First publ. in: Mining Complex Data / Djamel A. Zighed ... (eds.). (Studies in Computational Intelligence, Vol. 165). Berlin: Springer, 2009, pp. 189-205

Full Perfect Extension Pruning for Frequent Graph Mining

Christian Borgelt[1] and Thorsten Meinl[2]

[1] European Center for Soft Computing
c/ Gonzalo Gutirrez Quirs s/n, 33600 Mieres, Spain
christian.borgelt@softcomputing.es
[2] Nycomed Chair for Bioinformatics and Information Mining
Dept. of Computer and Information Science
University of Konstanz, Box M712, 78457 Konstanz, Germany
Thorsten.Meinl@uni-konstanz.de

Summary. 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 [4]. With this method the number of generated fragments and visited search tree nodes can be reduced, often considerably, thus accelerating the search. We describe the method in detail and present experimental results that demonstrate its usefulness.

1 Introduction

In recent years the problem how of 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 [7]. However, the vast majority transfers techniques developed originally for frequent item set mining. Examples include MolFea [11], FSG [12], MoSS/MoFa [3] , gSpan [16], Closegraph [17], FFSM [9], and Gaston [14]. A related, but slightly different approach, which is strongly geared towards graph compression, is used in Subdue [5].

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, [1] for a family of such canonical forms and details of the procedure). As the properties of canonical forms are widely used throughout this chapter, we will briefly review them in Sect. 4. To further improve the algorithms one may restrict the search to so-called closed graph fragments (Sect. 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 [4], or equivalent occurrence pruning, as it is called in [17]. 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 (Sect. 3).

In addition, the approach in [4] avoided redundant search with the help of a repository of found fragments instead of using the more elegant 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 (Sect. 5). This was first shown in Closegraph [17]. 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 (Sect. 6). We demonstrate the usefulness of the enhanced approach with experiments on molecular data sets (Sect. 7).

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 of graphs in the given database. As an example consider the three molecules (no chemical meaning attached – they were constructed merely for demonstration purposes) shown in Fig. 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 Fig. 2 (how the extensions of

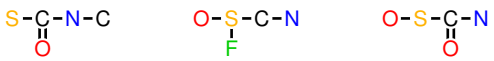

Fig. 1. Three simple example molecules

fragments are chosen and ordered is explained later). The numbers below or to

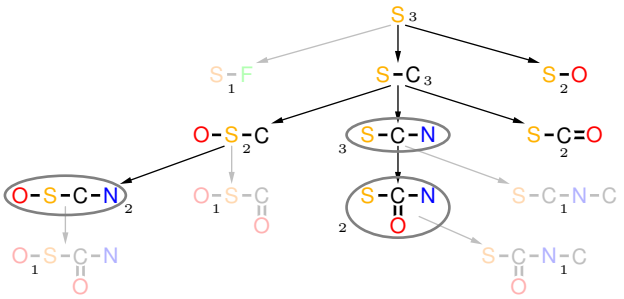

Fig. 2. Search tree for the three molecules in Fig. 1; infrequent fragments (contained in only one molecule) are drawn in grey/light colors, closed fragments are encircled

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). Note that, for example, the fragment `O-S-C` is not closed, since the fragment `O-S-C-N`, which contains `O-S-C` as a proper subgraph, has the same support (namely 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 Fig. 2) can be constructed from the closed ones by simply forming all substructures of closed fragments that are not closed fragments themselves. Knowledge of the support of a non-closed frequent fragment is also preserved: it is simply the maximum of the support values of those closed fragments of which it is a substructure. 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 given 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 the 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 only one path in the search tree that leads to this maximal common fragment and to start branching only from there.

As an example consider again the set of molecules shown in Fig. 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 (that is, occurrences of the fragment in the database graphs) 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 of a fragment 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.

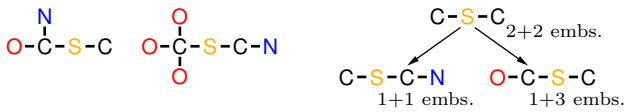

Fig. 3. Example of an imperfect extension

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 Fig. 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, because the extension can be done only once in the left molecule, but three times in the right. 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 and compare the number of embeddings per database graph.

A second problem (which was overlooked in both [17] as well as in [4]) is the behavior of rings (cycles) in the search, as we demonstrate with the example molecules shown in Fig. 4. A search tree for these molecules (with

Fig. 4. Rings/cycles can cause problems

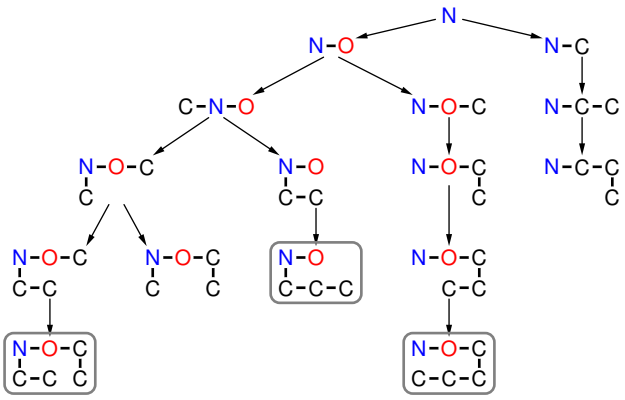

Fig. 5. Search tree for the two molecules in Fig. 4; closed fragments are encircled

only such fragments that are contained in both molecules) is shown in Fig. 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 extension 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. As a consequence we cannot restrict the search to the corresponding branch, since we would lose fragments. This can be seen clearly from the location of the closed fragments in the search tree shown in Fig. 5: there are three closed fragments (for a minimum support of 2, encircled in grey), but we cannot reach all of them if we see adding an edge from the nitrogen atom to a carbon atom as a perfect extension (even after the oxygen atom has been added, which could actually be seen as a perfect extension).

The problem obviously is that there are two ways of reaching the carbon atom that is directly connected to the nitrogen atom. Even though only one of them is possible in both molecules, both have to be considered, because part of the second possibility is the same in both molecules, thus leading to a relevant frequent fragment. Unfortunately there is no way to determine this locally, that is, by looking only at the grown fragment and its direct extension. 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 a necessary condition. As a closer inspection easily reveals, that extensions closing a ring (that is, extensions by an edge leading to a node that is already in the base fragment) are also safe and thus can be allowed as candidates for perfect extensions: since the destination node is already in the fragment, there cannot be a problem with multiple ways of reaching it. Hence we can slightly relax the constraints.

Note, however, that these relaxed constraints are still only sufficient, but not necessary. There are other situations in which an extension can be considered perfect, even though it does not meet the abovementioned 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, it is harmless and thus can be considered a perfect extension. Even the extension by a bond from the nitrogen atom to the oxygen atom in Fig. 5 can be considered a safe perfect extension, despite the fact that the rings have different size. Unfortunately, checking necessary and sufficient conditions is complicated and costly and thus we confine ourselves to the rule that an extension edge must either be a bridge in all molecules or must close a ring (cycle) in all molecules in order to be considered perfect.

4 Canonical Codes for Graphs

As we already mentioned in the introduction, perfect extension pruning, as it was described in the previous section, is not a problem unless canonical forms are used to identify redundant fragments. However, since canonical forms are a lot more elegant than, for example, a repository of already processed fragments, need less memory and make it easier to parallelize the search, it is desirable to be able to use perfect extension pruning together with canonical form pruning. In this section we briefly review some fundamentals of canonical forms for (attributed) graphs, which are necessary to know in order to understand the code word reorganization we discuss in Sect. 6.

The core idea of canonical forms of graphs is to describe an (attributed) graph by a code word, which uniquely identifies it up to automorphisms, and from which the graph can be reconstructed. The letters of such a code word describe the edges of the graph and which nodes they connect as well as the node and edge attributes (or labels). In order to capture the connection structure, the nodes are numbered (or, more generally, endowed with unique labels), since the node attributes are not enough to identify them uniquely: the same attribute may be assigned to several nodes in a graph. Of course, there are several possible ways of numbering the nodes, each of which gives rise to a different code word. Generally, all of these code words are taken into account and the lexicographically smallest (or greatest) code word is then defined to be the canonical code word. Note, however, that due to the way in which code words are used in the search (see below), the possible node numberings (and thus the possible code words) one has to consider can actually be restricted to

those compatible with traversals of spanning trees (see [1] for more extensive explanations).

Canonical code words are used in the search as follows: during the mining process the fragments are grown by adding an edge in each step. This edge is characterized by the node in the fragment from which it starts, by its attribute, and by the node it leads to. (Note that this does not mean that the edges are directed; the "source" and "destination" node are solely defined by how the extension is done: the node that is extended is the "source" and the other node, the extension edge is incident to, is the "destination"). In addition, the nodes are numbered in the order in which they are added to the fragment. Hence the search process naturally constructs a code word for each grown fragment, namely by simply concatenating the descriptions of the edge extensions that led to it. Of course, there are many possible ways of building a fragment by adding edges, each of which leads to a different code word. However, there is obviously only one way that leads to the canonical code word for a fragment (since a code word fixes a specific order of the extensions). Hence we may choose to extend only those fragments that have been built in such a way that their code word is canonical. Eliminating all other fragments is called canonical form pruning, which obviously rules out all redundant search: each fragment is considered at most once. Note that the way in which the search process builds code words also explains why we can confine ourselves to node numberings (and thus code words) compatible with traversals of spanning trees (as mentioned above): no other code words can be constructed by the search.

For the rest of the paper we focus on code words resulting from node numberings that are obtained by breadth-first traversals of spanning trees, that is, the canonical form used in MoSS/MoFa [3]. Note, however, that the described approach is also applicable for code words resulting from node numberings that are obtained from depth-first traversals of spanning trees, that is, the canonical form used in gSpan [16] or Closegraph [17]. The necessary adaptations of the procedure are straightforward and thus not described in detail.

A breadth-first 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 (or number) of the source node of an edge, and i_d the index (or number) of the destination node of an edge (it is always $i_s < i_d$). The letter m denotes the number of edges of the fragment. Each parenthesized expression describes one edge. As an example, consider the left molecule shown in Fig. 1. If this molecule is built from left to right, that is, if we choose the left oxygen atom as the root of a spanning tree, a possible code is `O 0-S1 1-C2 2=O3 2-N4`. As can be seen from this code word, the bond added first is the one from the oxygen atom (index 0) to the sulfur atom (index 1), the bond added last is the one from the carbon atom (index 2) to the nitrogen atom (index 4). However, there is another possibility of building the same fragment, which leads to the code word `O 0-S1 1-C2 2-N3 2=O4` (that is, the last two bonds are added in inverse

order). If these two code words are compared lexicographically, the latter is smaller than the former (assuming that single bonds are "smaller" than double bonds, that is, `- < =`). Therefore we can conclude that the first code word is not the canonical code word, but neither is the second. The canonical code word for this molecule is actually `S 0-C1 0-O2 1-N3 1=O4` (if we use the order `S < C < N < O` and `- < =`, which we always use for all following examples). Note, however, that the canonical code word is `C 0-N1 0-S2 0=O3 2-O4` if we use the order of the periodic table of elements (that is, `C < N < O < S`, and `- < =`), showing that which code word is the canonical one also depends on the order we choose for the node and edge attributes. Empirical evidence suggests that it is recommendable to use an order that reflects the frequency of the attributes in the graph database to mine (less frequent attributes should precede more frequent ones), as this usually leads to fewer generated fragments and thus shorter search times. Note also that (canonical) code words for graph fragments provide a natural way of ordering the fragments in the search tree: the children of a search tree node are listed from left to right in the order of lexicographically increasing code words. This makes precise what we mean by "to the left" and "to the right" of a search tree branch: "to the left" are fragments with smaller, "to the right" fragments with greater (canonical) code words.

Checking whether a given code word is canonical usually requires testing all possible code words for a fragment (at least w.r.t. all possible node numberings resulting from traversals of spanning trees) and thus has essentially the same complexity as a graph isomorphism test. (Pseudo-code for such a canonical form check can be found, for example, in [1].) Nevertheless, canonical code words are very effective in pruning the search tree, because they use "global" information in contrast to only "local" rules, as they were used originally in [3]. These "local" or "simple" rules, however, can still be applied to support canonical form pruning, as they specify necessary (though not sufficient) prerequisites for code words to be canonical, which can be tested very efficiently and help to avoid a costly canonical form test in many cases. For example, if we use a breadth-first canonical form (as it was described above), one may not extend a node that has an index smaller than another node in the fragment, which has already been extended (maximum source extension: only nodes with an index no less than the maximum source index may be extended). The reason is that an extension violating this rule necessarily leads to a non-canonical code word, as can easily be checked with a spanning tree rooted at the same node. As an example consider the fragment `S-C-N` in the search tree in Fig. 2: this fragment may not be extended by an edge from the sulfur atom (index 0) to an oxygen atom, because an atom with a higher index, namely the carbon atom (index 1), has already been extended. Indeed, if we add such an edge, the code word of the resulting fragment is `S 0-C1 1-N2 0-O3`, while the canonical code word for this fragment is `S 0-C1 0-O2 1-N3` (using again the order `S < C < N < O`). More details on canonical forms and the "local" or "simple" rules, which result from them and restrict the possible extensions,

can be found in [1]. Of course, there are "local" or "simple" rules not only for breadth-first, but also for depth-first code words, which specify a restricted set of extensions known as rightmost path extensions. However, for the discussion in this paper it suffices to know that regardless of the canonical form used the "local" or "simple" rules basically state that extensions that generated the sibling nodes to the left of a search tree branch may not be done in the search tree branch itself or in branches to the right of it.

5 Partial Perfect Extension Pruning

If one wants to combine perfect extension pruning with canonical form pruning as it was described above, the following problem has to be solved: growing the maximal common fragment can interfere with canonical form pruning and in particular the extension restrictions resulting from it (note that this was no problem in [4] due to the use of a repository of found fragments to avoid redundant search). However, perfect extensions should not lead to such a restriction, because otherwise search results may be lost. Even the fact that the code word of a fragment, as it is built by the search, is not canonical is not sufficient to prune it.

As an example consider again the search tree shown in Fig. 2. If we simply confined the search to the sub-tree 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 sub-tree (as described in the preceding section), because an atom with a higher number, namely the carbon atom, has already been extended in the preceding step.

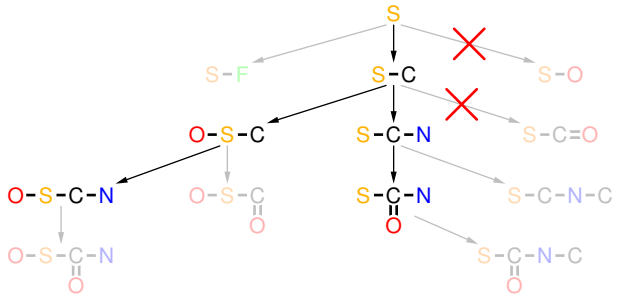

Fig. 6. Search tree for the three molecules in Fig. 1 with partial perfect extension pruning (crossed out branches are pruned)

Fortunately, this only affects search tree branches to the left of the perfect extension branch, since the corresponding extensions are ruled out by the perfect extension and the "local" or "simple" rules (see above). 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: they cannot contain any closed fragment, because the perfect extension cannot be done in them without violating canonical form, but has to be done in order to reach a closed fragment. 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 [17] uses only this form of pruning.

How partial perfect extension pruning changes the search tree for the molecules in Fig. 1 is shown in Fig. 6. Note that only non-closed fragments are removed from the search tree (compare to Fig. 2, in which the closed fragments are highlighted). The gains consist in the fact that the two pruned fragments need not be processed: neither do they have to be checked for canonical form nor do we have to consider possible extensions of them.

6 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 (with or without the "local" or "simple" rules). 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 Fig. 7 for the molecules in Fig. 1.

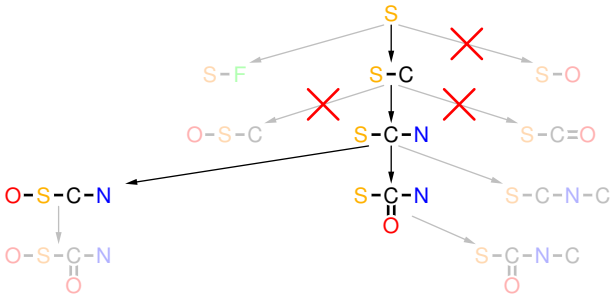

Fig. 7. Search tree for the three molecules in Fig. 1 with full perfect extension pruning (crossed out branches are pruned)

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 `S 0-C1 0-O2 1-N3`. However, with the search tree in Fig. 7 it is

assigned the code word `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 by a perfect extension with no search tree branches to the left of it. 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 of it. Note that we can check for the existence of branches to the left of a perfect extension branch after minimum support pruning, that is, after eliminating all fragments that occur in less than the user-specified minimum number of database graphs. The reason is that we can be sure that extensions leading to infrequent fragments in branches to the left will also lead to infrequent fragments in the perfect extension branch or in branches to the right of it and thus need not be considered in these branches.

The construction of the code word of a fragment is then modified as follows: instead of always simply appending the description of the extension edge to the end of a code word, the description of the 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 the new edge description does not enter 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 sulfur atom to the oxygen atom) is shifted left past the perfect extension bond (that is, the bond from the carbon atom to the nitrogen atom), the oxygen and the nitrogen atom get new indices. The reason is that the nodes must be numbered in the order in which they would be added if the edges were added in the order in which their descriptions are listed in the (reorganized) code word (see Fig. 8).

1. Base fragment: `S-C-N`	canonical code: `S 0-C1 1-N2`
2. Extension to `O-S-C-N`	code: `S 0-C1 1-N2 0-O3` (not canonical!)
3. Shift the non-perfect extension	code: `S 0-C1 0-O3 1-N2`
4. Renumber nodes	canonical code: `S 0-C1 0-O2 1-N3`

Fig. 8. Fixing a fragments code by shifting a non-canonical extension over perfect extensions (marked in gray) to the proper place

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 edge in the volatile part 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 (renumbering nodes as needed). 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.

To make the process clearer, we execute it step by step for the example shown in Fig. 8. The root node (here the sulfur atom) is, of course, always in the fixed part. Hence it receives the initial node index, that is, 0. Since the next edge is already in the volatile part, this finishes processing the fixed part. Since by assigning the index 0 to the sulfur atom, one node incident to the new edge (sulfur to oxygen) is thus already numbered, we have to start immediately to compare edge descriptions. We compare two possibilities, namely appending the description of the new edge, which assigns the node index 1 to the oxygen atom, or appending the already present first perfect extension edge (sulfur to carbon), which assigns the node index 1 to the carbon atom. This yields two possible code word prefixes, namely `S 0-O1` and `S 0-C1`. Since the latter is smaller (as `C < O`), it is fixed (that is, the new edge is not yet inserted) and we move to the next position. Here we compare the code word prefixes `S 0-C1 0-O2` and `S 0-C1 1-N2`. Since the former is smaller (as `O < N`), the position of the new edge has been found and we fix the first prefix. In a final step, the remaining perfect extension edge is appended, assigning the node index 3 to the nitrogen atom. Note that the fixed part of the resulting code word now contains not only the root atom, but two bonds: the first perfect extension bond, which is rendered fixed by the fact that a non-perfect extension was inserted after it, and the new bond, which is fixed, simply because it is not a perfect extension. The volatile part contains only the second perfect extension (the bond from the carbon to the nitrogen atom).

Note that generally, provided the new edge is not a perfect extension itself, this edge is recorded for the restricted extensions as required by the "local" or "simple" rules of maximum source extensions (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 (as described 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. For example, the new edge may actually have to be inserted into the fixed part in order to make the

code word canonical. In this case the fragment must not be adapted, so that the code word becomes canonical, but has to be pruned.

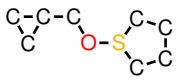

Fig. 9. Example molecule used to demonstrate full perfect extension pruning

To further illustrate the process described above, we study another complete example, which also shows the different cases that can occur. Consider the molecule shown in Fig. 9. Our goal is to build this molecule using full perfect extension pruning[3]. As the order of the elements we use again the order `S < C < O`, which is in line with the order used in the preceding example. As a consequence the search has to start at the sulfur atom, because all other starting points obviously lead to non-canonical code words (as even the first letter is greater for them). Three extensions of this one-node fragment (code word: `S`) are possible: we may add one of the two ring bonds to carbon atoms (which lead to the same fragment `S-C`) or we may add the bond to the oxygen atom. Without perfect extension pruning, both child fragments (`S-C` and `S-O`) would have to be considered. However, the bond to the oxygen atom occurs in all molecules (only one in this example) and the number of embeddings of the extended fragment is the same as for the single sulfur atom. Hence adding this bond is a perfect extension, while the bond to a carbon atom is not eligible as a perfect extension, since it is a ring bond (and thus no bridge, see Sect. 3). This leads to the code word `S 0-O1`. The extension is marked as perfect, and the volatile part of the code word starts directly after the oxygen atom (indicated by a grey background). Note that the other extension (leading to the fragment `S-C`) would have to be considered if we only used partial perfect extension pruning, since its code word, that is, `S 0-C1`, is smaller than `S 0-O1`. Only full perfect extension pruning allows us to eliminate this fragment from the search.

In the next step, all possible extensions are considered (no restriction by "local" or "simple" rules), which are the two ring bonds (again leading to the same fragment, now `O-S-C`) and the bond from the oxygen atom to the next carbon in the chain atom. The latter is a perfect extension and thus the other two extensions are pruned, resulting in the code word `S 0-O1 1-C2`. Since the new edge is a perfect extension, the volatile part grows to two edge descriptions (grey background). In the third step, the two ring bonds incident to the sulfur atom are again eliminated due to the perfect extension to the next carbon atom in the chain, which is in the left ring: `S 0-O1 1-C2 2-C3`.

[3] Mining only one molecule is, of course, not very sensible in practice, but it keeps the example simple and the process is the same as when mining a larger number of molecules.

Now there are no perfect extensions left, because all remaining bonds are part of rings (and thus no bridges). It should be noted that the maximum source index is still 0 (sulfur atom), because all extensions made so far were perfect and thus their source indices are not counted for the "local" or "simple" rules characterizing maximum source extensions. Without this special handling, we would not be allowed to add any of the bonds of the right ring. But since the sulfur atom is still extendable, we add, in the next step, one of the two ring bonds to it, which results in the code word `S 0-O1 1-C2 2-C3 0-C4`. As one can immediately see from the source index 0 in the last bond, this code word is not canonical. Therefore we have to start the process of rebuilding the code word. First, the sulfur atom is numbered 0 and this already determines three of the four parts of the description of the newly added bond from the sulfur to the carbon atom, namely `0-C_` (the still to be assigned destination index is replaced by an underscore; alternatively it may be set to the next free node index, which is 1 in this case, as we did it before). This "incomplete" extension is compared to the first perfect extension in the volatile part. Since the incomplete description `0-O_` of this extension is greater (as carbon precedes oxygen), the position of the new edge has been found and the description of this edge is appended. Therefore we have as a code word prefix `S 0-C1`, which forms the new fixed part of the code word. The three perfect extensions in the volatile part are renumbered accordingly (the indices of the destination nodes are increased by one) and their descriptions are appended, yielding the code word `S 0-C1 0-O2 2-C3 3-C4`. The next extension adds the other ring bond from the sulfur atom to a nitrogen atom: we reorganize from `S 0-C1 0-O2 2-C3 3-C4 0-C5` to `S 0-C1 0-C2 0-O3 3-C4 4-C5`. The sixth extension adds another ring bond, yielding – before reorganization – the code word `S 0-C1 0-C2 0-O3 3-C4 4-C5 1-C6`. This time, the new edge description is not inserted before all perfect extensions, but after the first, because its source node index is greater than that of the first perfect extension: `S 0-C1 0-C2 0-O3 1-C4 3-C5 5-C6`. This has two effects: in the first place, the volatile part now consists of only the last two perfect extensions (as the insertion of a non-perfect extension edge after the first perfect extension renders the first perfect extension edge fixed). Secondly, the atom with the maximum source index (from which on extension are still allowed) is now the one with index 1, namely the source atom of the added edge.

The next edge that is added is another ring bond and it is inserted before the volatile part, since its source index is smaller than the source index of the next perfect extension bond: `S 0-C1 0-C2 0-O3 1-C4 2-C5 3-C6 6-C7`. The next bond closes the right ring and it is inserted in the middle of the volatile part: `S 0-C1 0-C2 0-O3 1-C4 2-C5 3-C6 4-C5 6-C7` (since its source node index is larger than that of the first perfect extension bond in the volatile part, but smaller than that of the second perfect extension). The last three edges, that is, the three bonds of the left ring (3 carbons), are added in the normal order (after the volatile part, or actually simply by appending to a fixed code word, since adding the first bond of the left ring renders the last

perfect extension fixed). No code word reorganization is necessary any more. The final code word is:
`S 0-C1 0-C2 0-O3 1-C4 2-C5 3-C6 4-C5 6-C7 7-C8 7-C9 8-C9`.

7 Experiments

In order to test full perfect extension pruning, we implemented it as an extension of the MoSS program[4], which is written in Java. As the test dataset we used the well-known subset of the Index Chemicus 1993 [10] and a small dataset of 17 steroids. The results on these datasets with different search modes are shown in Fig. 10, Fig. 11, and Fig. 12, which display the number of

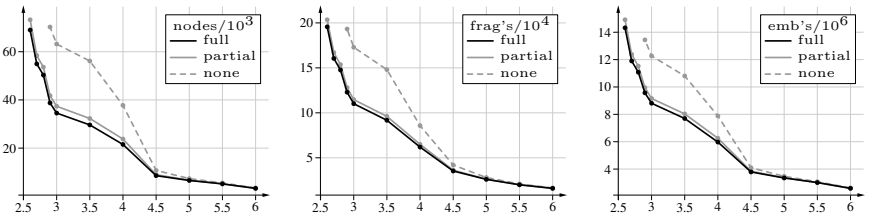

Fig. 10. Experimental results on the IC93 data without ring mining (pure single bond extensions)

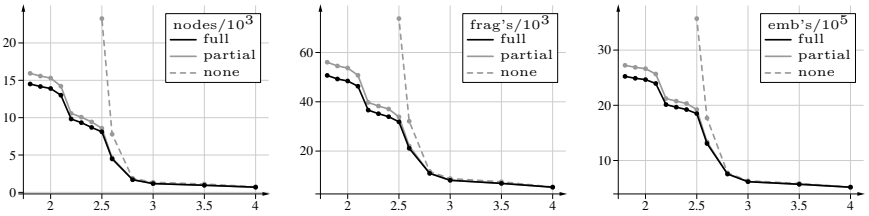

Fig. 11. Experimental results on the IC93 data with ring mining (complete ring extensions)

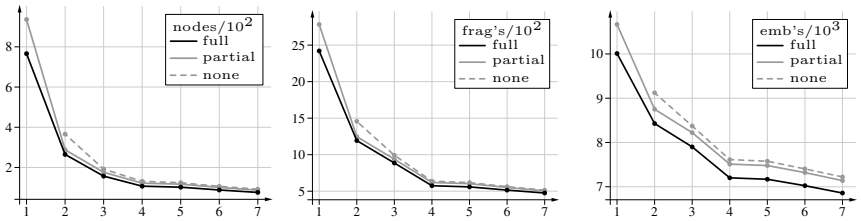

Fig. 12. Experimental results on the steroids data with ring mining (complete ring extensions)

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

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 Fig. 11 and Fig. 12 we used ring mining, which means that rings in a user-defined size range (here: 5 to 6 bonds) are not built edge by edge, but added in one step. The technique underlying such ring mining was introduced in [8] for a repository of processed fragments to avoid redundant search, but later extended in [2] to work with perfect extension pruning (using a reorganization technique similar to the one presented in this paper).

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, even though the main gains clearly result from partial perfect extension pruning. Tests we ran during the development of the program indicated that relaxing the constraints for perfect extensions (that is, also edges closing rings/cycles are allowed as perfect extensions instead of only bridges) improved performance by up to an additional 3%.

8 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 [17]), 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. 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
2. C. Borgelt. Combining Ring Extensions and Canonical Form Pruning. *Proceedings of the 4th International Workshop on Mining and Learning in Graphs*

(MLG06), Berlin, Germany, 109–116. ECML/PKDD 2006 Organization Committee, Berlin, Germany 2006
3. 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
4. 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
5. 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
6. 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
7. 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
8. 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
9. 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
10. *Index Chemicus — Subset from 1993.* Institute of Scientific Information, Inc. (ISI). Thomson Scientific, Philadelphia, PA, USA 1993 http://www.thomsonscientific.com/products/indexchemicus/
11. 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
12. 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
13. *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
14. 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
15. 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
16. 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
17. 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

Index