Graph Mining: Repository vs. Canonical Form

Christian Borgelt and Mathias Fiedler

European Center for Soft Computing
c/ Gonzalo Gutiérrez Quirós s/n, 33600 Mieres, Spain
christian.borgelt@softcomputing.es, mail@mathias-fiedler.info

Summary. In frequent subgraph mining one tries to find all subgraphs that occur with a user-specified minimum frequency in a given graph database. The basic approach is to grow subgraphs, adding an edge and maybe a node in each step, to count the number of database graphs containing them, and to eliminate infrequent subgraphs. The predominant method to avoid redundant search (the same subgraph can be grown in several ways) is to define a canonical form that uniquely identifies a graph up to automorphisms. The obvious alternative, a repository of processed subgraphs, has received fairly little attention yet. However, if the repository is laid out as a hash table with a carefully designed hash function, this approach is competitive with canonical form pruning. In experiments we conducted, the repository-based approach could sometimes outperform canonical form pruning by 15%.

1 Introduction

Frequent subgraph mining consists in the task to find all subgraphs that occur with a user-specified minimum frequency in a given database of (attributed) graphs. Since this problem appears in applications in biochemistry, web mining, and program flow analysis, it has attracted a lot of attention, and several algorithms were proposed to tackle it. Some of them rely on principles from inductive logic programming and describe graphs by logical expressions (Finn *et al.* 1998). However, the vast majority transfers techniques developed originally for frequent item set mining. Examples include MolFea (Kramer *et al.* 2001), FSG (Kuramochi and Karypis 2001), MoSS/MoFa (Borgelt and Berthold 2002), gSpan (Yan and Han 2002), Closegraph (Yan and Han 2003), FFSM (Huan *et al.* 2003), and Gaston (Nijssen and Kok 2004). A related, but slightly different approach is used in Subdue (Cook and Holder 2000).

The basic idea of these approaches is to grow subgraphs into the graphs of the database, adding an edge and maybe a node (if it is not already in the subgraph) in each step, to count the number of graphs containing each grown subgraph, and to eliminate infrequent subgraphs. All found frequent subgraphs are reported (or often only the subset of so-called *closed* subgraphs).

While in frequent item set mining it is trivial to ensure that each item set is checked only once, it is a core problem in frequent subgraph mining how to avoid redundant search. The reason is that the same subgraph can be grown in several ways, namely by adding the same nodes and edges in different orders. Although multiple tests of the same subgraph do not invalidate the result of a subgraph mining algorithm, they can be devastating for its execution time.

One of the most elegant ways to avoid redundant search is to define a canonical description of a (sub)graph. Combined with a specific way of growing the subgraphs, such a canonical description can be used to check whether a given subgraph has been considered in the search before. For example, Borgelt (2006) studied a family of such canonical forms, which comprises the special forms used in gSpan (Yan and Han 2002) and Closegraph (Yan and Han 2003) as well as the one underlying MoSS/MoFa (Borgelt and Berthold 2002).

However, canonical form pruning is not the only way to avoid redundant search. A simpler and much more straightforward approach is a repository of already processed subgraphs, against which each grown subgraph is checked. Nevertheless this approach is rarely used, has actually not even been properly investigated yet. To our knowledge only two existing algorithms use a repository, namely MoSS/MoFa, which prunes with a canonical form by default, but offers the optional use of a repository, and Gaston (Nijssen and Kok 2004), in which a repository is used in the final phase for general graphs, since Gaston's canonical form is restricted to trees. In order to close this gap, this paper examines repository-based pruning and compares it to canonical form pruning. Surprisingly enough, a repository-based approach is highly competitive and could sometimes outperform canonical form pruning by 15%.

2 Canonical Form Pruning

The core idea underlying a canonical form is to construct a code word that uniquely identifies a graph up to automorphisms. The characters of this code word describe the connection structure of the graph. If the graph is attributed (labeled), they also comprise information about edge and node attributes. While it is straightforward to capture the attribute information, it is less obvious how to describe the connection structure. For this, the nodes of the graph must be numbered (more generally: endowed with unique labels), because we need to specify the source and the destination node of an edge. Unfortunately, different ways of numbering the nodes of a graph yield different code words, because they lead to different descriptions of an edge (simply because the indices of source and destination node differ). In addition, the edges can be listed in different orders. Different possible solutions to these two problems give rise to different canonical forms (see Borgelt (2006) for details).

However, given a (systematic) way of numbering the nodes of a graph and a sorting criterion for the edges, a canonical description is derived as follows: each numbering of the nodes yields a code word, which is the concatenation of the sorted edge descriptions. The resulting code words are sorted lexicographically. The lexicographically smallest code word is the canonical description. (It should be noted that the graph can be reconstructed from this code word.)

Canonical code words are used in the search as follows: the process of growing subgraphs is associated with a way of building code words for them. Most naturally, the code word of a subgraph is obtained by simply concatenating the descriptions of its edges in the order in which they are added in the search. Since each possible subgraph needs to be checked only once, we may choose to process it only in the node of the search tree, in which its code word (as constructed by the search) is the canonical code word. Otherwise the subgraph (and thus the search tree rooted at it) is pruned.

It follows that we cannot use just any possible canonical form. If extended code words are built by appending the next edge description to the code word of the current subgraph, then the canonical form must have the so-called *prefix property*: any prefix of a canonical code word must be a canonical code word itself. Since we plan to extend only graphs in canonical form, the prefix property is needed to ensure that all possible subgraphs can be reached in the search. A simple way to ensure that a canonical form has the prefix property is to confine oneself to spanning tree numberings of the nodes of a graph.

In a straightforward algorithm (the code words of) all possible extensions of a subgraph are created and checked for canonical form. Extensions in canonical form are processed further, the rest is discarded. However, canonical forms also give rise to restrictions of the extensions of a subgraph, because for certain extensions one can see immediately that they lead to a non-minimal code word. For the two most important canonical forms, namely those that are based on a breadth-first (MoSS/Mofa) and a depth-first spanning tree numbering (gSpan/Closegraph), these are (for details see Borgelt (2006)):

- *maximum source extensions*
  Only nodes having an index no less than the maximum source of an edge may be extended (the source of an edge is the node with the smaller index).
- *rightmost path extensions*
  Only the nodes on the rightmost path of the spanning tree used for numbering the nodes may be extended (children of a node are sorted by index).

While reasons of space prevent us from reviewing details, restricted extensions are important to mention here. The reason is that they can be exploited for the repository approach as well, because they are an inexpensive way of avoiding most of the redundancy imminent in the search. (Note, however, that they cannot rule out all redundancy, as there are no perfect "simple rules".)

3 Repository of Processed Subgraphs

A repository of processed subgraphs is the most straightforward way of avoiding redundant search. Every encountered frequent subgraph is stored in a data structure, which allows us to check quickly whether a given subgraph is contained in it or not. Whenever a new subgraph is created, this data structure is accessed and if it contains the subgraph, we know that it has already been processed and thus can be discarded. Only subgraphs that are not contained in the repository are extended and, of course, inserted into the repository.

There are two main issues one has to address when designing such a data structure. In the first place, we have to make sure that each subgraph is stored using a minimal amount of memory, because the number of processed subgraphs is usually huge. (This consideration may be one of the main reasons why a subgraph repository is so rarely used.) Secondly, we have to make the containment test as fast as possible, since it will be carried out frequently.

In order to achieve the first objective, we exploit that we only want to store graphs that appear in at least one graph of the database (which usually resides in memory anyway). Therefore we can store a subgraph by listing the edges of one embedding (that is, one occurrence of the subgraph in a graph of the database). Note that it suffices to list the edges, since the search is usually restricted to connected subgraphs and thus the edges also identify all nodes.[1]

It is pleasing to observe that this way of storing a subgraph can also make it easier to check whether a given subgraph is equivalent to it (isomorphism test). The rationale is to fix an order of the database graphs and to create the embeddings of all subgraphs in this order. Then we do not store an arbitrary embedding, but one into the first database graph it is contained in. For a new subgraph, for which we want to know whether it is in the repository, we can then check whether the first database graph containing it coincides with the one underlying the stored embedding. If it does not, we already know that the subgraphs (the new one and the stored one to which it is compared) cannot be equivalent, since equivalent subgraphs have the same embeddings.

However, if the database graphs coincide, we carry out the actual isomorphism test by also relying on the embeddings. We mark the embedding that is stored in the repository (that is, its edges) in the containing database graph. Then we traverse all embeddings of the new subgraph into the same graph[2] and check whether for any of them all edges are marked. If such an embedding exists, the two subgraphs (the new one and the stored one) must be equivalent, otherwise they differ. Obviously, this isomorphism test is linear in the number of edges and thus very efficient. It should be kept in mind, though, that it can be costly if a subgraph possesses a large number of embeddings into the same graph, because in the worst case (that is, if the two subgraphs are not isomorphic) all of these embeddings have to be checked. However, our experiments showed that this is an unlikely case, since especially larger subgraphs most of the time possess only a single embedding per database graph.

[1] The only exception are subgraphs consisting of a single node. Fortunately, such subgraphs need not be stored, since they cannot be created in more than one way, thus making it unnecessary to check whether they have been processed before.

[2] This is straightforward in our implementation, since in order to facilitate and accelerate forming extensions, we keep a list of all embeddings of a subgraph.

Even though an isomorphism test of the described form is fairly efficient, one should try to avoid it. Apart from the obvious checks whether the number of nodes and edges, the support in the graph database and the number of embeddings coincide (naturally these must all be equal for isomorphic subgraphs), we employ a hash function that is computed from local graph properties. The basic idea is to combine the node and edge attributes and the node degrees, hoping that this allows us to distinguish non-isomorphic subgraphs. In particular, we combine for each edge the edge attribute and the attribute and degree of the two incident nodes into a number. For each node we compute a number from the node attribute, the node degree, the attributes of its incident edges and the attributes of the other nodes these edges are incident to. These numbers (one for each node and one for each edge) are then combined with the total numbers of nodes and edges to yield a hash code.[3]

The computed hash code is used in the standard way to build a hash table, thus making it possible to restrict the isomorphism test to (a subset of) the subgraphs in one hash bin (a subset, because some collisions can be resolved by comparing the support etc., see above). By carefully tuning the parameters of the hash function we tried to minimize the number of collisions.

4 Comparison

Considering how canonical form pruning and repository-based pruning work, we can make the following observations, which already give hints w.r.t. their relative performance (and which we use to explain our experimental findings):

Canonical form pruning has the advantage that we only have to carry out one test (for canonical form) in order to determine whether a subgraph needs to be processed or not (even though this test can be expensive). It has the disadvantage that it is most costly for the subgraphs that are in canonical form (and thus have to be processed), because for these subgraphs all possibilities to construct a code word have to be tried. For non-canonical code words the test usually terminates earlier, since it can often construct fairly quickly a prefix that is smaller than the code word of the subgraph to test.

Repository-based pruning has the advantage that it often allows to decide very quickly that a subgraph has not been processed yet (for example, if a hash bin is empty). Together with comparing the numbers of nodes and edges, the support etc., this suggests that a repository-based approach is fastest for subgraphs that actually have to be processed. Only if these simple tests fail (as for equivalent subgraphs), we have to carry out the isomorphism test.

As a consequence, we expect repository-based pruning to perform well if the number of subgraphs to be processed is large compared to the number of subgraphs to be discarded (as the repository is usually faster for the former).

[3] A technical remark: we do not only combine these numbers by summing them and computing their bitwise exclusive or, but also apply bitwise shifts of varying width in order to cover the full range of values of (32 bit) integer numbers.

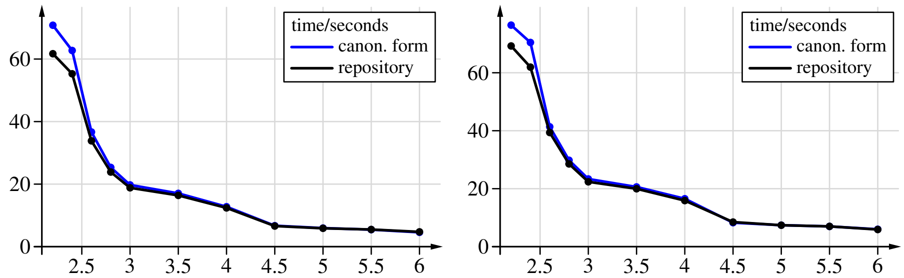

Fig. 1. Experimental results on the IC93 data set, search time vs. minimum support in percent. Left: maximum source extensions, right: rightmost path extensions.

5 Experiments

In order to test our repository-based pruning experimentally, we implemented it as part of the MoSS program[4], which is written in Java. As a test dataset (to which we confine ourselves here due to limits of space) we used a subset of the *Index Chemicus* from 1993. The results we obtained with different restricted extensions (maximum source and rightmost path, see Section 2) are shown in Figures 1 to 3. The horizontal axis shows the minimal support in percent.

Figure 1 shows the execution times in seconds. The upper graph refers to canonical form pruning, the lower to repository-based pruning. The times do not differ much, but diverge for lower support values, reaching 15% advantage for the repository-based approach together with maximum source extensions.

Figure 2 shows the numbers of subgraphs considered in the search and provides a basis for explanations of the observed behavior. The graphs refer (from top to bottom) to the number of generated subgraphs, the number checked for duplicates, the number of processed subgraphs, and the number of (discarded) duplicates (difference between the two preceding curves).

Note that about half of the work is done by minimum support pruning (which discards all subgraphs that do not appear in the user-specified minimum number of database graphs), as it is responsible for the difference between the two top curves. The subgraphs discarded in this way may be unique or not—we need not care, since they do not qualify anyway.

Canonical form or repository-based pruning only serve the purpose to get rid of the subgraphs between the two middle curves. That the gap between them is fairly small compared to their vertical location indicates the high quality of restricted extensions: most redundancy is already removed by them and only fairly few redundant subgraphs still need to be detected. (Note that the gap is smaller for maximum source extensions, which is the main reason for the usually lower execution times achieved by this approach).

[4] MoSS is available for download under the Gnu Lesser (Library) General Public License at http://www.borgelt.net/moss.html.

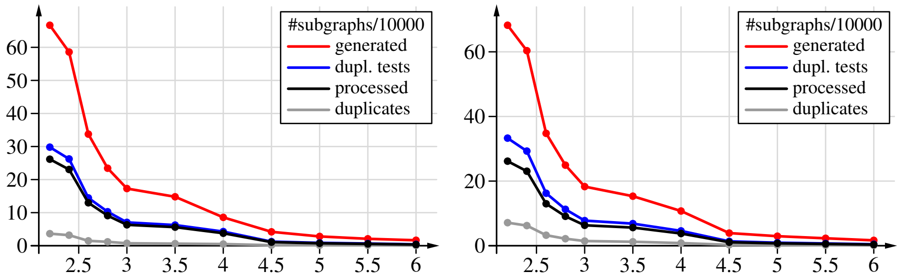

Fig. 2. Experimental results on the IC93 data set, numbers of subgraphs used in the search. Left: maximum source extensions, right: rightmost path extensions.

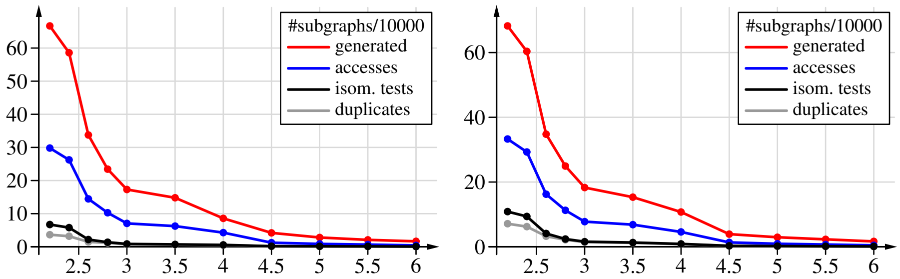

Fig. 3. Experimental results on the IC93 data set, performance of repository-based pruning. Left: maximum source extensions, right: rightmost path extensions.

Figure 3 finally shows the performance of repository-based pruning (mainly the effectiveness of the hash function). All curves are the same as in the preceding figure, with the exception of the third curve from the top, which shows the number of isomorphism tests. Subgraphs in the gap between this curve and the one above it have to be processed and are identified as such without any isomorphism test. Only subgraphs in the (small) gap between this curve and the bottom curve (the number of actual duplicates) have to be identified and discarded with the help of isomorphism tests.

Note that for a perfect hash function (which maps only equivalent subgraphs to the same value) the two bottom curves would coincide. Note also that a canonical form can be seen as a perfect hash function (with a range of values that does not fit into an integer), since it uniquely identifies a graph.

6 Summary

In this paper we investigated the widely neglected possibility to avoid redundant search in frequent subgraph mining with a repository of already encountered subgraphs. Even though it may be less elegant than the more popular approach of canonical forms and, of course, requires additional memory for storing the subgraphs, it should not be dismissed too easily. If the repository is designed carefully, namely as a hash table with a hash function computed from local graph properties, it is highly competitive with a canonical form approach. In our experiments we observed execution times that were up to 15% lower for the repository-based approach than for canonical form pruning, while the additional memory requirements were bearable.

References

BORGELT, C., and BERTHOLD, M.R. (2002): Mining Molecular Fragments: Finding Relevant Substructures of Molecules. *Proc. IEEE Int. Conf. on Data Mining (ICDM 2002, Maebashi, Japan)*, 51–58. IEEE Press, Piscataway, NJ, USA

BORGELT, C., MEINL, T., and BERTHOLD, M.R. (2005): MoSS: A Program for Molecular Substructure Mining. *Workshop Open Source Data Mining Software (OSDM'05, Chicago, IL)*, 6–15. ACM Press, New York, NY, USA

BORGELT, C. (2006): Canonical Forms for Frequent Graph Mining. *Proc. 30th Ann. Conf. of the German Classification Society (GfKl 2006, Berlin, Germany).* Springer-Verlag, Heidelberg, Germany

COOK, D.J., and HOLDER, L.B. (2000) Graph-Based Data Mining. *IEEE Trans. on Intelligent Systems* 15(2):32–41. IEEE Press, Piscataway, NJ, USA

FINN, P.W., MUGGLETON, S., PAGE, D., and SRINIVASAN, A. (1998): Pharmacore Discovery Using the Inductive Logic Programming System PROGOL. *Machine Learning*, 30(2-3):241–270. Kluwer, Amsterdam, Netherlands

HUAN, J., WANG, W., and PRINS, J. (2003): Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. *Proc. 3rd IEEE Int. Conf. on Data Mining (ICDM 2003, Melbourne, FL)*, 549–552. IEEE Press, Piscataway, NJ, USA

INDEX CHEMICUS — Subset from 1993. Institute of Scientific Information, Inc. (ISI). Thomson Scientific, Philadelphia, PA, USA 1993
http://www.thomsonscientific.com/products/indexchemicus/

KRAMER, S., DE RAEDT, L., and HELMA, C. (2001): Molecular Feature Mining in HIV Data. *Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2001, San Francisco, CA)*, 136–143. ACM Press, New York, NY, USA

KURAMOCHI, M., and KARYPIS, G. (2001): Frequent Subgraph Discovery. *Proc. 1st IEEE Int. Conf. on Data Mining (ICDM 2001, San Jose, CA)*, 313–320. IEEE Press, Piscataway, NJ, USA

NIJSSEN, S., and KOK, J.N. (2004): A Quickstart in Frequent Structure Mining Can Make a Difference. *Proc. 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD2004, Seattle, WA)*, 647–652. ACM Press, New York, NY, USA

YAN, X., and HAN, J. (2002): gSpan: Graph-Based Substructure Pattern Mining. *Proc. 2nd IEEE Int. Conf. on Data Mining (ICDM 2003, Maebashi, Japan)*, 721–724. IEEE Press, Piscataway, NJ, USA

YAN, X., and HAN, J. (2003): Closegraph: Mining Closed Frequent Graph Patterns. *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2003, Washington, DC)*, 286–295. ACM Press, New York, NY, USA