Full Perfect Extension Pruning for Frequent Graph Mining

Christian Borgelt
European Center for Soft Computing
c/ Gonzalo Gutiérrez Quirós s/n,
33600 Mieres, Spain
christian.borgelt@softcomputing.es

Thorsten Meinl
Dept. of Computer and Information Science
University of Konstanz,
Box M712, 78457 Konstanz, Germany
meinl@inf.uni-konstanz.de

Abstract

Mining graph databases for frequent subgraphs has recently developed into an area of intensive research. Its main goals are to reduce the execution time of the existing basic algorithms and to enhance their capability to find meaningful graph fragments. Here we present a method to achieve the former, namely an improvement of what we called "perfect extension pruning" in an earlier paper [2]. With it the number of generated fragments and visited search tree nodes can be reduced, thus accelerating the search.

1. Introduction

In recent years the problem how to find common subgraphs in a database of (attributed) graphs, that is, subgraphs that appear with a user-specified minimum frequency, has gained intense and still growing attention. For this task—which has useful applications in, for example, biochemistry, web mining, and program flow analysis—several algorithms have been proposed. Some of them rely on principles from inductive logic programming and describe the graph structure by logical expressions [6]. However, the vast majority transfers techniques developed originally for frequent item set mining. Examples include MolFea [10], FSG [11], MoSS/MoFa [1], gSpan [15], Closegraph [16], FFSM [8], and Gaston [13]. A related, but slightly different approach is used in Subdue [4].

The basic idea of these approaches is to grow subgraphs into the graphs of the database, adding an edge and maybe a node in each step, counting the number of graphs containing each grown subgraph, and eliminating infrequent subgraphs. Unfortunately, with this method the same subgraph can be constructed in several ways, adding its nodes and edges in different orders. The predominant method to avoid the ensuing redundant search is to define a canonical form of a graph that uniquely identifies it up to automorphisms: together with a specific way of growing the subgraphs it enables us to determine whether a given subgraph can be pruned from the search tree (see, for example, [3] for a family of such canonical forms and details of the procedure).

To further improve the algorithms one may restrict the search to so-called *closed graph fragments* (Section 2), which capture all information about frequent subgraphs, but lead to considerably smaller output (in terms of the number of reported fragments). This restriction also enables us to employ additional pruning techniques, one of which is *perfect extension pruning*, as we called it in [2], or *equivalent occurrence pruning*, as it is called in [16]. Unfortunately, neither of these approaches, in the form in which they were described in these papers, works correctly, as they can miss certain fragments. This flaw we fix in this paper (Section 3).

In addition, the approach in [2] avoided redundant search with the help of a repository of found fragments instead of using the more efficient canonical form pruning. As a consequence, perfect extension pruning was easier to perform, since it was not necessary to pay attention to the canonical form. With canonical form pruning, part of perfect extension pruning is easy to achieve, namely pruning the search tree branches to the *right* of the perfect extension (Section 4). This was first shown in Closegraph [16]. In this paper we show how one may also prune the search tree branches to the *left* of the perfect extension by introducing a (strictly limited) code word reorganization (Section 5). We demonstrate the usefulness of the enhanced approach with experiments on molecular data sets (Section 6).

2. Mining Closed Graph Fragments

The notion of a *closed fragment* is derived from the corresponding notion of a *closed item set*, which is defined as an item set no superset of which has the same support, i.e., is contained in the same number of transactions. Analogously, a *closed fragment* is a fragment no superstructure of which has the same support, i.e., is contained in the same number

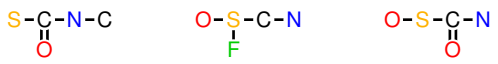

Figure 1. Three simple example molecules.

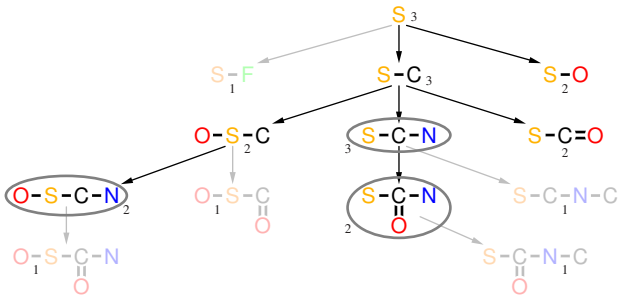

Figure 2. Search tree for the three molecules in Figure 1; infrequent fragments (contained in only one molecule) are drawn in grey/light colors, closed fragments are encircled.

of graphs in the given database. As an example consider the three molecules (no chemical meaning attached) shown in Figure 1 as the given database of attributed graphs. A corresponding search tree (starting from sulfur as a seed and with fragments being extended only if they appear in at least two molecules) is shown in Figure 2. The numbers below or to the left/right of the fragments state their support, i.e., the number of molecules a fragment is contained in. Infrequent fragments (i.e. with a support less than two molecules) are drawn in grey/light colors. The encircled fragments are closed and thus constitute the output of the search (for a minimum support of two molecules).

As for item sets, restricting the search for molecular fragments to closed fragments does not lose any information: all frequent fragments (drawn in black/dark color in Figure 2) can be constructed from the closed ones by simply forming all substructures of closed fragments that are not closed fragments themselves and assigning to them as their support the maximum of the support values of those closed fragments of which they are substructures. Consequently, restricting the search to closed fragments is a very convenient and lossless way to reduce the size of the output.

3. Perfect Extensions

Perfect extension pruning is based on the observation that sometimes there is a fairly large common fragment in *all* currently considered molecules (that is, in all molecules considered in a certain branch of the search tree). From the definition of a closed fragment it is clear that in such a situation, if the current fragment is only a part of the common substructure, then any extension that does not grow the current fragment towards the maximal common one can be postponed until this maximal common fragment has been reached. That is, as long as the search has not grown a fragment to this maximal common one, it is not necessary to branch in the search tree. The reason is, obviously, that the maximal common fragment is part of all closed fragments that can be found in the currently considered set of molecules. Consequently, it suffices to follow one path in the search tree that leads to this maximal common fragment and to start branching only from there.

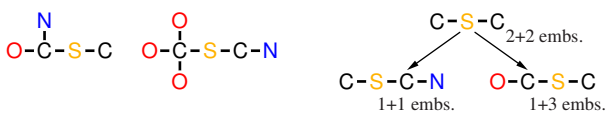

Figure 3. Example of an imperfect extension.

As an example consider again the set of molecules shown in Figure 1. If the search is seeded with a single sulfur atom, considering extensions by a single bond starting at the sulfur atom and leading to an oxygen atom can be postponed until the structure `S-C-N` common to all molecules has been grown (provided that the extensions of this maximal common fragment are not restricted in any way—see below).

Technically, the search tree pruning is based on the notion of a *perfect extension*. An extension of a fragment, consisting of an edge and possibly a node (if the edge does not close a ring), is called *perfect* if all of its embeddings can be extended in exactly the same way by this edge and node. (Note that there may be several ways of extending an embedding by this edge and node. Then all embeddings must be extendable in the same number of ways.) If there is a perfect extension, all closed (super-)fragments can, in principle, be found by searching only the corresponding branch.

However, when identifying perfect extensions, one has to be careful. In the first place, it does not suffice to check whether the number of embeddings of the extended fragment is equal to or a multiple of the number of embeddings of the base fragment (as one may think at first sight). This is only a necessary, but not a sufficient condition, as the example shown in Figure 3 demonstrates. Even though the total number of embeddings in the right branch is the same as for the root, the extension is not perfect. The left branch is not perfect, because the number of extended embeddings, even though the same for each parent embedding, is reduced from the number of extensions of its parents. Such a reduction, which also occurs in the right branch for the left molecule, indicates that some symmetry has been destroyed by the extension, which therefore cannot be perfect. As a consequence, a test for perfect extension actually has to count the number of embeddings per database graph.

A second problem (which was overlooked in both [16]

Figure 4. Rings/cycles can cause problems.

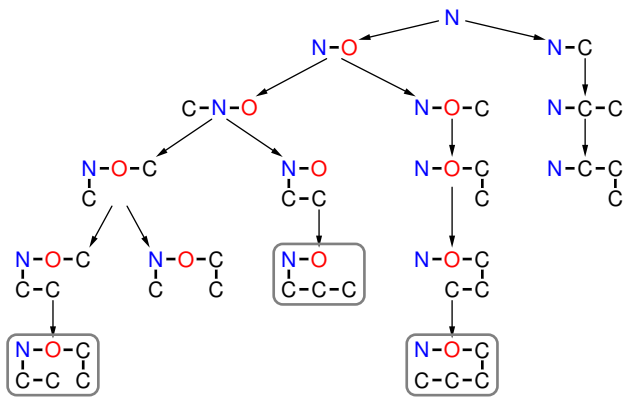

Figure 5. Search tree for the two molecules in Figure 4; closed fragments are encircled.

and [2]) is the behavior of rings (cycles) in the search, as we demonstrate with the example molecules shown in Figure 4. A search tree for these molecules (with only such fragments that are contained in both molecules) is shown in Figure 5. Here almost all extensions are perfect in the sense that they can be made in the same way in all molecules. However, the problem becomes clear when one considers adding a bond from the nitrogen atom to a carbon atom. This rules out certain ways of reaching the carbon atom via the oxygen atom and the rest of the ring. Hence the bond leading to the carbon atom is only "locally" perfect, but not "globally", that is, when the ring structure is taken into account.

In order to cope with this problem we require that perfect extensions must be bridges (that is, the extension edge must be a bridge in all embeddings of the extended fragment). This is surely a safe (i.e., sufficient) condition as it rules out any possibility that the destination of the perfect extension edge can be reached in any other way, and thus fixes the flaw mentioned above. However, it is not necessary. As a closer inspection easily reveals, extensions closing a ring (that is, leading to a node that is already in the base fragment) are also safe and thus can be allowed as candidates for perfect extensions. Hence we can slightly relax the constraints.

Note that these relaxed constraints are still only sufficient, but not necessary. There are other situations in which an edge is a perfect extension even though it does not meet the above conditions, for example, if an edge leads to a new node and is part of rings of the same size and composition in all supporting graphs. Checking necessary and sufficient conditions is, however, complicated and costly.

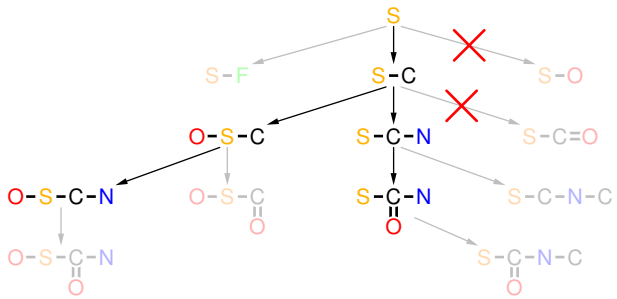

Figure 6. Search tree for the three molecules in Figure 1 with partial perfect extension pruning (crossed out branches are pruned).

4. Partial Perfect Extension Pruning

If one wants to combine perfect extension pruning with canonical form pruning, another problem has to be solved: growing the maximal common fragment can interfere with the extension restrictions resulting from canonical form pruning (see [3] for details, and note that this was no problem in [2] due to the use of a repository of found fragments to avoid redundant search). For example, with a breadth-first search based canonical form (as in MoSS/MoFa) one must not extend a fragment by edges starting from nodes with smaller numbers than the one extended in the preceding step. However, perfect extensions should not lead to such a restriction, because otherwise search results are lost.

As an example consider again the search tree shown in Figure 2. If we simply confined the search to the subtree rooted at the fragment `S-C-N`, we would lose the fragment `O-S-C-N` in the leftmost branch. The reason is that the extension of `S-C` to `S-C-N`, due to canonical form restricted extensions, prevents an extension of the sulfur atom in this subtree, because an atom with a higher number, namely the carbon atom, has been extended in the preceding step.

Fortunately, this only affects search tree branches to the *left* of the perfect extension branch, since these extensions are ruled out by the perfect extension. All extensions corresponding to branches to the *right* of the perfect extension are still possible for the fragment reached by the perfect extension. Therefore branches to the right can be pruned immediately without any loss. This type of pruning we call *partial perfect extension pruning* (because it prunes only part of the branches aside from the perfect extension one). Note that Closegraph [16] uses only this form of pruning.

How partial perfect extension pruning changes the search tree for the molecules in Figure 1 is shown in Figure 6. Note that only non-closed fragments are removed from the search tree (compare to Figure 2, in which the closed fragments are highlighted). The gains consist in the fact that the two pruned fragments need not be processed.

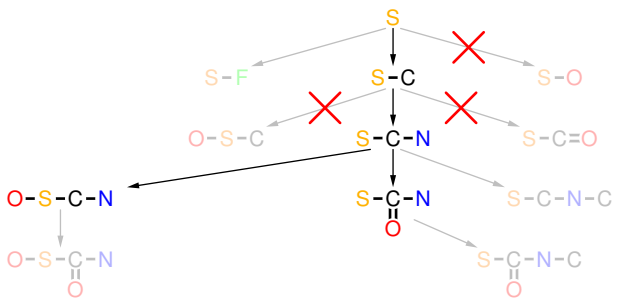

Figure 7. Search tree for the three molecules in Figure 1 with full perfect extension pruning (crossed out branches are pruned).

5. Full Perfect Extension Pruning

Although partial perfect extension pruning is already highly effective, it is desirable to prune also the search tree branches to the *left* of the perfect extension, thus completing partial perfect extension pruning into *full perfect extension pruning*. In order to do so, we must not restrict the extensions of the fragment that resulted from a perfect extension as it would be required by canonical form pruning. Otherwise we would lose fragments, as we demonstrated above. In other words, we would like to have a search tree like the one shown in Figure 7 for the molecules in Figure 1.

The core problem with this is how we can avoid that the fragment `O-S-C-N` is pruned as non-canonic. The breadth-first search canonical code word for this fragment is[1]

`S 0-C1 0-O2 1-N3.`

However, with the search tree in Figure 7 it is assigned

`S 0-C1 1-N2 0-O3,`

because this reflects the order in which the bonds have been added. Since this code word is not canonical, the fragment would be pruned and neither extended nor reported.

In order to avoid this, we allow for a (strictly limited) reorganization of code words as they result from a search tree, which takes care of the fact that perfect extension edges may have been added earlier than required by the canonical form. We split the code word into two parts: The first, fixed part consists of the (possibly empty) prefix up to and including the last edge that was added by a non-perfect extension or a perfect extension with no search tree branches to the left of it (after minimum support pruning). The second, volatile part consists of the remaining suffix of the code word, which is made up only of perfect extensions edges, which had search tree branches to the left. Instead of always appending it at the end of a code word, the description of a new edge may now be inserted anywhere in or even before the volatile part, but not in the fixed part. We may imagine this as first appending the new edge description and then shifting it to the left, as long as this makes the code word lexicographically smaller, but not entering the fixed part.

Note, however, that "shifting" an edge in the code word can make it necessary to renumber the nodes. For example, if in the fragment `O-S-C-N` the bond added last in the search (that is, the bond from the sulphur to the oxygen atom) is shifted left past the perfect extension bond (that is, the bond from the carbon to the nitrogen atom), the oxygen and the nitrogen atom get new indices.

Technically, we achieve this renumbering as follows: instead of actually shifting the extension edge from right to left, we rebuild the code word from left to right. First we traverse the fixed part, numbering all nodes in the order in which they are met. Then we continue with the volatile part until at least one of the two nodes incident to the new edge is numbered. (Note that this may already be the case before the first volatile edge is considered. In this case no edge of the volatile part is processed in this step.)

Finally we traverse the (remaining) volatile part edge by edge, each time comparing the next edge to the new edge. If the new edge (w.r.t. source and—possibly still to be assigned—destination index and edge and destination node attribute) is lexicographically smaller, it is inserted at the current position in the volatile part and the rest of the volatile part is appended. Otherwise any unnumbered node incident to the current volatile edge is numbered and the next volatile edge is considered. If all volatile edges have been traversed and the new edge has not been inserted, it is simply appended at the end of the code word.

Note that, provided the new edge is not a perfect extension itself, the insertion position of the new edge is recorded for the restricted extensions as required by the canonical form (that is, extensions preceding this edge are ruled out). In other words, if the new edge is not a perfect extension, the place at which it is inserted is the new end of the fixed part of the code word (see above).

Note also that the resulting code word still has to be checked for canonical form. Since the reorganization is strictly limited, the resulting code word may not be canonical. In this case the fragment has to be pruned.

6. Experiments

In order to test full perfect extension pruning, we implemented it as an extension of the MoSS program[2], which is

[1] A breadth-first search code word has the general form $a\,(i_s\,b\,a\,i_d)^m$, where a is node attribute, b an edge attribute, i_s the index of the source node of an edge, and i_d the index of the destination node of an edge. m is the number of edges of the fragment. Each parenthesized expression describes one edge. These edge descriptions are sorted lexicographically. The lexicographically smallest code word over all possible breadth-first numberings of the nodes of the fragment is the canonical code word. Details about a whole family of canonical forms can be found in [3].

[2] MoSS is available for free download under the Gnu Lesser (Library) Public License at http://fuzzy.cs.uni-magdeburg.de/∼borgelt/moss.html.

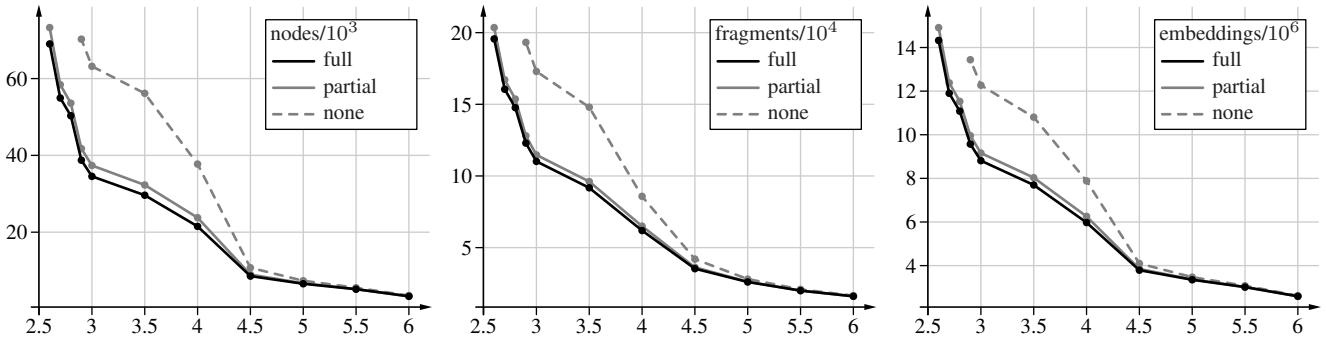

Figure 8. Experimental results on the IC93 data without ring mining (pure single bond extensions).

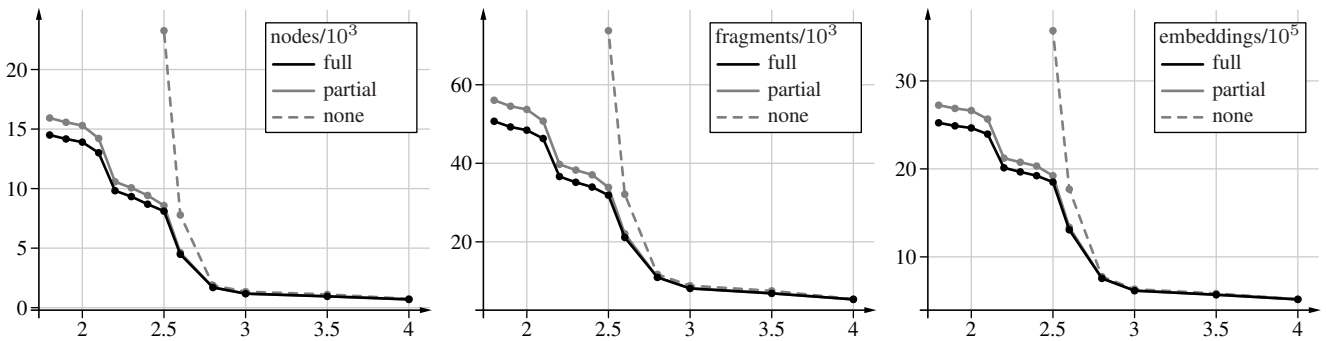

Figure 9. Experimental results on the IC93 data with ring mining (complete ring extensions).

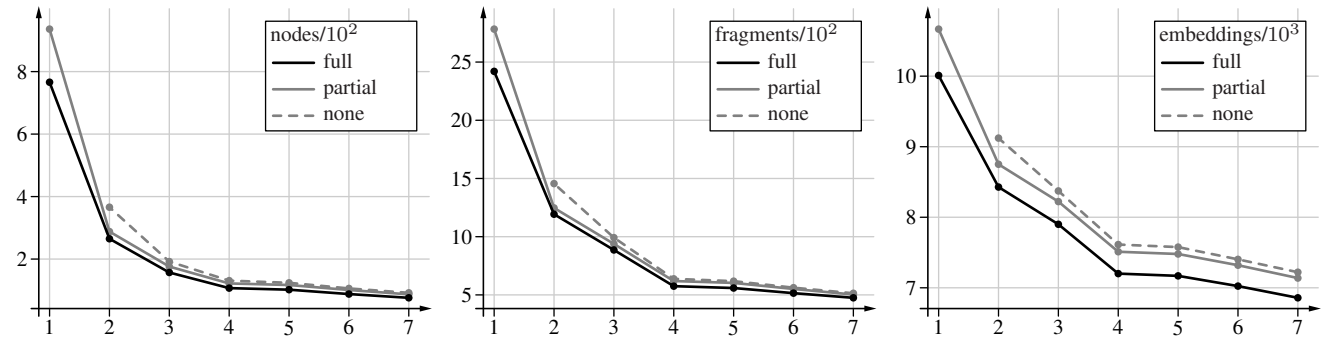

Figure 10. Experimental results on the steroids data with ring mining (complete ring extensions).

written in Java. As the test dataset we used the well-known subset of the *Index Chemicus* 1993 [9] and a small dataset of 17 steroids. The results on these datasets with different search modes are shown in Figures 8 to 10, which display the number of search tree nodes (left), created fragments (middle), and created embeddings (right). The horizontal axis shows the minimal support in percent (IC93) or as an absolute number (steroids). For the experiments of Figures 9 and 10 we used ring mining, which means that rings in a user-defined size range (here: 5 to 6 bonds) were not built edge by edge, but added in one step. In each diagram the dashed grey line refers to the basic algorithm without any perfect extension pruning, the grey solid line to partial perfect extension pruning and the black solid line to full perfect extension pruning.

These results show that full perfect extension pruning indeed leads to some non-negligible gains (in the order of about 5 to 10%) over partial perfect extension pruning. Tests we ran during the development of the program indicated that relaxing the constraints for perfect extensions (also edges closing rings/cycles instead of only bridges) improved performance by up to an additional 3%.

7. Conclusions

In this paper we fixed the flaw of the original descriptions of perfect extension pruning by requiring that perfect extensions must be bridges, but still allowing edges that close rings/cycles apart from bridges. In addition, we introduced *full perfect extension pruning*, which consists in pruning not only the search tree branches to the right (*partial perfect extension pruning* as it is used in Closegraph [16]), but also those to the left of the perfect extension branch. To make this possible in combination with canonical form pruning, we allowed for a (strictly limited) reorganization of code words as they result from the search. The experimental results show that this method can actually further reduce the complexity of the search, although the main improvement comes from partial perfect extension pruning. Future work is directed at combining sibling perfect extensions into one extension, so that perfect extensions, once found, need not be rediscovered and reprocessed.

References

[1] C. Borgelt and M.R. Berthold. 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 2002

[2] C. Borgelt, T. Meinl, and M.R. Berthold. Advanced Pruning Strategies to Speed Up Mining Closed Molecular Fragments. *Proc. IEEE Conf. on Systems, Man and Cybernetics (SMC 2004, The Hague, Netherlands)*, CD-ROM. IEEE Press, Piscataway, NJ, USA 2004

[3] C. Borgelt. On Canonical Forms for Frequent Graph Mining. *Proc. 3rd Int. Workshop on Mining Graphs, Trees and Sequences (MGTS'05, Porto, Portugal)*, 1–12. ECML/PKDD 2005 Organization Committee, Porto, Portugal 2005

[4] D.J. Cook and L.B. Holder. Graph-Based Data Mining. IEEE Trans. on Intelligent Systems 15(2):32–41. IEEE Press, Piscataway, NJ, USA 2000

[5] G. Di Fatta and M.R. Berthold. Distributed Mining of Molecular Fragments. *Workshop on Data Mining and the Grid, IEEE Int. Conf. on Data Mining*, 1–9. IEEE Press, Piscataway, NJ, USA 2004

[6] P.W. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacore Discovery Using the Inductive Logic Programming System PROGOL. *Machine Learning*, 30(2-3):241–270. Kluwer, Amsterdam, Netherlands 1998

[7] H. Hofer, C. Borgelt, and M.R. Berthold. Large Scale Mining of Molecular Fragments with Wildcards. *Intelligent Data Analysis* 8:495–504. IOS Press, Amsterdam, Netherlands 2004

[8] J. Huan, W. Wang, and J. Prins. 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 2003

[9] *Index Chemicus — Subset from 1993*. Institute of Scientific Information, Inc. (ISI). Thomson Scientific, Philadelphia, PA, USA 1993
http://www.thomsonscientific.com/products/indexchemicus/

[10] S. Kramer, L. de Raedt, and C. Helma. 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 2001

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

[12] *DTP AIDS Antiviral Screen (HIV Data Set) — Subset from 2001*. Developmental Therapeutics Program (DTP), National Cancer Institute, USA 2001
http://dtp.nci.nih.gov/docs/aids/aids_data.html

[13] S. Nijssen and J.N. Kok. 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 2004

[14] T. Washio and H. Motoda. State of the Art of Graph-Based Data Mining. *SIGKDD Explorations Newsletter* 5(1):59–68. ACM Press, New York, NY, USA 2003

[15] X. Yan and J. Han. 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 2002

[16] X. Yan and J. Han. 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 2003