Canonical Forms for Frequent Graph Mining

Christian Borgelt

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

Abstract. A core problem of approaches to frequent graph mining, which are based on growing subgraphs into a set of graphs, is how to avoid redundant search. A powerful technique for this is a canonical description of a graph, which uniquely identifies it, and a corresponding test. I introduce a family of canonical forms based on systematic ways to construct spanning trees. I show that the canonical form used in gSpan (Yan and Han (2002)) is a member of this family, and that MoSS/MoFa (Borgelt and Berthold (2002), Borgelt et al. (2005)) is implicitly based on a different member, which I make explicit and exploit in the same way.

1 Introduction

Recent years saw an intense and still growing interest in the problem how to find common subgraphs in a database of (attributed) graphs, that is, subgraphs that appear with a user-specified minimum frequency. For this task—which has 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 graph structures by logical expressions (Finn et al. (1998)). However, the vast majority transfers techniques developed originally for frequent item set mining.[1] 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 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 in each step, counting the number of graphs containing each grown subgraph, and eliminating infrequent subgraphs.

[1] See, for example, Goethals and Zaki (2003, 2004) for details and references on frequent item set mining.

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

One of the most promising ways to avoid redundant search is to define a canonical description of a (sub)graph. Together 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 and thus need not be extended. This approach underlies the gSpan algorithm (Yan and Han (2002)) and its extension CloseGraph (Yan and Han (2003)). In this paper I generalize the canonical form of gSpan, thus arriving at a family of canonical descriptions. I also show that a competing algorithm called MoSS/MoFa (Borgelt and Berthold (2002), Borgelt et al. (2005)) is implicitly based on a canonical description from this family, which is different from the gSpan one. By making this canonical form explicit, it can be exploited in MoSS/MoFa in the same way as in gSpan, leading to a significant improvement of the MoSS/MoFa algorithm.

2 Finding frequent subgraphs

Generally, a graph database is searched for frequent subgraphs as follows: Given an initial node (for which all possibilities have to be tried), a subgraph is grown by adding an edge and, if necessary, a node in each step. In this stepwise extension process one usually restricts the search to connected subgraphs (which suffices for most applications). In its most basic form the search considers all possible extensions of the current subgraph. (It will be shown later how the set of extensions can be reduced by exploiting a canonical description.)

Note that, as a consequence of the above, the search produces a numbering of the nodes in each subgraph: the index of a node simply reflects the order in which it was added. In the same way it produces an order of the edges—again the order in which they were added. Even more: the search builds a spanning tree of the subgraph, which is enhanced by additional edges (closing cycles).

3 Canonical forms of attributed graphs

In this section I describe the family of canonical descriptions that is introduced in this paper, using the special cases employed in gSpan and MoSS/MoFa as examples and pointing out alternatives. How these canonical forms define an extension strategy and thus a search order is discussed in the next section.

3.1 General idea

The core idea underlying a canonical form is to construct a code word that uniquely identifies a graph up to isomorphism and symmetry (i.e. automorphism). The characters of this code word describe the edges of the graph. If the graph is attributed or directed, they also comprise information about edge attributes and/or directions as well as attributes of the incident nodes.

While it is straightforward to capture the latter information about an edge (i.e. attributes and edge direction), how to describe the connection structure is less obvious. For this, the nodes of the graph must be numbered (or more generally: endowed with unique labels), because we need a way 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 specifications of an edge (simply because the indices of the source and the destination node differ). In addition, the edges can be listed in different orders. How these two problems can be treated is described in the following: the different possible solutions give rise to different canonical forms.

However, given a (systematic) way of numbering the nodes of a (sub)graph and a sorting criterion for the edges, a canonical description is generally derived as follows: each numbering of the nodes yields a code word, which is the concatenation of the sorted edge descriptions (details are given in Section 3.4). The resulting list of code words is sorted lexicographically. The lexicographically smallest code word is the canonical description. (Note that the graph can be reconstructed from this code word.)

3.2 Constructing spanning trees

From the review of the search process in Section 2 it is clear that we can confine ourselves to numberings of the nodes of a (sub)graph that result from spanning trees, because no other numberings will ever occur in the search. Even more: specific systematic ways of constructing a spanning tree suffice. The reason is that the basic search algorithm produces *all* spanning trees of a (frequent) (sub)graph, though usually in different branches of the search tree.[2] Since the extensions of a (sub)graph need to be checked only once, we may choose to form them only in the branch of the search tree, in which the spanning tree of the (sub)graph has been built in the chosen way. This can also be used to ensure that the canonical description has the *prefix property*, meaning that each prefix of a canonical code word is a canonical code word itself. Since in the search we extend only graphs in canonical form, the prefix property is needed to make sure that all (sub)graphs can be reached.

The best-known systematic methods for constructing a spanning tree of a graph are, of course, depth-first and breadth-first search. Thus it is not surprising that gSpan uses the former to define its canonical form (Yan and

[2] They occur in the same search tree node only if the graph exhibits some symmetry, i.e., if there exists an automorphism that is not the identity.

Han (2002)). However, the latter (i.e., breadth-first search) may just as well be chosen as a basis for a canonical form. And indeed: as will turn out later, the (heuristically introduced) local extension order of the MoSS/MoFa algorithm (Borgelt and Berthold (2002), Borgelt et al. (2005)) can be justified from a canonical form that is based on a breadth-first search tree. Thus MoSS/MoFa can be seen as implicitly based on this canonical form.

Other methods include a spanning tree construction that first visits all neighbors of a node (like breadth-first search), but then chooses the next node to extend in a depth-first manner (This may be seen as a variant of depth-first search.). However, in the following I confine myself to (standard) depth-first and breadth-first search trees to keep things simple. Nevertheless, it should be kept in mind that there are various other possibilities one may explore.

It should be noted that there is no restriction on the order in which the neighbors of a node are visited in the search. Hence there is generally a large number of different spanning trees, even if the root node is fixed. As a consequence choosing a method for constructing a spanning tree is not sufficient to avoid redundant search. Since usually several spanning trees of a (sub)graph can be constructed in the chosen way, there are several search tree branches that qualify for an extension of the (sub)graph. Although this freedom will be reduced below by exploiting edge and node attributes, it cannot be eliminated completely, since there are no local (i.e. node-specific) criteria that allow for an unambiguous decision in all cases (Borgelt and Berthold (2002)) gives an example). Therefore we actually need to construct and compare code words to avoid all redundancy.

3.3 Edge sorting criteria

Once we have a numbering of the nodes, we can set up edge descriptions and sort them. In principle, the edge descriptions can be sorted using any precedence order of the edge's properties (i.e. attribute of the edge and attributes and indices of the source and destination node). However, we can exploit the purpose for which the canonical form is intended to find appropriate sorting criteria. Recall that we construct different spanning trees of the same (sub)graph in different branches of the search tree. Each of these gives rise to a numbering of the nodes and thus a code word. In addition, recall that the canonical form is intended for confining the extensions of a (sub)graph to one branch of the search tree. Hence we need a way of checking whether the code word resulting from the node numbering in a search tree node is minimal or not: if it is, we descend into the search tree branch, otherwise we prune it.

In order to carry out this test, we could construct all other possible code words for a (sub)graph and compare them to the one resulting from the node numbering in the current search tree node. However, such a straightforward approach is much too costly. Fortunately, it can be made much more efficient, since the code words are compared lexicographically. Hence we may not need to know the full code words in order to decide which of them is lexicographically smaller—a prefix may suffice. This immediately gives rise to the idea to check all code words in parallel that share the same prefix. However, whether this is (easily) possible or not, depends on how we sort the edges.

Fortunately, for both canonical forms, depth-first and breadth-first search, there is a sorting criterion that yields such an order. The core idea is to define the order of the edges in such a way that they are sorted into the order in which they are added to the (sub)graph in the search. This has three advantages: in the first place, it ensures the prefix property. Secondly, we need no sorting to obtain the code word that results from the node numbering in the current search tree node. Since it is easiest to implement the search by always appending the added edge to a list of contained edges, this edge list already yields the code word. Thirdly, we can carry out the search for alternative code words in basically the same way as the whole search for frequent subgraphs. Doing so makes it possible to compare the prefixes of the code words after the addition of every single edge, thus making the test of a code word maximally efficient. Details about the comparisons are given below, after the exact form of the code words for the two canonical forms are defined.

3.4 Code words

In my definition of a code word I deviate slightly from the definition of gSpan (Yan and Han (2002)), where code words are simple lists of edge descriptions, each of which comprises all information about the edge and the incident nodes. The reason is that it is not necessary to compare the attribute of the source node, except for the first edge that is added. In other words, we may precede the sequence of edge descriptions by a character that specifies the attribute of the root of the spanning tree, while at the same time we cancel the attribute of the source node from the following edge descriptions. Then the general forms of code words (as regular expressions with non-terminal symbols) are:

- Depth-First Search: $a\,(i_d\,\underline{i_s}\,b\,a)^m$
- Breadth-First Search: $a\,(i_s\,b\,a\,i_d)^m$

Here a is a node attribute and b an edge attribute. i_s is the index of the source and i_d the index of the destination node of an edge. (Source and destination of an edge are defined by the relation $i_s < i_d$, that is, the incident node with the smaller index is the source.) Parentheses are for grouping characters; each parenthesized sub-word stands for one edge. The exponent m means that there are m repetitions of the group of characters to which it is attached.

The describing properties of an edge are compared in the order in which they appear in the parenthesized expressions. All characters are compared ascendingly, with the exception of the underlined i_s in the depth-first search form, which is compared descendingly. Note that the parenthesized expressions (that is, the edge descriptions) are sorted and are concatenated afterwards to form the code word. It is easy to see that in this way the code word describes how the edges have been added in the search process.

It should be noted that one may let spanning tree edges take absolute precedence over other edges. That is, the code word may start with the spanning tree edges, and only after all of them the other edges (which lead to cycles) are listed. Here, however, I make no such distinction of edge types. The order of the edge descriptions in the code words is defined by the stated edge properties alone and thus spanning tree edges may be intermingled with edges closing cycles. The reason is that I want the edges to be in exactly the order in which they have been added to the (sub)graph in the search tree. However, one may also choose to find spanning trees first before closing cycles. As the ideas underlying the Gaston algorithm (Nijssen and Kok (2004)) suggest, there may be good reasons for adopting such a strategy, as it may speed up the search.

3.5 Checking for canonical form

After the code words are defined, the test whether a code word is a canonical description of a (sub)graph can be stated formally. The pseudocode below describes the procedure. w is the code word to be tested, $G = (V, E)$ is the corresponding (sub)graph. Each node $v \in V$ has an attribute $v.a$, which I assume to be coded as an integer, and an index field $v.i$, which is filled by the algorithm. Likewise each edge $e \in E$ has an attribute $e.a$, again assumed to be coded as an integer, and a marker $e.i$, which is used to record whether it was visited. Since apart from node and edge attributes a code word contains only indices of nodes, it can thus be represented as an array of integers.

```
function isCanonical ($w$: int array, $G$: graph) : boolean;
var $v$ : node;                      (* to traverse the nodes of the graph *)
    $e$ : edge;                      (* to traverse the edges of the graph *)
    $x$ : node array;                (* to collect the numbered nodes *)
begin
  forall $v \in G.V$ do              (* traverse all nodes and *)
    $v.i := -1$;                     (* clear their indices *)
  forall $e \in G.E$ do              (* traverse all edges and *)
    $e.i := -1$;                     (* clear their markers *)
  forall $v \in G.V$ do begin        (* traverse the potential root nodes *)
    if $v.a < w[0]$ then return false;   (* abort on smaller root nodes *)
    if $v.a > w[0]$ then continue;       (* skip larger root nodes *)
    $v.i := 1$; $x[0] := v$;         (* number and record the root node *)
    if not rec($w$, 1, $x$, 1, 0)    (* check the code word recursively *)
    then return false;               (* abort if a smaller word is found *)
    $v.i := -1$;                     (* clear the node index again *)
  end
  return true;                       (* the code word is canonical *)
end
```

This function is the same, regardless of whether a depth-first or a breadth-first search canonical form is used. The difference lies only in the implementation of the function “rec”, mainly in the order in which edge properties are compared. Here I confine myself to the implementation for breadth-first search. However, for depth-first search the function can be implemented in a very similar way.

The basic idea is to add one edge in each level of the recursion. The description of this edge is generated and if it already allows to decide whether the generated code word is larger or smaller (prefix test!), the recursion is terminated. Only if the edge description coincides with the one found in the code word to check, the edge and the node at the other end (if necessary) are marked/numbered and the function is called recursively. Note that the loop over the edges incident to the node $x[i]$ in the pseudocode below assumes that the edges are considered in sorted order, that is, the edges with the smallest attribute are tested first, and among edges with the same attribute, they are considered in increasing order of the attribute of the destination node.

```
function rec ($w$: int array, $k$ : int, $x$: node array, $n$: int, $i$: int) : boolean;
var $d$ : node;                     (∗ node at the other end of an edge ∗)
    $j$ : int;                      (∗ index of destination node ∗)
    $u$ : boolean;                  (∗ flag for unnumbered destination ∗)
    $r$ : boolean;                  (∗ buffer for a recursion result ∗)
begin
  if $k \geq$ length($w$) return true;   (∗ full code word has been generated ∗)
  while $i < w[k]$ do begin        (∗ check for an edge with a ∗)
    forall $e$ incident to $x[i]$ do   (∗ source node having a smaller index ∗)
      if $e.i < 0$ then return false;
    $i := i + 1$;                   (∗ go to the next extendable node ∗)
  end
  forall $e$ incident to $x[i]$ (in sorted order) do begin
    if $e.i > 0$ then continue;     (∗ skip visited incident edges ∗)
    if $e.a < w[k+1]$ then return false;     (∗ check the ∗)
    if $e.a > w[k+1]$ then return true;      (∗ edge attribute ∗)
    $d :=$ node incident to $e$ other than $x[i]$;
    if $d.a < w[k+2]$ then return false;     (∗ check destination ∗)
    if $d.a > w[k+2]$ then return true;      (∗ node attribute ∗)
    if $d.i < 0$ then $j := n$ else $j := d.i$;
    if $j < w[k+3]$ then return false;       (∗ check destination ∗)
    if $j = w[k+3]$ then begin               (∗ node index ∗)
      $e.i := 1$; $u := d.i < 0$;   (∗ mark edge and number node ∗)
      if $u$ then begin $d.i := j$; $x[n] := d$; $n := n + 1$; end
      $r :=$ rec($w$, $k + 4$, $x$, $n$, $i$);       (∗ check recursively ∗)
      if $u$ then begin $d.i := -1$; $n := n - 1$; end
      $e.i := -1$;                  (∗ unmark edge (and node) again ∗)
      if not $r$ then return false;
    end                             (∗ evaluate the recursion result ∗)
  end
  return true;                    (* return that no smaller code word *)
end                                 (* than w could be found *)
```

3.6 A simple example

In order to illustrate the code words defined above, Figure 1 shows a simple molecule (no chemical meaning attached; it was constructed merely for illustration purposes). This molecule is represented as an attributed graph: each node stands for an atom and each edge for a bond between atoms. The nodes carry the chemical element of the corresponding atom as an attribute, the edges are associated with bond types. To the right of this molecule are two spanning trees for this molecule, both of which are rooted at the sulfur atom. Spanning tree edges are depicted as solid lines, edges closing cycles as dashed lines. Spanning tree A was built with depth-first search, spanning tree B with breadth-first search, and thus correspond to the two considered approaches.

If we adopt the precedence order S $\prec$ N $\prec$ O $\prec$ C for chemical elements (derived from the frequency of the elements in the molecule) and the order - $\prec$ = for the bond types, we obtain the two code words shown in Figure 2. It is easy to check that these two code words are actually minimal and thus are the canonical description w.r.t. a depth-first and breadth-first search spanning tree, respectively. In this case it is particularly simple to check this, because the root of the spanning tree is fixed, as there is only one sulfur atom.

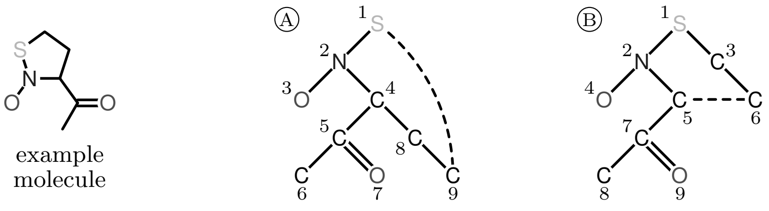


Fig. 1. An example fragment/molecule and two possible canonical forms: A – form based on a depth-first search tree, B – form based on a breadth-first search tree.

```
A:  S 21-N 32-O 42-C 54-C 65-C 75=O 84-C 98-C 91-S
B:  S 1-N2 1-C3 2-O4 2-C5 3-C6 5-C6 5-C7 7-C8 7=O9
```

Fig. 2. Code words describing the two canonical forms shown in Figure 1. Note that in form A the edges are sorted descendingly w.r.t. the second entries (i.e., i_s).

4 Restricted extensions

Up to now canonical descriptions were only used to test whether a search tree branch corresponding to a (sub)graph has to be descended into or not *after* the (sub)graph has been constructed. However, canonical forms can also be used to restrict the possible extensions directly. The idea is that for certain extensions one can see immediately that they lead to a code word that is not minimal. Hence one need not construct and test the (sub)graph, but can skip the extension right way. For the two special cases I consider here (depth-first and breadth-first search spanning trees), the allowed extensions are:

- Depth First Search: *Rightmost Extension* (Yan and Han (2002))

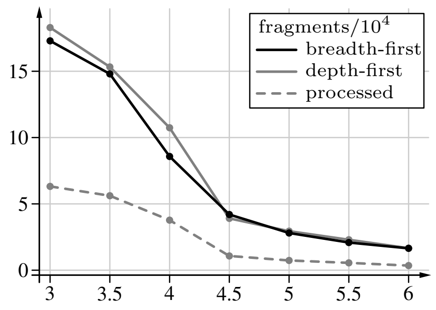
  Only nodes on the rightmost path of the spanning tree of the (sub)graph may be extended, and if the node is no leaf, it may be extended only by edges whose descriptions do not precede the description of the downward edge on the rightmost path. That is, the edge attribute must be no less than the attribute of the downward edge and if the edge attribute is identical, the attribute of its destination node must be no less than the attribute of the downward edge's destination node. Edges between two nodes that are already in the (sub)graph must lead from a node on the rightmost path to the rightmost leaf (that is, the deepest node on the rightmost path). In addition, the index of the source node of such an edge must precede the index of the source node of an edge already incident to the rightmost leaf.

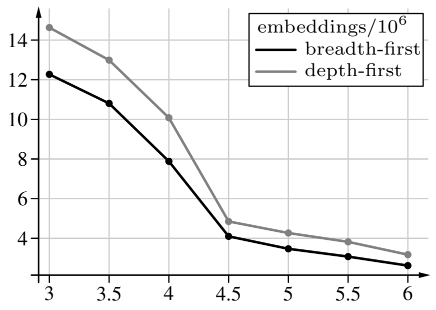
- Breadth First Search: *Maximum Source Extension* (Borgelt and Berthold (2002))

  Only nodes having an index no less than the maximum source index of an edge already in the (sub)graph may be extended.[3] If the node is the one having the maximum source index, it may be extended only by edges whose descriptions do not precede the description of any downward edge already incident to this node. That is, the attribute of the new edge must be no less than that of any downward edge, and if it is identical, the attribute of the new edge's destination node must be no less than the attribute of any corresponding downward edge's destination node (where corresponding means that the edge attribute is the same). Edges between two nodes already in the (sub)graph must start at an extendable node and must lead "forward", that is, to a node having a larger index.

In both cases it is easy to see that an extension violating the above rules leads to a (sub)graph description that is not in canonical form (that is, a numbering of the nodes of the (sub)graph that does not lead to the lexicographically smallest code word). This is easy to see, because there is a depth-first or breadth-first search numbering, respectively, *starting at the same root node*,

[3] Note that if the (sub)graph contains no edge, there can only be one node, and then, of course, this node may be extended without restriction.





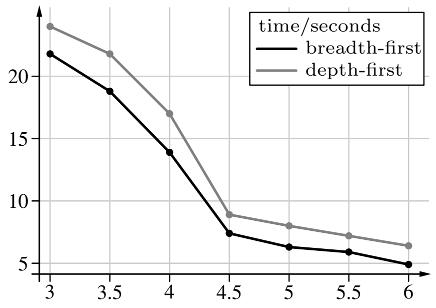


Fig. 3. Experimental results on the IC93 data. The horizontal axis shows the minimal support in percent. The curves show the number of generated and processed fragments (top left), number of generated embeddings (top right), and the execution time in seconds (bottom left) for the two canonical forms/extension strategies.

which leads to a lexicographically smaller code word (which may or may not be minimal itself—all that matters here is that it is smaller than the one derived from the current node numbering of the (sub)graph). In order to find such a smaller word, one only has to consider the extension edge. Up to this edge, the construction of the code word is identical, but when it is added, its description precedes (w.r.t. the defined precedence order) the description of the next edge in the code word of the unextended (sub)graph.

It is pleasing to see that the *maximum source extension*, which was originally introduced in the MoSS/MoFa algorithm based on heuristic arguments (Borgelt and Berthold (2002), Borgelt et al. (2005)), can thus nicely be justified and extended based on a canonical form.

As an illustration of these rules, consider again the two search trees shown in Figure 1. In the depth-first search tree A atoms 1, 2, 4, 8, and 9 are extendable (rightmost extension). On the other hand, in the breadth-first search tree B atoms 7, 8, and 9 are extendable (maximum source extension). Other restrictions are, for example, that the nitrogen atom in A may not be extended by a bond to another oxygen atom, or that atom 7 in B may not be extended by a single bond. Tree A may not be extended by an edge between two nodes already in the (sub)graph, because the edge $(1, 9)$ already has the smallest possible source (duplicate edges between nodes may be allowed, though). Tree B, however, may be extended by an edge between atoms 8 and 9.

5 Experimental results

In order to test the search tree pruning based on a breadth-first search canonical form, I extended the MoSS/MoFa implementation described in (Borgelt et al. (2005)). In order to compare the two canonical forms discussed in this paper, I also implemented a search based on rightmost extensions and a corresponding test for a depth-first search canonical form (that is, basically the gSpan algorithm (Yan and Han (2002)), with the exception of the minimal difference in the definition of the code word pointed out above). This was done in such a way that only the functions explicitly referring to the canonical form are exchanged (extension generation and comparison and canonical form check), so that on execution the two algorithms share a maximum of the program code.

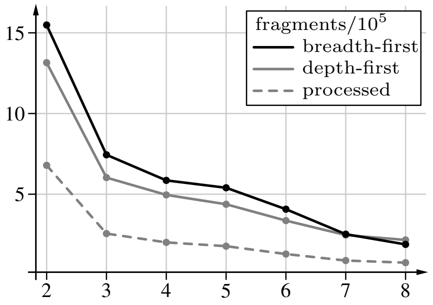
Figure 3 shows the results on the 1993 subset of the INDEX CHEMICUS (IC (1993)), Figure 4 the results on a data set consisting of 17 steroids. In all diagrams the grey solid line describes the results for a depth-first canonical form, the black solid line the results for a breadth-first canonical form. The diagram in the top left shows the number of generated and actually processed fragments (note that the latter, shown as a dashed grey line, is necessarily the same for both algorithms). The diagram in the top right shows the number of generated embeddings, the diagram in the bottom left the execution times.[4]

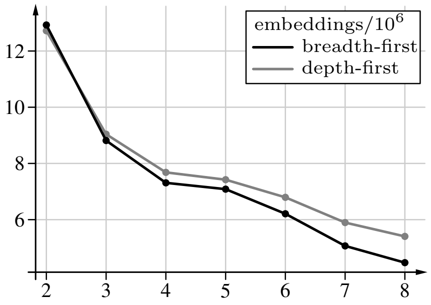
As can be seen from these diagrams, both canonical forms work very well (compared to an approach without canonical form pruning and an explicit removal of found duplicates, I observed speed-ups by factors between about 2.5 and more than 30). On the IC93 data the breadth-first search canonical form performs slightly better, needing about 10–15% less time. As the other diagrams show, this is mainly due to the lower numbers of fragments and embeddings that are generated. On the steroids data the depth-first search canonical form performs minimally better at low support values. Again this is due to a smaller number of generated fragments, which, however, is outweighed by a larger number of generated embeddings for higher support values.

6 Conclusions

In this paper I introduced a family of canonical forms of graphs that can be exploited to make frequent graph mining efficient. This family was obtained by generalizing the canonical form introduced in the gSpan algorithm (Yan and Han (2002)). While gSpan's canonical form is defined with a depth-first search tree, my definition allows for any systematic way of obtaining a spanning tree. To show that this generalization is useful, I considered a breadth-first search spanning tree, which turned out to be the implicit canonical form

[4] Experiments were done with Sun Java 1.5.0_01 on a Pentium 4C@2.6GHz system with 1GB main memory running S.u.S.E. Linux 9.3.





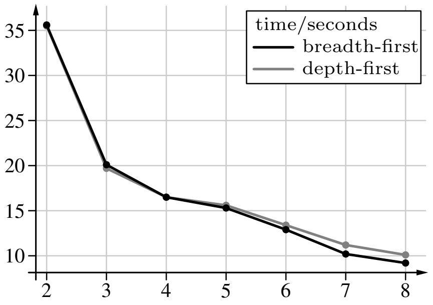


Fig. 4. Experimental results on the steroids data. The horizontal axis shows the absolute minimal support. The curves show the number of generated and processed fragments (top left), number of generated embeddings (top right), and the execution time in seconds (bottom left) for the two canonical forms/extension strategies.

underlying the MoSS/MoFa algorithm (Borgelt and Berthold (2002), Borgelt et al. (2005)). Exploiting this canonical form in MoSS/MoFa in the same way as the depth-first search canonical form is exploited in gSpan leads to a considerable speed up of this algorithm. It is pleasing to see that based on this generalized canonical form, gSpan and MoSS/MoFa can nicely be described in the same general framework, which also comprises a variety of other possibilities (an example alternative was pointed out above).

References

BORGELT, C. and BERTHOLD, M.R. (2002): Mining Molecular Fragments: Finding Relevant Substructures of Molecules. *Proc. 2nd IEEE Int. Conf. on Data Mining.* IEEE Press, Piscataway, 51–58.

BORGELT, C., MEINL, T. and BERTHOLD, M.R. (2004): Advanced Pruning Strategies to Speed Up Mining Closed Molecular Fragments. *Proc. IEEE Conf. on Systems, Man and Cybernetics, CD-ROM.* IEEE Press, Piscataway.

BORGELT, C., MEINL, T. and BERTHOLD, M.R. (2005): MoSS: A Program for Molecular Substructure Mining. *Proc. Open Source Data Mining Workshop.* ACM Press, New York, 6–15.

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

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.*

GOETHALS, B. and ZAKI, M. (2003/2004): Proc. 1st and 2nd IEEE ICDM Workshop on Frequent Itemset Mining Implementations. *CEUR Workshop Proceedings 90 and 126.* Sun SITE Central Europe and RWTH Aachen http://www.ceur-ws.org/Vol-90/, http://www.ceur-ws.org/Vol-126/.

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.* IEEE Press, Piscataway, 549–552.

INDEX CHEMICUS — Subset from 1993. Institute of Scientific Information, Inc. (ISI). Thomson Scientific, Philadelphia.

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.* ACM Press, New York, 136–143.

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

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.* ACM Press, New York, 647–652.

WASHIO, T. and MOTODA, H. (2003): State of the Art of Graph-based Data Mining. *SIGKDD Explorations Newsletter 5, 1, 59–68.*

YAN, X. and HAN, J. (2002): gSpan: Graph-based Substructure Pattern Mining. *Proc. 2nd IEEE Int. Conf. on Data Mining.* IEEE Press, Piscataway, 721–724.

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