomorphism. *Congr. Num.*, 30:45–87, 1981.

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

[8] C. Schellewald and C. Schnörr. Probabilistic subgraph matching based on convex relaxation. In *EMMCVPR*, pages 171–186, 2005.

[9] 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.

[10] X. Yan and H. Jiawei. gspan: Graph-based substructure pattern mining. In *Proc. 2002 IEEE Int. Conf. on Data Mining (ICDM'02)*, pages 721–724. IEEE Computer Society, 2002.