MGTS 2005

Proceedings of the
3rd International Workshop on
Mining Graphs, Trees and Sequences

in conjunction with ECML/PKDD 2005

Siegfried Nijssen
Thorsten Meinl
George Karypis
(Eds.)

Porto, Portugal, 7th October 2005

Workshop Co-Chairs

Siegfried Nijssen
Leiden University, The Netherlands
snijssen@liacs.nl

Thorsten Meinl
Friedrich-Alexander University Erlangen-Nuremberg, Germany
meinl@cs.fau.de

George Karypis
University of Minnesota, USA
karypis@cs.umn.edu

Program Commitee

Lise Getoor, University of Maryland, USA

Tamas Horvath, Fraunhofer Inst. for Autonomous Intelligent Systems, Germany

Michael Berthold, University of Konstanz, Germany

Lawrence B. Holder, University of Texas at Arlington, USA

Joost N. Kok, Leiden University, The Netherlands

Bart Goethals, Universiteit Antwerpen, Belgium

Jan Ramon, Katholieke Universiteit Leuven, Belgium

Thomas Gaertner, Fraunhofer Inst. for Autonomous Intelligent Systems, Germany

Yun Chi, NEC Research Labs, Cupertino, CA, USA

Takashi Washio, Osaka University, Japan

Hiroki Arimura, Hokkaido University, Japan

Akihiro Inokuchi, IBM, Japan

Tsuyoshi Murata, National Institute of Informatics, Japan

Michihiro Kuramochi, University of Minnesota, USA

Peter Murray-Rust, Unilever Centre for Molecular Sciences Informatics, England

Additional Reviewers

Hendrik Stange

Lukas Molzberger

Preface

The workshop on Mining Graphs, Trees and Sequences (MGTS), of which you have the proceedings now in front of you, is this year already in its third edition, and, just like in previous years, the number of submissions suggests that active research on MGTS is still going on, all over the world. It has been a tradition of MGTS to accept only half of the papers. However, in this year's edition there was such a large number of interesting submissions, that we have decided to relax the acceptance rate by giving room to a larger number of short presentations. As a result, we now have a program of 9 papers, out of 15 submitted papers.

So what is MGTS all about? We believe that the current program reflects very well everything that MGTS is about. For many problems in machine learning and data mining, the traditional "attribute-value" or "item-set" representations are not sufficient, and more complex formalisms are required. Although there are very powerful formalisms, such as first order logic, that could be used to deal with complexly structured data, for reasons of scalability and accuracy, one may wish to employ formalisms that are not more complex than necessary. Graphs, trees and sequences are some of the most basic representations that are still sufficiently complex to deal with many problems, and have been studied extensively in many branches of computer science. MGTS researchers therefore study those problems where graph, tree or sequential representations strike exactly the correct balance between the complexity of the formalism and the mining problem that has to be solved; they study those problems which are typical to mining data that is represented as graph, tree or sequence.

A problem which has drawn attention from MGTS researchers from the start, is that of mining molecules. This is no surprise as molecule mining involves many of the problems relevant also to other branches of MGTS. Molecules are essentially graph structures, and therefore the choice to mine their graph representation directly is very natural. The papers of Christian Borgelt, and of Katharina Jahn and Stefan Kramer show that molecule mining is still actively studied. The target of their work is to deepen our understanding of the computational complexity of the learning problems. Indeed, one of the interesting properties of graph mining is that it involves complexity problems that have been studied in other branches of computer science: when mining graphs, one needs a mechanism to find matchings of subgraphs in graphs (subgraph isomorphism or homomorphism) and one needs to traverse a search space (which may involve graph isomorphism). While Christian Borgelt's paper provides a better understanding in the problem of graph isomorphism, Katharina Jahn and Stefan Kramer's paper study the problem of subgraph isomorphism more closely.

Graphs are however not only of interests for molecular applications. The papers of Alex Markov and Mark Last, and of Lukas Molzberger study applications that have not yet been studied extensively in the MGTS context: Alex Markov and Mark Last study the representation of (WEB) documents as graphs,

with the purpose of getting better classifiers; Lukas Molzberger studies graph representations for learning natural languages. Of interest is how in both applications, which start from sequential data, graph representations are employed to try to match the real, underlying complexity of the learning problem better. This approach is becoming popular: it is also employed in the work Stéphanie Jacquemont et al., where probabilistic automata are used to represent sequential data and sequences are mined from these automata. Another essential part of the work of Stéphanie Jacquemont et al. is to find patterns with statistical relevance, and also this problem turns out to be popular: a similar problem is studied by Edgar de Graaf and Walter Kosters, who optimize the discovery of discriminating patterns by exploiting background knowledge.

The third part of the study of Stéphanie Jacquemont et al. involves the use of constraints. Constraint-based mining is also considered by Jeroen De Knijf, but this time in the context of frequent tree mining. Jeroen De Knijf introduces several optimizations for mining frequent trees under constraints. Trees are of interest as they provide more expressiveness than sequences, but on the other hand many efficient algorithms for them are known, which makes tree-based problems easier to solve than graph-based problems. This may be useful for applications in WEB mining.

A further study of tree mining is performed by Alexandre Termier et al.. In comparison with de Knijf's work, Alexandre Termier et al. limit their patterns to attribute trees, which are even more restricted trees. Alexandre Termier et al. study optimizations to speed-up the discovery of such frequent trees.

What all these papers share, is that they mostly consider large collections of small(er) graphs, trees, or sequences. Equally relevant are however problems in which the data consists of one large graph, tree or sequence. The most famous example here is the web graph, which consists of webpages and their links. While most algorithms for mining graphs, trees and sequences have concentrated on small graphs, it is of interest how they would apply to large graphs, and to study how modifications of these algorithms may compare to more special algorithms for mining large graphs. Such algorithms are discussed by Tayfun Gürel and Kristian Kersting, who study the problem of classification in large graphs, and distinguish several types of algorithms.

Overall we are convinced that the workshops covers all topics that are of relevance to MGTS, among others: efficiency issues of algorithms, issues of how to represent data, issues of mining under constraints, and issues of how to learn in graph representations. It is our hope that the presence of researchers working on so many related topics will spark some interesting discussions at the workshop.

Program Committee Co-chairs
Siegfried Nijssen, Thorsten Meinl, George Karypis.

Table of Contents

Full Papers

On Canonical Forms for Frequent Graph Mining 1
Christian Borgelt (Otto-von-Guericke-University of Magdeburg)

Using a Probable Time Window for Efficient Pattern Mining in a Receptor Database 13
Edgar H. de Graaf and Walter A. Kosters (Leiden University)

On the Trade-Off Between Iterative Classification and Collective Classification: First Experimental Results 25
Tayfun Gürel and Kristian Kersting (University of Freiburg)

Mining Probabilistic Automata: a New Way to Sequence Mining 37
Stéphanie Jacquemont, Francois Jacquenet, and Marc Sebban (Université Jean Monnet de Saint-Etienne)

Efficient Graph-Based Representation of Web Documents 51
Alex Markov and Mark Last (Ben-Gurion University of Negev)

Computation-time efficient and robust attribute tree mining with DryadeParent 63
Alexandre Termier, Marie-Christine Rousset, Michèle Sebag, Kouzou Ohara, Takashi Washio, and Hiroshi Motoda (Osaka University and Université Paris-Sud)

Short Papers

Optimizing gSpan for Molecular Datasets 77
Katharina Jahn and Stefan Kramer (Technische Universität München)

Frequent Tree Mining with Selection Constraints 89
Jeroen De Knijf (Utrecht University)

A Graph-Based Rule-Mining Framework for Natural Language Learning and Understanding 103
Lukas Molzberger (Fraunhofer Institut Autonome Intelligente Systeme)

Author Index 118

On Canonical Forms for Frequent Graph Mining

Christian Borgelt

Dept. of Knowledge Processing and Language Engineering
Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany
borgelt@iws.cs.uni-magdeburg.de

Abstract. In approaches to frequent graph mining that are based on growing subgraphs into a set of graphs, one of the core problems is how to avoid redundant search. A powerful technique to overcome this problem is a canonical description of a graph, which uniquely identifies it, and a corresponding test. This paper introduces a family of canonical forms that are based on systematic ways to construct spanning trees. I show that the canonical form used in gSpan [14] is a member of this family, and that MoSS/MoFa [1, 3] is implicitly based on a different member, which I make explicit and exploit in the same way as in gSpan.

1 Introduction

In recent years there emerged 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 the graph structure by logical expressions [5]. However, the vast majority transfers techniques developed originally for frequent item set mining.[1] Examples of algorithms developed in this way include MolFea [10], FSG [11], MoSS/MoFa [1], gSpan [14], CloseGraph [15], FFSM [8], and Gaston [12]. A related 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.

While in frequent item set mining it is trivial to ensure that the same item set is checked no more than once in the search (using an arbitrary, but fixed global order of the items), in frequent subgraph mining it is one of the core problems how to avoid redundant search. Since the same subgraph can be grown in several different ways, adding the same nodes and edges in different orders, it is difficult to guarantee that each subgraph is considered only once. Although such multiple tests of the same subgraph do not invalidate the result, they can be devastating for the execution time of the algorithm. Therefore methods that rule out such redundant search are very important to make the algorithms efficient.

[1] See, for example, [6, 7] for details and references on frequent item set mining.

One of the most promising ideas 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 [14] and its extension CloseGraph [15]. In this paper I generalize the canonical form introduced in gSpan, thus arriving at a family of canonical descriptions of which the one used in gSpan is a special case. However, it is not the only usable one. I show that a competing algorithm called MoSS/MoFa [1, 3] is implicitly also based on a canonical description from this family, 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

For the following considerations it is important to review how a graph database is searched for frequent subgraphs: 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 requires that at least one node incident to an added edge must already be in the subgraph, thus restricting 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 by an edge and, if necessary, a node. (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 step 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.

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 the family of canonical forms introduced in this paper 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, in particular which nodes they connect. If the graph is attributed or directed, they also comprise information about the attribute and/or direction of the edge as well as the attributes of the incident nodes.

While it is straightforward to capture the latter information about an edge (i.e. attributes and edge direction) in some alphabet, how to describe the connection structure is not so 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, there are many different ways of numbering the nodes of a graph, all of which may lead to different code words, because each numbering leads to a different specification 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 two subsections: 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 about the form of such a code word are given in Section 3.4). The resulting list of code words is sorted lexicographically. Then 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 node numberings will ever occur in the search. Even more: we can confine ourselves to a specific systematic way of constructing a spanning tree. 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.

The most straightforward systematic methods for constructing a spanning tree of a graph are, of course, depth-first and breadth-first search. Thus it is not surprising to see that gSpan uses the former to define its canonical form [14]. 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 [1, 3] can be justified from a canonical form that is based on such a breadth-first search tree. Thus MoSS/MoFa can be seen as implicitly based on this canonical form.

Other alternatives 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 search 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 is a variety of other possibilities one may explore.

[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.

It should be noted that at this point 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 there are usually several spanning trees of a (sub)graph that are 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 ([1] 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 of the graph, we can set up the edge descriptions and sort them. In principle, the edge descriptions can be sorted using any precedence order of the properties of an edge (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 the search constructs 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 the same (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 by the insight that 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 have been added to the (sub)graph in the search. This has two advantages: in the first place, we need no sorting to obtain the code word of the (sub)graph under consideration 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. Secondly, 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 considered here are defined.

3.4 Code Words

In my definition of a code word I deviate slightly from the definition of gSpan [14], where code words are simple lists of edge descriptions, each of which comprises all information about the edge and the incident nodes. This deviation is triggered by the insight 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: $a\,(i_d\,[n-i_s]\,b\,a)^m$
- Breadth-First Search: $a\,(i_s\,b\,a\,i_d)^m$

In these words a is a node attribute and b an edge attribute. n is the total number of nodes in the (sub)graph and m the total number of edges. i_s is the index of the source node 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. Square brackets indicate one character, the value of which is computed by the expression inside them. That is, in the depth-first search code word the second character for each edge is the number $n - i_s$.

The describing properties of an edge are compared in the order in which they appear in the parenthesized expressions. All characters are compared ascendingly. (This explains the expression in square brackets; alternatively one may define that i_s has to be compared descendingly here.) Note that the parenthesized expressions are sorted (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.

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 have been listed, the other edges (which lead to cycles) are listed. In my definition, however, there is 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 [12] 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 formally stated. 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 $e.a$, which again I assume 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: array of int, G: graph) : boolean;
var v : node;                     (* to traverse the nodes of the graph *)
    e : edge;                     (* to traverse the edges of the graph *)
    x : array of node;            (* to collect the numbered nodes *)
begin
  forall v ∈ G.V do               (* traverse all nodes and *)
    v.i := −1;                    (* clear their indices *)
  forall e ∈ G.E do               (* traverse all edges and *)
    e.i := −1;                    (* clear their markers *)
  forall v ∈ G.V do begin         (* traverse the potential root nodes *)
    if v.a = w[0] then begin      (* if v is acceptable as a root node *)
      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 and *)
      then return false;          (* abort if a smaller code word is found *)
      v.i := −1;                  (* clear the node index again *)
    end
  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 a breadth-first search. However, for a depth-first search the function can be implemented in a very similar way.

The basic idea of the recursion 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 immediately. 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: array of int, k : int, x: array of node, 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 node *)
    r : boolean;                       (* buffer for a recursion result *)
begin
  if k ≥ length(w) return true;        (* full code word has been generated *)
  while i < w[k] do begin              (* check whether there is an edge with *)
    forall e incident to x[i] do       (* a 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 begin              (* traverse the unvisited 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
  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 has been constructed merely for illustration purposes). This molecule is represented as an attributed graph, in which 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. The spanning tree edges are depicted as solid lines, the 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.

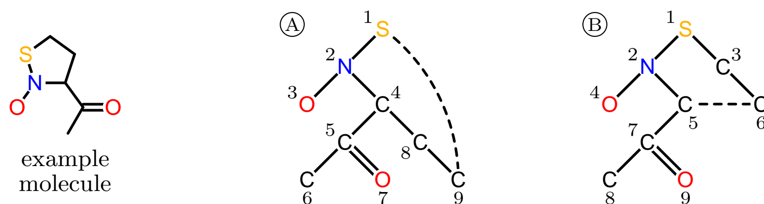


Fig. 1. An example fragment/molecule and two possible canonical forms: A – form based on a depth first search tree, B – form based on a breadth first search tree.

```
A:  S 28-N 37-O 47-C 55-C 64-C 74=O 85-C 91-C 98-S
       1    2    2    4    5    5    4    8    1

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. The second entries in the bond strings in A are $n - i_s$, where n is the number of nodes and i_s the index of the source node. The actual values of i_s are shown below the code.

If we adopt the precedence order $\mathsf{S} \prec \mathsf{N} \prec \mathsf{O} \prec \mathsf{C}$ for chemical elements (which is 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 represent 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.

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 by an edge and maybe a node 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* [14]
  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* [1]

  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*, 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 the description of the next edge in the code word of the unextended (sub)graph.

It is pleasant to see that the *maximum source extension*, which was originally introduced in the MoSS/MoFa algorithm based on heuristic arguments [1, 3], can thus nicely be justified and extended based on a canonical form.

As an illustration of the above extension 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.

[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.

5 Experimental Results

In order to test the pruning based on a breadth-first search canonical form, I extended the MoSS/MoFa implementation described in [3]. 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 [14]). 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 (IC93) [9], Figure 4 shows 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 [14]. 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 underlying the MoSS/MoFa algorithm [1, 3]. 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 pleasant 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.

[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.

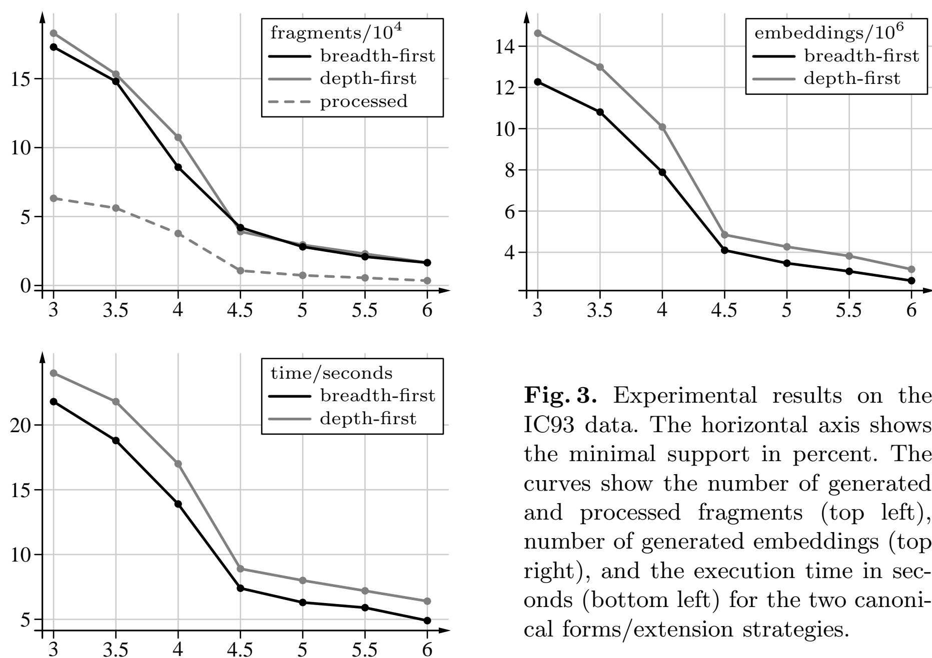


Fig. 3. Experimental results on the IC93 data. The horizontal axis shows the minimal support in percent. The curves show the number of generated and processed fragments (top left), number of generated embeddings (top right), and the execution time in seconds (bottom left) for the two canonical forms/extension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

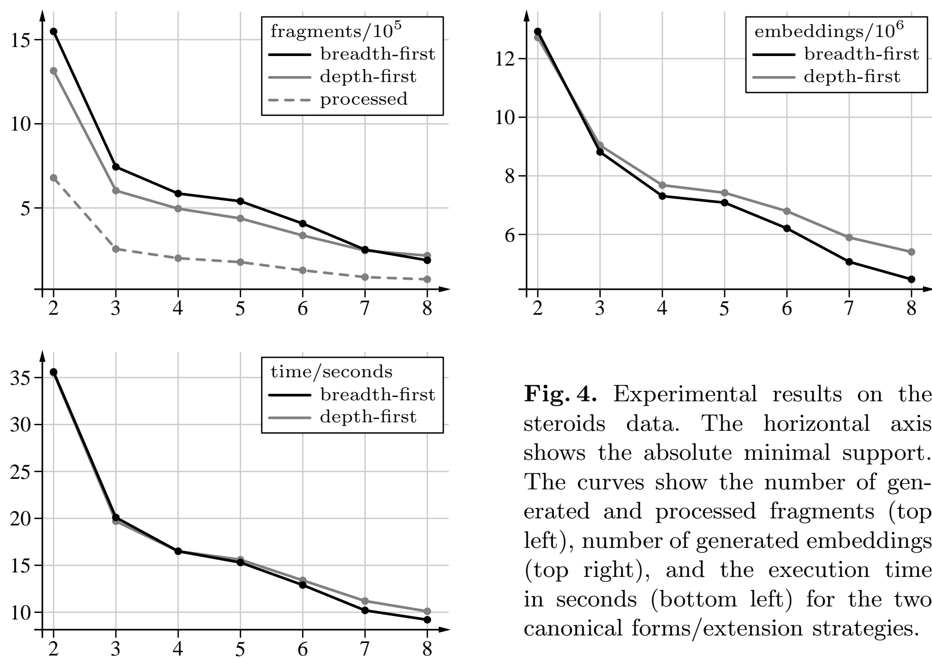


Fig. 4. Experimental results on the steroids data. The horizontal axis shows the absolute minimal support. The curves show the number of generated and processed fragments (top left), number of generated embeddings (top right), and the execution time in seconds (bottom left) for the two canonical forms/extension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

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, T. Meinl, and M.R. Berthold. MoSS: A Program for Molecular Substructure Mining. *Proc. Open Source Data Mining Workshop (OSDM 2005, at KDD2005, Chicago, IL)*, to appear.
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. 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
6. B. Goethals and M. Zaki. *Proc. 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003.
http://www.ceur-ws.org/Vol-90/
7. B. Goethals and M. Zaki. *Proc. 2nd IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 126, Aachen, Germany 2003.
http://www.ceur-ws.org/Vol-126/
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
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. 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
13. 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
14. 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
15. 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

Using a Probable Time Window for Efficient Pattern Mining in a Receptor Database*

Edgar H. de Graaf and Walter A. Kosters

Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands
{edegraaf,kosters}@liacs.nl

Abstract. The analysis of sequences is one of the major research areas of bio-informatics. Inspired by this research, we investigate the discovery of sequential patterns for use in classification. We will define variations of a fit function that enables us to tell if one pattern is "better" than another. Furthermore we will show how domain knowledge can be used for faster discovery of better sequential patterns in specific types of databases, in our case a receptor database.

1 Introduction

Sequence analysis has many application areas, e.g., protein sequence analysis and customer behavior analysis. We investigate extraction of features for protein sequence classification where features are *sequential patterns*: ordered lists of items (for proteins the items are amino acids). As a motivating example, we would like to know if a protein sequence, an ordered list of amino acids, belongs to the Olfactory family or not, where the Olfactory family is a group of proteins than deals with smell. We focus on a special group of proteins called GPCRs. These G-protein-coupled receptors (GPCRs) play fundamental roles in regulating the activity of virtually every body cell [17]. Usually classification is done unsupervised using alignment, however in the case of GPCRs this turned out to be difficult. Fortunately, we know for some protein sequences whether they are of the Olfactory family or not. These sequences can thus be divided into two disjoint classes: Olfactory and No-olfactory, and from these classes we can extract sequential patterns to be used as attributes in a classification algorithm (as is being proposed in [12]). The question we try to answer in this paper is which sequential patterns are *best* used as features/patterns? And how can domain knowledge be used to improve the search for such patterns?

Classification based on sequential patterns is also applicable in many other areas. For example, in the case of customer behavior analysis, we might want to characterize groups of clients based on sequential patterns in their behavior.

The "best" sequential patterns are discovered through a function that judges patterns. In Section 2 we will discuss different instances of this function and

* This research is carried out within the Netherlands Organization for Scientific Research (NWO) MISTA Project (grant no. 612.066.304).

select one for our purposes. Section 3 adapts the PrefixSpan algorithm of [15] to deal with this function. In addition, a pruning strategy is introduced in Section 4, increasing efficiency by first searching in a certain area of the sequence, the *probable time window*. Section 4 also describes how preferring small patterns can further increase classification performance. The effectiveness of these improvements will be shown in Section 5. Earlier work on this subject has been presented at BNAIC2005 [7].

Related work. Our algorithms will be based on the pattern growth approach called PrefixSpan proposed in [15]. Classification by means of patterns has been done before but not so much in the sequence domain. We now mention related work in the non-sequence domain. Apriori-C [9] constructs classification rules by extending the Apriori algorithm [2, 3]. Apriori-C discovers a large number of rules from which a fixed number of rules with the highest support are selected. Apriori-SD [10] solves the problem of selecting the right rules with *subgroup discovery.* This algorithm selects a subgroup of rules by calculating their *weighted relative accuracy.* This means that the probability of a pattern occurring in a class is compared with the probability of its occurrence outside the class. This is weighted with the probability of a class. Most class association rule mining algorithms work with unordered sets of items frequently occurring together in *item sets.* Classification with association rules is presented in [13] and [14]. Furthermore CorClass [18] describes an algorithm that also works with item sets. It introduces a new method of pruning. Specialized rules are only added if the upper bound of its correlation is higher than the minimal correlation of k rules. In our work we use a similar method of pruning. Much work has been done in the field of molecular feature mining, e.g., the MolFea algorithm described in [11]. MolFea employs a level-wise version space algorithm to discover those molecule fragments often occurring in one data set and less often in another. Finally some other researchers try to use domain knowledge to speed up the search for frequent patterns, e.g., the Carpenter algorithm presented in [5]. In this work the authors perform row enumeration instead of the standard column enumeration done in Apriori-like algorithms. This is done because biological data sets often have many columns/items and only a few rows.

2 The Maximal Discriminating Patterns

One would like to select the best patterns for use as attributes in a classification algorithm. But how can we tell if one pattern is better than the other? In this section we will first explain the notion of support and why it is less useful for selecting the best pattern. Next we introduce the notion of confidence which will give more useful patterns, but it also has disadvantages. Finally we will discuss and motivate so-called maximal discriminating patterns, enabling us to have patterns specific to one class, but without the disadvantages of confidence.

Assume given a database D with $D = D_1 \cup D_2 \cup \ldots \cup D_c$, with c classes. The D_i's ($1 \leq i \leq c$) are mutually disjoint and not empty.

Each record in the database is a non-empty finite *sequence* (i.e., an ordered list) of items from the set $\Sigma = \{\texttt{A},\texttt{B},\texttt{C},\ldots\}$, e.g., (C,B,G,A,A,A,C,B). Now fit_0 is defined as support (as used in association rule mining algorithms like APRIORI [3]), because support can be seen as a measure of how well a pattern fits the data. Commonly a sequence d is said to support a *pattern* s if the pattern is contained (in the set sense) in the sequence:

$$\mathit{supp}_0(s,d) = \begin{cases} 1 & \text{if for all } i \ (1 \leq i \leq k) \text{ there is a } j \ (1 \leq j \leq \ell) \text{ with } s_i = d_j; \\ 0 & \text{otherwise,} \end{cases}$$

for $s = (s_1, s_2, \ldots, s_k)$ and $d = (d_1, d_2, \ldots, d_\ell)$. This means that s is a *subset* of d. We then can define fit_0:

$$\mathit{fit}_0(s, D_i) = \frac{1}{|D_i|} \sum_{d \in D_i} \mathit{supp}_0(s,d)$$

$(1 \leq i \leq c)$, where s is a pattern.

We now specialize support to sequences. A sequence $d = (d_1, d_2, \ldots, d_m)$ is called a *super-sequence* of a sequence $s = (s_1, s_2, \ldots, s_k)$ if $k \leq m$ and for each s_i $(1 \leq i \leq k)$ there is a d_{j_i} $(1 \leq j_i \leq m)$ with $s_i = d_{j_i}$ and $j_{i-1} < j_i$ $(i > 1)$. We denote this with $s \prec d$. The sequence s is called a *sub-sequence* of d. This defines *sequential patterns* on sequences of items. (Another definition of sequential patterns was given by Agrawal et al. in [3], in which they define sequential patterns on sequences of item sets). We now let

$$\mathit{supp}_1(s,d) = \begin{cases} 1 & \text{if } s \prec d; \\ 0 & \text{otherwise,} \end{cases}$$

and define fit_1 in the same way as fit_0 was defined using supp_0.

Now fit_1 or fit_0 by itself is not useful for selection of features for classification. One of the patterns of size 1 will always have the highest fit and these small patterns are probably often present in more than one D_i. Thus the presence of such a pattern will not give a good distinction between classes.

The next most logical step is to use confidence to select the best patterns. The patterns x_r $(1 \leq r \leq c)$, one for each class, are then chosen to maximize *confidence* $(\mathit{fit}_1(x_r, D_r)|D_r|)/(\mathit{fit}_1(x_r, D)|D|)$. The class t of sequence s is the t $(1 \leq t \leq c)$ where $x_t \prec s$. If more than one t is possible we select based on the highest confidence. One is selected at random if more than one class t has a pattern with the highest confidence. If there is no t where $x_t \prec s$ then the sequence could be said to be "undecided".

A problem is that we only pick one pattern per class. This is plausible if a family of a sequence is only decided by one sequence of features. However, it is often the case that the class of a sequence is decided by multiple patterns. Moreover there can be constraints on the pattern. This means that the class deciding pattern x_t *with* the constraint is not necessarily equal to the x_t *without* the constraint. As a consequence it is usually possible to find a combination of patterns with a better classification performance. Finally it is possible that a

single sequential pattern x_t is equal for two or more classes, and as a consequence a classification will be done at random. This problem will occur with a lower probability if we use multiple patterns for each class.

Another major drawback of the confidence method is that the size of the D_i's seriously influences the classification. E.g., assume we have databases D_1 and D_2. Furthermore assume D_1 contains 500 sequences and D_2 only 100. The pattern p_1 occurs 100 times in D_2 and 60 times in D_1, thus a confidence with respect to D_2 of 0.625. Another pattern p_2 occurs 70 times in D_2 and 10 times in D_1, giving a confidence of 0.875. The pattern p_2 will be used for classification if no other pattern has a higher confidence. However p_1 occurs in every sequence of D_2 and only in a small percentage of the sequences in D_1. One could argue that p_1 should be preferred over p_2.

Therefore we define fit_2, which we use in the sequel. For a pattern s and $1 \leq q, r \leq c$ we define $\delta(s, D_q, D_r) = \mathit{fit}_1(s, D_q) - \mathit{fit}_1(s, D_r)$, and we let $\mathit{fit}_2(s, D_r) = \min\{\delta(s, D_r, D_q) \mid 1 \leq q \leq c \ \wedge \ q \neq r\}$. We then choose patterns x_r $(1 \leq r \leq c)$ with maximal $\mathit{fit}_2(x_r, D_r)$. We can then use them to classify sequences as before, without the drawbacks mentioned above. We will usually find those patterns that are characteristic for one class. With *characteristic* we mean that fit_1 will have a high value in D_t and a lower value in the other $D_i{}'s$, $i \neq t$.

Our new fit has some similarities with the concept of *emerging patterns* presented in [4] and [6]. In order to discover emerging patterns patterns are preferred where the ratio $\mathit{fit}_1(s, D_1)/\mathit{fit}_1(s, D_2)$ is the highest, where D_1 and D_2 are two databases each containing one class of sequences. Bailey et al. [4] further investigate jumping emerging patterns. These are patterns that have a support of zero in D_2 and a non-zero support in D_1. Emerging patterns can also be defined in a way similar to fit_2, but now using $\mathit{fit}_1(s, D_q)/\mathit{fit}_1(s, D_r)$ instead of $\delta(s, D_q, D_r)$. Dong et al. [6] point out that the growth rate measure used by emerging patterns doesn't take into account the coverage, a problem they solve with a score function. However in the case of fit_2 coverage is less of a problem, a pattern with a low $\mathit{fit}_1(s, D_1)$ is less likely to have a high fit_2 value. Also the fit_2 measure allows us to more easily explain and implement the pruning rules that will be discussed in the remainder of this paper.

Classification algorithms usually need a limited number of attributes. In order to classify a sequence s we use a finite number of n sequential patterns $p_1^t, p_2^t, \ldots, p_n^t$ per class t, where $\mathit{fit}_2(p_1^t, D_t) \geq \mathit{fit}_2(p_2^t, D_t) \geq \ldots \geq \mathit{fit}_2(p_n^t, D_t)$ and p_n^t has the n-th highest fit_2 for all possible patterns. These patterns, the so-called *maximal discriminating patterns*, could be used by any classification algorithm when we first convert each sequence to a vector indicating for each pattern if it is contained in the sequence, see [12]. However it is possible that, e.g., p_1^t is supported by all or most of the sequences supporting p_2^t. Thus p_2^t might not improve classification. This problem could be solved by removing all sequences containing p_1^t from D_t. The algorithm for searching the sequence with maximal fit is then again applied to this subset of D_t in order to find p_2^t. In this paper we do not further focus on the precise classification performance, but

rather on the discovery of the discriminating patterns. Our algorithm aims at finding the set $P = P^t$ of maximal discriminating patterns.

3 Algorithm without Domain Knowledge

Our pattern search algorithm, coined PrefixTWEAC (Time Window Exploration And Cutting), is based on PrefixSpan. The algorithm does not generate candidates, but it grows patterns from smaller patterns. This principle makes it faster than most Apriori like algorithms [15]. PrefixSpan is a depth first algorithm, which will be explained in more detail in Section 4 when we adapt this algorithm to our current needs (see Algorithms 1 and 2). PrefixSpan as described in [15] searches for those patterns with *support* larger than or equal to a given support threshold *minsupp*, where support is defined as fit_1. The algorithm starts with all frequent sub-sequences of size one. For each sub-sequence a projected database is created. These frequent sub-sequences are extended to all frequent sub-sequences of size two by only looking in the projected database. This *projected database* is a database of pointers to the first item occurring after the current pattern, also called the *prefix*. A sequence is only in the projected database if it contains the prefix. Again for each frequent sub-sequence of size two a corresponding projected database is created. This process continues recursively until no extension is frequent anymore.

PrefixTWEAC (Algorithm 1) is different from PrefixSpan in that it searches for the maximal fit_2 instead of the maximal support fit_1. The function fit_2 is by definition not anti-monotone (so $fit_2(s_1, D_t) > fit_2(s_2, D_t)$ might happen, where s_1 is a super-sequence of s_2). However the anti-monotone property for fit_1 can still be used in two ways, when looking for the one pattern with maximal fit_2. First of all in PrefixTWEAC we only examine an extended pattern p if $fit_1(p, D_t) \geq minsupp$ where *minsupp* is the support threshold. Secondly p is not further examined if $fit_1(p, D_t) <$ *current n-th maximal fit*, where *current n-th maximal fit* is the current n-th best fit of all patterns found while searching. The value of $fit_2(p, D_t)$ will never become larger than the current n-th maximal fit, because it can at most become $fit_1(p, D_t)$. Note that CorClass uses similar methods to prune [18].

4 Domain Specific Improvements

In the previous section we stated that fit_2 can be used to "prune": certain pattern extensions are not further examined because they can never lead to the maximal fit_2. The faster we get to a large fit_2 for the n-th pattern in $P = P^t$ the better, because all extensions with a lower $fit_1(p, D_t)$ can be pruned. The improved version of PrefixTWEAC will be explained in the sequel.

If we consider protein sequences then pattern discovery might be done faster and/or classification might improve when using certain knowledge about the sequences:

Algorithm 1 The PrefixTWEAC algorithm

PrefixTWEACCore(prefix, projected_database)
1. For all items i that can extend the prefix
2. new_prefix = prefix extended with item i
3. Count $w_1 = \mathit{fit}_1$ in the projected_database$_t$ for new_prefix
4. Calculate $f_2 = \mathit{fit}_2$ for new_prefix
5. Create a projected database new_projected_database with new_prefix
6. Get δ_{min}, the lowest fit_2 in P
7. Get s_{min}, fit_1 corresponding with the lowest fit_2 in P
8. **if** $w_1 \geq \mathit{minsupp}$ **and** $|P| < n$ **then**
9. Add new_prefix to P
10. Call PrefixTWEACCore(new_prefix, new_projected_database)
11. **else if** $w_1 \geq \mathit{minsupp}$ **and** $w_1 \geq \delta_{min}$ **then**
12. **if** $f_2 > \delta_{min}$ **or**
13. ($f_2 = \delta_{min}$ **and** $w_1 > s_{min}$) **or**
14. ($f_2 = \delta_{min}$ **and** $w_1 = s_{min}$ **and** new_prefix $\prec p_n$) **then**
15. Remove p_n from P and add new_prefix to P
16. Call PrefixTWEACCore(new_prefix, new_projected_database)

- Protein sequences are sequences of amino acids. Certain parts of such a sequence are shaped like a helix in 3D space. These helices will probably contain most of the maximal fitting sequences since parts outside the helix have more variation in size and content. Patterns (partially) outside the helix are less likely to occur in most members of the protein family.
- Small patterns are preferred. Smaller patterns are less specific and biologists prefer smaller patterns in their analysis.

For certain problems we know the approximate area of important features, e.g., protein sequences should have most of the discriminating patterns in the helix. Also in other problems this might be the case, for example — in the case of customer relations — customers tend to behave differently during the night. These *probable time windows* can easily be defined with an *inclusion vector*. An inclusion vector is a vector $v = (v_1, v_2, \ldots, v_n)$, $v_i \in \{0, 1\}$ $(1 \leq i \leq n)$. This vector will indicate where to search in the first phase of the algorithm, see Algorithm 2. We then let

$$\mathit{supp}_1^{\mathrm{PTW}}(s, d) = \begin{cases} 1 & \text{if } s \prec d/v; \\ 0 & \text{otherwise,} \end{cases}$$

where $(d/v)_i = d_i$ if $v_i = 1$ and \$ otherwise (\$ $\notin \Sigma$), so only positions with nonzero v_i are considered.

First PrefixTWEACExt (Algorihtm 2) is applied to the databases D_t, one at a time, each time starting with an empty $P = P^t$. After using PrefixTWEACExt with the inclusion vector we apply PrefixTWEAC (Algorithm 1) without the vector to the remaining states stored in the state database S.

Algorithm 2 PrefixTWEAC Extended: extension using the probable time window

PrefixTWEACExt(prefix, projected_database)
1. For all items i that can extend the prefix
2. new_prefix = prefix extended with item i
3. Count fit_1 for new_prefix:
4. $w_1 = fit_1$ in the projected_database$_t$ without the inclusion vector, using $supp_1$
5. $w_2 = fit_1$ in the projected_database$_t$ with the inclusion vector, using $supp_1^{\mathrm{PTW}}$
6. Calculate $f_2 = fit_2$ for new_prefix (without the inclusion vector)
7. Create new_projected_database (without using the inclusion vector)
8. Get δ_{min}, the lowest fit_2 in P
9. Get s_{min}, fit_1 of the lowest fit_2 in P
10. **if** $w_1 \geq minsupp$ **and** $|P| < n$ **then**
11. Add new_prefix to P
12. Call PrefixTWEACExt(new_prefix, new_projected_database)
13. **else if** $(w_1 \geq minsupp$ **and** $w_2 < minsupp)$ **or**
14. $(w_2 \geq minsupp$ **and** $w_1 \geq \delta_{min}$ **and** $w_2 < \delta_{min})$ **then**
15. storeState(S,new_prefix, new_projected_database)
16. **else if** $w_2 \geq minsupp$ **and** $w_2 \geq \delta_{min}$ **then**
17. **if** $f_2 > \delta_{min}$ **or**
18. $(f_2 = \delta_{min}$ **and** $w_1 > s_{min})$ **or**
19. $(f_2 = \delta_{min}$ **and** $w_1 = s_{min}$ **and** new_prefix $\prec p_n)$ **then**
20. Replace p_n with new_prefix
21. Call PrefixTWEACExt(new_prefix, new_projected_database)

Figure 1 shows an example of the extensions made to a sequence `A`. The dotted lines are extensions that do not have a high enough *fit1* and *fit2* inside and outside the probable time window. These extensions and their extensions are pruned. The dashed lines indicate extensions that are currently good enough with regards to the entire sequence only. Finally the solid lines are already good enough when we only count patterns inside the probable time window.

If we prefer small patterns, then we can add a new rules, using so-called *smallest maximal discriminating patterns* to improve classification:

- $fit_1(s, D_r) = 0$ for all r $(1 \leq r \leq n, r \neq t)$. Then fit_2 of the extended patterns will never increase.
- $fit_1(s, D_t) \leq fit_2(p, D_t)$ where both p and s are sequences and s is created by extending p. Then fit_2 of the extended patterns will never be better than the fit_2 of p.

These rules sacrifice some completeness for classification performance; if extensions do not improve a smaller pattern then they are not always explored further. These pruning rules will not lower classification performance because they leave out only non-improving extensions. Rather the classification is expected to improve because the set of patterns will contain less small variations of the same

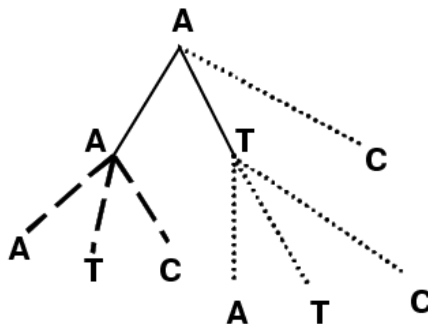


Fig. 1. Extending the single item sequence A

pattern. We will from now on abbreviate the use of these rules with SP or "small patterns".

Protein sequences usually are very long, about 300 amino acids. However these sequences are constructed out of only 20 types of amino acids. We need to use constraints to make the problem tractable. It was chosen to use the time window constraint, because the discovered patterns will be concentrated in one area. The *time window constraint* means that the distance between the first and last item of the pattern in the sequence is bounded by some constant. This is easily implemented in the algorithm used. It was also considered to use the *gap constraint* [1], that allows some gaps in the matches. However this constraint would have required more memory, e.g., if we count fit_1 of (A,C,G) and we want to know whether the sequence (A,C,C,C,G) contains it. Furthermore assume the maximal gap is 1, thus in the sequence one letter is allowed between two letters of the pattern. If the algorithm only looks at the first C then the gap constraint will be broken because the gap between the C and the G is 2. An algorithm has to check two C's to match (A,C,G). PrefixSpan will have to add both projections to the projected database for at least two C's. One other reason for not using the gap constraint is that it would allow patterns to be spread all over the sequence as long as it doesn't break the gap constraint.

5 Experimental Results

The experiments are aimed at showing the effectiveness of the pruning rules we described. The protein sequences used during our experiments where extracted from the GPCRDB website [8]. The effectiveness was also tested on a synthetic data set: the two classes consist of 1000 sequences of length 130, having 20 item types. First each item is chosen with a uniform probability and then we insert one of ten patterns at each starting position within the time window (position 20 to 60) of class one with 80% probability.

The results are shown in Figure 2 and Figure 3. All experiments were done on a Pentium 4 2.8 GHz with 512MB RAM. On the horizontal axis in the graphs we have the number of used sequences in the data set. As both synthetic and protein data set have two classes, we take one half of these sequences from the first class and the rest from the second class. In the case of the GPCRDB Olfactory data set the first class contain the Olfactory sequences and the second class the No-olfactory sequences. Furthermore the GPCRDB Amine data set contains Amine and Peptide sequences. With this data we want to show that some groups of sequences are harder to distinguish. On all the vertical axis we have the pruning effectiveness indicated by a real number between 0 and 1. This effectiveness is calculated by dividing the search time by the worst search time in the experimental results. During the experiments we searched for the 100 maximal discriminating patterns in the GPCRDB and 10 in the synthetic data set, each with a time window of 8 and a *minsupp* of 0. Note that time window and probable time window are different concepts. The experiments on the synthetic data are done to indicate that the probable time window can improve pruning efficiency. Other experiments will show the effectiveness of the method in the case of GPCRDB data.

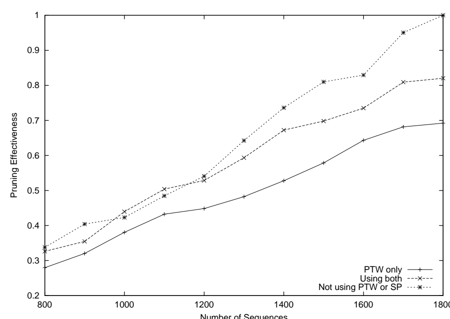


Fig. 2. Effectiveness on the GPCRDB Olfactory/No-olfactory data set

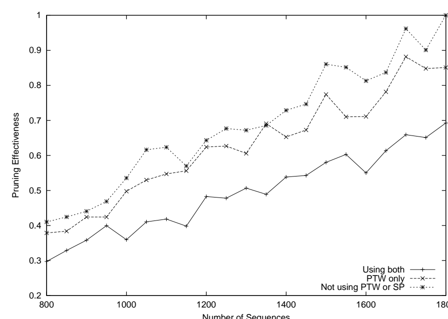


Fig. 3. Effectiveness on the synthetic data set

Figure 2 shows the effectiveness of using probable time windows (PTW) of Algorithm 2 and pruning when using "small patterns" (SP) on the GPCRDB Olfactory data. The algorithm not using PTW or SP is shown in Algorithm 1. Note that SP lowers pruning effectiveness with regards to the GPCRDB Olfactory data, because less variations of the same pattern fill up the set of patterns. Some of the patterns discovered with this data set were used for classification: these two protein families (Olfactory and No-olfactory) could be correctly distinguished in more than 90% of the cases, depending on the chosen time window size and the classification algorithm at hand.

In the synthetic data set we have most of the best patterns in the probable time window. The n-th pattern p will get a large fit_2 earlier in the search,

thus more extensions can be ignored. Figure 3 shows the effectiveness as the number of sequences in the synthetic data set increases when searching for the 10 maximal discriminating patterns. The "small pattern" rules (SP) increase the effectiveness even further, because in the synthetic data set many patterns are quickly non-improving.

Table 1. Confusion matrices of Olfactory (GPCRDB) patterns without (left) and with (right) "small patterns" (SP)

	classified as no-olfactory	classified as olfactory
no-olfactory	2015	22
olfactory	16	1909

	classified as no-olfactory	classified as olfactory
no-olfactory	2024	13
olfactory	22	1903

Table 2. Confusion matrices of Amine/Peptide (GPCRDB) patterns

	classified as amine	classified as peptide
amine	489	16
peptide	3	1091

The confusion matrices of Table 1 and Table 2 were generated using the C4.5 implementation by Weka [16] with the 10 (Olfactory) and 20 (Amine) best patterns discovered in the GPCRDB data. In Table 1 we get a slightly better classification in 10-fold cross-validation when using SP: 99.12% instead of 99.04%. This is as expected because the set of 10 patterns used in Table 1 will contain less small variations of the same pattern. The results of Table 2 required 20 patterns instead of 10. The Amine/Peptide problem is more difficult than the Olfactory/No-olfactory problem and it requires more patterns. The effect of SP on classification is small, however to show that the difference in classification performance is significant a two-tailed unpaired t-test was performed. Ten-fold crossover with 1999 sequences was done 100 times with two groups of 50 Amine/Peptide patterns, with and without SP, and a time window of 4. The t-value of 6.420 with a probability of less than 0.001 of happening by chance shows that the patterns found with SP classify significantly better when using the C4.5 algorithm with these patterns as attributes.

Figure 4 shows less improvement of the pruning effectiveness. This is because the patterns in the probable time window of the Amine sequences are less discriminating compared to the patterns in the probable time window of the Olfactory sequences. We still need to evaluate many patterns if the δ_{min} stays low, even though we might find the maximal discriminating patterns quickly.

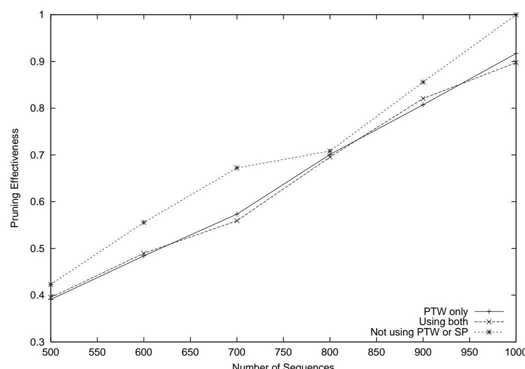


Fig. 4. Effectiveness on the GPCRDB Amine/Peptide data set

6 Conclusion

In this paper we introduced and compared two sequential pattern mining algorithms by using knowledge from the application area of protein sequence analysis. Given some assumptions, we can improve mining for the maximal discriminating patterns. The effectiveness depends on the quality of the assumptions, e.g., how probable a discriminating pattern is within a certain time window. Our method also depends on the discriminative power of the patterns. Pruning will be less effective if this is low, even though we might find the maximal discriminating patterns quickly. It is shown that using probable time windows in protein sequences can speed up the search. Protein sequences are long but contain only a few types of items; constraints are required to make the discovery of patterns in these sequences tractable.

In future research we will further investigate methods for automatically discovering the probable time window. Furthermore we plan to use maximal discriminating patterns in other application areas like workflow analysis.

References

1. Antunes, C., Oliveira, A.L.: Generalization of Pattern-Growth Methods for Sequential Pattern Mining with Gap Constraints. In Machine Learning and Data Mining in Pattern Recognition (MLDM 2003), Lecture Notes in Computer Science 2734, Springer, pp. 239–251.
2. Agrawal, R., Imielinski, T., Srikant, R.: Mining Association Rules between Sets of Items in Large Databases. In Proc. of ACM SIGMOD Conference on Management of Data (1993), pp. 207–216.
3. Agrawal, R., Srikant, R.: Mining Sequential Patterns. In Proc. International Conference Data Engineering (ICDE 1995), pp. 3–14.

4. Bailey, J., Manoukian, T., Ramamohanarao, K.: Fast Algorithms for Mining Emerging Patterns. In Proc. 6th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2002), Lecture Notes in Artificial Intelligence 2431, Springer, pp. 39–50.
5. Cong, G., Pan, F., Yang, J., Zaki, M.J.: CARPENTER: Finding Closed Patterns in Long Biological Datasets. In Proc. Conference on Knowledge Discovery in Data (SIGKDD 2003), pp. 637–642.
6. Dong, G., Zhang, X., Wong, L., Li, J.: CAEP: Classification by Aggregating Emerging Patterns. In Proc. International Conference on Discovery Science (DS-1999), pp. 30–42.
7. Graaf, E.H. de, Kosters, W.A.: Efficient Feature Detection for Sequence Classification in a Receptor Database, to appear in the Proceedings of the Seventeenth Belgium-Netherlands Conference on Artificial Intelligence, BNAIC2005.
8. GPCRDB: Information System for G Protein-Coupled Receptors (GPCRs), Website `http://www.gpcr.org/7tm/`.
9. Jovanoski, V., Lavrač, N.: Classification Rule Learning with APRIORI-C. In Proc. 10th Portuguese Conference on Artificial Intelligence (EPIA 2004), pp. 44–51.
10. Kavsẽk, B., Lavrač, N., Jovanoski, V.:APRIORI-SD: Adapting Association Rule Learning to Subgroup Discovery. In Proc. International Symposium on Intelligent Data Analysis (IDA 2003), Lecture Notes in Computer Science 2810, Springer, pp. 230–241.
11. Kramer, S., Raedt, L. De, Helma, C.: Molecular Feature Mining in HIV Data. In Proc. Conference on Knowledge Discovery in Data (SIGKDD 2001), pp. 136–143.
12. Lesh, N., Zaki, M.J., Ogihara, M.: Mining Features for Sequence Classification. In Proc. International Conference Knowledge Discovery and Data Mining (KDD 1999), pp. 342–346.
13. Liu, B., Hsu, W., Ma, Y.: Integrating Classification and Association Rule Mining. In Proc. Conference on Knowledge Discovery in Data (SIGKDD 1998), pp. 80–86.
14. Li, W., Han, J., Pei, J.: CMAR: Accurate and Efficient Classification Based on Multiple Class-Association Rules. In Proc. of the 2001 IEEE International Conference on Data Mining (ICDM 2001), pp. 369–376.
15. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. IEEE Trans. Knowl. Data Eng. 16(11) (2004), pp. 1424–1440.
16. Weka 3: Data Mining Software in Java, Website `http://www.cs.waikato.ac.nz/ml/weka/`.
17. Wess, J.: G-Protein-Coupled Receptors: Molecular Mechanisms Involved in Receptor Activation and Selectivity of G-Protein Recognition, FASEB Journal 11 (5) (1997), pp. 346–354.
18. Zimmermann, A., Raedt, L. De: CorClass: Correlated Association Rule Mining for Classification. In Proc. International Conference on Discovery Science (DS-2004), pp. 60–72.

On the Trade-Off Between Iterative Classification and Collective Classification: First Experimental Results

Tayfun Gürel and Kristian Kersting

Machine Learning Lab
Institut für Informatik
University of Freiburg
Georges-Koehler-Allee 79 79110, Freiburg Germany
{guerel,kersting}@informatik.uni-freiburg.de

Abstract. There have been two major approaches for classification of networked (linked) data. *Local* approaches (iterative classification) learn a model locally without considering unlabeled data and apply the model iteratively to classify unlabeled data. *Global* approaches (collective classification), on the other hand, exploit unlabeled data and the links occurring between labeled and unlabeled data for learning. Naturally, global approaches are computationally more demanding than local ones. Moreover, for large data sets, approximate inference has to be performed to make computations feasible.
In the present work, we investigate the benefits of collective classification based on global probabilistic models over local approaches. Our experimental results show that global approaches do not always outperform local approaches with respect to the classification accuracy. More precisely, the results suggest that global approaches considerably outperform local approaches only for low ratios of labeled data.

1 Introduction

Networked data has a key importance since many real world data can be represented in networks of data containing nodes. The most typical example of networked data is the world wide web. Web pages can be modeled as data instances, which are nodes of a directed graph. Hyperlinks give graph characteristics to billions of web pages, hence, gives us the opportunity to represent the world wide web as a huge network. The set of scientific publications exhibits network characteristics, as scientific publications cite each other. Groups of interacting people (social networks), transmission of epidemic diseases are examples of networked data, where the nodes represent human beings. Recently, there has been an increasing interest in mining networked data, see i.e. Lise Getoor's (2003) overview on link mining.

The problem of labeling networked data can be stated as follows:

Given: An undirected graph (V, E), where E is the set of edges and each node in V corresponds to a vector of features $A_1, ..., A_n, C$, where C denotes the class attribute; the values for the A_i are known for all $v \in V$, but class labels for C are only known for a proper subset T of V . We will call a node also a **data instance**.

Find: The class labels for $U := V - T$.

Several approaches to solve the *networked data labeling problem*:

Type 1: The **AVL** approach to tackling this problem is to treat this as a classical learning problem, where the nodes L are the examples and the nodes in L have to be classified (while ignoring E). The traditional AVL approach is, however, incapable of using the network features. (**type 1**: AVL approach)

Type 2: The **iterative classification** scheme (Chakrabarti et. al. 1998, Neville and Jensen 2000, Macskassy and Provost 2003, Lu and Getoor 2003a) employs a traditional AVL learner on the nodes in L but enriches the features $A_1, ..., A_n$ with additional features $F_1, ..., F_m$, which are derived from the neighborhood of the node in E. Examples of F_i include the existence of a neighbor of a particular class c. Only the labeled data is employed in learning, but in contrast to the previous method, the structure of the network is employed. The resulting classifiers is then iteratively applied on the unlabeled nodes. (**type 2**: Iterative Classification)

Type 3: The **collective classification** (Taskar et al. 2001, Taskar et al. 2002, Lu and Getoor 2003b) approach treats the problem as a global optimization problem, in which a particular function, e.g. the maximum likelihood w.r.t. the overall nodes V , is maximized. In this framework, one typically employs the Expectation Maximization (EM) algorithm (Dempster et al.,1977) over the features $A_1, ..., A_n, F_1, ..., F_m$. So, in this framework, both labeled and unlabeled data are employed, as well as the structure of the network. (**type 3** : Collective Classification)

Here, the approaches are listed according to their complexity:

> **Type 1** *approaches employ less information than* **type 2**, *and* **type 2** *less information than* **type 3**

Consequently, learning of **type 3** models can be expected to be computationally more demanding and algorithmically more complex than learning **type 2** and **type 1** approaches. Identifying the cases in which the higher computational costs of **type 3** approaches pay off is an interesting research question.

Our main contribution is such an experimental investigation showing that

> *the additional complexity of* **type 3** *approaches based on naive Bayes only pays off for low ratios of labeled data.*

This is somewhat surprising as Lu and Getoor (2003b) report that **type 3** approaches based on logistic regression improve the classification/labeling performance of **type 1** and **type 2** approaches for any ratio.

The paper is structured as follows. After reviewing related work, we will introduce a collective classification algorithm, which makes use of structural information as well as that from unlabeled nodes to improve the classification performance. The algorithm is similar to the one employed by Lu and Getoor (2003b). Instead of logistic regression, however, we use a Naive Bayes approach. We will argue that the algorithm can be viewed to learn a global probabilistic model. More precisely, the model can be represented as a Probabilistic Relational Model (Getoor et al, 2001). Before concluding, we will present our experimental results. We compare our algorithm with Naive Bayes as baseline approach (**type 1**) and iterative Naive Bayes classification (Neville and Jensen, 2000) (**type 2**).

2 Related Work

In the last few years, there has been a lot of interesting work on labeling networked data. They can be roughly divided into two approaches, *iterative classification* and *collective classification*. We will now discuss each of them in turn.

Influence propagation by Iterative Classification: Collective inference can be traced back to Chakrabarti et al. (1998). Based on collective inference, they employ a local relaxation labeling algorithm, in a sub-graph around the entity to be classified. Computations of the probability distributions are based on the naive bayes classifier.

Neville and Jensen (2000) present an iterative classification algorithm for relational data. They introduce the concept of dynamic attributes, whose values are subject to change, according to the classification results. For the task of predicting the company type in a relational corporate data set, one example of dynamic attribute might be the dominating type of the companies that the owner of a particular company owns. A naive bayes classifier is employed, which is trained on fully labeled data. The trained classifier is eventually used iteratively to label unseen examples. Neville and Jensen report an improvement of approximately 12 % in the accuracy.

Macskassy and Provost (2003) introduce Relational Neighborhood classifier (RN), which classifies the entities of a graph. RN starts with a partially labeled graph and completes labels by simply computing the weighted counts of each class among the neighbors (weighted with the weight of the edge for each neighbor) and by selecting the class that gives the maximum weighted count. There is also an iterative version of the RN classifier, which repeatedly applies RN classification as its inner loop until convergence.

Collective Classification approaches: Getoor et. al.(2001) develop a probabilistic relational model (PRM) to classify web pages. The PRMs are extensions of Bayesian networks, which were developed for relational databases. PRMs are expressive enough to compute the influence of the labels and the words on the hyperlinked web pages on the label of another web page.

Taskar et. al. (2002) employ relational Markov networks for collective classification tasks. A relational Markov Network defines undirected dependencies and

joint distributions for relational databases. When unrolled, a relational Markov network instantiates a Markov network, which defines the dependencies among the entities and their attributes. Similar to the PRMs, learning the Markov networks is based on the expected count of the values for each attribute. Inference is performed with belief propagation as in the PRMs.

Lu and Getoor (2003b) present an EM like algorithm employing logistic regression to make use of link structure for classification of networked data. They enrich the feature space with the ones from the link structure and employ iterative learning of a logistic regression classifier.

3 Collective Classification Using the Expectation Maximization Algorithm

The Expectation Maximization (EM) algorithm is a popular iterative algorithm to learn the parameters of a probabilistic model in case of missing data. The idea of using the EM algorithm for learning from unlabeled data is not new. Nigam et al. (2000) employ the EM algorithm together with the naive bayes classifier for text classification making use of the unlabeled data. In this work, we extend the naive bayes-EM algorithm to the relational case, by introducing additional features from network structure.

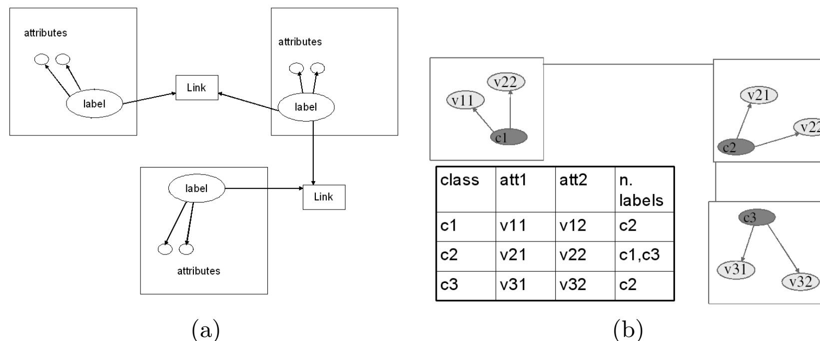
3.1 Generation of Second Order Attributes

In order to exploit the network structure, we create **second order attributes**, which hold the information about the labels of the neighboring instances. Since the information about the neighboring labels is not complete due to missing labels, second order attributes may have missing values. We applied two settings for second order attribute generation.

Neighboring Labels Our first setting for second order attribute generation is based on a graphical generative model, which is a Bayesian network. We treat the attributes and the class labels of the instances as random variables and assume that the attributes of an instance are mutually independent, given its class label (naive bayes assumption).

Additionally, we assume the existence of a link random variable between two instances (only if they are linked by an edge). Note that this is only a conceptual random variable, since we know the value of the random variable will always be 'yes' ('yes':if the edge exists and 'no': if the edge does not exist). This random variable cannot take the value 'no', because the link random variables are not included in the Bayesian network for non-existent edges. Note that this is not a complete generative probabilistic model, because we discard the influence of the non-existence of an edge.

Figure 1 illustrates a mini dataset (b) and the corresponding Bayesian network (a). In the figure a square represents a data instance with its attributes and class label. Edges between squares stand for links.



class	att1	att2	n. labels
c1	v11	v12	c2
c2	v21	v22	c1,c3
c3	v31	v32	c2

Fig. 1. (a) The underlying probabilistic model (b) Second Order Attribute Generation: Neighboring Labels

From the Bayesian network, we can infer that attributes of the instances are independent of the attributes of the other instances and the values of the link random variables are independent of each other, given the class labels. To learn the parameters of this Bayesian network, we can make use of the fact that the the parameters will be the same for each link random variable and for the attributes of the each instance. It is, therefore, possible to learn the parameters for the nodes of the same type together. This idea is analogous to that of PRMs (Getoor et al., 2001). Learning the parameters of the Bayesian network can be achieved by flattening the data into a table, with an additional attribute. We call this attribute "**Neighboring Labels**". We extract all the labels of the neighboring instances and use these labels as the values of the "Neighboring Labels". Since it is possible that one instance has more than one neighboring instance, this second order attribute is a multi-set valued attribute. An example of second order attribute generation with the "Neighboring Labels" setting is given in Figure 1b. In this example, for the sake of simplicity, we assume that there are 2 attributes and 3 instances. Squares represent the instances, whereas the connections between the squares represent the links. In this example c_i denote the ith class label. The flattened table is also shown in the same figure.

With a complete training set, the class dependent attribute probabilities (model parameters), could be found by counting in this table and they would correspond to the parameters of the Bayesian network. The parameters for the attribute nodes ($p(v_{ak}|c_j)$, where a is an index over attributes, k is an index over values and j is an index over class labels) would correspond the parameters on the first order attributes on the table. The Bayesian network parameters of the Link nodes would correspond the parameters on the second order attribute from the table, i.e. $p(Link =' yes'|c_j, c_k) = p(NeighborClass = c_k|c_j)$, since links are undirected.

class	att1	att2	(neighbor to c1)	(neighbor to c2)	(neighbor to c3)
c1	v11	v12	no	yes	no
c2	v21	v22	yes	no	yes
c3	v31	v32	no	yes	no

Fig. 2. Second Order Attribute Generation: Existence Attributes

Existence Attributes The second setting for second order attribute generation is the generation of the **existence attributes**. These second order attributes hold information about existence of links to instances from each class label. For example, if instance A has link to an instance, class label of which is c_j, then, the value of the attribute ExistsLinkTo-c_j will be *yes*. Otherwise, the value of this attribute will be *no*. Doing so for each class label, the number of classes times second order attributes are generated. An example second order attribute generation in this setting is shown in Figure 2. The data is the same as in Figure 1b. $P(there\, is\, a\, link\, to\, class\, c_n|c_j))$.

3.2 Maximum Likelihood Parameters and the EM Algorithm

Our goal is to optimize the parameters of the model globally by finding the maximum likely parameters, given the data. According to Bayes' law, the likelihood of the parameters can be written as: $P(\theta|D) = P(D|\theta)P(\theta)/P(D)$.

Dropping $P(D)$ from the expression will not change the maximizing parameters since it is not conditional on θ. We assume that the prior probability for each parameter set is equal. Therefore, we can drop $P(\theta)$ from the expression too. Since the logarithm is a monotonic function, the maximum likely parameter set is given as follows: $\theta' = \arg\max_\theta \log(P(D|\theta))$.

In our case, all the other nodes of the Bayesian network are independent of each other if the class labels are given. Therefore, the likelihood of the data can be written as follows (In a Bayesian network, a node is independent from its non-descendants given its parents):

$$\begin{aligned} P(D|\theta) =& \{\prod_{i=1}^{N} P(c(i)|\theta) \prod_{a=1}^{|A|} P(v_{ia}|c(i);\theta)\} \\ &* \{\prod_{l=1}^{|L|} P(c(l_1)|c(l_2);\theta)\} \end{aligned} \tag{1}$$

In the above equation, i is an index over the instances, N is the total number of instances, a is an index over the attributes, and $|A|$ is the number of attributes. $c(l_1)$ and $c(l_2)$ denote the class labels on two sides of a link.

We employ the EM algorithm, since the values of the second order attributes are partially missing. We introduce the hidden variables z_{ij}. i denotes an index

over the instances, and j denotes an index over the classes. The true value of z_{ij} is 1 if instance i belongs to class c_j; 0 otherwise. The computation of the expected value z_{ij} corresponds to finding the probability that the instance i belongs to class c_j. With the introduction of the hidden variables, the likelihood in Equation (1), can be rewritten as follows:

$$\begin{aligned} P(D|\mathbf{Z};\theta) =& \{\prod_{i=1}^{N}\prod_{j=1}^{|C|}\prod_{a=0}^{|A|}(P(v_{ia}|c_j;\theta)P(c_j|\theta))^{z_{ij}}\} \\ & * \{\prod_{d_i\in D}\prod_{d_m<d_i}\prod_{j=1}^{|C|}\prod_{n=1}^{|C|}P(Link='yes'|c_n,c_j;\theta)^{t_{im}z_{ij}z_{mn}}\} \end{aligned} \tag{2}$$

Applying the logarithm yields:

$$\begin{aligned} \log P(D|\mathbf{Z};\theta) =& \{\sum_{i=1}^{N}\sum_{j=1}^{|C|}\sum_{a=0}^{|A|}z_{ij}\log(P(v_{ia}|c_j;\theta)P(c_j|\theta))\} \\ & + \{\sum_{i=1}^{N}\sum_{m<d}\sum_{j=1}^{|C|}\sum_{n=1}^{|C|}t_{im}z_{ij}z_{mn}\log P(Link='yes'|c_n,c_j;\theta)\} \end{aligned} \tag{3}$$

In the above equation, i and m are indices over the instances, N is the total number of instances, j is an index over the class labels, c_j is the jth class label, v_{ia} is the value of the ath attribute, a is an index over the attributes. t_{im} is defined as 1 if there is a link between the ith and the mth instances; and 0 else. Note that $P(Link='yes'|c_n,c_j;\theta) = p(c_n|c_j) = p(c_j|c_n)$. The two latter parameters can be found by counting in the flattened table.

Equation (3) is composed of merely sum of logs and computable in closed form (Dempster et al, 1977). The expectation and the maximization step of the algorithm are as follows (The formulas follow our first setting of attribute generation, which consists of constructing a set valued second order attribute, namely "Neighboring Labels"):

Expectation

$$\begin{aligned} E^{(k+1)}[z_{ij}] &= P(classLabel = c_j|d_i, L, E^{(k)}[\mathbf{Z}]) \\ &= \alpha p(c_j)\prod_{a}p(v_{ia}|c_j)\prod_{u\in Neighbors(i)}(\sum_{t=1}^{|C|}E^{(k)}[z_{ut}]p(c_t|c_j)) \end{aligned} \tag{4}$$

In the above equation $E^{(k+1)}[z_{ij}]$ denotes the expected value of the hidden variable z_{ij} at step k+1. d_i denotes the first order attribute values of the instance i. L denotes the set of links. $E^{(k)}[\mathbf{Z}]$) stands for the expected values of the hidden variables at step k. v_{ia} stands for the value of the ath attribute of instance i. Please note that the probability is computed under naive bayes assumption that

the attributes are independent of each other. This assumption is also made for second order attributes as well. $p(NeighborClass = c_t|c_j) = p(c_t|c_j)$ is the probability that an instance from class c_t is situated at one side of a link, given that an instance from c_j is situated on the other side. This corresponds to the probability that the second order attribute value is c_t given that class label is c_j.

Maximization The maximization step corresponds to the learning the model classifier parameters based on the expected values of the hidden variables.

$$p(v_{ak}|c_j) = \frac{\hat{N}(v_{ak}, c_j)}{\hat{N}(c_j)} = \frac{\sum_i^N E[z_{ij}]\delta(v_a - v_{ak})}{\sum_i^N E[z_{ij}]} \tag{5}$$

$$\begin{aligned} p(c_l|c_k) &= \frac{\hat{N}(c_k, c_l)}{\hat{N}(c_k, DontCare)} \\ &= \frac{\sum_{i=1}^N \sum_{m=1}^N (E[z_{ik}]E[z_{ml}]t_{im} + E[z_{il}]E[z_{mk}]t_{im})}{\sum_{j=1}^{|C|} \sum_{i=1}^N \sum_{m=1}^N (E[z_{ik}]E[z_{mj}]t_{im} + E[z_{ij}]E[z_{mk}]t_{im})} \end{aligned} \tag{6}$$

$$p(c_j) = \frac{\hat{N}(c_j)}{N} = \frac{\sum_i^N E[z_{ij}]}{N} \tag{7}$$

The maximization step is nothing more than finding the Naive Bayes parameters by counting. Since the exact counts are missing, the expected counts are used. $\delta(v_a - v_{ak})$ is defined as 1 if $v_a = v_{ak}$; 0 else. t_{im} denotes the existence of a link between the instances i and m. By $\hat{N}(c_k, c_l)$, we point to the expected counts of the links, which have class labels c_k and c_l on its two sides. By $\hat{N}(v_{ak}, c_j)$, we point to the expected counts of the instances, the ath attributes of which have values v_{ak}. N denotes the total number of instances. $\hat{N}(c_j)$ denotes the expected count of the instances, which have class labels c_j.

4 Experimental Results

We compare the performances of **type 1** (naive bayes), **type 2** (iterative naive bayes with second order attributes), **type 3** (collective classification via the EM, see table 1) approaches. For **type 3**, we also implemented a hard version of the EM algorithm, which finds most likely values of the hidden values instead of expected values at each iteration. We investigate whether **type 3** approach has a significant improvement on **type 2** approach. The **type 2** approach uses the same features (second order features) as the **type 3** approaches, but instead of global optimization, learns from labeled data only and applies the learned model to unlabeled data iteratively. We also compare **type 2** and **type 3** approaches to **type 1** approach.

We ran our experiments on three datasets: Gene (KDD-Cup 2001, http://www.cs.wisc.edu/ dpage/kddcup2001/), Cora (McCallum et al, 2000)

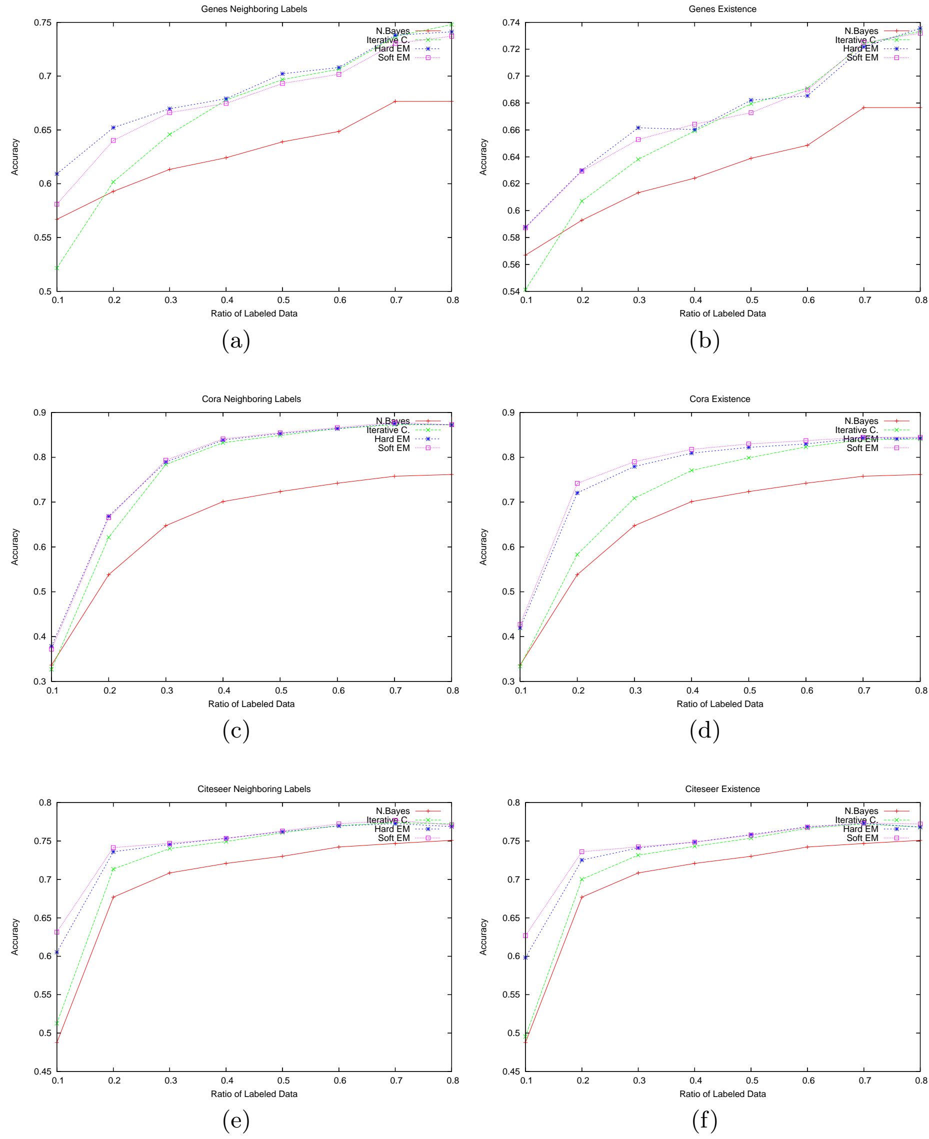


Fig. 3. Hard EM and (Soft) EM outperform the Iterative Classification **significantly** (t-test, 95%) only below a threshold of labeled data ratio (a) Hard EM threshold = 0.4 , Soft EM threshold = 0.4 (b) Hard EM threshold = 0.4 , Soft EM threshold = 0.3 (c) Hard EM threshold = 0.3 , Soft EM threshold = 0.4 (d) Hard EM threshold = 0.7 , Soft EM threshold = 0.7 (e) Hard EM threshold = 0.5 , Soft EM threshold = 0.3 (f) Hard EM threshold = 0.6 , Soft EM threshold = 0.4

Algorithm 1 The (soft) EM algorithm for collective classification

- **Input:** a set of labeled instances T, a set of unlabeled instances U and a set of links L.
- Learn a local naive bayes model on T ignoring the links
- for each node i in U
  - for each possible class label c_j
    - compute $P(label(i) = c_j | Attributes(i))$ using the local naive bayes model
  - end for
- end for
- repeat
  - Learn the Bayesian network parameters (M-Step, Equations (5), (6), (7))
  - for each node i in U
    - for each possible class label c_j
      - · Compute $E[z_{ij}] = P(label(i) = c_j | d_i, L, E^{(k)}[\mathbf{Z}])$ (E-Step, Equation (4))
    - end for
  - end for
- until distributions over possible class labels converge
- for each node i in U
  - $label(i) := \arg\max_{c_j} P(label(i) = c_j)$
- end for
- **Output:** The labels of the nodes in U

and CiteSeer (Lu and Getoor, 2003a) datasets. The Gene dataset contains 1242 examples. The task is to predict the localization of the protein in the cell (among 15 locations), given values of six attributes (some of which may be missing) and given the interaction network between proteins. There are 1806 interactions among proteins. The Cora data set contains 4187 examples. The examples are machine learning papers, which are categorized into seven topics. The Cora dataset contains 6185 citations. The CiteSeer dataset includes around 3600 computer science papers in six categories as well as 7522 citations. For Cora and CiteSeer datasets, we used the document frequency pruned dictionaries (Lu and Getoor, 2003a). The pruned dictionary for Cora has 1400 words and the pruned dictionary for CiteSeer has 3000 words.

For each data set and for each second order attributes setting, we ran the four algorithms for different ratios of labeled data, ranging from 0.1 to 0.8 . Each experiment was repeated 5 times. Each time, the instances are randomly initialized as labeled or unlabeled, according to the probability resulting from labeled data ratio. For example if the 80 % percent of the data should be labeled, the probability of any instance being initially labeled is 0.8. We performed paired t-tests (95% significance level), to assess the significance of outperformances. The results are presented in Figure 3. With our second order attributes, naive bayes (**type 1**) is significantly outperformed by the iterative and collective classification algorithms (**type 2** and **type 3**) significantly. Whether **type 3** algorithms can outperform the **type 2** algorithm does strongly depend on the labeled data

ratio. If the unlabeled data ratio is equal or below a threshold (0.4-0.5), **type 3** algorithms are no more significantly better than the **type 2** algorithm (Exception: Cora Dataset with existence attributes where threshold is 0.7).

Lu and Getoor (2003b) did a similar analysis with an EM-like iterative logistic regression algorithm. In contrast to our results for Naive Bayes, Lu and Getoor report that **type 3** algorithm improve the predictive performance of **type 1** and **type 2** approaches for any ratio of labeled data. Explaining the different results is an open and interesting future research question.

5 Conclusions

We experimentally compared global and local approaches for labeling/classifying networked data. To do so, we devised a simple global algorithm based on Naive Bayes and EM making use of labeled and unlabeled data. Our experimental results suggest that global approaches improve the performance of corresponding local ones only for low ratios of labeled data.

Acknowledgements

We would like to thank Luc De Raedt and Lise Getoor for valuable discussions. Further thanks to Lise Getoor and Prithviraj Sen for the CiteSeer and Cora data sets. This work was partially supported by the European Union IST programme, contract no FP6-508861, Applications of Probabilistic Logic Programming 2.

References

1. Chakrabarti, S., Dom B., Indyk, P.: Enhanced hypertext classification using hyperlinks. Proc. ACM SIGMOD (1998)
2. Dempster, A. P., Laird, N. M., Rubin, D. B.: Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society. Series B. **39** (1977) 1–38
3. Getoor, L., Segal, E., Taskar, B., Koller, D.: Probabilistic Models of Text and Link Structure for Hypertext Classification. IJCAI Workshop on "Text Learning: Beyond Supervision". Seattle, WA. (2001)
4. Getoor, L.: Link Mining: A New Data Mining Challenge. SIGKDD Explorations. volume 5, issue 1. (2003)
5. Lu, Q. & Getoor, L.: Link-based Classification. International Conference on Machine Learning. Washington, DC. (2003)
6. Lu, Q. & Getoor, L.: Link-based Classification using Labeled and Unlabeled Data. ICML Workshop on the Continuum from Labeled to Unlabeled Data in Machine Learning and Data Mining. Washington, DC. (2003)
7. Macskassy, S. A. & Provost, F.: A Simple Relational Classifier. ICML Workshop on the Continuum from Labeled to Unlabeled Data in Machine Learning and Data Mining. Washington DC. (2003)

8. McCallum, A., Nigam, K., Rennie, J., Seymore, K.: Automating the Construction of Internet Portals with Machine Learning. Information Retrieval Journal. **3** (2000) 127–163
9. Neville, J. & Jensen, D.: Iterative Classification in Relational Data. In Proc. AAAI-2000 Workshop on Learning Statistical Models from Relational Data. (2000)
10. Nigam, K., McCallum, A., Thrun, S., Mitchell T.: Text Classification from Labeled and Unlabeled Documents using EM. Machine Learning. **39(2/3)** (2000) 103–134
11. Taskar, B., Segal, E., Koller, D.: Probabilistic Clustering in Relational Data. Seventeenth International Joint Conference on Artificial Intelligence (IJCAI01). Seattle, Washington. (2001)
12. Taskar, B., Abbeel, P., Koller, D.: Discriminative Probabilistic Models for Relational Data. Eighteenth Conference on Uncertainty in Artificial Intelligence (UAI02). Edmonton, Canada (2002)

Mining Probabilistic Automata: a New Way to Sequence Mining

Stéphanie Jacquemont, Francois Jacquenet, and Marc Sebban

EURISE – Université Jean Monnet de Saint-Etienne
23, rue du Dr Paul Michelon – 42023 Saint-Etienne cedex 2 – France

Abstract. We propose a new sequence mining algorithm that extracts constrained frequent patterns from a probabilistic finite state automaton (PDFA). Even if PDFAs have not received wide attention in sequence mining, we show that the use of a learned compact summary of the input-sequences, rather than a costly exact representation, is very relevant in this domain. We propose two kinds of constraints for extracting particular patterns from the PDFA, reducing then the search space. Experiments show the utility of considering such constraints in sequence mining.

1 Introduction

The sequence mining task consists in finding patterns, *i.e.* sequences of events shared in a database (DB) by a large number of instances which can take the form of texts, DNA sequences, web site usage logs, etc. By automatically extracting such *frequent* patterns, one aims at discovering knowledge about customer behaviors, network alarms, web site access strategies, etc. [1]. A pattern w is *frequent* if the number (called *support*) of sequences of the DB that contain w is greater than a *minimal support* given by the user [2]. In such a context, many sequence mining algorithms have been proposed during the last decade [2–5]. However, over the past few years, two new trends seem to independently emerge.

The first one concerns the way the DB is scanned while discovering frequent patterns. Some work has been done [6, 7] to build a subtle representation of the DB to avoid multiple naive scanning of it. However, in case of huge DBs, this *exact* representation requires all the same high computational and storage costs. Then, rather than building a costly *exact* representation of the data, an other solution would consist in *learning* a more compact summary of the DB in the form of a generative model, such as a grammar or an automaton. This idea of using a generalized representation of the sequences has been used by Hingston in [8]. However, the advantages of using PDFAs have not been thoroughly studied. In this paper, we achieve this task and we provide an extension of Hingston's approach by introducing *constraints* on the frequent sequences extracted from a PDFA. This concept of constraint is a second current trend in sequence mining.

Actually, despite the use of a *minimal support*, an *unconstrained* search can produce millions of patterns or may even be intractable. A recent strategy consists in extracting frequent patterns under constraints: length and width restrictions, minimum or maximum gap between events, time window of occurrence

[9], or regular expressions [10] (see also [11]). Moreover, as Zaki claims in [9], there exist many domains (such as in bio-informatics) where the user may be interested in interactively adding syntactic constraints on the mined sequences. In this paper, we introduce two new constraints that we adapt to the specific context of PDFAs. The first belongs to the same family as the one of *time window* proposed in [9], and consists in extracting only frequent sequences which begin after a given prefix length. To take it into account in PDFAs, we propose an extension of Hingston's formulas. The second constraint is based on the statistical relevance of the extracted frequent sequences. Roughly speaking, we mean that a frequent pattern does not always express a significant information. Let us take an example. Two series of experiments A and B are carried out by tossing a coin respectively 10 and 10000 times. Suppose we obtain respectively 8 and 8000 *tails* (and 2 and 2000 *heads*), resulting in two DBs of 10 and 10000 sequences (of only one event), and we fix the minimal support to 70%, *i.e.* respectively 7 and 7000 sequences. So, the sequence *tails* is frequent in both DBs. Does it mean that the knowledge "*tails is frequent with a support of* 80%" expresses the same information in both DBs? Absolutely not. While it is highly probable, with a *non-unbalanced* coin, to obtain more than 70% of *tails* over 10 trials, this event is so improbable over 10000 that it could lead to challenge the balance of the coin itself. Since tuning the minimal support is a tricky task, we propose to constrain a frequent sequence to be also statistically relevant. We introduce in this paper a statistical test-based approach that we adapt to sequence mining from PDFA.

The paper is organized as follows. First, after a presentation of the advantages of an automaton-based sequence mining algorithm, we describe Hingston's method. We carry out a series of experiments that shows the efficiency of a PDFA for estimating true probabilities. In Section 3, we outline the main steps of our methodology by defining our constraints and combining them in a sequence mining algorithm. Section 4 deals with experiments, carried out using the algorithm ALERGIA [12] for learning the PDFA, that show the efficiency of our approach.

2 On using PDFA for sequence mining

Using a PDFA for mining sequences can allow us to extract frequent patterns by analyzing its paths. However, we must make here a remark concerning the kind of information we are able to find. By definition, the problem of mining sequences is to extract all frequent patterns according to a minimal support *sup*. However, by learning a generalized representation of the data in the form of an automaton, and by mining it instead of the input sequences themselves, we can not *a priori* claim that the extracted sequences occur exactly more than *sup* times in the DB. In other words, since we use a probabilistic representation of the data, we can *only* ensure that the extracted sequences are *probably* frequent. Fortunately, if the size of the DB is large enough, we will see that the probabilities estimated from the PDFA converge to the ones observed in the set of sequences. In this case, all extracted sequences from the PDFA will be *probably* also frequent in the DB.

Using a PDFA for sequence mining might also result in the discovery of new knowledge, *i.e.* patterns that do not occur in the DB. This amazing phenomenon is due to the fact that most of the PDFA learning algorithms work by state merging, starting from a prefix tree acceptor and merging states that are judged compatible. From a theoretical standpoint, if the DB contains characteristic sequences, it is possible to prove that one can exactly learn the target distribution of the sequences [13]. Let us now formally define a PDFA.

Definition 1. *a* PDFA $A =< Q, \Sigma, q, q_0, \pi, \pi_F >$ *is a tuple where* Q *is a finite set of states;* Σ *is the alphabet;* $q : Q \times \Sigma \rightarrow Q$ *is a transition function;* q_0 *is the initial state;* $\pi : Q \times \Sigma \times Q \rightarrow [0, 1]$ *is a probability function on the transitions;* $\pi_F : Q \rightarrow [0, 1]$ *is a probability function which associates to each state* $S \in Q$ *a probability to be final; Moreover,* A *must be deterministic, i.e.* $\forall S \in Q, \forall z \in \Sigma$, *the cardinality of the set* $\{x|q(S, z) = x\}$ *is bounded by 1.*

Since A is deterministic, the two first arguments of π are sufficient to describe a transition. $\pi(S, z)$ will then represent the probability $\pi(S, z, q(S, z))$. Fig.1 shows a PDFA where $Q = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $q(0, a) = 2$, $q_0 = 0$, and $\pi_F(0) = 0.338$.

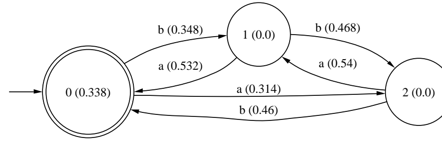


Fig. 1. An example of PDFA.

The first related work is probably the one of Hingston [8], which used an Apriori-like system for mining sequences from PDFA. Since we aim at extending his method to constrained sequence mining, we now detail the main steps of his method which estimates the probability of occurrence of particular patterns.

Let $A =< Q, \Sigma, q, q_0, \pi, \pi_F >$ be a PDFA. To assess with $\hat{p}(x)$ the proportion $p(x)$ of sequences of the DB that contain x, he defines the probability $P(S, x)$ that a path in A starting from state S contains an x. This is ensured either if a path begins with an x (of probability $\pi(S, x)$), or with some other symbol $z \in Q$ and is followed by a path starting at the next state (given by $q(S, z)$) and containing an x. This can be written with the recursive formula: $P(S, x) = \pi(S, x) + \sum_{z \neq x \in \Sigma} (\pi(S, z) \times P(q(S, z), x))$ that we rewrite as follows:

$$P(S, x) = \pi(S, x) + \sum_{T \in Q} \left(\sum_{z \neq x, q(S,z)=T} \pi(S, z) \right) \times P(T, x) \quad (1)$$

If $S = q_0$, $P(S, x)$ is exactly equal to $\hat{p}(x)$, *i.e.* the probability we are looking for. Computing $P(S, x)$ requires to solve a system of linear equations for which

Hingston proposes an efficient solution based on matrix products. He defines the matrix $\rho(x)$ whose components are $\rho_{S,T}(x) = \sum_{z \neq x, q(S,z)=T} \pi(S,z)$. Let $P(x)$ (resp. $\pi(x)$) be the vector of values of $P(S,x)$ (resp. $\pi(S,x)$), Eq.1 becomes:

$$P(x) = \pi(x) + \rho(x)P(x) = (I - \rho(x))^{-1}\pi(x) \tag{2}$$

where I is the identity matrix. Since the matrix $\rho(x)$ and the vector $\pi(x)$ are directly built from the conditional probabilities of the PDFA, the computation of the vector $P(x)$ becomes very easy to achieve.

Example: let us take again the PDFA of Fig.1. We aim at assessing with $\hat{p}(a) = P(0,a)$ the true proportion $p(a)$ of sequences that contain the letter a. Vector $\pi(a)$ has the components $\pi(0,a) = 0.314$, $\pi(1,a) = 0.532$, $\pi(2,a) = 0.54$. For matrices $\rho(a)$ and $(I - \rho(x))^{-1}$, we get

$$\rho(a) = \begin{pmatrix} 0 & 0.348 & 0 \\ 0 & 0 & 0.468 \\ 0.46 & 0 & 0 \end{pmatrix} \quad and \quad (I - \rho(x))^{-1} = \begin{pmatrix} 1.081 & 0.376 & 0.176 \\ 0.233 & 1.081 & 0.506 \\ 0.498 & 0.173 & 1.081 \end{pmatrix}.$$

We deduce that $P(x) = (0.635, 0.921, 0.832)$ and $\hat{p}(a) = P(0,a) = 0.635$.

Based on the same principle, one can estimate the proportion of strings that contain a bi-gram xy. Let $P(S,xy)$ be the probability that a path starting at state S contains xy. One can derive as previously:

$$P(xy) = (I - \rho(x))^{-1}\tau(xy), \tag{3}$$

where $\tau(xy) = \pi(S,x)\pi(q(S,x),y)$. Hingston shows that it is also possible to assess the proportion of strings containing x, followed *later* by y, noted $P(S, < x, y >)$. By combining $P(S,x)$, $P(S,xy)$ and $P(S, < x, y >)$, one can derive formulas to compute probabilities for any ordering of symbols and n-grams.

To experimentally assess the efficiency of a PDFA to estimate the true proportions in the DB, we implemented Hingston's algorithm and carried out a series of experiments. We simulated a target distribution from a given alphabet Σ. To test different configurations, this target was modeled in the form of an automaton with 1, 10 or 30 states. From it, we sampled learning sets of different sizes, and for each of them, we learned a PDFA using ALERGIA [12], and then we computed $P(q_0, x)$ and $P(q_0, < x, y >)$[1], $\forall x, y \in \Sigma$, and compared them with the true frequencies $p(x)$ and $p(< x, y >)$ observed in the DB. Fig.2 shows the behavior of the average difference between the estimated and the true frequencies. We can observe that in all cases it converges rapidly toward 0. Hingston proposed a sequence mining algorithm in [8] using the previous estimates. We will use it in some of our experiments in Section 4.

3 Constrained sequence mining

We extend here Hingston's approach by constraining the sequence mining algorithm to discover *particular* frequent patterns. Two types of constraints are

[1] Since the behavior of $P(q_0, xy)$ follows the one of $P(q_0, x)$, we did not carry out experiments for it.

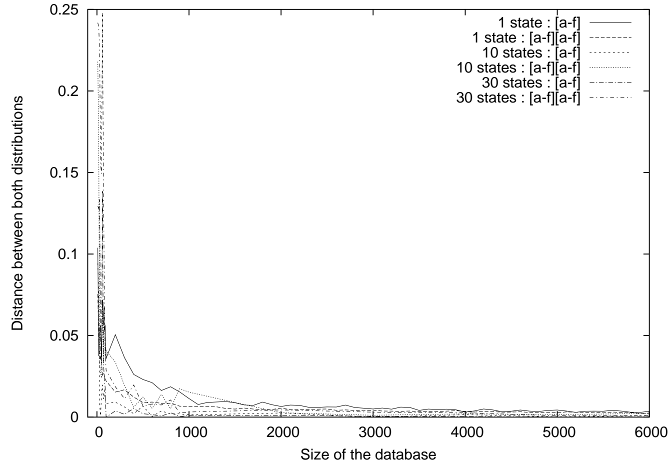


Fig. 2. Average difference between estimated and observed probabilities of given patterns (according to regular expressions on the alphabet).

proposed. The first one allows us to discover patterns satisfying a given prefix-length. This is interesting in domains where the location in the sequence expresses the meaning of the pattern. To compute the new estimates, we must extend Hingston's formulas. Then, we present a second type of constraints aiming at extracting only the patterns that are statistically relevant. Adapted to the context of PDFA, they can be used by standard sequence mining algorithms.

3.1 A prefix length constraint

We have to modify $P(S,x)$ and $P(S,xy)$ to take into account a prefix length constraint, that could result in a large reduction of the search space. Let us recall that $P(S,x)$ (resp. $P(S,xy)$) is an estimate of the proportion of sequences that contain, from state S, the symbol x (resp. the bi-gram xy). We want to extend them respectively to $P(S,x,\delta)$ and $P(S,xy,\delta)$, *i.e.* the proportion of sequences that contain an x (resp. xy) after a prefix of length δ. Note that we will not extend $P(S,<x,y>)$, that explains why we did not enter in the previous section in the details of its calculation. We think that imposing such a constraint in this context would not be relevant, because the important information is that the symbol y occurs *after* the symbol x (the location is here not important).

From $P(S,x)$ to $P(S,x,\delta)$ A component $P(S,x,\delta)$ of $P(x,\delta)$ is the proportion of sequences containing an x (maybe not the first one) at a distance δ from S. From Fig.1, if we are looking for the proportion $P(0,a,2)$ of sequences containing an a in third position, we can establish that:

$$
\begin{aligned}
P(0,a,2) &= \pi(0,b)\times\pi(1,b)\times\pi(2,a)+\pi(0,b)\times\pi(1,a)\times\pi(0,a)\\
&\quad +\pi(0,a)\times\pi(2,a)\times\pi(1,a)+\pi(0,a)\times\pi(2,b)\times\pi(0,a)\\
&= \sum_{z\in\Sigma}\pi(0,z)\times P(q(0,z),a,1) = 0.282.
\end{aligned}
$$

Generalizing, we get

$$
P(S,x,\delta) = \sum_{z\in\Sigma}\pi(S,z)\times P(q(S,z),x,\delta-1) = \sum_{T\in Q}\left(\sum_{z,q(S,z)=T}\pi(S,z)\right)\times P(T,x,\delta-1). \quad (4)
$$

Since an x can occur before the distance δ, the constraint $z \neq x$ does not exist, and then we can not use the matrix $\rho(x)$ of Eq.2. Let us introduce the following matrix μ which is now *independent* of x, $\mu_{S,T} = \sum_{z,q(S,z)=T}\pi(S,z)$. We can rewrite Eq.4 as

$$
P(S,x,\delta) = \sum_{T\in Q}\mu_{S,T}\times P(T,x,\delta-1). \quad (5)
$$

Let $P(x,\delta)$ be the vector of values of $P(S,x,\delta)$, Eq.5 becomes:

$$
P(x,\delta) = \mu\times P(x,\delta-1). \quad (6)
$$

This is a geometric series of common ratio μ and first term $P(S,x,0) = \pi(S,x)$. Writing $\pi(x)$ for the vector of values of $\pi(S,x)$, we obtain:

$$
P(x,\delta) = \mu^{\delta}\times\pi(x). \quad (7)
$$

All the components of μ and $\pi(x)$ are directly obtained from the PDFA. As for Hingston, we are only interested in the first one, *i.e.* in the case $S = q_0$.

From $P(S,xy)$ to $P(S,\omega,\delta)$ Rather than extending the probability of occurrence of a bi-gram xy, we extend directly $P(S,xy)$ to $P(S,\omega,\delta)$ where w is a *pattern, i.e.* an ordered set of k symbols of Σ. First, we have to generalize the function $\tau(S,xy)$ (Eq.3) to a pattern w. We get, $\tau(S,w_1\ldots w_n) = \pi(S,w_1)\times\tau(q(S,w_1),w_2\ldots w_n)$. Then,

$$
P(S,w,\delta) = \sum_{z\in\Sigma}\left(\pi(S,z)\times P(q(S,z),w,\delta-1)\right) \quad (8)
$$

$$
P(S,w,0) = \pi(S,w_1)\times\pi(q(S,w_1),w_2)\times\ldots = \tau(S,w) \quad (9)
$$

Using exactly the same principle as in Eq.7 we get:

$$
P(w,\delta) = \mu^{\delta}\times\tau(w) \quad (10)
$$

3.2 Statistical relevance constraints

We introduce here a second type of constraints based on the relevance of the extracted frequent patterns. We claim that a *frequent* pattern can be statistically *irrelevant*. The relevance can be assessed using statistical tests applied on the estimates of the PDFA. We propose two tests, that we will call *relevance constraints*, to validate *step by step* the symbols of a *relevant* frequent pattern. The first test verifies an *absolute* condition. Let $w = (w_1...w_l)$ be a current *relevant* pattern of size l in the PDFA, at a distance δ from q_0 (that we will note δ-relevance to simplify). The δ-relevance of an additional $(l+1)^{th}$ symbol w_{l+1} will be ensured if the proportion of sequences that contain the pattern $w' = (w_1...w_{l+1})$ at a distance δ from q_0 (estimated by $P(q_0, (w_1...w_{l+1}), \delta)$) covers a significant part of the probability density of all sequences. The second constraint expresses a *relative* condition. Given a current δ-relevant frequent pattern $w = (w_1...w_l)$ and a symbol w_{l+1} satisfying the first test. The proportion of sequences that satisfy w at a distance δ from q_0 must be approximately the same as the one of the new pattern $w' = (w_1...w_{l+1})$. According to these conditions, we can define *recursively* the notion of δ-relevance (of a symbol and of a pattern). For this reason, let us assume that we have already a frequent and δ-relevant pattern w.

Definition 2. *Let A be a* PDFA *in which a pattern $w = (w_1...w_l)$ is already frequent and δ-relevant. Let w_{l+1} be a new symbol which, concatenated with w, makes a pattern $w' = (w_1...w_{l+1})$. w_{l+1} is δ-relevant for w' iff $P(q_0, (w_1...w_{l+1}), \delta)$ is significantly higher than 0, using a test of proportion.*

The notion of *significance* is defined by a classic test of proportion (called PROP_TEST) aiming at verifying if $P(q_0, (w_1...w_{l+1}), \delta)$ is high enough. For simplifying notations, let us consider that $P(q_0, (w_1...w_{l+1}), \delta) = \hat{p}(w', \delta)$. To verify if this estimate of the true proportion $p(w', \delta)$ (of sequences containing w' after a prefix of length δ) is significant, we test the null hypothesis $H_0 : p(w', \delta) = 0$, against the alternative one $H_a : p(w', \delta) > 0$. If the number of sequences is large enough, $\hat{p}(w', \delta)$ asymptotically follows the normal law. Let us determine the threshold k which defines the bound of rejection of H_0, and which corresponds to the $(1-\alpha)$-percentile (U_α) of the distribution of $p(w', \delta)$ under H_0. It is easy to show that $P(\hat{p}(w', \delta) > k) = \alpha$ iff $k = U_\alpha \sqrt{\frac{\hat{p}(w',\delta)(1-\hat{p}(w',\delta))}{n}}$, where n is the number of sequences in the DB. We get then the decision rule: *if $\hat{p}(w', \delta) > k$, the δ-relevance of w_{l+1} is satisfied.*

So far, we assumed that we had a current δ-relevant frequent pattern w. From an algorithmic standpoint, we will initialize w to the empty string, w' containing only the additional symbol w_{l+1}. By doing this, we obtain a first set of δ-relevant patterns with PROP_TEST. Fig.3 shows a PDFA with 6 states, where $\Sigma = \{a, b, c\}$, built from a set of 100 sequences. Given the pattern $w' = (a)$ and a minimal support of 40%, w' is *frequent* at a distance 0 from state 0 ($P(0, a, 0) > 0.4$). Is this new (and unique here) symbol a also 0-relevant? With a risk $\alpha = 2.5\%$, $U_\alpha = 1.96$ and $k = 0.088$, since $P(0, a, 0) > k$, the symbol a is 0-relevant for the pattern w'. Since $w' = (a)$, we can also deduce here that w' is 0-relevant.

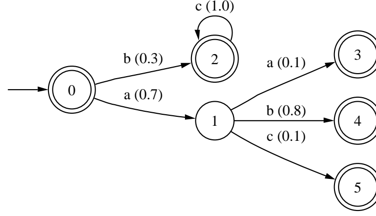


Fig. 3. An other example of PDFA with 6 states.

We think that a unique constraint on each additional symbol is not sufficient to accept w' as being a δ-relevant pattern. We propose to also verify if there exists a statistical dependence in the PDFA between w' and the previous pattern w. Roughly speaking, the high majority of the sequences that contained w must also satisfy w'. This is what we call a *conditional constraint.*

Definition 3. *Let A be a* PDFA *in which a pattern $w = (w_1...w_l)$ is already frequent and δ-relevant. Let w_{l+1} be a new symbol which, concatenated with w, makes a new pattern $w' = (w_1...w_{l+1})$. w' is δ-relevant conditionally to w iff w' is significantly dependent of w, using a Chi-Square test.*

This dependence can be assessed by analyzing the nature of all the different symbols occurring in the PDFA after the pattern w. Consider again Fig.3 for which we have already validated the δ-relevance of $w' = (a)$ for $\delta = 0$. According to the conditional probabilities of this PDFA, is $w' = (ab)$ also δ-relevant? To deal with this problem, we must achieve two tasks. First, the PROP_TEST must be run to verify if the additional symbol b is δ-relevant for w'. In this case, with $\alpha = 2.5\%$ and $U_\alpha = 1.96$, $\hat{p}(w', 0) = P(0, ab, 0) = \pi(0, a).\pi(1, b) = 0.56$ and $k = 0.083$. Since $\hat{p}(w', 0) > k$, b is δ-relevant for w'. We must now verify if there exists a statistical dependence between $w = (a)$ and $w' = (ab)$. To carry out this task, we generate an output vector $\overrightarrow{V_{out}}$ of dimension m, where m is the number of different outgoing transitions from states in which the last symbol w_l of w ends. In our example, $\overrightarrow{V_{out}}$ has three components $\overrightarrow{V_{out}}(i), i = 1, ..., 3$ because there is a set $O = \{a, b, c\}$ of outgoing transitions from state 1. Each component $\overrightarrow{V_{out}}(i)$ is the expected number of times the symbol $z_i \in O$ follows the pattern w in the DB, that means that $\overrightarrow{V_{out}}(i) = P(q_0, z_i, \delta) \times |\text{DB}|$. We arrange $\overrightarrow{V_{out}}$ such that the considered symbol w_{l+1} (here b) is the first component of the vector (the order of the others does not matter). In our example, $\overrightarrow{V_{out}} = (56, 7, 7)$. We now aim at testing the dependence between $\overrightarrow{V_{out}}$ and an input vector $\overrightarrow{V_{in}} = (70, 0, 0)$ for which the first component is the expected number of times the pattern w occurs in the DB (the other components are null). From $\overrightarrow{V_{in}}$ and $\overrightarrow{V_{out}}$, we run a test of independence based on a Chi-square test (called CHI2_TEST). We build the following statistic X^2, such that:

$$X^2 = \sum_{i=1}^{m} \frac{[(\overrightarrow{V_{in}}(i) - \Psi(i))^2 + (\overrightarrow{V_{out}}(i) - \Psi(i))^2]}{\Psi(i)},$$

where $\Psi(i) = \frac{\overrightarrow{V_{in}}(i)+\overrightarrow{V_{out}}(i)}{2}$. X^2 follows a Chi-square distribution with $2 \times m-1$ degrees of freedom. It is then possible to test if X^2 is higher than $X_{\alpha}^{2\times m-1}$, which is the $(1-\alpha)$-percentile of the Chi-square law. We get then the decision rule: *if $X^2 < X_{\alpha}^{2\times m-1}$, the conditional δ-relevance is verified.* By combining the two previous constraints, we can define a frequent and δ-relevant pattern.

Definition 4. *Let A be a* PDFA *in which a pattern $w = (w_1...w_l)$ is already frequent and δ-relevant. Let w_{l+1} be a new symbol which, concatenated with w, makes a new pattern $w' = (w_1...w_{l+1})$. w' is frequent and δ-relevant iff (i) $P(q_0, w', \delta)$ is higher than a minimal support, (ii) the new symbol w_{l+1} is δ-relevant for w' and (iii) w' is δ-relevant conditionally to w.*

Algorithm 1: Pseudo-code of ACSM

```
    Input: A PDFA A = (Q, Σ, q ,q0, π, πF), a support threshold σ, two risks α1 and α2, a
           prefix length δ
    Output: a set of relevant frequent patterns G
 1  begin
 2      G1 ← ∅ ;
 3      foreach l ∈ Σ do
 4          if P(q0, l, δ) ≥ σ then
 5              if PROP_TEST (P(q0, l, δ), α1) then
 6                  G1 ← G1 ∪ l^δ // l^δ means l at distance δ;
 7              end
 8          end
 9      end
10      G ← G1; n ← 1 ;
11      while Gn ≠ ∅ do
12          Gn+1 ← ∅ ;
13          foreach ω^δ = (w1...wn)^δ ∈ Gn do
14              foreach l ∈ Σ do
15                  ω^δ ← (w1...wn l)^δ;
16                  if P(q0, ω^δ, δ) ≥ σ then
17                      if PROP_TEST (P(q0, (ωn l), (δ + n − 1)), α1) then
18                          if CHI2_TEST (V_in→, V_out→, α2) then
19                              Gn+1 ← Gn+1 ∪ ω^δ ;
20                          end
21                      end
22                  end
23              end
24          end
25          G ← G ∪ Gn+1 ;
26          n ← n + 1 ;
27      end
28      return G ;
29  end
```

3.3 A new sequence mining algorithm

Combining all the concepts we presented before, we propose a new constrained sequence mining algorithm. It aims at discovering from a PDFA all frequent and δ-relevant patterns in the form of n-grams, according to a minimal support σ and a prefix length δ. Of course, we can also run it several times with different values of δ to avoid to have to fix in advance a given prefix length without any knowledge about the studied domain. The pseudo-code of our algorithm ACSM (for Automata-based Constrained Sequence Mining) is presented in Algorithm 1. From lines 2 to 9, it initializes a set of δ-relevant frequent patterns composed of only one symbol. Since no patterns have been extracted yet, only the support test (line 4) and the PROP_TEST (line 5) are run. The paths of the PDFA that do not satisfy these two tests will not be studied anymore, that allows us to dramatically reduce the search space. The second part of ACSM tests additional symbols to search for larger frequent δ-relevant patterns. In this case, three conditions must be satisfied: the support test (lines 16), the PROP_TEST (line 17) and the CHI2_TEST (line 18). Due to the prefix length constraint, note that in its first version, ACSM only deals with patterns without gaps. Of course, it is possible to directly apply our relevance constraints on other sequence mining algorithms, that will be done in Section 4 with the one of Hingston.

4 Experimental results

We aim at studying the potential advantages of our approach. We are going to assess the impact of our constraints in terms of *number* and *quality* of the patterns extracted from a PDFA. To achieve these tasks, we carried out many series of experiments from two DBs. The first one (called FIRSTNAME) is the top 1000 first names given in 2004 in the USA[2]. Removing names in common, it remains 1872 strings, among which 936 are female, that will constitute the target to learn. We also used the synthetic generator IBM DATAGEN[3] for generating a larger DB (called SYND) containing 100000 sequences of average size 10 events.

4.1 Number of extracted patterns

Showing that the number of extracted patterns can drop because of our constraints would allow us to prove the potentiality of our approach to reduce the search space. We learned two PDFAs with ALERGIA from a part of FIRSTNAME (only the 936 female names) and SYND. Using our algorithm ACSM, we evaluated the effect of each constraint $(\alpha_1, \alpha_2, \delta)$ on the number of extracted patterns.

Fixing $\delta = 0$, the first two charts of Fig.4 show the effects of the relevance constraints on SYND. We tested the influence of the relevance constraint of a symbol without incorporating the conditional relevance constraint of a pattern (first chart, fixing α_2 to 100%) and reciprocally (second chart, fixing α_1 to 100%).

[2] http://www.ssa.gov/OACT/babynames/

[3] http://www.almaden.ibm.com/software/quest/Resources/index.shtml

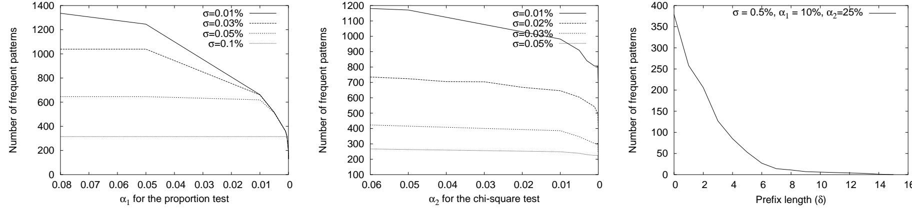


Fig. 4. Effect of our constraints $(\alpha_1, \alpha_2, \delta)$ on the number of extracted patterns.

We can observe that the stronger one of these constraints is (*i.e.* the lower the α_1 or α_2 parameter is), the more the number of extracted patterns decreases. Moreover, we can note that the more the minimum support increases, the more the relevance constraints become obviously useless. The last chart shows the influence of the prefix length constraint δ in the case of FirstName. We can see that strengthening this constraint (for a given configuration of σ, α_1 and α_2) leads to extracting a decreasing number of patterns. Fig.4 shows then the impact of our constraints to dramatically decrease the number of extracted patterns. Now, our second objective is to assess their *quality.*

4.2 Quality assessment of the PDFA-based patterns

We have experimentally shown (Fig.2) that a PDFA provides a sufficiently close approximation to the original set of sequences. Moreover, since automata learning algorithms generalize the original DB, we hope that the PDFA contains not only all the frequent patterns which would be extracted by a classic sequence mining algorithm, but also new interesting knowledge in the form of paths not or rarely used by learning sequences. To achieve this task, we run first (with a support of 5%) the original Hingston's algorithm (*i.e.* without any constraints) on X female names randomly chosen from FirstName ($X = 10, ..., 936$), using ALERGIA for learning the PDFA. Then, we run SPAM[4][14], known as one of the most efficient sequence mining algorithm, on the same growing datasets. For each size, we computed the number of "forgotten" patterns (*i.e.* found by SPAM but not from the PDFA) and the number of "added" patterns (*i.e.* only found from the PDFA). The two curves ("forgotten" and "added") are presented in the first chart of Fig.5. We can note that once the size of the DB is sufficiently large, almost all the frequent patterns found by SPAM are also extracted from the PDFA (the curve "forgotten" tends to 0). This proves, once again, the ability of a PDFA to correctly approximate the original DB. Moreover, there are "new" patterns extracted from the PDFA, which come, as we explained before, from paths built by generalization.

We can wonder if among these additional patterns, some of them are *relevant*, *i.e.* really contain interesting knowledge. To answer this question, we applied our

[4] http://himalaya-tools.sourceforge.net/Spam/

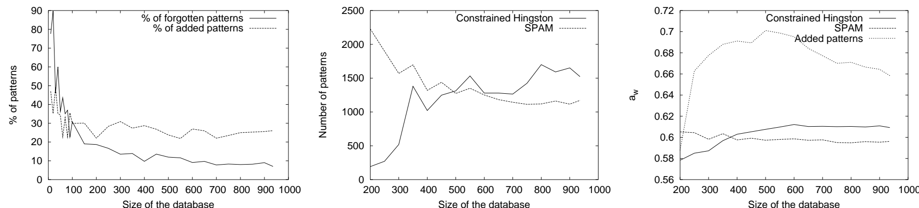


Fig. 5. Quality of the new patterns extracted from a PDFA.

relevance constraints on the patterns extracted from the PDFA in order to keep only the most significant ones. To assess the quality of the sets of patterns extracted by SPAM and from the PDFA, we computed the following criterion $a_w = \frac{\sum_{i=1}^{m} n_{i+}}{\sum_{i=1}^{m} n_{i+} + \sum_{i=1}^{m} n_{i-}}$, where n_{i+} (resp. n_{i-}) is the number of female (resp. male) first names covered by the set of m patterns. Note that this criterion takes into account the negative examples (male names). Even if we could propose other measures, this one allows us to evaluate the risk to extract non-informational patterns about the target concept. The two remaining charts of Fig.5 describe the comparison between SPAM and our constrained approach. The first one shows that from about 500 names, the number of *relevant* patterns extracted from the PDFA is higher than the number of patterns discovered by SPAM. To assess the quality of this additional information, the last chart of Fig.5 shows the evolution of the criterion a_w on 3 different sets of patterns extracted (i) by SPAM, (ii) by a constrained Hingston and (iii) by a constrained Hingston but not by SPAM. From about 400 names, we can note that the quality of the sets of patterns extracted from the PDFA are always better than the one of SPAM. More interestingly, the curve "added patterns" shows that the quality measure only computed from the additional patterns is much more higher in average than the one of SPAM. This proves that the generalization of the automaton provides new relevant knowledge about the target to learn.

5 Conclusion

In this paper, we have presented an automata-based approach for constrained sequence mining. We have seen that building a PDFA from the data and then mining that structure presents many advantages. Our framework extends the one of Hingston by incorporating a prefix length constraint and two statistical relevance constraints. The experiments we made have shown that those constraints lead to extracting less frequent patterns which is really important for the users that are often overwhelmed by huge amount of useless patterns while mining data. Moreover, using a quality measure, we noted that it is possible to extract new knowledge from the PDFA allowing us to improve the performances of the patterns from a classification standpoint. We now want to focus on several points. First we would like to integrate other constraints in the ACSM algorithm.

Then, we plan to study the impact of noisy data on the system and the way it deals with them. We also want to use it in the context of biological data in order to explore the power of our new constraints on such a field of applications.

References

1. Han, J., Altman, R.B., Kumar, V., Mannila, H., Pregibon, D.: Emerging scientific applications in data mining. Comm. of the ACM **45** (2002) 54–58
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. of the 12th Inter. Conf. on Data Engineering, IEEE Computer Society (1995) 3–14
3. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. In: Proc. of EDBT'96, Springer-Verlag (1996) 3–17
4. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. Data Min. Knowl. Discov. **1** (1997) 259–289
5. Zaki, M.J.: Efficient enumeration of frequent sequences. In: Proc. of CIKM'98, ACM Press (1998) 68–75
6. Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., Hsu, M.C.: Freespan: Frequent pattern-projected sequential pattern mining. In: Proc. of the Inter. Conf. on Knowledge Discovery and Data Mining. (2000) 355–359
7. Pei, J., Han, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: Proc. of the Inter. Conf. on Data Engineering. (2001) 215–224
8. Hingston, P.: Using finite state automata for sequence mining. In: Proc. of the 25th Australasian conference on Computer science. (2002) 105–110
9. Zaki, M.J.: Sequence mining in categorical domains: incorporating constraints. In: Proc. of CIKM 2000, ACM Press (2000) 422–429
10. Garofalakis, M., Rastogi, R., Shim, K.: Mining sequential patterns with regular expression constraints. IEEE Trans. on Knowl. and Data Eng. **14** (2002) 530–552
11. Pei, J., Han, J., Wang, W.: Mining sequential patterns with constraints in large databases. In: Proc. of CIKM 2002, ACM Press (2002) 18–25
12. Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: Proc. of ICGI'94, Springer-Verlag (1994) 139–152
13. de la Higuera, C.: Characteristic sets for polynomial grammatical inference. Mach. Learn. **27** (1997) 125–138
14. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM (2002) 429–435

Efficient Graph-Based Representation of Web Documents

Alex Markov and Mark Last

Ben-Gurion University of Negev, Department of Information Systems Engineering
Beer-Sheva 84105, Israel

Abstract. In this paper we describe a new approach to classification of web documents. Most web classification methods are based on the vector space document representation of information retrieval. Recently the graph based web document representation model was shown to outperform the traditional vector representation using k-Nearest Neighbor (k-NN) classification algorithm. Here we suggest two new hybrid approaches to web document classification built upon both graph and vector representations. K-NN algorithm and three benchmark document collections were used to compare this method to graph and vector based methods separately. The results demonstrate that we succeed in most cases to outperform graph and vector approaches in terms of classification accuracy along with a significant reduction in classification time.

1 Introduction

Automated classification of previously unseen data items has been an active research area for many years. Many efficient and scalable classification techniques were developed in the areas of Machine Learning [7] and Data Mining [1]. Those techniques are used in wide range of research domains, text and web document categorization being one of them. Huge quantity of text documents, stored by organizations in local information systems, and billions of online web pages makes manual classification too costly and sometimes impossible. This situation requires development of more accurate and faster algorithms.

Most web categorization methods come from information retrieval where the "vector space model" for document representation is typically used. According to this model, a set of terms $T(t_1, t_2 \dots t_{|T|})$ called vocabulary is constructed from words located in the training document set. Each document d_i is represented by vector $(w_{\mathrm{i}1}, w_{\mathrm{i}2} \dots w_{\mathrm{i}|\mathrm{T}|})$ where number of vector dimensions $|T|$ is a number of terms in vocabulary and value of each dimension w_{ij} in vector symbolizes the weight of term t_j in that document. Weight can be a frequency of term t_j in document d_i or some defined measure as $TF * IDF$ [12] (term frequency * inverse document frequency) that gives a higher weight to terms that occur a lot in one document but little in any other documents. In most cases differences between various approaches are: (1) in the way of defining a term and (2) in the way of calculating the weight of a term.

Advantage of such representation model is that it can be used by most classification algorithms. For instance, many methods for distance or similarity calculation between two vectors were developed [1], [2], [7] so *lazy* k-NN algorithm can easily be used with one of those methods.

Such vector collection can also be transformed into one of conventional classification models. Examples of available classification models include decision trees [9], [10] IFN - info-fuzzy networks [5], artificial neural networks, NBC - Naïve Bayes Classifier [7] and many others. Those models associate vector collection with attribute table where every term in dictionary is an attribute and each $d_{i,j}$ is the value of attribute j in document i. Examples of applications of such approach to text documents can be found in [6], [15]. Ability to create a model is extremely important for systems where classification needs to be done online.

However, this popular method of document representation does not capture important structural information, such as the order and proximity of word occurrence or the location of a word within the document. Vector space, as most other existing information retrieval methods, also ignores the fact that web documents contains markup elements (HTML tags), which are an additional source of information. Thus, HTML tags can be used for identification of hyperlinks, title, underlined or bold text, etc. This kind of structural information may be critical for accurate internet page classification.

In order to overcome the limitations of the vector-space model, several methods of representing web document content using graphs instead of vectors were introduced [13], [14]. The main benefit of the proposed graph-based techniques is that they allow us to keep the inherent structural information of the original document. Ability to calculate similarity between two graphs allows to classify graphs with some distance-based *lazy* algorithms like k-NN, but available eager algorithms (like ID3, C4.5, NBC etc) work only with vectors and cannot induce even a simple classification model from a graph structure. On the other hand, *lazy* algorithms are very problematic in terms of classification speed and cannot be used for online massive classification of web documents represented by graphs.

In this paper we present a new method of web document representation, based on frequent sub-graph extraction, that can help us to overcome problems of both vector space and graph techniques [4]. Our method has two main benefits: (1) we keep important structural web page information by extracting relevant sub-graphs from a graph that represents this page; (2) we can use most eager classification algorithms for inducing a classification model because, eventually, a web document is represented by a simple vector with Boolean values.

The methodology we propose in this work is based on frequent sub-graph recognition that is beyond information retrieval or web content mining and belongs to the graph mining domain. As a general data structure, a graph can be used to model many complex relationships in data. Frequent sub-graph extraction or graph frequent pattern mining has been active research area in recent years. [8] is an example of using graphs for chemical compounds representa-

tion where labeled nodes represent different atoms and edges - different types of bonds among them. Most popular sub-structure detection algorithms based on BFS and DFS approaches are presented in [3] and [16] respectively.

This paper is organized as follows. In Section 2 we explain, step by step, the graph-based document representation approach. Our hybrid methods for document representation and classification with k-NN algorithm will be described in Sections 3 and 4 respectively. Benchmark document collections and comparative results are described in Section 5. Some conclusions are presented in Section 6.

2 Graph Based Web Document Representation

Before we describe the graph-based methodology, the definition of a graph should be given. A graph G is a 4-tuple: *G= (V, E, a, b)*, where V is a set of nodes (vertices), $E \subseteq V \times V$is a set of edges connecting the nodes, $\alpha : V \rightarrow \sum v$ is a function labeling the nodes, and$\beta : V \times V \rightarrow \sum e$ is a function labeling the edges ($\sum v$ and $\sum e$ being the sets of labels that can appear on the nodes and edges, respectively).

In [13] five different ways for graph representation of web documents were introduced. All those are based on the adjacency of terms in a document and some can also be used for plain (non-HTML) text document representation. In our work the *standard* graph representation was used because of the best results shown by this method compared to other techniques.

In order to convert document into graph some preprocessing steps where taken. First - all meaningless words (stop words such as "the", " of", and "and" in English) were removed from text. Those words do not convey information about document's specific subject so they are not needed for classification purposes. Second - stemming was done to bring all words with identical stem into one form (e.g. "students" and "student"). Stemming is often used in information retrieval to reduce the size of term vectors by conflating those terms which are considered to be identical after the removal of their suffixes. Porter stemming algorithm was used [11]. Third - extraction of document's most frequent words. To reduce the graph size and the resulting computational complexity only N most frequent words were taken for creation of graphs. N is the maximum number of nodes in future graphs (graph size).

Under the *standard* method each unique term (keyword) appearing in the document becomes a node in the graph representing that document. Each node is labeled with the term it represents. The node labels in a document graph are unique, since a single node is created for each keyword even if a term appears more than once in the text. Second, if word a immediately precedes word b somewhere in a "section" s of the document, then there is a directed edge from the node corresponding to term a to the node corresponding to term b with an edge label s. An edge is not created between two words if they are separated by certain punctuation marks (such as periods). Sections defined for the standard representation are: *title* (TI), which contains the text related to the document's

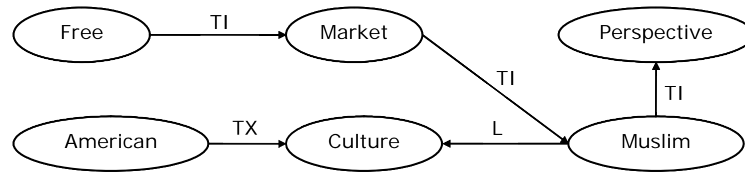


Fig. 1. Example of a Standard Graph Representation

title and any provided keywords (meta-data); *link* (L), which is text that appears in hyper-links on the document; and *text* (TX), which comprises any of the visible text in the document. An example of a standard graph representation of a short English web document is shown in Fig. 1.

3 Hybrid Document Representation

Our *hybrid* process for web document representation for the $k-NN$ classification algorithm is shown in Fig. 2.

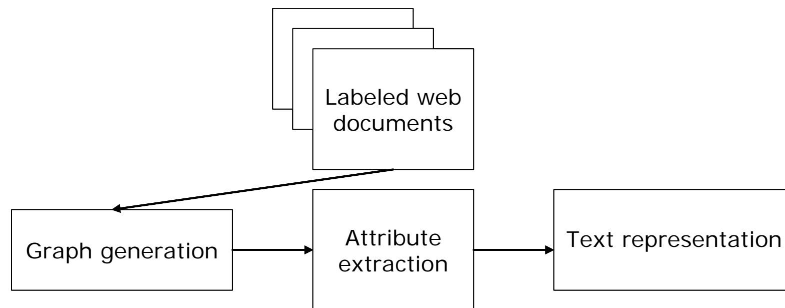


Fig. 2. Labeled web documents representation

The first, document representation stage, begins with a training collection of labeled documents $D = (d_1 \ldots d_{|D|})$ and a set of categories as $C = (c_1 \ldots c_{|C|})$, where each document $d_i \in D; 1 \leq i \leq |D|$ belongs to one and only one category $c_v \in C; 1 \leq v \leq |c|$. Three main actions need to be done at this stage. First - graph generation where graph representation of document is generated and a set of labeled graphs $G = (g_1 \ldots g_{|D|})$ is obtained. It is possible to use a limited graph size by defining parameter N, which is the maximum number of nodes in the graph, and using only N most frequent terms for graph construction, or use all document terms except stop words. Second - extraction of relevant

attributes (sub-graphs). Main goal of this stage is to find the attributes that are relevant for classification and create the vocabulary. We developed two different approaches to this task that are explained below. A group of attribute sub-graphs (terms) $T = (t_1 \ldots t_{|\mathrm{T}|})$ is the output of this stage. Third - text representation, that is representation of all document graphs as vectors of Boolean features corresponding to every sub-graph in G' ("1" - a sub-graph from the set appears in a graph). A set of binary vectors $V = (v_1 \ldots v_{|\mathrm{D}|})$ is the output of this stage.

Before term extraction methods will be explained, sub-graph and term should be defined. A graph $G_1 = (V_1, E_1, a_1, b_1)$ is a sub-graph of a graph $G_2 = (V_2, E_2, a_2, b_2)$, denoted $G_1 \subseteq G_2$, if $V_1 \subseteq V_2$,$E_1 \subseteq E_2 \cap (V_1 \times V_1)$, $\alpha_1 (x) = \alpha_2 (x)\ \forall x \in V_1$ and $\beta_1 (x, y) = \beta_2 (x, y)\ \forall (x, y) \in E_1$. Conversely, graph G_2 is also called a super-graph of G_1. In out representation methods, we define *terms* as sub-graphs selected by us to represent the document. Two optional term selection procedures are described below.

3.1 Naïve Extraction

All graphs representing the web documents are divided into groups by class attribute value (for instance: positive and negative in case of binary categorization). Then the frequent sub-graphs extraction algorithm is activated on each group with a user-specified threshold value $0 < t_{min} < 1$. Every sub-graph more frequent than t_{min} is chosen by algorithm to be a term (classification attribute) and stored in the vocabulary. All accepted groups of discriminating sub-graphs are combined to one set.

The Naïve method is based on a simple postulate that an attribute explains the category best if it is frequent in that category but in real life cases it is not necessarily the case. For example if sub-graph g is frequent in more than one category it can be chosen to be an attribute but cannot help us to discriminate instances belonging to those categories. The "smart" extraction method has been developed by us to overcome this problem.

3.2 Smart Extraction

Like in the Naïve representation, all graphs representing the web documents should be divided into groups by class attribute value. In order to extract discriminating sub-graphs several measures should be defined, as follows:

SCF — Sub-graph Class Frequency:

$$SCF\left(g'_k\left(c_i\right)\right) = \frac{g'_k f\left(c_i\right)}{N\left(c_i\right)}$$

where:

$SCF\left(g'_k\left(c_i\right)\right)$ — Frequency of sub-graph g'_k in category c_i.
$g'_k f\left(c_i\right)$ — Number of graphs that contains sub-graph g'_k.
$N\left(c_i\right)$ — Number of graphs in category c_i.

ISF — Inverse Sub-graph Frequency:

$$ISF\left(g'_k\left(c_i\right)\right) = \begin{cases} \log_2\left(\frac{\sum N(c_j)}{\sum g'_k f(c_j)}\right) & \text{if } \sum g'_k f(c_j) > 0 \\ \log_2\left(2 \times \sum N(c_j)\right) & \text{if } \sum g'_k f(c_j) = 0 \end{cases} \quad \{\forall c_j \in C;\ j \neq i\}$$

where:

$ISF\left(g'_k\left(c_i\right)\right)$ — Measure for inverse frequency of sub-graph g'_k in category c_i.

$N(c_j)$ — Number of graphs belonging to all categories except of c_i.

$g'_k f(c_j)$ — Amount of graphs that contain g'_k belonging to all categories except c_i.

And finally: *CR* — Classification Rate:

$$CR\left(g'_k\left(c_i\right)\right) = SCF\left(g'_k\left(c_i\right)\right) \times ISF\left(g'_k\left(c_i\right)\right)$$

where:

$CR\left(g'_k\left(c_i\right)\right)$ — Classification Rate of sub-graph g'_k in category c_i.

This measure indicates how well g'_k can explain category c_i. $CR\left(g'_k\left(c_i\right)\right)$ will reach its maximum value when every graph in category c_i contains g'_k and graphs in other categories do not contain it at all.

According to the *Smart* method, CR_{min} (minimum classification rate) will be defined and only sub-graphs with higher *CR* value than CR_{min} will be chosen as terms and entered into the vocabulary.

Calculation of Classification Rate for each candidate sub-graph is a slightly more complicated and time expensive procedure than finding frequency only in the *Naïve* approach, because of *ISF* calculation when graphs from other categories are taken into account. Notwithstanding, as can be seen below, in most cases using *Smart* representation brings to better accuracy results.

3.3 Frequent sub-graph extraction problem

The input of sub-graph discovery problem, in our case is a set of labeled, directed graphs and parameter t_{min} such that $0 < t_{min} < 1$. The goal of the frequent sub-graph discovery is to find all connected sub-graphs that occur in at least (t_{min}*100) % of the input graphs. Additional property of our graphs is that labeled vertex is unique in each graph. This fact makes our problem mach easier then standard sub-graph discovery case [16], [3] where such limitation does not exist. The most complex task in frequent sub-graph discovery problem is *sub-graph isomorphism identification*[1]. It is known as NP-complete problem where nodes in the graph are not uniquely labeled but in our case it has polynomial $O(n^2)$ complexity.

We use *breadth first search* (BFS) approach to find frequent sub-graphs. Our extraction process is given in Algorithm 1. First we detect all frequent nodes in the input set of graphs and insert them into frequent subgraph set. Each

[1] Means that graph is isomorphic to a part of another graph.

iteration of the loop (row 3) we try to extend each frequent subgraph by finding subgraph isomorphism between it and the graphs from input set and adding to the subgraph outgoing edge (row 7). Then we construct set of all possible candidate subgraphs (rows 8 to 13). We place appropriate candidates in the frequent set (row 14) and return the union of all frequent subgraph sets we got after each iteration (row 16). All relevant notations are given in Table 1.

Algorithm 1 Frequent subgraph extraction $(G,\ t_{min})$

1: $F^0 \leftarrow$ Detect all frequent 1 node subgraphs (nodes) in G
2: $k \leftarrow 1$
3: **While** $(F^{k-1} \neq \theta)$ **Do**
4:   **For Each** subgraph $sg^{k-1} \in F^{k-1}$ **Do**
5:     **For Each** graph $g \in G$ **Do**
6:       **If** $sg^{k-1} \subseteq g$ **Then**
7:         $E^k \leftarrow$ Detect all possible k edge extensions of sg^{k-1} in g
8:   **For Each** extension subgraph $sg^k \in E^k$ **Do**
9:     **If** sg^k allready a member of C^k **Then**
10:       $\{sg^k \in C^k\}.Count++$
11:     **Else**
12:       $sg^k.Count \leftarrow 1$
13:       $C^k \leftarrow sg^k$
14:   $F^k \leftarrow \{sg^k \in C^k | sg^k.Count > t_{min} \times |G|\}$
15:   $k++$
16: **Return** $F^1, F^2 \ldots F^{k-2}$

Table 1. Notation used in Algorithm 1

Notation	Description
G	Training set of documents represented by graphs
t_{min}	Subgraph frequency threshold
k	Number of edges in the graph
g	Single graph
sg^k	Subgraph with k edges
F^k	Set of frequent subgraphs with k edges
E^k	Set of extension subgraphs with k edges
C^k	Set of candidate subgraphs with k edges

4 Document Classification

Classification process with $k-NN$ algorithm is executed as follows (see Fig. 3). Previously unseen data item (web document) should be converted into graph g. Then $|g'|$ dimensional vector v of Boolean features corresponding to every sub-graph in G' ("1" - a sub-graph from the set appears in a graph) is generated from g. Next, distance between v and each vector in V is calculated and k vectors closest to v are chosen. Once we have found the k nearest training instances using some distance measure, we examine them and determine which class is held by the majority of those instances. This class is then assigned as the predicted class for the input instance. Since we deal with vectors, a lot of distance or similarity measures like Cosine [12], Manhattan Distance [1], etc are available.

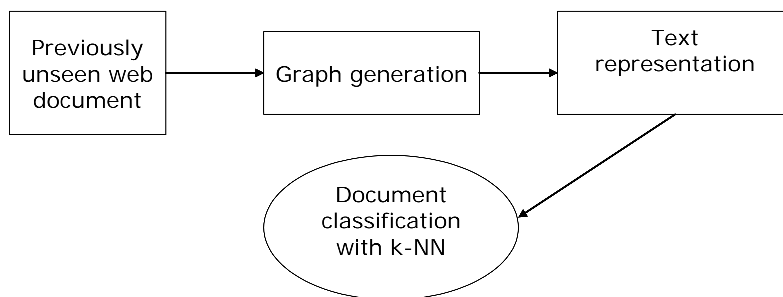


Fig. 3. Classification of previously unseen document

5 Comparative evaluation of results

In order to evaluate the performance of proposed method we performed several experiments on three different benchmark collections of web documents, called the F-series, the J-series, and the K-series. The same collections were used in [13] and [14] for comparative evaluation. Documents in those collections were originally news pages hosted at Yahoo (www.yahoo.com) and they were downloaded from ftp://ftp.cs.umn.edu/dept/user/boley/PDDPdata/.

To evaluate our classification approach we used k-NN as classification algorithm and Manhattan Distance [1] as distance measure. Manhattan Distance was chosen because of its ability to work with Boolean vectors and it was calculated as follows: $Distance(i,j) = |d_{\mathrm{i1}} - d_{\mathrm{j1}}| + |d_{\mathrm{i2}} - d_{\mathrm{j2}}| + \ldots + |d_{\mathrm{i|d|}} - d_{\mathrm{j|d|}}|$. As benchmark, we have taken the most accurate results of *vector space* and *graph based* models presented in [14]. Similarly the results of our methods (*Hybrid Naïve* and *Hybrid Smart*) are shown only for the values of input parameters that maximizing classification accuracy. These parameters include estimation Graph Size

N, Minimum Sub-graph Frequency Threshold $t_{\min}$ and Minimum Classification Rate $CR_{\min}$. In all cases *leave-one-out* method was used for accuracy evaluation. In J and K series (see Fig. 5 and Fig. 6), our hybrid method tends to outperform

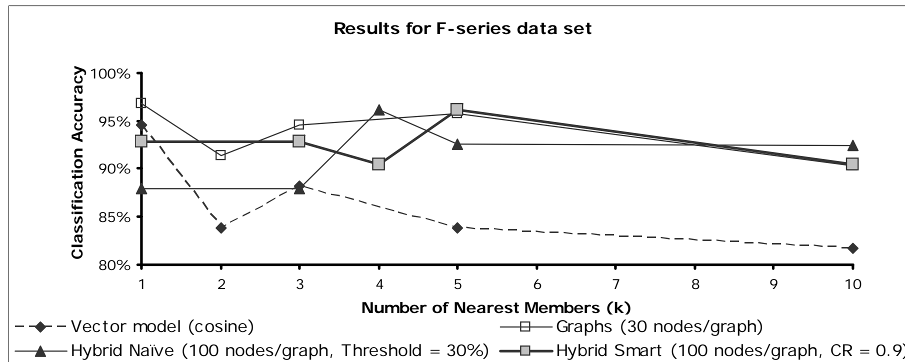


Fig. 4. Comparative results for the F-series

graph and vector methods in terms of classification accuracy. Especially, in J series (see Fig. 5), the *Smart hybrid* approach has reached better accuracy for all values of k. As to the F series collection (Fig. 4), *Graph* method has shown better accuracy results than *Hybrid* methods for most values of k, but we still succeed to get best accuracy for $k = 5$. Inferior result for other values of k can possibly be explained by the collection size. F is the smallest document collection in our experiments.

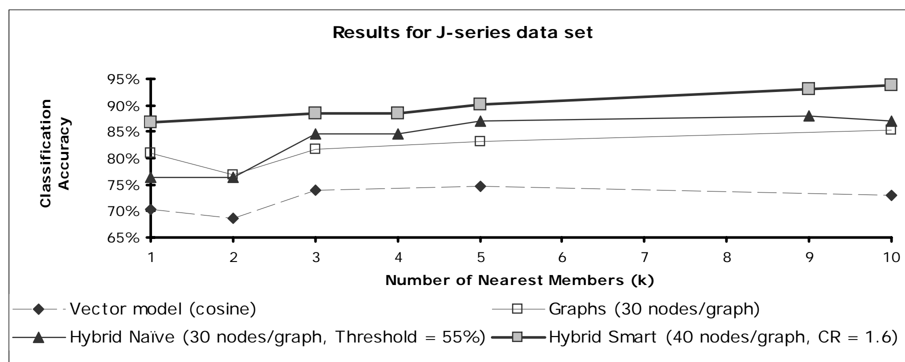


Fig. 5. Comparative results for the J-series

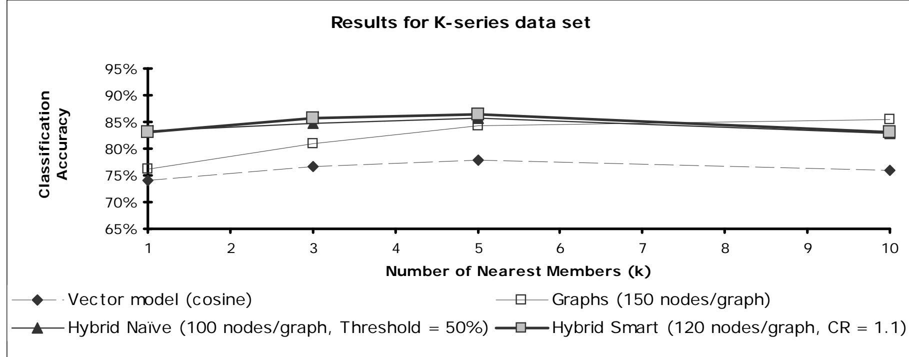


Fig. 6. Comparative results for the K-series

We also measured and compared the execution time needed to classify one document in the K-series data set, which was the most time-consuming collection for each method. Results are presented in Table. 2. Average time to classify one document for vector and graph models was taken from [13]. That time for our *hybrid* method was calculated under the same system conditions for more than 50 experiments.

Timing results are shown for higher accuracy cases with each method. The major improvement in execution time can be explained by the fact that we used shorter vectors (156 dimensions in *Naïve* and 137 in *Smart* case respectively) in contrast with 1458 in the vector space model to reach even better accuracy. In addition, our vectors take binary values, giving us the ability to use *xor* function for calculating the Manhattan Distance between each two vectors as follows: $Distance(i,j) = |d_{\mathrm{i1}} \otimes d_{\mathrm{j1}}| + |d_{\mathrm{i2}} \otimes d_{\mathrm{j2}}| + \ldots + |d_{\mathrm{i|d|}} \otimes d_{\mathrm{j|d|}}|$, which is computationally faster than calculating the cosine distance between non-binary vectors.

Table 2. Average time to classify one K-series document for each method

Method	Average time to classify one document
Vector (cosine)	7.8 seconds
Graphs, 100 nodes/graph	24.62 seconds
Hybrid Naïve, 100 nodes/graph, $t_{\mathrm{min}} = 50\%$	0.017 seconds
Hybrid Smart, 120 nodes/graph, $CR_{\mathrm{min}} = 1.1$	0.016 seconds

6 Conclusions

This paper has empirically compared four different representations of web documents in terms of classification accuracy and execution time. The proposed *hybrid* approaches were found to be more accurate in most cases and generally much faster than their vector-space and graph-based counterparts. Finding the optimal Graph Size N, Minimum Sub-graph Frequency Threshold t_{min} and Minimum Classification Rate CR_{min} is a subject for our future research. In addition, we are going to classify web documents using our *hybrid* representations with model-based algorithms such as ID3, C4.5 or Naïve Bayes. We expect an additional reduction of classification time as a result of using the *hybrid* representation with these algorithms. Then we will evaluate the ability of our *hybrid* representation approaches to classify documents in other languages such as Hebrew and Arabic.

7 Acknowledgments

This work was partially supported by the National Institute for Systems Test and Productivity at University of South Florida under the USA Space and Naval Warfare Systems Command Grant No. N00039-01-1-2248. We thank Dr. Adam Schenker for his technical assistance.

References

1. J. Han, M. Kamber, "Data Mining Concepts and Techniques", Morgan Kaufmann 2001.
2. A.K. Jain, M.N. Murty, and P.J. Flynn, "Data Clustering: A Review", ACM Computing Surveys, Vol. 31, No. 3, 1999
3. M. Kuramochi and G. Karypis, "An Efficient Algorithm for Discovering Frequent Subgraphs", Technical Report TR♯ 02-26, Dept. of Computer Science and Engineering, University of Minnesota, 2002.
4. A. Markov and M. Last, "A Simple, Structure-Sensitive Approach for Web Document Classification", in P.S. Szczepaniak et al. (Eds.), Advances in Web Intelligence, Proceedings of the 3rd Atlantic Web Intelligence Conference (AWIC 2005), Springer-Verlag, LNAI 3528, pp. 293298, Berlin Heidelberg 2005.
5. O.Maimon, and M.Last, Knowledge Discovery and Data Mining - The Info-Fuzzy Network (IFN) Methodology, Kluwer Academic Publishers, 2000.
6. A. McCallum, K. Nigam,"A Comparison of Event Models for Naive Bayes Text Classification", AAAI-98 Workshop on Learning for Text Categorization, 1998.
7. T. M. Mitchell, "Machine Learning", McGraw-Hill, 1997.
8. D. Mukund, M. Kuramochi and G. Karypis, "Frequent sub-structure-based approaches for classifying chemical compounds", ICDM 2003, Third IEEE International Conference, 2003.
9. J.R. Quinlan, "Induction of Decision Trees", Machine Learning, 1:81–106, 1986.
10. J.R. Quinlan, "C4.5: Programs for Machine Learning", 1993.
11. M. Porter, "An algorithm for suffix stripping", Program Vol. 14, No. 3, 130-137, 1980.

12. G. Salton, A. Wong, and C. Yang, "A vector space model for automatic indexing", Communications of the ACM, 18(11):613–620, 1971.
13. A. Schenker, H. Bunke, M. Last, A. Kandel, "Graph-Theoretic Techniques for Web Content Mining", World Scientific, 2005.
14. A. Schenker, M. Last, H. Bunke, A. Kandel, "Classification of Web Documents Using Graph Matching", International Journal of Pattern Recognition and Artificial Intelligence, Special Issue on Graph Matching in Computer Vision and Pattern Recognition, Vol. 18, No. 3, pp. 475-496, 2004.
15. S. M. Weiss, C. Apte, F. J. Damerau, D. E. Johnson, F. J. Oles, T. Goetz and T. Hampp, "Maximizing Text-Mining Performance", IEEE Intelligent Systems, Vol.14, No.4. Jul. /Aug. 1999. Pp.63-69.
16. X. Yan and J. H. Gspan," Graph-based substructure pattern mining", Technical Report UIUCDCS-R-2002-2296, Department of Computer Science, University of Illinois at UrbanaChampaign, 2002.

Computation-time efficient and robust attribute tree mining with DRYADEPARENT

Alexandre Termier[1], Marie-Christine Rousset[2], Michèle Sebag[2], Kouzou Ohara[1], Takashi Washio[1], and Hiroshi Motoda[1]

[1] I.S.I.R., Osaka University
8-1, Mihogaoka, Ibarakishi, Osaka, 567-0047, Japan
termier@ar.sanken.osaka-u.ac.jp
[2] CNRS & Université Paris-Sud (LRI) & INRIA (Futurs)
Building 490, Université Paris-Sud, 91405 Orsay Cedex, France

Abstract. In this paper, we present a new tree mining algorithm, DRYADEPARENT, based on the hooking principle first introduced in DRYADE [1]. In the experiments, we demonstrate that the branching factor and depth of the frequent patterns to find are key factor of complexity for tree mining algorithms, even if often overlooked in previous work. We show that DRYADEPARENT outperforms the current fastest algorithm, CMTreeMiner, by orders of magnitude on datasets where the frequent patterns have a high branching factor.

1 Introduction

In the last ten years, the frequent pattern discovery task of data mining has expanded from simple itemsets to more complex structures: for example sequences, episodes, trees or graphs. In this paper we focus on *tree mining*, that is finding frequent tree-shaped patterns in a database of tree-shaped data. Tree mining can lead to many practical applications in the areas of computer networks, bioinformatics, XML documents databases mining, and hence have received a lot of attention from the research community in recent years. Most of the well-known algorithms use the same generate-and-test principle that made the success of frequent item set algorithms. The main adaptation to the tree case is the design of efficient candidate tree enumeration algorithms in order to avoid generating redundant candidates, and to enable efficient pruning. However, the search space of tree candidates is huge, particularly when the frequent trees to find have both high depth and high branching factor. Especially the high branching factor case has received very little attention by the tree mining community. However, performances of existing algorithms are dramatically affected by the branching factor of the patterns to find, as shown in our experiments.

Starting from this observation, we have developped the DRYADEPARENT algorithm. This algorithm is an adaptation of our earlier algorithm DRYADE [1]. DRYADE is based on a more general tree inclusion definition appropriate for mining highly heterogeneous collections of tree data. DRYADEPARENT follows

the same principles of Dryade, but uses a standard inclusion definition [2, 3] to make possible performance comparison with other existing systems based on different principles. We will show in this paper that DryadeParent outperforms the up-to-date CMTreeMiner algorithm [3], and conduct a thorough study on the influence of structural characteristics of the patterns to find, like depth and branching factor, on the computing time performance of both algorithms.

The outline of the paper is as follows. Section 2 introduces the notations and definitions used throughout the paper. Section 3 presents and discusses the state of the art in tree mining. Section 4 gives an overview of the DryadeParent algorithm. Section 5 reports detailed comparative experiments, both on real and artificial datasets. In section 6, we conclude and give some directions for future work.

2 Formal Background

Let $L = \{l_1, ..., l_n\}$ be a set of labels. A *labelled tree* $T = (N, A, root(T), \varphi)$ is an acyclic connected graph, where N is the set of nodes, $A \subset N \times N$ is a binary relation over N defining the set of edges, $root(T)$ is a distinguished node called the *root*, and φ is a labelling function $\varphi : N \mapsto L$ assigning a label to each node of the tree. We assume without loss of generality that edges are unlabelled: as each edge connects a node to its parent, the edge label can be considered as part of the child node label.

A tree is an *attribute tree* if φ is such that two sibling nodes cannot have the same label (more details on attribute trees can be found in [2]).

Let $u \in N$ and $v \in N$ be two nodes of a tree. If there exists an edge $(u, v) \in A$, then v is a *child* of u, and u is the *parent* of v. If there exists a path from u to v in the tree, then v is a *descendant* of u, and u is an *ancestor* of v.

Tree inclusion: Let $AT = (N_1, A_1, root(AT), \varphi_1)$ be an attribute tree and $T = (N_2, A_2, root(T), \varphi_2)$ be a tree. AT is an *induced subtree* of T if there exists an injective mapping $\mu : N_1 \mapsto N_2$ such that: 1) μ preserves the labels: $\forall u \in N_1 \;\; \varphi_1(u) = \varphi_2(\mu(u))$ and 2) μ preserves the parent relationship: $\forall u, v \in N_1$ $(u, v) \in A_1 \Leftrightarrow (\mu(u), \mu(v)) \in A_2$.

This relation will be written $AT \sqsubseteq T$, and we will sometimes say that AT is included into T.

If $AT \sqsubseteq T$, the set of mappings supporting the inclusion is denoted $\mathcal{EM}(AT, T)$. The set of *occurrences* of AT in T, denoted $Locc(AT, T)$, is the set of nodes of T onto which the root of AT is mapped by a mapping of $\mathcal{EM}(AT, T)$.

We also introduce the notion of *image* of an attribute tree AT in a tree T. The set of images of AT into T is the set of (attribute) trees obtained by mapping AT onto T by applying the mappings from $\mathcal{EM}(AT, T)$.

Frequent attribute trees: We can now define the problem of finding *frequent attribute trees* in a tree database. Let $TD = \{T_1, ..., T_m\}$ be a tree database. The *datatree* D_{TD} is the tree whose root is an unlabelled node, having the trees $\{T_1, ..., T_m\}$ as its direct subtrees.

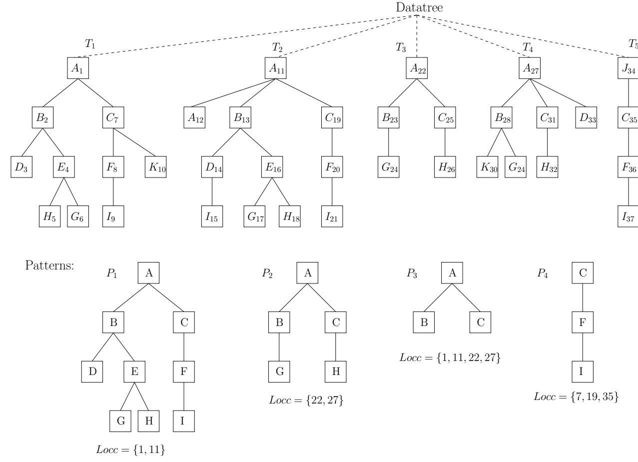


Fig. 1: Datatree example (node identifiers are subscripts of node labels), and patterns for $\varepsilon = 2$

The *support* of an attribute tree AT in the datatree can be defined in two ways:

- $support_d(AT) = \sum_{i=1}^{m} \sigma_d(AT, T_i)$ where $\sigma_d(AT, T_i) = 1$ if $AT \sqsubseteq T_i$, 0 otherwise. *(document support)*
- $support_o(AT) = \sum_{i=1}^{m} \sigma_o(AT, T_i)$ where $\sigma_o(AT, T_i) = |Locc(AT, T_i)|$ *(occurrences support)*

In this paper, we are interested in finding attribute trees frequent by document support. The term *support* will now be used for document support. But for the sake of completeness, our algorithm needs to keep track of all frequent occurrences, and will use the occurrences support for processing.

Let ε be an absolute frequency threshold. AT is a frequent attribute tree of D_{TD} if $support_d(AT) \geq \varepsilon$. The set of all frequent attribute trees is denoted by $\mathcal{F}(D_{TD}, \varepsilon)$, and by abuse of notation we will only denote it as $\mathcal{F}$ in the rest of this paper.

In the example of Fig. 1, with a support threshold of $\varepsilon = 2$, the attribute trees P_1, P_2, P_3, P_4 are all frequent by document support in the datatree.

Closed trees: A frequent attribute tree is *closed* if it is maximal, according to inclusion, for its set of occurrences.

Definition 1. *A frequent attribute tree $AT \in \mathcal{F}$ is closed either if it is not included into any other frequent attribute trees, or if it is included into a frequent attribute tree $AT' \in \mathcal{F}$, there exists a mapping in $\mathcal{EM}(AT, D_{TD})$ which is not in the mappings of $\mathcal{EM}(AT', D_{TD})$.*

We will denote the set of all closed frequent attribute trees as $\mathcal{C}$, with the same abuse of notation as before.

In our example P_1 and P_2 are closed because they are not included into any other frequent attribute tree, P_3 is closed because even if it is included into P_1 and P_2, neither the occurrences of P_1 nor the occurrences of P_2 can cover all the occurrences of P_3, and in the same way P_4 is also closed as even if it is included into P_1, its mapping starting at occurrence 35 is not contained in any mapping of P_1.

Tree mining problem: The tree mining problem we are interested in is to find all the closed frequent attribute trees for a given datatree and support threshold. The merit of this problem is that the number of closed frequent attribute trees is smaller than the number of all frequent attribute trees, but the amount of information is the same in both cases: all the frequent attributes trees can be easily deduced from the closed frequent attribute trees. Thus finding such closed trees enables faster mining without loss of information. From now on, we will refer to the closed frequent attribute trees as *patterns*.

3 Related work

Most tree mining algorithms deal with finding *all* the frequent subtrees from a collection of trees. One pioneering work is Asai & al.'s Freqt algorithm [4], discovering all frequent induced subtrees with preservation of the order of the siblings. The other pioneering work is Zaki's *TreeMiner* [5], using a more relaxed inclusion definition where the order still has to be preserved, but instead of the parent relationship the mapping has only to preserve the ancestor relationship. Both these algorithms extend the Apriori algorithm [6] principle to trees: they use efficient candidate tree generation procedures, that cover all the search space without generating twice the same tree, and for each candidate test its frequency against the data. The enumeration technique builds a new candidate by adding one edge to a previously found frequent tree, along its *rightmost branch*.

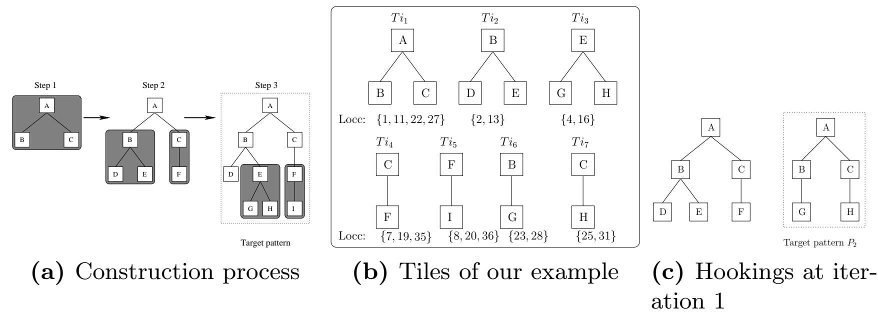
The second generation of tree mining algorithms has been designed to get rid of the order preservation constraint. This was realised by basing the enumeration procedures on canonical forms, one canonical form representing all trees that are isomorphic except for the order of siblings. Such work include the Unot algorithm by Asai & al. [7], the work of Nijssen & al. [8] and the recent Sleuth algorithm by Zaki [9].

There are still very few algorithms mining closed frequent trees. We already mentioned our Dryade algorithm [1], which relies on a very general tree inclusion definition and a new *hooking* principle. The only algorithm mining closed frequent induced subtrees is the CMTreeMiner algorihm of Chi & al. [3]. It uses the same generate and test principle as other tree mining algorithms, extended to handle closure. This algorithm has shown excellent experimental results. Recently, Arimura & Uno proposed the Clott algorithm [2] for mining closed frequent attribute trees, in the same settings as those of this paper. This algorithm has a proved output-polynomial time complexity, which should also give excellent performances. Up to now there is not yet an implementation available.

4 The DryadeParent algorithm

We propose in this section an improved way of exploring the search space, namely the DryadeParent algorithm. This method provides important perfomance gains for complex patterns.

Our goal is to find all the patterns in $\mathcal{C}$. Instead of discovering them edge by edge as done by most algorithms, we are interested in discovering the patterns depth level by depth level, starting with the root and finishing with the deepest leaves in a breadth-first fashion. An example of this discovery process is shown in Fig. 2(a).



(a) Construction process **(b)** Tiles of our example **(c)** Hookings at iteration 1

Fig. 2: Tiles and hookings

We call *tiles* the attribute trees that must be added to discover a new depth level of a pattern. In Fig. 2(a) such tiles are enclosed in shaded boxes. A tile is a frequent attribute tree made from a node of a pattern of $\mathcal{C}$ and all its children. The set of all tiles is noted $\mathcal{T}$.

The essence of the DryadeParent algorithm is to first discover all the tiles that are in the datatree, and then apply a fast levelwise strategy to *hook* these tiles together and compute the patterns of $\mathcal{C}$.

4.1 Discovering the tiles

Let us denote by $\mathcal{F}_1$ the subset of $\mathcal{F}$ made of the frequent attribute trees of depth 1. The set of closed frequent attribute trees of depth 1, denoted by $\mathcal{C}_1$, is the closure of $\mathcal{F}_1$, defined with Definition 1 where $\mathcal{F}$ is replaced by $\mathcal{F}_1$. DryadeParent relies on the following property:

Property 1. The set of tiles is exactly the set of closed frequent attribute trees of depth 1, i.e.: $\mathcal{T} = \mathcal{C}_1$

Computing such closed frequent attribute trees of depth 1 can be efficiently done by constructing for each label l a matrix M_l where each line corresponds to

a node of label l in the datatree, and with as many columns as labels in L, so that a 1 in cell (i, j) indicates that the node corresponding to line i has a child with label l_j. Applying a closed frequent itemset discovery algorithm like LCM2 [10] on matrix M_l will discover all the closed frequent attribute trees of depth 1 with a root of label l (to comply with document support, one only has to prune the occurence-frequent trees that do not meet the document frequency constraint). By repeating the process for all the labels of L, all the closed frequent attribute trees of depth 1, i.e. all the tiles, are discovered. The Fig. 2(b) shows the tiles for our example.

4.2 Hooking the tiles

Having found the tiles, the goal of DryadeParent is to compute efficiently all the patterns through hookings of these tiles. We have chosen a levelwise strategy, where each iteration computes the next depth level for the patterns being constructed.

Root tiles To begin with, the tiles that correspond to the depth levels 0 and 1 of the patterns must be found in the set of tiles. Such tiles are called *root tiles*, for they are the starting point of pattern construction by our *hooking* principle. Some of these root tiles can be found in a straightforward manner: these are the tiles whose root cannot be mapped to the same node as the leaves of any other tiles. We call them *initial root tiles*. In our example Ti_1 is the only initial root tile because its occurrences 1, 11, 22 and 27 are never leaves in any other tile.

The other root tiles are not as easily found. As some of their root nodes can be mapped to the leave nodes of other tiles, these tiles are subtrees in some patterns, and root in some other patterns. For example, Ti_4 is as well a subtree in P_1, and the root tile of P_4. For the sake of efficiency, it must be avoided as much as possible to identify incorrectly a tile as a root tile, this would lead to the construction of an attribute tree that would in fact only be a subtree of a pattern, i.e. having done redudant computation and getting an unclosed result. To avoid this, the starting points of DryadeParent are the initial root tiles, and at the end of each iteration new root tiles are looked for. We will explain how in the "Preparing next iteration" subsection.

In the following, we will denote by $\mathcal{RP}_i$ the attribute trees actually constructed by the algorithm at iteration i, and by $\mathcal{C}_{\mathcal{RP}_i}$ the patterns that will be obtained by successive hookings on the attributes trees of $\mathcal{RP}_i$ at the end of the algorithm. $\mathcal{C}_{\mathcal{RP}_i}$ is for illustration purposes, and is not actually constructed by the algorithm. In the example, $\mathcal{RP}_0 = \{Ti_1\}$ and $\mathcal{C}_{\mathcal{RP}_0} = \{P_1, P_2, P_3\}$ of Fig. 1.

Hooking The initial root tiles are the entry point to the main iteration of DryadeParent. In iteration i, for each element T of $\mathcal{RP}_i$ the algorithm will discover all the possible ways to add one depth level to T w.r.t. the patterns to get. This is done via the **hooking** operation: for an integer i, let T be an element of $\mathcal{RP}_i$, and $C \in \mathcal{C}_{\mathcal{RP}_i}$ such that the structures of T and C are isomorphic for

all depth until i. The *hooking* operation consists in constructing a new attribute tree T' by hooking a set of *hooking tiles* $\{Ti_1, ..., Ti_k\}$ on the leaves of T such that the occurrences of T' include those of C, and the structures of T' and C are isomorphic for all depths until $i + 1$.

The subtle point is to find all the frequent hooking tile sets for an element T of $\mathcal{RP}_i$. The potential hooking tiles on T are all tiles whose root is mapped to a leaf node of T. In our example, the potential hooking tiles on Ti_1 are $\{Ti_2, Ti_4, Ti_6, Ti_7\}$. Among all these potential hooking tiles, we want to find those which frequently appear together according to the occurrences of T. This is a closed frequent itemset discovery problem, and we can solve it by creating a matrix M whose each line k corresponds to an occurrence o_k of T, and each column j corresponds to a potential hooking tile Ti_j. $M[i, j] = 1$ iff. for the occurrence o_k of T, a leaf of T is mapped to the same node as the root of Ti_j. Applying a closed frequent itemset discovery algorithm like LCM2 on M enables discovering efficiently all the closed frequent hooking tile sets.

In our example, the frequent hooking tile sets on Ti_1 are $\{Ti_2, Ti_4\}$ and $\{Ti_6, Ti_7\}$. These hookings are illustrated in Fig. 2(c). It can be seen that the pattern P_2 has been discovered.

The frequent attribute trees discovered that are not yet patterns are inserted into $\mathcal{RP}_{i+1}$ for further expansion in the next iteration.

Preparing next iteration *Next level root tiles:* Once all the uses of a tile as a hooking tile have been discovered, either all the occurrences of this tile have been involved in a hooking for the creation of a particular attribute tree, or some occurrences have never been used together in a hooking. In the latter case, this tile becomes a root tile at the next iteration, for we are sure that starting from all the occurrences of this tile will produce a new closed attribute tree. For example at the end of the first iteration, all the possible hookings of Ti_4 have been discovered. But the occurrence 35 of this tile has never been used, so tile Ti_4 becomes a root tile in iteration 2, enabling the discovery of pattern P_4.

This way all the patterns of $\mathcal{C}$ are discovered by DryadeParent (this is proved in an even more general case in [11]).

Closure checking: Another important point is that in some cases hooking can lead to attribute trees that are not closed. Such cases can be detected quickly by analysing the hookings, for this purpose DryadeParent keeps a database of all the hookings performed so far. When a new hooking is proposed, the algorithm checks that this new hooking satisfies the closure property w.r.t. the hookings of the database. Two non-closure cases can arise: 1) the new hooking is included into an existing hooking, then the new hooking is discarded; 2) the new hooking includes an existing hooking, then the existing hooking and the corresponding pattern are erased from the database, and a new pattern is created from the new hooking, which is registered into the hooking database.

5 Experiments

This section reports on the experimental validation of DryadeParent on real-world and artificial datasets. All runtimes are measured on 2.8 GHz Intel Xeon processor with 2GB memory (Rocks 3.3.0 Linux). DryadeParent is written in C++, involving the closed frequent itemset algorithm LCM2 [10], kindly provided by Takeaki Uno. Reported results are wall-clock runtimes, including data loading and preprocessing.

5.1 Real datasets

In the tree mining litterature, two real-world datasets are widely used for testing: the NASA dataset sampled by Chi & al. from multicast communications during a shuttle launch event [12], and the CSLOGS dataset consisting of web logs collected over one month at the CS department of Rensselaer Institute [5].

The runtimes obtained for various frequency thresholds for both DryadeParent and CMTreeMiner are displayed on Fig. 3.

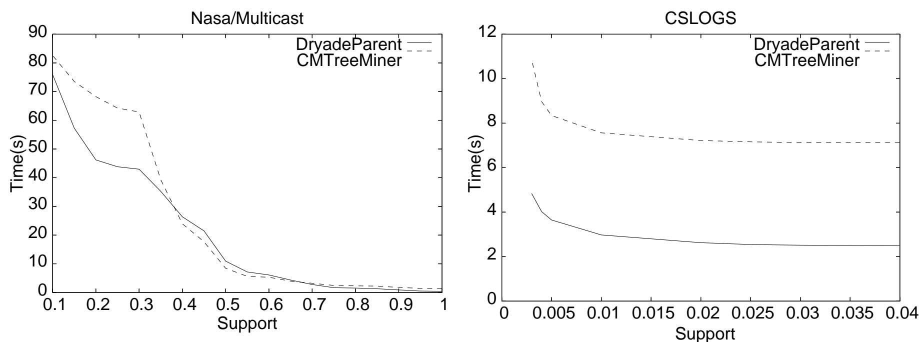


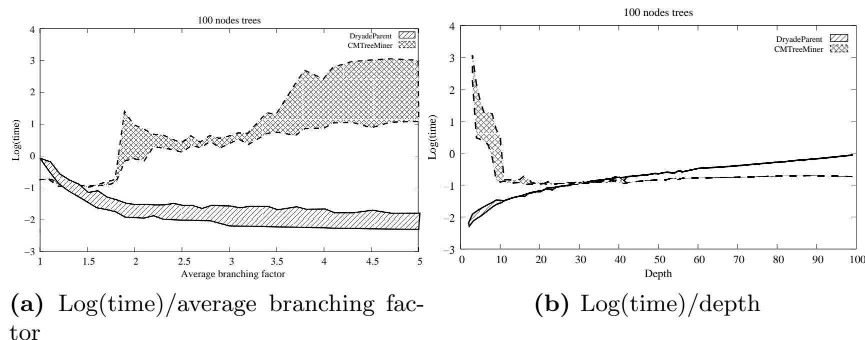
Fig. 3: Running time w.r.t. support for the Nasa/Multicast and CSLOGS datasets.

DryadeParent is more than twice faster than CMTreeMiner on the CSLOGS dataset. For the NASA dataset the performances are similar for high and medium support values, DryadeParent having a distinct advantage for the lowest support values. Note that we obtained similar results with simplified CSLOGS and NASA datasets consisting only of attribute trees. We were interested to know why DryadeParent and CMTreeMiner have a bigger performance difference on the CSLOGS dataset than on the NASA dataset. Analysing the structure of the computed patterns in both cases, we found that in the CSLOGS dataset, for the support value 0.003 (lowest value tested), there are 924 patterns, with 3 nodes on average, and an average branching factor of 1.6. For the NASA dataset, the picture is different: at the support value 0.1, there are 737 patterns, with 42 nodes on average, an average depth of 12 and an average branching factor of 1.2. So patterns of NASA and patterns of CSLOGS have very different

characteristics, and lead to different performance results for CMTreeMiner and DryadeParent. Artificial datasets will be used to get a deeper understanding of the influence of the structure of the patterns on performance of these two algorithms.

5.2 Artificial datasets

In the usual tree mining algorithms studies, at most the length (i.e. the number of nodes) of the found patterns is reported, without any information about the structure of these patterns. However, branching factor and depth of the patterns intervene directly in the candidate generation process, so they are likely to play a major role w.r.t. the computation time. To ascertain this hypothesis, we wrote a random tree generator that can generate trees with a given node number N and a given average branching factor b. Nodes are labelled with their pre-order identifier, so there are no couples of nodes with the same label in a tree. We generated trees with $N = 100$ nodes and $b \in [1.0; 5.0]$, b increasing by increment of 0.1. For each value of b, 10000 trees were generated. Let T be such a tree. For each T a dataset D_T was generated, consisting simply of 200 identical copies of T (we perform this 200-times duplication of each T to increase the processing time for D_T and so reduce the error rate on time measurement). Each D_T was processed by both algorithms, with a support threshold of 200 (hence the pattern to find is the tree T), and the processing time was recorded. Eventually, for each value of b we regrouped the trees by their depth d, and got a point (b, d) by averaging the processing times for all the trees of average branching factor b and depth d. Fig. 4(a) shows the logarithms of these averaged time values w.r.t. the average branching factor b, and Fig. 4(b) shows the logarithms of these averaged time values w.r.t. the depth d.



(a) Log(time)/average branching factor

(b) Log(time)/depth

Fig. 4: Random trees with 100 nodes

The Fig. 4(a) shows that DryadeParent is orders of magnitude faster than CMTreeMiner as long as the branching factor exceeds 1.3, that is the case in most

of the experiments space. For lower branching factor values, CMTreeMiner has a small advantage. Patterns with such a low branching factor necessarily have a high depth, this is confirmed by Fig. 4(b). This figure shows that DryadeParent exhibits a linear dependency on the depth of the patterns. This is not surprising: each iteration of DryadeParent computes one more depth level of the patterns, so very deep patterns will need more iterations.

CMTreeMiner, on the other hand, shows a dependency on the average branching factor, but for a given value of b the computation time varies greatly, being especially high for low depth values. Because of the constraints on the random tree generator, a tree that has a low depth with a high average branching factor will necessarily have some nodes with a very large branching factor. We plotted in Fig. 5 a new curve, showing the computation time with respect to the *maximal* branching factor.

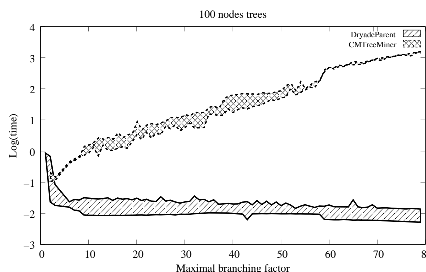


Fig. 5: Random trees with 100 nodes, log(time) w.r.t. maximal branching factor

DryadeParent is nearly unaffected by the maximal branching factor, but the computation time of CMTreeMiner depends strongly on this parameter.

In order to understand how much the behavior of CMTreeMiner and DryadeParent differ, we analyze below the reasons of the dependency to branching factor of CMTreeMiner, and of the variability of its performances in general.

We give a brief reminder of the candidate enumeration technique of CMTreeMiner, called rightmost branch expansion. To generate candidates with k nodes from a frequent tree with $k-1$ nodes, CMTreeMiner tries to add a new edge leading to a frequent node and starting at a node of the rightmost branch of the $k-1$ node tree. All the nodes of the rightmost branch are explored successively in a top-down fashion, from the root to the rightmost leaf.

1. Branching factor leads CMTreeMiner to generate more unclosed candidates by backtracking. For a node with high branching factor, finding correctly the set of its frequent children is a classical frequent itemset mining problem, and the highly combinatorial nature of this problem often leads to the generation of useless candidates. CMTreeMiner is no exception to this rule: its top-down rightmost branch expansion technique finds very quickly all the chidren of a node, but then systematically needs to backtrack to check for frequent subsets of these children. In most cases, this leads to the generation of non-

closed candidates. For example, compare the two patterns of Fig. 6. The linear pattern P_1 is found without generating any unclosed candidates. But the flat pattern P_2 is found after the generation of 3 unclosed candidates, so according to our experiments finding P_2 needs 7% more time than finding P_1 in this simple setting with 4 nodes, and 100% more time in a similar setting with 11 nodes.

Fig. 6: CMTreeMiner candidate enumeration for a linear pattern and for a flat pattern

DryadeParent also has to confront such a combinatorial problem in high branching factor cases, but it does so by using the LCM2 closed frequent itemset mining algorithm, which provides as of now the most efficient way to explore the search space of closed frequent itemsets. Furthemore, by discovering the tiles once and for all at the beginning of the algorithm, DryadeParent avoids to repeat these complex computations if the same tile appears more than once in the patterns.

On this problem, CMTreeMiner could probably be improved by modifying its enumeration technique in order to use LCM2 for sibling enumeration. Such a modified algorithm should be similar to the recent Clott algorithm by Arimura and Uno, which is an extension of the LCM2 principles to the closed attribute tree case.

2. Candidate generation asymmetry The previous problem explains partly why CMTreeMiner is slower than DryadeParent in most cases. As we have seen, this problem can theoretically be overcome. However, another problem remains, that cannot be overcome easily, and this problem is essential to the superior performances of our hooking strategy over any algorithm based on rightmost branch expansion.

Consider the simple pattern of Fig. 7. As it can be seen, during candidate enumeration, unwanted candidates are generated, because the rightmost leaf expansion technique has to test “blindly” all the potential expansions on the

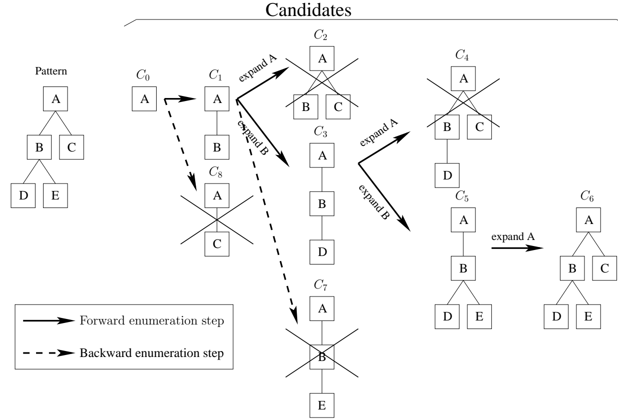


Fig. 7: CMTreeMiner enumeration for a left-balanced pattern

rightmost branch, but can only grow good candidates for certain expansions. For example, the candidate C_2 contains correct information: it corresponds to the first level of the pattern to find. But as some expansions must be made on the node labelled B, which is not on the rightmost branch of C_2, then C_2 is eliminated. In the same way, C_4 is computed for nothing. The children with label C of the root node will have to be recomputed in candidate C_6, even if it could have been discovered much earlier.

This behavior is not only sub-optimal, it also undermines the robustness of CMTreeMiner. Consider the two patterns of figure 8.

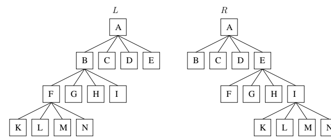


Fig. 8: L: left-balanced pattern, R: right-balanced pattern

Except for the names of labels, both these patterns exhibit the same tree structure, so it is expected that they are discovered in exactly the same amount of time. However, assuming that the sibling processing order is the ascending order of labels (this is the case in the actual implementation of CMTreeMiner), pattern R, which is right-balanced, is an ideal case for enumeration by rightmost tree expansion. CMTreeMiner will check 43 candidates to discover it. Oppositely, the left-balanced pattern G is a worst case, and CMTreeMiner will require to check 79 candidates for its discovery. The computation times reflect this difference in candidates checking: time for finding G is 50 % higher than time for finding R, as shown in Tab. 1.

Table 1: Computation time for finding patterns R and G

Pattern	R	L
CMTreeMiner	0.0010 s	0.0015 s
DryadeParent	0.0013 s	0.0013 s

On the other hand, thanks to its tree-orientation neutral hooking technique, DryadeParent requires exactly the same amount of time for processing these two patterns. For both L and R, DryadeParent will generate 3 candidates: first the initial tile with root A, then a candidate generated by hooking of a tile on respectively B or E, and then the pattern L or R by hooking of another tile on respectively F or I.

Discussion: Our artificial experiments have shown that the structure of the patterns to find, and especially their branching factor, is a crucial performance factor. The closed tree mining algorithm CMTreeMiner, based on candidate enumeration by rightmost branch expansion, has performances which vary considerably with the branching factor of the patterns, and even with their balance. The fact that CMTreeMiner and DryadeParent have similar performances on the NASA dataset, with patterns having quite low branching factor, and that CMTreeMiner is slower than DryadeParent on the CSLOGS dataset, with patterns having a higher branch factor, is consistent with our experiment on artificial data.

Experiments have shown that the new method for finding closed frequent attribute trees of our DryadeParent algorithm is not only computation-time efficient but also robust w.r.t. tree structure, delivering good performances with most tree structure configurations. Such a robustness is a desirable feature for most applications, especially the applications which deal with trees having a great diversity of structure, which cannot predict what will be the typical structure of patterns.

6 Conclusion and perspectives

In this paper, we have presented the DryadeParent algorithm, based on the computation of tiles (closed frequent attribute trees of depth 1) in the data, and on an efficient hooking strategy that reconstructs the patterns from these tiles.

Thorough experiments have shown that in most settings DryadeParent is faster than CMTreeMiner , and that its performances are robust w.r.t. the structure of the patterns to find.

We have proposed new benchmarks taking into account the structure of the patterns to test the behavior of tree mining algorithms. As far as we know, such kind of tests are new in the tree mining community.

Improving these benchmarks and making more detailed analyses are some of our future research directions. We think that our experiments proved that such tools are valuable for the tree mining community. We also plan to extend DryadeParent to structures more general than attribute trees.

Acknowledgements

We wish to thank especially Takeaki Uno for the LCM2 implementation, and Yun Chi for making available the CMTreeMiner implementation and giving us the Nasa dataset. This work is partly supported by the grant-in-aid of scientific research No. 16-04734.

References

1. Termier, A., Rousset, M., Sebag, M.: Dryade : a new approach for discovering closed frequent trees in heterogeneous tree databases. In: International Conference on Data Mining ICDM'04, Brighton, England. (2004) 543–546
2. Arimura, H., Uno, T.: An output-polynomial time algorithm for mining frequent closed attribute trees. In: 15th International Conference on Inductive Logic Programming (ILP'05). (2005)
3. Chi, Y., Yang, Y., Xia, Y., Muntz, R.R.: Cmtreeminer: Mining both closed and maximal frequent subtrees. In: The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04). (2004)
4. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient substructure discovery from large semi-structured data. In: In Proc. of the Second SIAM International Conference on Data Mining (SDM2002), Arlington, VA. (2002) 158–174
5. Zaki, M.J.: Efficiently mining frequent trees in a forest. In: In Proc. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. (2002)
6. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of the 20th VLDB Conference, Santiago, Chile (1994)
7. Asai, T., Arimura, H., Uno, T., ichi Nakano, S.: Discovering frequent substructures in large unordered trees. In: the Proc. of the 6th International Conference on Discovery Science (DS'03). (2003) 47–61
8. Nijssen, S., Kok, J.N.: Efficient discovery of frequent unordered trees. In: First International Workshop on Mining Graphs, Trees and Sequences, 2003. (2003)
9. Zaki, M.J.: Efficiently mining frequent embedded unordered trees. Fundamenta Informaticae, special issue on Advances in Mining Graphs, Trees and Sequences **65** (2005) 33–52
10. Uno, T., Kiyomi, M., Arimura, H.: Lcm v.2: Efficient mining algorithms for frequent/closed/maximal itemsets. In: 2nd Workshop on Frequent Itemset Mining Implementations (FIMI'04). (2004)
11. Termier, A.: Extraction of frequent trees in an heterogeneous corpus of semi-structured data: application to xml documents mining. Technical Report 1388, LRI (2004) http://www.lri.fr/∼termier/publis/phdTermierEN.ps.gz.
12. Chalmers, R., Almeroth, K.: Modeling the branching characteristics and efficiency gains of global multicast trees. In: Proceedings of the IEEE INFOCOM'2001. (2001)

Optimizing gSpan for Molecular Datasets

Katharina Jahn[1,2] and Stefan Kramer[1]

[1] Technische Universität München, Institut für Informatik
Boltzmannstr. 3, 85748 Garching bei München, Germany
kramer@in.tum.de
[2] Ludwig-Maximilians-Universität München, Institut für Informatik,
Oettingenstr. 67, 80538 München, Germany
jahnka@cip.ifi.lmu.de

Abstract. We propose two optimizations for mining molecular databases with gSpan, one of the state-of-the-art graph mining algorithms. Both optimizations apply to the enumeration of subgraph occurrences in a graph database, which is, also according to our profiling, the most expensive operation of gSpan. The first optimization reduces the number of subgraph isomorphisms that need to be accessed for proper support computation in considering the symmetries inherent in many chemical molecules, and the second speeds up subgraph isomorphism tests by making use of the non-uniform frequency distribution of atom and bond types. The optimizations are part of a reimplementation of the original gSpan algorithm and are shown to significantly increase the performance on two chemical datasets.

1 Introduction

In graph mining, two lines of research have been followed in the past years. The first line focused on less expressive pattern languages such as paths, but on queries more complex than just those for patterns of minimum support [3, 5]. The second line focused on more expressive pattern languages, mostly general (connected) subgraphs, and gave rise to algorithms for finding frequent subgraph patterns in graph databases [4, 6, 11, 12, 2]. The first of the latter algorithms, AGM [4], was a graph-variant of the levelwise search algorithm APriori for mining frequent itemsets [1]. Further speed-up was achieved using a depth-first traversal of the search space and dropping the costly candidate generation of AGM in favor of a straightforward growth from a single subgraph, combined for the first time in the gSpan algorithm [11]. Subsequently, optimizations were proposed based on ideas for closed patterns [12] and keeping lists of embeddings [2, 7]. However, a recent study [10] showed that gSpan is still competitive, at least for smaller subgraphs and larger datasets.

In this paper, we present two optimizations particularly tailored for databases of molecular graphs, that can improve the performance of gSpan significantly. As will be shown below, the ideas are not restricted to gSpan and could be transferred to other algorithms. In gSpan, the search strategy demands the retrieval of

all occurrences of a frequent subgraph, which is the major source of complexity in the algorithm. We show that under certain conditions it is sufficient to access only some of the occurrences and present an efficient strategy for the search of subgraph occurrences in a graph. The first method relies on symmetries of graphs and the second on the non-uniform frequency distribution of label types, which are both typical features of molecular datasets.

2 Preliminaries

In this section, we briefly review some basic concepts of graph theory and introduce the notation used in the remainder of the paper.

A graph is a pair $G = (V, E)$ consisting of a set of vertices V and a set of edges $E \subseteq V \times V$, such that every edge $e \in E$ relates to a pair of vertices (v_1, v_2). If there is an edge $e = (v_1, v_2)$, the vertices v_1, v_2 are called adjacent, and e is said to be incident to v_1 and v_2. We consider only undirected graphs, which means that the pair (v_1, v_2) is unordered. A *labeled graph* is represented as a tuple $G = (V, E, L, l)$ consisting of a set of vertices V, a set of edges $E \subseteq V \times V$, a set of labels L and a function $l : V \cup E \rightarrow L$ that assigns a label to every vertex and every edge. A graph is *connected* if $\forall u, v \in V$ there is a sequence of edges $(v_0, v_1), (v_1, v_2), ..., (v_{n-1}, v_n)$ such that $(v_i, v_{i+1}) \in E$ for $i \in 0, ..., n-1$ and $v_0 = u$ and $v_n = v$. An *articulation vertex* is a vertex $v \in V$ such that the removal of v and its incident edges turns a connected graph G into an unconnected graph. If $V \subseteq V'$ and $E \subseteq E' \cap V \times V$, then $G = (V, E)$ is a *subgraph* of $G' = (V', E')$, denoted by $G \subseteq G'$. A *graph isomorphism* between two labeled graphs $G = (V, E, L, l)$ and $G' = (V', E', L, l')$ is a bijective function $f : V \rightarrow V'$, such that (i) $\forall v \in V$, $l(v) = l'(f(v))$ and (ii) $\forall (u, v) \in E$, $(f(u), f(v)) \in E'$ and $l(u, v) = l'(f(u), f(v))$. A graph isomorphism that maps a graph to itself is called an *automorphism*. If the mapping from G to G' is injective, but not surjective, we call it a *subgraph isomorphism*. Then there is a subgraph $H \subseteq G'$ such that f defines a graph isomorphism between G and H. Such a mapping is called a *subgraph occurrence* of G in G' or a *subgraph matching*.

We can extend a graph $G = (V, E, L, l)$ by adding a new edge $e = (u, v)$. The resulting graph is denoted by $G \diamond e$, which is similar to the notation used by Yan and Han [12]. Edge e can either (1) link two vertices of G if $u, v \in V$, or (2) introduce a new vertex into G, if either u or $v \notin V$. Without loss of generality, we assume $v \notin V$. When f is a subgraph matching of G in $G' = (V', E', L, l')$, we denote by $\Diamond(G|f)$ the set of all extensions $G \diamond e$ for which there is an edge $e' \in E'$ compatible with e such that f can be extended to map $G \diamond e$ to G'. This means that for $e' = (u', v')$, $l(e) = l'(e')$ and either (1) $u' = f(u)$ and $v' = f(v)$, or (2) $u' = f(u)$, $v' \notin f(V)$ and $l'(v') = l(v)$.

3 Brief Review of gSpan

In this section we summarize the basic features of gSpan, focusing on those being essential for our optimizations. In gSpan, a graph is represented as a linear

sequence of its edges, called a DFS code, with each edge being represented as 5-tuple $(i, j, l_i, l_{i,j}, l_j)$ with i and j being the identifier numbers of its incident nodes (assigned by their order of appearance in the sequence), l_i and l_j being their labels and $l_{i,j}$ being the label of the edge. Not every possible edge sequence forms a valid DFS code, but only those representing a depth-first traversal of a graph. The DFS code representation of a graph is not unique, but a linear order on DFS codes is defined by the insertion position of the edges into the sequence and by the lexicographic order of the label types [11]. The smallest DFS code representation of a graph is defined as its minimum DFS code, which is unique and used as canonical form for the mining process. The search space is then defined as a DFS code tree consisting of nodes that represent the DFS codes and edges that indicate that the child node grows from the parent node by adding one new edge at the end of its DFS code. Yan and Han [11] also showed that each minimum DFS code is the child of another minimum DFS code so that the search space for frequent subgraph mining can be pruned wherever non-minimum DFS codes occur.

The mining process of gSpan works as follows: After initial identification of frequent and minimum one edge subgraphs the search space is traversed in a depth-first manner, and for each frequent and minimum DFS code encountered during the traversal all its occurrences in the database are accessed for support computation of its candidate extensions. Whenever an extension turns out to be infrequent or not minimal, it need not be accessed for further extension, but can be pruned from the search space.

4 Optimizations

Subgraph isomorphism testing, in general NP-complete, is the most expensive operation of the gSpan algorithm consuming about 70% of the re-implementation's computation time according to our profiling. In this section we present two methods to speed up this process when mining in molecular databases.

4.1 Symmetries in Graphs

The purpose of subgraph isomorphism testing in gSpan is in support computation of a subgraph's candidate extensions. It has been assumed that this requires the enumeration of all mappings of the subgraph into a graph unless all possible extensions are previously discovered. However, not every mapping will necessarily lead to the discovery of a new extension, and an extension redundantly found in a graph is useless for support computation so that (at least theoretically) it is sufficient to access only a subset of the mappings for proper support computation. Whereas in the general case the "prediction" of redundant mappings is extremely difficult, we will show that for graphs with inherent symmetries part of the redundancy can be determined by computing their automorphism groups. Remember that an automorphism of a graph G is a graph isomorphism with itself, i.e., a mapping from the vertices of the given graph G back to vertices of G

such that the resulting graph is isomorphic with G. The sets of automorphisms define a permutation group. The automorphism groups of a graph characterize its symmetries, and are therefore very useful in determining certain of its properties [9]. In the following theorem the basic idea behind the proposed optimization is given for unlabeled graphs. The extension to labeled graphs is straightforward.

Theorem 1. *Let $G = (V, E)$ be a graph with vertices $v_1, v_2 \in V$ and a graph automorphism $f : V \to V$ with $f(v_1) = v_2$. Then for every graph $H = (V', E')$ with a subgraph isomorphism $g_1 : V' \to V$, such that $\forall (u', v') \in E'$, $(g_1(u'), g_1(v')) \in E$ and $v_1 \in g_1(V')$ there is a subgraph isomorphism $g_2 : V' \to V$ such that $\forall v' \in V'$, $g_2(v') = f(g_1(v'))$ and $v_2 \in g_2(V')$.*

Proof. Define $g_2 : V' \to V$ such that $g_2(v') = f(g_1(v'))$ for all $v' \in V'$. With g_1 being a monomorphism and f being an isomorphism, g_2 is a monomorphism. Further we have $\forall (u', v') \in E'$, $(g_2(u'), g_2(v')) \in E$ because $(g_2(u'), g_2(v')) = (f(g_1(u')), f(g_1(v')))$ and $(f(g_1(u')), f(g_1(v'))) \in E$ follows from $(g_1(u'), g_1(v')) \in E$ which follows from $(u', v') \in E'$. Finally $v_2 \in g_2(v')$ since $v_1 \in g_1(V')$ and $v_2 = f(v_1)$. $\square$

For the enumeration of the subgraph occurrences of H in G this means that the position of a second matching follows immediately from the first one (if $v_1 \neq v_2$). Moreover, both matchings g_1 and g_2 will yield the same extensions as the following lemma states.

Lemma 1. $\diamondsuit(H|g_1) = \diamondsuit(H|g_2)$.

Proof. We show $\forall H \diamond e \in \diamondsuit(H|g_1)$ it holds that $H \diamond e \in \diamondsuit(H|g_2)$: Let $H \diamond e \in \diamondsuit(H|g_1)$, then g_1' can be defined by extending g_1 such that it is a subgraph isomorphism from $H \diamond e$ to G with $g_1(v) = g_1'(v)$ for all $v \in V'$ and $(g_1(u), g_1(v)) = (g_1'(u), g_1'(v))$ for all $(u, v) \in E'$. According to theorem 1 we can derive from g_1' and f a subgraph isomorphism g_2' such that $g_2(v) = g_2'(v)$ for all $v \in V'$, and $(g_2(u), g_2(v)) = (g_2'(u), g_2'(v))$ for all $(u, v) \in E'$, thus $g_2' \in \diamondsuit(H|g_2)$. The other direction follows analogously when interchanging g_1 and g_2 and replacing f by its inverse. $\square$

We call matchings like g_1 and g_2 in theorem 1 equivalent. Evidently there can be more than one subgraph isomorphism from H to G that maps the vertex v' to v_1. However all of them have an equivalent matching that maps v' to v_2. Thus no matching mapping v' to v_1 yields an extension that is not also found in a matching that maps v' to v_2. Consequently, if both vertices v_1 and v_2 can be matched to the first vertex of a subgraph's DFS code we will not loose any candidate extensions if we skip the entire matching process starting at v_1 since all successful matchings starting at v_1 have an equivalent matching starting at v_2. Figure 1 shows a symmetric molecule and two ways of fitting in a fragment. Both matchings are equivalent, thus one of them will be completely skipped from subgraph isomorphism testing. Incidentally, it can be shown that there are only three different vertices at all to be considered as start vertices for subgraph isomorphism tests in this molecule.

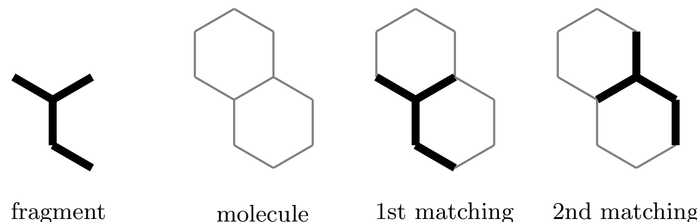


Fig. 1. Equivalent matchings with different start vertices.

Later in a matching having already matched the first $k-1$ tuples of subgraph H's DFS code to a graph G we might encounter a vertex v in G with two incident edges $e_1 = (v, w_1)$ and $e_2 = (v, w_2)$, so that there exists an automorphism f with $f(w_1) = w_2$, $f(w_2) = w_1$ and $f(v) = v$ and both edges can be matched to tuple k. If then additionally the nodes of G, to which those nodes of H occurring in the first $k-1$ tuples were matched, are mapped to themselves by f, every subgraph isomorphism that maps e_1 to position k of the DFS code has an equivalent matching that maps the first $k-1$ tuples to the same edges and the k-th tuple to e_2. This follows immediately from the previous theorem. The only difference is that we now impose an additional constraint on the mapping f, which is necessary to keep the discovery of equivalent matchings simple. After all we can skip the mapping of e_1 to DFS code position k as all matchings that could emanate from it lead to exactly the same extensions as their equivalent matchings that map e_2 to the position k and have the same mappings for the first $k-1$ tuples.

Whereas for the first pruning strategy we only need to know that there generally is a mapping from a vertex v_1 to v_2 , for this one we need information on the actual mapping. Thus, it is not sufficient to compute and store for which edge pairs there is a suitable mapping, but we would need the complete automorphism groups of the graphs in the database, which is too expensive in terms of computation time and memory consumption. Thus, we restrict this kind of pruning to two special cases: In the first $k = 1$, so that there are no edges priorly matched to the DFS code and in the second the vertex v shared by e_1 and e_2 is an articulation vertex that separates w_1 and w_2 from the vertices that have been previously matched to the DFS code. For the first case, it is apparently sufficient to know that there is an arbitrary matching in which e_1 and e_2 can be mapped to each other to fulfil the additional constraint. For the second, we need the result of the following theorem to see what information is required:

Theorem 2. *Let $G = (V, E, L, l)$ be a labeled graph with an articulation vertex $v \in V$, $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = v$ such that $\forall(u, w) \in E$ either $u, w \in V_1$ or $u, w \in V_2$. If there are two graph automorphisms $f, g : V \to V$ with $f(V_1) = V_1$, $f(V_2) = V_2$, $g(V_1) = V_1$, $g(V_2) = V_2$ and $f(v) = v$ and $g(v) = v$, then $h : V \to V$ with $h(v') = f(v')$ if $v' \in V_1$ and $h(v') = g(v')$ if $v' \in V_2$ for all $v' \in V$ is also a graph automorphism.*

Proof. The function $h : V \to V$ is bijective due to f and g being bijective and $f(v) = g(v)$. It remains to be shown that for all $u, w \in V$: $(u, w) \in E$ iff

$(h(u), h(w)) \in E$. Without loss of generality assume $u, w \in V_1$ then $h(u) = f(u)$ and $h(w) = f(w)$. With f being a graph automorphism it follows from $(u, w) \in E$ that $(h(u), h(w)) = (f(u), f(w)) \in E$ and vice versa. □

If we can show that every automorphism f, that maps e_1 and e_2 onto each other, divides V into V_1 and V_2 as required in theorem 2 we can infer the existence of a matching that satisfies the necessary constraints in combining an arbitrary matching f, that maps e_1 and e_2 onto each other, with the identity function. But this is easy to see: Set V_1 to contain all vertices that can be reached from w_1 or w_2 without passing the articulation vertex v and V_2 to the remaining vertices. If then a vertex w' that has such a path to w_1 was mapped to a vertex in V_2, i.e. $f(w') \in V_2$, one of the vertices on the path from w' to w_1 must be mapped to v since $f(w_1) \in V_1$ and v is the only connection between V_1 and V_2 but this can not be true, since v is already mapped to itself by f. (The same is true if w_1 is replaced by w_2.) Thus for the second case it is also sufficient to know that there exists a matching that maps e_1 and e_2 to each other.

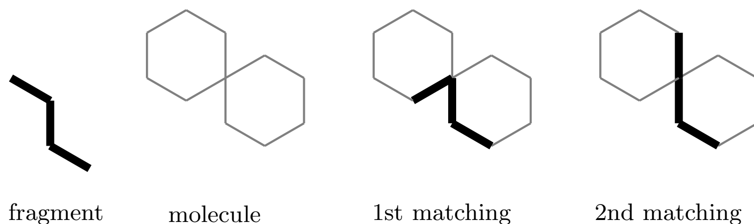


Fig. 2. Equivalent matchings with identical start vertices.

In Figure 2, an example is given for the second case. The two matchings share the first two edges and differ in the third. However the edges matched at the third position can be mapped onto each other in an automorphism and are separated from the previous edges by an articulation vertex. Thus, there must also be an automorphism mapping the complete matchings onto each other so that according to theorem 1 both matchings need to be equivalent.

The computation and marking of articulation vertices can be done in linear time. For the computation of the automorphisms we used a similar backtracking algorithm as for subgraph isomorphism testing (see below). We did not compute the complete automorphism group which can contain billions of mappings for highly symmetric graphs but checked for each vertex pair and each pair of edges that share an articulation vertex if there is an automorphism that maps them onto each other and marked them accordingly. This is still a cost-intensive preprocessing step, but the idea behind is that it will pay off later when many costly subgraph isomorphism tests benefit from it.

4.2 Sorting Labels in Ascending Frequency

For matching a minimum DFS code against a graph a backtracking algorithm is applied that repeatedly adds new edges to the matching until the whole DFS code

is represented and backtracks if an edge fails to be matched. Being exponential in worst case, the performance of this algorithm extremely benefits from failed matches as they allow for pruning a branch from the search space with the performance gain being the larger, the earlier a failed matching takes place. Consequently the identification of edge types that are more likely to fail and matching them as early as possible would speed up the mining process. However, the order in which the edges of a subgraph are matched is not arbitrary, but equals their sequence in the minimum DFS code that in turn is fixed by the DFS lexicographic order introduced in Section 3.

Generally the first tuple $(0, 1, l_i, l_{i,j}, l_j)$ of a subgraph's minimum DFS code will always encode the lexicographic smallest edge the subgraph contains, that means l_i will be the smallest vertex label in the subgraph, $l_{i,j}$ will be the smallest label of an edge incident with a vertex labeled l_i and l_j will be the label of the smallest node with an incident edge labeled $l_{i,j}$ that is also incident with a vertex of type l_i, otherwise the DFS code will not be minimal. For all other tuples the topology needs to be considered (i.e., the position at which the new edge is attached to the graph defined by the previous tuples and the type of the new edge), but whenever there are several topological identical edges to choose from, the lexicographic smallest edge will be the next tuple in the minimum DFS code to guarantee its minimal form. Thus, lexicographically small edges will tend to occur earlier in minimum DFS codes than those with large labels, but for edges others than the first one this is not a strict rule due to topology restrictions.

In relabeling the vertex and edge types in ascending frequency (in contrast to descending frequency in the original gSpan algorithm), our optimization makes use of this observation. The rare edges then carry small labels and thus are more likely to occur early in the minimum DFS code leading to an early pruning of the search space in the many cases they fail to be matched.

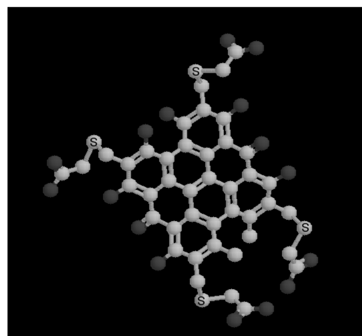


Fig. 3. An example molecule

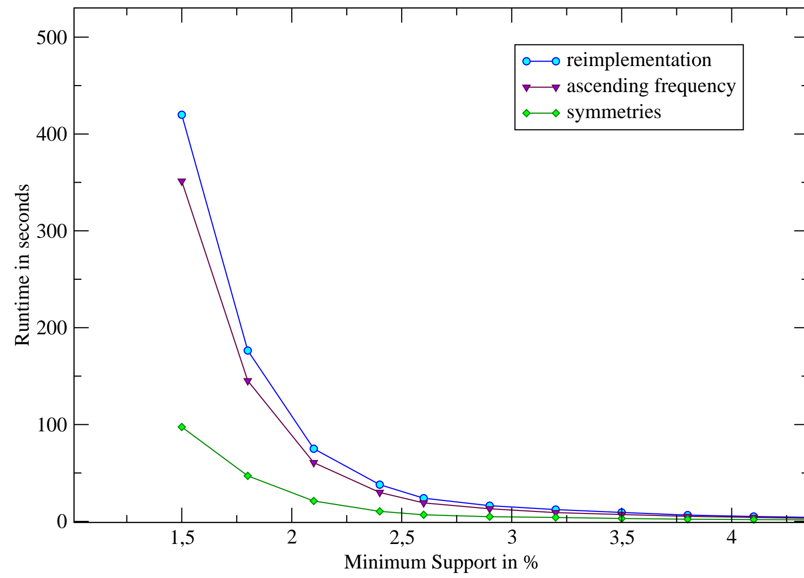
As an example imagine the fragment `C-C=C-C-S` is matched against the molecule displayed in Figure 3. Using the descending frequency order, the edges

of the fragments will be matched from left to right into the graphs, whereas in ascending frequency the matching order would be reverse, starting with the `S-C` edge. With only four sulfur atoms occurring in the molecule, but plenty of carbons, the search space for the latter ordering is much smaller: There are only four possibilities to match the first atom and eight possibilities to match the first edge, whereas for the descending ordering there are 42 ways to match the first `C` atom and 62 ways to match the first `C-C` edge, most of which will not contribute to the retrieval of a complete matching of the fragment. Thus, a lot of time is wasted on matching the `C-C=C-C` part of the fragment, before realizing that the crucial `C-S` edge is not found at the required position at the end of the matching.

5 Experiments

We evaluated the performance of our optimizations on two molecular datasets. All tests were performed on a 1.4 GHz Intel Pentium M machine with 512 MB main memory running the S.u.S.E Linux 9.0 operating system. The first test set contains the 340 molecules from the Predictive Toxicology Evaluation Challenge [8], which have been used previously for the performance evaluation of graph mining algorithms [11, 6]. A detailed description of the conversion process of the molecules into graphs was described by Kuramochi and Karypis [6]. In Figure 4, the runtime of our reimplementation and its optimizations on the data set is shown for different minimum support thresholds. (The time for the preprocessing step to compute symmetries is included in the runtime of the first optimization.) For both methods a significant performance gain was observed with the method considering the symmetries of the graphs reducing the runtime by up to 77%, and the method using the different labeling strategy by up to 20%. The combination of both methods did not improve the performance any further and is not displayed in the figure. The second dataset obtained from the website of the National Cancer Institute [13] contains 42,687 molecules screened for anti-HIV activity (October 99 release). Apart from its size, it differs from the first data set in three aspects: First, it does not contain any information on aromatic bonds, but labels the edges of an aromatic ring alternatingly with a single and a double bond label. Second, during the conversion process into the gSpan input format, we did not distinguish further between atoms of the same element type, and third, we completely removed hydrogen atoms and their incident edges to accomplish the mining process in reasonable time.

As can be seen in Figure 5, the use of symmetries does not enhance performance on this dataset, but at least the time spent on the preprocessing is compensated. We explain this result by the absence of hydrogens and aromatic bonds (i.e., aromatic rings cannot be detected directly), which are both, according to our observations, the major source of symmetries in small molecules. To test this hypothesis, we determined the number and sizes of automorphism groups for both datasets (see Table 1). We considered only graphs that consist of a single component, which explains the smaller values of the total graph counts. Besides, we have no values for one molecule in the NCI dataset and for three



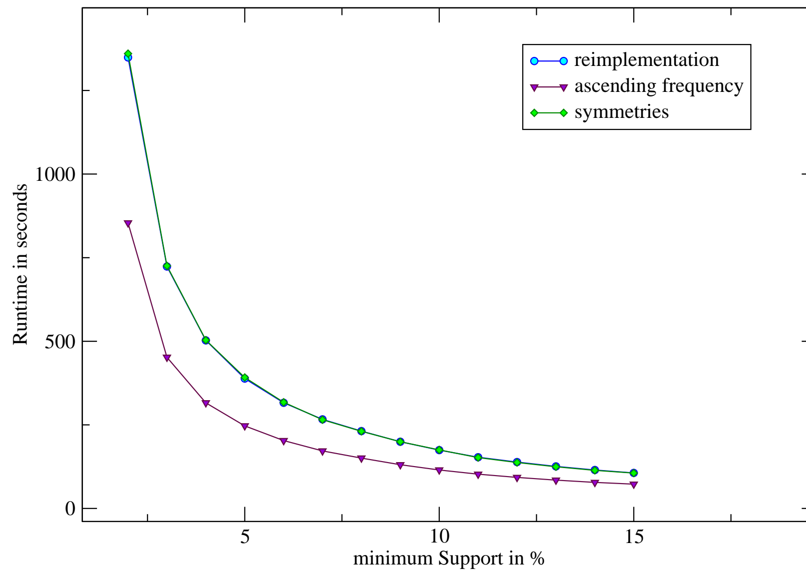


Fig. 5. Runtime vs. minimum support for the NCI HIV dataset. The points for the gSpan reimplementation with and without the first optimization (symmetries) coincide.

molecules in the PTE dataset, for which the computation of the automorphism group was timed out as it did not finish within reasonable time. In summary, these figures clearly indicate that the first optimization should in fact be much more effective on the PTE than on the NCI data.

Table 1. The frequency of automorphism group sizes for the molecules in the two test datasets. If no symmetries occur in a graph the automorphism group size is one.

#Autom.		total	1	2	3	4	6	8	10	12	16	24	32	36
NCI HIV	abs.	41685	21830	11423	25	2948	1665	1264	3	602	333	142	112	180
	%	100.00	52.37	27.42	0.06	7.07	4.00	3.03	0.01	1.44	0.80	0.34	0.27	0.43
PTE-2	abs.	325	10	41	0	47	3	33	0	25	11	21	7	1
	%	100	3.08	12.62	0.0	14.46	0.92	10.15	0.0	7.69	3.38	6.46	2.15	0.31

#Autom.		48	64	72	96	120	128	144	192	216	240	256	288	>288
NCI HIV	abs.	134	77	351	20	17	46	149	13	12	3	16	46	274
	%	0.32	0.18	0.84	0.05	0.04	0.11	0.36	0.03	0.03	0.01	0.04	0.11	0.66
PTE-2	abs.	12	7	12	2	0	1	9	3	0	0	1	11	68
	%	3.69	2.15	3.69	0.61	0.0	0.31	2.77	0.92	0.0	0.0	0.31	3.38	20.92

The performance gain of our labeling strategy is more pronounced on the NCI dataset, reaching values up to 38%. We explain this by the even stronger imbal … ıt
type … ın
subgr … ıe
time … ıe
order

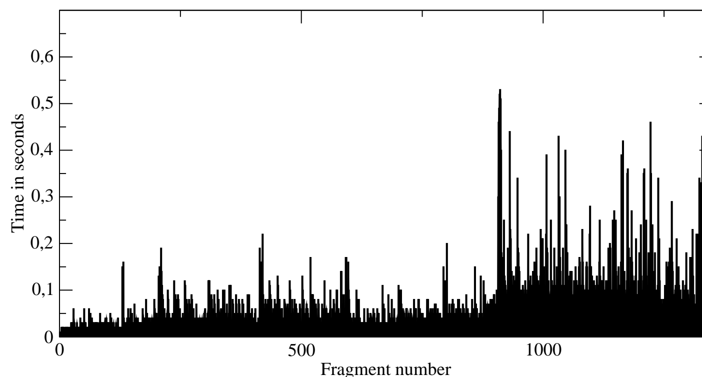


Fig. 6. Time spent on each of the 1350 frequent fragments found during a run of the program on the NCI dataset with minimum support of 4000 and label ordering in ascending frequency

The severe increase of time spent per fragment coincides with the entering of the search branch that contains only C=C and C-C edges, which are by far the most frequent edges in the database. Compared to this, the time spent on subgraphs found in earlier search branches, i.e., those containing also less frequent edge types, is almost negligible due to our relabeling strategy.

6 Discussion and Related Work

We presented two methods for reducing the effort for subgraph isomorphism testing in molecular datasets in the context of the graph mining algorithm gSpan, which is currently among the fastest methods for frequent subgraph mining. The speed-up was achieved by the consideration of two inherent properties of chemical molecules which are the non-uniform frequency distribution of atom and edge types and the symmetries occurring in many molecules. It remains to be seen whether it is possible to take advantage of other properties of chemical compounds to further decrease the expense of subgraph isomorphism testing, and whether other types of graph data share some of the characteristics so that similar optimizations work for them as well.

Several graph mining algorithms developed after gSpan make use of embedding lists to avoid costly subgraph isomorphism tests at the expense of excessive memory consumption. However, in a recent study [10] it was observed that gSpan is still competitive with these approaches unless the subgraphs become very large. For large databases it was even shown to outperform Gaston [7], the fastest among these algorithms.

Our optimizations for molecular graphs cannot be compared directly with those presented recently by Borgelt *et al.* [2]. The optimizations presented here prune the search space for the embedding of a subgraph into a graph, whereas in MoFa the search space of frequent subgraphs in pruned. Since MoFa keeps tracks of lists of embeddings, the presented optimizations would not be necessary there. However, the idea of using symmetries could be transferred to reduce the size of embedding lists, because it would be sufficient to store one representative of a set of equivalent matchings. Vice versa, MoFa's "equivalent sibling pruning" is not necessary in gSpan-type approaches, because the canonical form of DFS codes avoids the generation of syntactic variants in the first place. Since we are not searching for closed frequent graph patterns [11], the second optimization of MoFa is not applicable either. It is an open question whether the present optimizations can be combined with ideas for mining closed frequent graph patterns only. Clearly, the second optimization is compatible with CloseGraph, since only the label ordering is changed. As for the first optimization, we conjecture that it is compatible as well, but it appears much less obvious. Finally, we also find a commonality with MoFa: The idea of starting with infrequent elements has been shown to be useful in this (quite different) approach as well.

Acknowledgements

We would like to thank Ulrich Rückert and the anonymous reviewers for their helpful comments.

References

1. R. Agrawal, R. Srikant: Fast algorithms for mining association rules. Proc. 20th Int. Conf. on Very Large Data Bases (VLDB 1994, Santiago de Chile, Chile), 487-499. Morgan Kaufmann, San Francisco, CA, USA 1994.
2. C. Borgelt, T. Meinl, M. Berthold: Advanced pruning strategies to speed up closed molecular fragments. Proc. IEEE Conf. on Systems, Man and Cybernetics (SMC 2004, The Hague, Netherlands). IEEE Press, Piscataway, NJ, USA 2004.
3. L. De Raedt, S. Kramer: The levelwise version space algorithm and its application to molecular fragment finding. Proc. 17th Int. Joint Conf. on Art. Intell. (IJCAI 2001, Seattle, USA), 853-862. Morgan Kaufmann, San Francisco, CA, USA 2001.
4. A. Inokuchi, T. Washio, H. Motoda: An apriori-based algorithm for mining frequent substructures from graph data. Proc. of the 4th European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD 2000, Lyon, France), 13-23. Springer, Berlin, Germany 2000.
5. S. Kramer, L. De Raedt, C. Helma: Molecular feature mining in HIV data. Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2001, San Francisco, USA), 136-143. ACM Press, New York, NY, USA 2001.
6. M. Kuramochi, G. Karypis: Frequent subgraph discovery. Proc. 1st IEEE Int. Conf. on Data Mining (ICDM 2001, San Jose, USA), 313-320. IEEE Press, Piscataway, NJ, USA 2001.
7. S. Nijssen, J. N. Kok: A quickstart in frequent substructure mining can make a difference. Technical report, Leiden Institute of Advanced Computer Science, Leiden University (2004).
8. A. Srinivasan, R.D. King, D.W. Bristol: An assessment of submissions made to the Predictive Toxicology Evaluation Challenge. Proc. 16th Int. Joint Conf. on Artificial Intelligence (IJCAI 1999, Stockholm, Sweden), 270-275. Morgan Kaufmann, San Francisco, CA, USA 1999.
9. E.W. Weisstein: Graph automorphism. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GraphAutomorphism.html
10. M. Wörlein, Th. Meinl, I. Fischer, M. Philippsen: A quantitative comparision of the subgraph miners MoFa, gSpan, FFSM and Gaston. Proc. 9th Eur. Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005, Porto, Portugal), Springer, Berlin, Germany 2005.
11. X. Yan, J. Han: gSpan: Graph-based substructure pattern mining. Proc. 2nd IEEE Int. Conf. on Data Mining (ICDM 2002, Maebashi, Japan), 721-724. IEEE Press, Piscataway, NJ, USA 2002.
12. X. Yan, J. Han: Closegraph: Mining closed frequent graph patterns. Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2003, Washington DC, USA), 286-295. ACM Press, New York, NY, USA 2003.
13. http://dtp.nci.gov/docs/aids/aids_data.html/.

Frequent Tree Mining with Selection Constraints

Jeroen De Knijf

Utrecht University, Institute of Information and Computing Sciences
PO Box 80.089, 3508 TB Utrecht
The Netherlands
jknijf@cs.uu.nl

Abstract. Data that can conceptually be viewed as tree structures occurs extensively in domains such as bio-informatics, web logs, XML databases and computer networks. One important problem in mining tree structured data is to find all frequent subtrees. Due to combinatorial explosion, the number of frequent subtrees grows exponentially with the size of the trees. This causes severe problems with the completion time of the mining algorithm and the huge amount of potentially uninteresting patterns. In related areas such as frequent itemset mining and mining sequential patterns, the use of constraints has resulted in a considerable speedup of the mining application. Furthermore, the discovered patterns are focused on the users interest, which reduces the effort for the user to investigate the output of the mining process. In this paper we define selection constraints, both on the structure and on the labels of trees. We show how to efficiently compute all ordered and unordered induced closed subtrees that satisfy the selection constraints. We illustrate the use of these constraints within the application of web log mining. Finally, the effect of applying selection constraints on a synthetic data set and a real data set is evaluated. Our approach leads to a considerable speedup of the mining algorithm.

1 Introduction

Frequent treemining has become an important and popular problem in the field of knowledge discovery and data mining. The main reasons for the increase in interest is the growing amount of semi-structured data (e.g. XML databases) and the urge to exploit and mine these databases. Furthermore, the availability of treemining algorithms such as [2, 13, 16] to exploit these databases, without losing information on the structure of the data, has increased the interest of the research community. Briefly, given a set of tree data, the problem is to find all subtrees that satisfy the minimum support constraint, that is all subtrees must occur in at least $n\%$ of the data records. Applications of frequent treemining include the following:

- Web log mining: frequent access trees from a database of web logs, where each record corresponds to the entire forward access of a user, are explored in [16].

- Classification and clustering: the work presented in [17] uses a frequent treemining algorithm to extract frequent substructures from XML data, and subsequently classifies the data according to its structure.
- Database indexing: in [15] a frequent treemining algorithm is used to extract frequent tree query patterns from a large collection of XML queries. The answers to the frequent tree query patterns are then stored and indexed for faster retrieval.

In this work we introduce selection constraints on the labels and the structure of trees. We present an efficient mining algorithm (SCTreeminer) that combines closed frequent treemining with the exploitation of user-defined selection constraints to prune the search space. Experimental evaluation of SCTreeminer is performed on both a synthetic and a real data set, and we compared the run time for the constraint based algorithm with a post processing approach.

This paper is organized as follows: in the next section we describe related work and the motivation for a constraint based treemining approach. In section 3 we discuss the basic concepts required for frequent treemining. Section 4 describes selection constraints in detail, their properties, and problems related to mining with these constraints. In this section we also describe the SCTreeminer algorithm. In section 5 we experimentally evaluate our algorithm and compare the performance with a post pruning approach. In the last section we draw conclusions and indicate further research directions.

2 Motivation and Related Work

The design of effective algorithms for mining frequent subtrees has been the subject of several studies in recent years, see for example [2, 4, 5, 9, 12, 13, 16]. These algorithms differ in the type of trees handled (free trees, rooted unordered trees, rooted ordered trees) and the kind of tree matching relation used (induced, embedded, incorporated). A drawback of these approaches is the lack of user controlled focus in the pattern mining process, the only control mechanism the user has being the minimum support threshold. This results in:

1. High computational cost. Given a database of trees, the computation cost for treemining algorithms is fixed for a given minimum support threshold. For users that have a clear idea of which kind of patterns are interesting, the mining algorithm spends computation time on patterns that are of no interest to the user.
2. Huge number of potentially useless results. Since the number of frequent subtrees grows exponentially with the size of the tree, the user has to put a lot of effort into finding interesting results.

There are two basic approaches to address these problems: condensed representations and user-defined constraints. Chi et al. [6] introduce an algorithm to find all closed and maximal frequent subtrees. This approach results in a considerable speedup and smaller output size. For example, on the synthetic data

set described in section 5 which consists of 10,000 records the number of frequent (induced) subtrees is more then 118,000,000, while the number of closed subtrees is slightly above 164,000, using a minimum support of 1%. The computation time is reduced by a similar factor as the output size.

Another approach to tackle these problems is to provide users with a constraint specification language, in which they can express a "family" of patterns in which they are interested [7,8]. If these constraints can be pushed deep into the mining process—as opposed to post-pruning—this results in a considerable speedup of the mining algorithm in addition to a reduction of the output.

In this paper we combine both approaches. Mining closed trees does not give the user more control of the mining process, but it compresses the output such that all frequent subtrees are derivable from it. On the other hand the output of constraint query patterns can be further compressed by presenting only the closed patterns that satisfy the constraints without any loss of information.

3 Preliminaries

In this section we provide the basic concepts and notation that is used in the remainder of this paper. A labeled rooted tree $T = \{V, E, \Sigma, L, v_0\}$ consists of a vertex set V, an edge set E, an alphabet Σ for vertex labels and a labeling function $L : V \to \Sigma$ that assigns labels to vertices. The special node v_0 is called the root. If $(u, v) \in E$ then u is the parent of v and v is the child of u. For a node v, any node u on the path from the root node to v is called an ancestor of v. If u is an ancestor of v then v is called a descendant of u. If in addition the tree is also ordered there is a binary relation '$\leq$' $\subset V^2$ that represents an ordering among siblings. The size of a tree is defined as the number of vertices; we refer to a tree of size k as a k-tree.

Given two labeled rooted trees T_1 and T_2 we call T_2 an *induced* subtree of T_1 and T_1 an *induced* supertree of T_2, denoted by $T_2 \preceq T_1$, if there exists an injective matching function Φ of V_{T_2} into V_{T_1} satisfying the following conditions for any $v, v_1, v_2 \in V_{T_2}$:

- Φ preserves the parent relation: $(v_1, v_2) \in E_{T_2}$ iff $(\Phi(v_1), \Phi(v_2)) \in E_{T_1}$.
- Φ preserves the labels: $L_{T_2}(v) = L_{T_1}(\Phi(v))$.

If in addition T_1 and T_2 are ordered, the mapping Φ should also preserve the order among the siblings; that is, if $v_1 \leq_{T_2} v_2$ then $\Phi(v_1) \leq_{T_1} \Phi(v_2)$.

In this work we refer to induced subtrees simply as subtrees. Let D denote a database where each transaction $d \in D$ is a labeled rooted tree. For a given pattern tree t, which is also a labeled rooted tree, we say t occurs in a transaction d if t is a subtree of d. Let $\phi_d(t)$ denote the distinct occurrences of t in d, i.e. $\phi_d(t)$ is the set $\{\{\Phi^1(v_1), \ldots, \Phi^1(v_k)\}, \ldots, \{\Phi^n(v_1), \ldots, \Phi^n(v_k)\}\}$ where $(v_1, \ldots, v_k) \in V_t$ and $|t| = k$; with $\Phi^1, \ldots, \Phi^n$ the distinct matching functions from t into d. With $\phi(t)$ we denote the union of all occurrences in the database of t: $\phi(t) = \cup_{d \in D} \phi_d(t)$. Let $\psi_d(t) = 1$ if $|\phi_d(t)| > 0$ and 0 otherwise. The support of a pattern tree t in the database D is then defined as $\psi(t) = \sum_{d \in D} \psi_d(t)$, that

is the number of records in which it occurs one or more times. A pattern tree t is called frequent if $\psi(t)$ is greater than or equal to a user defined minimum support (minsup) value. The goal of frequent treemining is to find all frequent trees in a given database. Frequent treemining algorithms make use of the apriori property, which states that any subtree of a frequent tree is also frequent and any supertree of an infrequent tree is also infrequent.

To mine all frequent trees that satisfy the selection constraints we use some properties of closed trees: a tree t is closed if all supertrees of t have lower support. The blanket of t, denoted by B_t, is defined as the set of frequent supertrees of t that can be constructed from t by adding one vertex. Using the definition of blanket, we can rewrite the definition of closed trees in terms of the blanket of a tree: a tree t is closed iff $\forall t' \in B_t : \ \psi(t') < \psi(t)$. For two trees t_1 and t_2, we call t_1 occurrence matched with t_2, if there is an extension t_3 of t_1, with one or more nodes of t_2, such that t_3 is a supertree of t_2 and for every occurrence of t_1 there is a distinct occurrence of t_3. More formally, t_1 is occurrence matched with t_2, with $t_2 \not\preceq t_1$, if there exists a tree t_3 with $t_1 \preceq t_3 \wedge t_2 \preceq t_3$ and for every transaction $d \in D$ in which t_1 occurs we have that

$$\forall X \in \phi_d(t_1) \exists Y \in \phi_d(t_2) : X \cup Y \in \phi_d(t_3).$$

If t_1 is occurrence matched with t_2 then t_1 is not closed, because we can extend t_1 with the vertices of t_2 that are not part of t_1; the support of this new tree is equal to the support of t_1. Let $\phi(t|t')$ denote the occurrences of t that have a matching occurrence for t', i.e.

$$\phi(t|t') = \cup_{d \in D} \phi'_d(t) \ \text{ where } \phi'_d(t) =$$

$$\cup \{X | X \in \phi_d(t) \wedge \exists Y \in \phi_d(t') \exists t'' : X \cup Y \in \phi(t'') \wedge t \preceq t'' \wedge t' \preceq t'' \wedge t' \not\preceq t\}.$$

Let $\psi(t|t') = \sum_{d \in D} \psi_d(t|t')$ denote the support of t restricted to t' where $\psi_d(t|t') = 1$ if $|\phi_d(t|t')| > 0$ and 0 otherwise. A tree t is closed with respect to t' iff $\forall t'' \in B_t : \psi(t''|t') < \psi(t|t')$. Our definition of occurrence matching is an extension of the definition given in [6] where occurrence matching is only defined between a tree t and the trees in B_t.

To enumerate all frequent closed trees we use CMtreeminer, described in [6]. Here we give a brief summary of the CMtreeminer algorithm. Enumerating all frequent subtrees is done by using the rightmost extension techniques described in [2]: a $(k-1)$-tree is expanded to a k-tree by adding a new node *only* to a node on the rightmost branch of the $(k-1)$-tree. The rightmost branch of a tree is the unique path from the root to the rightmost leaf. Note that for each k-tree its parent is uniquely defined by removing the rightmost vertex. This procedure of extending pattern trees ensures that each pattern tree is counted exactly once. To compute the closed trees some extra pruning techniques are used: left-blanket pruning and right-blanket pruning. If a tree t is occurrence matched with a tree $t' \in B_t$, we distinguish two possible locations at which the additional vertex (v) should be added to extend t into t': **i)** v is added to the rightmost path of t, **ii)** v is added elsewhere. In the first case (right-blanket pruning) we should not

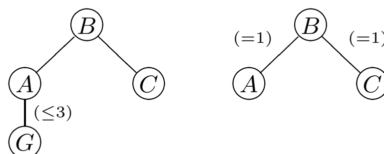


Fig. 1. Example of selection constraints.

extend t by adding vertices to any proper ancestor of v. In the second case (left-blanket pruning) t nor any extension of t can be closed, hence t can be pruned. When the trees are unordered rooted trees, multiple rooted unordered trees are equivalent to one rooted ordered tree. To solve this a canonical representation is needed, so in [5, 6, 9] fast canonical representations are developed. When the unordered trees are in canonical form, they can be further processed as ordered trees. All techniques used in CMtreeminer also apply to unordered trees that are in DFCF (depth first canonical form) as defined in [6]. In the rest of this work we assume that an unordered tree is in DFCF. Although CMtreeminer is also able to mine all maximal trees, we did not use this in our algorithm, because the computation of all maximal trees as done in CMtreeminer does not speed up the mining process. In fact, when mining only for maximal trees, all closed trees are a byproduct of the algorithm. Furthermore, maximal frequent trees have the disadvantage that with the extraction of all frequent trees from the set of maximal trees the exact support is lost; only a lower bound of the support can be determined.

4 Selection Constraints

Consider the web log of an online store. Suppose a data analyst is interested in customers who were in doubt between buying two different computers sold by the store. More specifically, the analyst is interested in users who visited both web pages that describe computers of type A and web pages that describe computers of type B. Suppose the analyst wants to know which points were discriminating for customers into making their choice (described by pages that can be reached from the pages about the specific computer types) and what other products these customers were interested in. With the selection constraints we define below, the data analyst can specify such constraints in the treemining algorithm. To specify the constraints in the constraint language, the node that represents the start page of the department that sells computer supplies is added as a root. To further specify the aforementioned constraints, the node that represents the top of the web pages describing computer type A, is added as the first child of the root. The second child of the root is the top of the web pages that describe computer type B. In general, selection constraints are defined as follows:

1. A labeled root ordered tree that consists of one or more nodes where the distance between an ancestor and a descendant can be constrained ($\leq, =, \geq$).

2. If S_1 and S_2 are two selection constraints then $S_1 \vee S_2$ is also a selection constraint.

A tree T satisfies the constraints if T is an *embedded* supertree—with respect to the distances specified between ancestors and descendants—of at least one of the terms in the disjunction. More formally, a tree T satisfies a selection constraint if there exists a term C of the disjunction such that: there exists an injective function $\Phi : V_C \rightarrow V_T$ satisfying the following conditions for any $v_1, v_2 \in V_C$:

- Φ preserves the labels: $L_C(v_1) = L_T(\Phi(v_1))$.
- Φ preserves the order among the siblings: if $v_1 \leq_C v_2$ then $\Phi(v_1) \leq_T \Phi(v_2)$.
- Φ preserves the ancestor-descendants relation and respects the constrained distance between them : if $(v_1, v_2) \in E_C$ and $dist(v_1, v_2) \odot n$ then $\Phi(v_1)$ is an ancestor of $\Phi(v_2)$ in T and the path length between $\phi(v_1)$ and $\phi(v_2) \odot n$, with $\odot \in \{\leq, =, \geq\}$.

Two examples of selection constraints are given in figure 1. On the left, the distance between the root node and its descendants is unspecified, so they may occur at any distance. The node labeled G must however occur in maximum distance of 3 from node A. On the right, we insist the nodes labeled A and C are children of the root node. As a third example, in figure 2 the selection constraint C_t does not specify a distance between the root and its children, so any tree with label B that has descendants A and C that are not descendants of each other satisfies the constraint.

The selection constraints are inspired on succinct constraints used in frequent itemset mining [8] and regular expression constraints in sequence mining [7]. For example, suppose the selection constraints consist of two trees, the first tree with a root labeled A and a descendant node labeled B and the second tree with a root labeled B and a descendant node labeled A. These constraints, which are in fact all possible rooted trees with the nodes A and B, are similar to the succinct constraints for which the items A and B must be in the itemset. In sequence mining, with an alphabet $A - Z$ the regular expression constraint $A(C{-}Z)^*B|B(C{-}Z)^*A$ is similar as well. However, because trees consist of more structures than sequences or sets, it is natural to incorporate more structure in constraints for trees.

4.1 Optimization Using Selection Constraints

In order to compute all frequent trees that satisfy the constraints, we first find all *minimal* trees that satisfy the constraints (*minimal* in the sense that no subtree satisfies the constraints). This selection is done in the first scan through the database; due to space limitations we will not elaborate on this further. These minimal trees are henceforth called base trees. Any frequent tree satisfying the constraints must be a supertree of *at least* one of the frequent base trees.

The idea is to extend each frequent base tree—similar to the frequent one patterns in other treemining algorithms—in such a way that all closed frequent

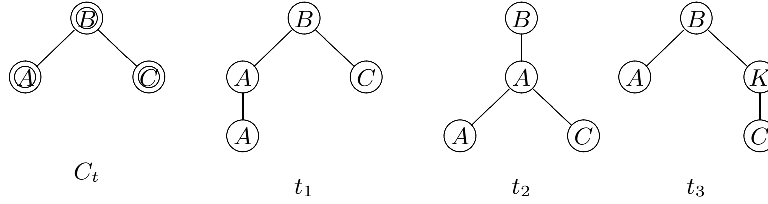


Fig. 2. A selection constraint (C_t) and three trees (t_1, t_2, t_3) that satisfy the constraint, since all the nodes of C_t occur in-order. t_1 is however not a base tree because it is not minimal.

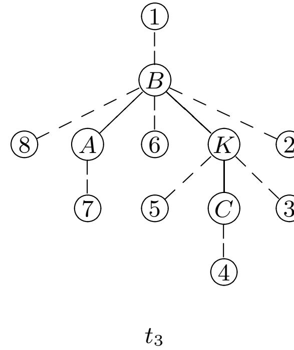


Fig. 3. Locations for an additional vertex to be added to the base tree t_3.

trees that satisfy the constraints are enumerated. With the standard rightmost extension technique, we can however not generate all supertrees if the enumeration is started from the base trees. To illustrate this point, in figure 3 all possible locations to add new nodes to the base tree t_3 (the rightmost tree in figure 2) are shown. For example, if a node labeled H occurs frequently on location 6 of t_3, the frequent tree obtained by extending t_3 with H at location 6 should clearly be part of the output. However, this tree can not be enumerated by using the rightmost extension technique, because the position of the new node is not on the rightmost path. Besides the rightmost extension (extension positions 2, 3 and 4 in figure 3), base trees should therefore be extended in the parent direction (marked with label 1 in figure 3) and depending on the shape of the base tree also on the leftmost path (extension positions 7 and 8) and in the area between the leftmost child and the rightmost child of the root (extension positions 5 and 6).

These extension problems can be solved by using the properties of closed trees. Consider an arbitrary base tree bt_0. Rather than starting the enumeration from bt_0 itself, we start the enumeration from its root t_1. Starting from t_1 we are able to enumerate all closed frequent supertrees of bt_0 with root t_1 that satisfy the constraints using *only the rightmost extension* technique as follows: let t_k be a tree constructed from t_1 by applying the rightmost extension technique one or

more times. If tree t_k can be extended to t_{k+1} such that $\phi(t_{k+1}|bt_0) = \phi(t_k|bt_0)$, then t_k is not closed with respect to bt_0, since then $\psi(t_{k+1}|bt_0) = \psi(t_k|bt_0)$. If there is a node in bt_0 that is not contained in t_k, then t_k cannot be closed with respect to bt_0, because we can extend t_k to t_{k+1}, with the node from bt_0 not contained in t_k, such that $\phi(t_{k+1}|bt_0) = \phi(t_k|bt_0)$. With this approach we condense a base tree to one node (its root), because we have enough information at this vertex together with the occurrences of the base tree.

For example consider the base tree t_3 in figure 3. We start by taking the root node of t_3 (B); we extend B with the node A—call this tree t_k. Because there is a supertree t_{k+1} of t_k—the extension of t_k with node K – for which $\phi(t_k|t_3) = \phi(t_{k+1}|t_3)$, t_k is not closed with respect to t_3; hence we keep extending t_k. Furthermore, suppose there is a frequent extension with a node labeled H, that is a sibling of A. This tree t_{k+1} is not closed because there is still a supertree that is occurrence matched with respect to t_3; hence we keep extending t_{k+1}. With this approach the positions $2-8$ of t_3 as shown in figure 3 are considered for extension and all valid results are supertrees of t_3.

With the previous observations we have solved the enumeration problem from the base pattern to extensions on the leftmost path and the area between the first child of the root and the last child. For extension of the base tree (bt_0 with root node t_1) in the parent direction, we compute the parents of t_1; call this set P_{t_1}. If there is a node in P_{t_1}, t_c, that is occurrence matched with t_1, we can condense the base tree to the new node t_c. As a result we extend bt_0 with t_c as the new root of the base tree; the computation is then repeated for the extended base tree and its new root. If t_1 is not occurrence matched with any node in P_{t_1} we add t_1 to the frequent one-patterns and for each frequent node in P_{t_1} we construct a new base tree bt_{0_i}, that is, bt_0 extended with the frequent node as new root node. For each extended base tree of bt_0, to which we further refer as $bt_{0_1}, \ldots, bt_{0_n}$, this procedure is repeated. In algorithm 1 we give our selection constraint algorithm in pseudo-code.

With the algorithms previously specified, we introduce duplicates. For example, when root nodes with the same label of different base patterns have the same occurrences in the database, the enumeration of the subtree starting at these nodes will be initiated twice, leading to duplicate trees in the output. Also the MineParent algorithm causes duplicates: in figure 4 the data tree d_1 contains two base patterns that match the constraint C_t in figure 2; call these base patterns bt_0 (occurring in the left subtree of the root node) and bt_1 (occurring in the right subtree). When the parent sets of bt_0 and bt_1 are computed the base pattern with root node E is present twice. These base patterns $E - D - bt_0$ and $E - bt_1$ need not be the same, because the occurrence list of both patterns might be different; but when we mine all closed trees starting from the two 1-trees with root E and occurrences $\phi(E-D|bt_0)$ and $\phi(E|bt_1)$ respectively we can get d_1 twice in the output.

To overcome this problem, the algorithm requires extra rules when we start the enumeration from a base pattern as well as when extending the current pattern at the root node of another base pattern. Note that the root nodes

Algorithm 1 Pseudo-code of the SCTreeminer algorithm.

Algorithm SCTreeminer

$S \leftarrow$ all frequent base trees $1 \dots n$
$i \leftarrow 1$
$out \leftarrow \varnothing$
while $(S \neq \varnothing)$
  $bt_i \leftarrow$ a tree from S
  $S \leftarrow S \setminus \{bt_i\}$
  $v_i \leftarrow v_0$ /* the root node of bt_i */
  $\phi(v_i) \leftarrow \phi(v_0|bt_i)$
  $F_1 \leftarrow$ MineParent(v_i, bt_i)
  $out \leftarrow out \cup$ ComputeClosedtrees(F_1, bt_i)
  $i \leftarrow i + 1$
return *out*

Function MineParent (set S, current base tree bt_i)

$out \leftarrow \varnothing$
while $(S \neq \varnothing)$
  $t \leftarrow$ a tree from S
  $S \leftarrow S \setminus t$
  $P_t \leftarrow$ the frequent parents of t
  if $\exists t' \in P_t : t$ is occ. matched with t'
    then delete t
    else mark $\phi(t)$ with a tag from bt_i
      $out \leftarrow out \cup \{t\}$
  $S \leftarrow S \cup P_t$
  return *out*

of a base pattern are all "frequent one patterns" previously discovered by the MineParent algorithm.

The basic idea is to define an ordering on the initially discovered base patterns. We use the support of the base patterns as ordering criterion, but the order is in fact arbitrary. Suppose we have n base patterns with ordering: $bt_1 < bt_2 < \dots < bt_{n-1} < bt_n$. Notice that for each base pattern bt_i, a set $bt_{i_1}, \dots, bt_{i_n}$ is associated that are the extensions (generated by the MineParent algorithm) of bt_i; the order of these extended base trees equals the order of the associated base tree. In the rest of this paper, we use bt_i as an abbreviation of the base tree bt_i or an extended base tree from the associated set of bt_i.

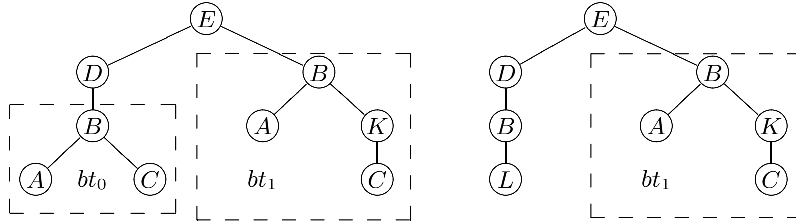


Fig. 4. Two example data trees d_1 (left) and d_2 (right).

When *extending* bt_i the mining algorithm can enumerate all base patterns that have a higher order, i.e. $bt_{i+1}, bt_{i+2}, \dots, bt_n$. In the other case, when we are extending the current base pattern bt_j with a root node from another base pattern bt_i, with $bt_i < bt_j$ we have to determine if the extension t_k of the current base pattern is occurrence matched with bt_i. As long as it is not occurrence

matched we should extend t_k, because t_k may have children that are part of bt_i or children that can not be reached starting from bt_i. Note that in this case one can not enumerate the whole base tree bt_i from t_k. When *starting* from bt_i, suppose the root node of bt_i is also the root node of other base patterns. If all these base patterns have a higher order, we proceed with bt_i. Otherwise, we proceed with bt_i until it is occurrence matched with one of the lower ordered base patterns.

To compute $\phi(t_k|bt_i)$ in an efficient way, we mark in the database all root labels of the base trees and the extended base trees with its corresponding order; this is done in the same sweep trough the database as the computation of the frequent one patterns. With this approach we can determine $\phi(t_k|bt_i)$ without any overhead. For the example with data tree d_1 in figure 4, if we have computed $E - D - bt_0$, we extend it with the nodes of the right subtree of d_1, because $bt_0 < bt_1$. Hence, if d_1 is frequent, it is part of the result. For another example, again consider the two data trees in figure 4. Suppose the extension of bt_1 with nodes E, D and B is frequent, call this tree t_k. Because some occurrences of B are part of the base pattern bt_0 and $bt_1 > bt_0$, we determine if t_k and bt_0 are occurrence matched. Since this is not the case we extend t_k to t_{k+1} by adding a new label A. Since t_{k+1} and bt_0 are occurrence matched, we prune t_{k+1}. Another extension possibility is adding node L to t_k. In this case t_{k+1} is a valid $k+1$ candidate tree and if this tree is frequent it should be be further extended. In algorithm 2 the pseudo-code is shown to compute closed trees; including our duplicate elimination technique.

Algorithm 2 Pseudo-code of CMtreeminer with the duplicate elimination algorithm.

Function ComputeClosedtrees (set of k-trees F_k, current base pattern bt_m)
  $out \leftarrow \varnothing$
  while $F_k \neq \varnothing$
    $t \leftarrow$ a tree from F_k
    $F_k \leftarrow F_k \setminus t$
    if $\exists bt_n : bt_n < bt_m$ **and** t is occurrence matched with bt_n
      then delete t
      else $F_{k+1} \leftarrow F_{k+1} \cup$ all valid and frequent extensions of t
        according to CMtreeminer
    if t is closed
      then $out \leftarrow out \cup \{t\}$
  $k \leftarrow k + 1$
  return out

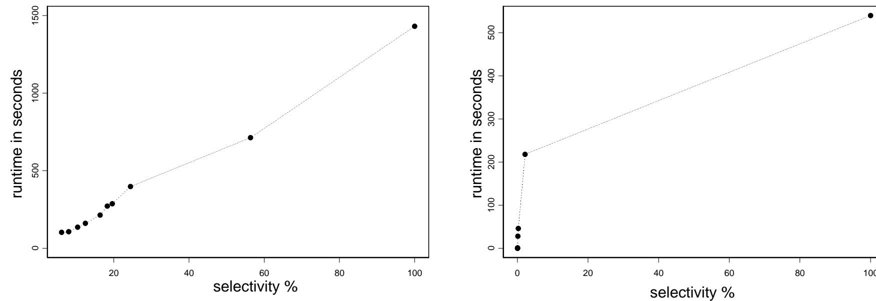


Fig. 5. The running time for different levels of selectivity. The rightmost point in both plots (where the selectivity equals 100) is the case where there are no constraints specified, i.e. the closed treemining algorithm. *Left*: the results for the data set T1D with minimum support value of 1%. *Right*: the results for RPI data set with minimum support of 0.042%.

5 Experimental Results

We implemented all algorithms in C++ and studied the performance of the applications extensively. All experiments were run on a 2.8GHz PC with 500 MB of RAM, running Red Hat Linux. We performed experiments on two data sets: a synthetic data set T1D and the RPI CS web log data. Both the synthetic data set generator and the web log data were kindly provided by Mohamed Zaki and are described in [16]. We briefly describe the synthetic data set generator. First a master tree is generated with the following parameters: the number of distinct node labels $N = 50$, the total number of nodes in the master tree $M = 100,000$, the maximum fanout of a node $F = 10$ and the maximal depth of the tree $D = 15$. From this master tree the data set is created by selecting a number $T = 10,000$ of subtrees from the master tree. The average number of nodes in the trees equals 230. The real data set RPI CS consists of logs of the RPI computer science website. After processing, RPI CS consists of $13,361$ unique web-pages and $59,691$ user browsing subtrees.

The goal of the experiments is to examine the use of selection constraints in frequent treemining. Because our SCTreeminer algorithm is an extension of the closed treeminer algorithm, we chose data sets that require some running time of the closed treeminer algorithm, and compared it to the run time of our SCTreeminer algorithm. The run time of the closed treemining algorithm is a lower bound for the time needed to compute all frequent subtrees that satisfy selection constraints with a post processing approach. We compared the closed treemining algorithm with SCTreeminer for the two data sets with several sets of constraints, with different selectivity. A selectivity of $x\%$ means that $x\%$ of the closed frequent trees satisfies the constraints. Another option was to compare a frequent treemining algorithm with post processing selection constraints and our SCTreeminer algorithm, with a post processing step to extract all frequent trees

from the closed trees. But since on the data set we used, the closed treemining algorithm needed about one quarter of an hour to complete, while a standard frequent treemining algorithm needed more than 3 days, the comparison would not have been very illustrative.

In figure 5 the results of the experiments are shown. For the synthetic data set $T1D$ the run time appears to increase linearly with selectivity of the output. The maximum speedup achieved is about 14. In the real data set RPI, five out of nine runs with selection constraints used less than 0.5 seconds of CPU time. The number of closed frequent trees that satisfy the constraints for these runs was between 28 and 65, a maximum selectivity of 0.06%. Note that we did not get a selectivity higher than 3% for any run with selection constraints. An explanation for this is the high number of labels compared with the number of closed frequent trees, i.e. many node labels occur in only very few trees. Note that the computation time for the run with a selectivity of 3% is fairly high (218 seconds). This is because when mining this data set with such a low minimum support (0.042%) value, the size distribution of the frequent closed trees has a long right tail. This tail is likely to be caused by webcrawlers which contrary to regular users visit a large number of webpages. The run with a selectivity of 3% contained many of these trees with large size: 58% of the trees were of size greater than 18, while in the run with no constraints only 3% of the trees were of size greater than 18.

6 Conclusion

In this paper we proposed the use of selection constraints for frequent treemining. Selection constraints allow the user to give a partial specification of patterns of interest. We have shown in our experiments that the use of selection constraints leads to a reduction of the search space (and hence computation time), and may yield a considerable reduction of patterns that are of no interest to the user. The reduction of the search space is achieved by pre-selecting records of the database that satisfy the constraints.

Further research can be divided into two directions. The first direction is to change the subtree inclusion relation, i.e. the extension of the selection constraints treemining algorithm in such a way that also embedded and incorporated trees are covered. Another interesting possibility in this direction is to define similar constraints and develop similar algorithms for graphs as well. The second research direction is to extend the constraint specification language such that also constraints on attributes of nodes can be incorporated. We plan to further investigate these topics in the near future.

Acknowledgment

We would like to to thank professor Mohamed Zaki for providing the data set and the synthetic data generator. This work is supported by the Netherlands Organisation for Scientific Research (NWO) under grant no. 612.066.304.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.
2. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In R. L. Grossman, J. Han, V. Kumar, H. Mannila, and R. Motwani, editors, *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.
3. R. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of the 15th International Conference on Data Engineering*, page 188, 1999.
4. B. Bringmann. Matching in frequent tree discovery. In *ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 335–338, 2004.
5. Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, page 509, 2003.
6. Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. Cmtreeminer: Mining both closed and maximal frequent subtrees. In *The Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, May 2004.
7. M. N. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 223–234, 1999.
8. R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In A. T. Laura and M. Haas, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 13–24, 1998.
9. S. Nijssen and J.N. Kok. Efficient discovery of frequent unordered trees. In *In Proceedings of the first International Workshop on Mining Graphs, Trees and Sequences (MGTS2003)*, pages 55–64, 2003.
10. N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT '99: Proceeding of the 7th International Conference on Database Theory*, pages 398–416, 1999.
11. J. Pei and J. Han. Can we push more constraints into frequent pattern mining? In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000.
12. U. Rückert and S. Kramer. Frequent free tree discovery in graph data. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 564–570, 2004.
13. K. Wang and H. Liu. Discovering structural association of semistructured data. *Knowledge and Data Engineering*, 12(2):353–371, 2000.
14. X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, 2003.
15. L. H. Yang, M. Lee, W. Hsu, and S. Acharya. Mining frequent quer patterns from xml queries. In *Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03)*, pages 355–362, 2003.

16. M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of the Int'l Conf. on Knowledge Discovery and Data Mining*, pages 71–80, 2002.
17. M. J. Zaki and C. C. Aggarwal. Xrules: an effective structural classifier for xml data. In L. Getoor, T. E. Senator, P. Domingos, and C. Faloutsos, editors, *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 316–325, 2003.

A Graph-Based Rule-Mining Framework for Natural Language Learning and Understanding

Lukas Molzberger

Fraunhofer Institut Autonome Intelligente Systeme
lukas.molzberger@ais.fraunhofer.de

Abstract. Learning and understanding natural languages are usually considered as independent tasks in natural language processing. These two tasks, however, are strongly interrelated and are presumably unsolvable as separate problems. In this paper, we present an algorithm called Frequent Rule Graph Miner (FRGM) that tackles these problems by alternately improving on the language model and the example interpretations. FRGM is based on an effective graph-mining algorithm adapted for enumerating frequent rule-graphs and is applicable to different layers of natural language processing such as morphology, syntax, semantics and pragmatics.

1 Introduction

Learning and understanding natural languages are two of the most challenging problems in artificial intelligence. One reason why these problems are so hard is that they can not be solved independent from one another [3]. Before deeper linguistic knowledge can be learned from an example sentence, an interpretation of the meaning of this sentence is required. But the construction of such an interpretation requires that a sufficiently good language model has been learned before. In this paper, we propose a rule-based approach able to integrate both the learning and the understanding steps.

Another problem in natural language processing is that the different layers of language such as morphology, syntax, semantics and pragmatics can not be processed separately from one another. There is also no predefined order in which rules acting on these layers have to be processed. Consider, for instance, the problem that grammar is in many cases syntactically ambiguous, i.e. there are often multiple possible parse trees for a given sentence. Choosing the most appropriate one usually requires semantic and contextual information [5]. One possible way to overcome this problem is to provide a common representation scheme in which all the information needed for a useful interpretation of text can be expressed. In our approach we chose a powerful class of labeled graphs, called *text-graphs* which meets these demands. Text-graphs allow to represent words, semantic annotations and concepts as vertices and to put them in relation to each other by the means of edges.

To represent knowledge in our language model, we introduce *rule-graphs*. A rule-graph is a pair $A \rightarrow B$, where A and B are labeled graphs. In this rule-graph, A is the rule body which is required to occur in a text-graph before the rule-graph can be applied. When that happens, B the rule head will be added to the text graph. Since these rules are based on graphs they are able not only to recognize a relational pattern, but also to provide a resulting relational pattern that references the original pattern. The ability to process relational data is an important trait that distinguishes rule-graphs from more conventional propositional rules.

In our approach the task of understanding a given sentence is performed by inferring an interpretation from this sentence using a previously learned language model. This is done by iteratively applying the rules in the model on a text-graph representation of the sentence. Hereby, each rule can rely on the information that has been added during the previous round so that more and more information is accumulated in each round. Ideally, the final interpretation assigns each word a unique meaning and also describes the relations between the phrases of the sentence. The rules are applied in a forward chaining fashion, meaning that we start from the data, and successively apply the rules. We note that this contrasts the approach taken by most parsing algorithms, where the starting point is a predefined goal that these algorithms try to "explain" by chaining the rules backwards until a complete parse tree is found. The other task that we try to tackle in our approach is to learn the language model from a set of example text-graphs. To do so, we formulate the learning task as a problem of finding all rule-graphs that occur frequent as positive embeddings and infrequent as negative embeddings in these examples. In other words we search for rule-graphs that are as general as possible, but at the same time do not make too many mistakes. The assumption is that rule-graphs that occur frequent in the training examples, will accurately predict the target values on unknown data, too. To determine the frequency threshold we use a linear function that requires higher frequency for larger rule-graphs, because those are less likely to be correct according to Occam's razor. If the quality of the training data is good enough, it makes sense to set the maximum threshold for negative embeddings to zero in order to disallow false embeddings. The forward-chaining property of our rule system allows to facilitate false parses as negative training examples.

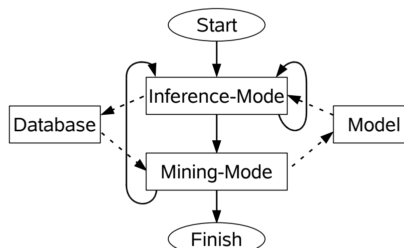


Fig. 1. The main routine

The idea behind our approach is to go through several cycles of mining (learning) and inference (understanding) steps, so that alternately the language model and the training database are improved (Fig.1). During the first round only very basic rules are learned from the example database, but those rules are then applied to the database so that the next mining round can learn from a richer, better understood set of examples. Thus, complex training examples can profit from rules that have been previously learned from simpler examples. Once the model has been sufficiently trained it can be used to process a new previously unknown text. The output of this processing step is solely determined by the training examples that have been used to train the model.

For efficiency reasons we store the model in a generalized version space tree (GVST) as proposed by Rückert and Kramer [4]. But that means, that we have to define a unique or canonical form in which to store the rule-graphs.

Another application area where forward-chaining based inference has been quite successfully used is expert systems. Here, however, rules are usually based on a first order logic representation instead of a graph based one. The most popular algorithm for matching such rules is RETE [1].

In production rule systems runtime efficiency has always been a critical issue. Fortunately, new advances in the field of graph-mining such as GVSTs, allow to improve the performance not only of the rule-mining step but also for the rule-application step. The FRGM algorithm itself is based on a basic graph-mining algorithm similar to the gSpan [6] algorithm. The task of such an algorithm is to enumerate all frequent subgraphs in a database of graphs. A new technique that we employ in our algorithm are the *refinement candidates*, which simplify the generation of rule-graph refinements. Another approach concerned with mining for rule-graphs is the Subdue system [2]. Their graph grammar rules however replace (compress) parts of the instance graphs instead of adding new subgraphs.

2 A running Example

Before we go into further details, we want to illustrate our algorithm on a simple example sentence. We start the example by transforming the sentence "the dog chased the cat" into a text-graph where each word has a corresponding vertex. The correct sequence of words in the text-graph is maintained by edges specifying the previous word relations between the vertices. In this step, additional edges and vertices can be introduced to represent morphological features or long ranging word relations. In addition to the input data (marked by continuous lines), we need to specify the learning target (marked by dotted lines) as a set of additional vertices and edges. The learning target are those elements that we hope to predict on unknown data. According to Fig.1 processing would start with the rule application (inference) step, but since the model is still empty we can directly jump to the mining step. In mining mode the algorithm attempts to find all rule-graphs that occur frequently as positive example embeddings and at the same time infrequent as negative example embeddings. Positive example embeddings are those where the complete rule, body and head, can be embedded

into the example text-graph such that the edges, labels and roles (INPUT ⇔ BODY, TARGET ⇔ HEAD) match together. A positive example embedding is for instance the embedding $\{(1 \mapsto 1), (2 \mapsto 6)\}$ of rule-graph G_r^1 in text-graph G_t^1. Example embeddings where only an embedding of the rule-body exists, but not of the rule-head are counted as either negative or neutral. This depends on whether the rule-head and the target part of the text-graph have common labels. Consider for instance the embedding $\{(1 \mapsto 4)\}$ of the rule-body of rule-graph G_r^1 into the text-graph G_t^1. This embedding is negative, because the label A is present as the label of a target vertex (e.g. Vertex 6) but there is no target vertex corresponding to the head of the rule-graph. The intention behin d this scheme is to implicitly define the negative training examples. This scheme, however, demands that the target vertices of a certain label have always to be completely provided for a given text-graph since otherwise valid occurrences of a rule would be counted as negative ones.

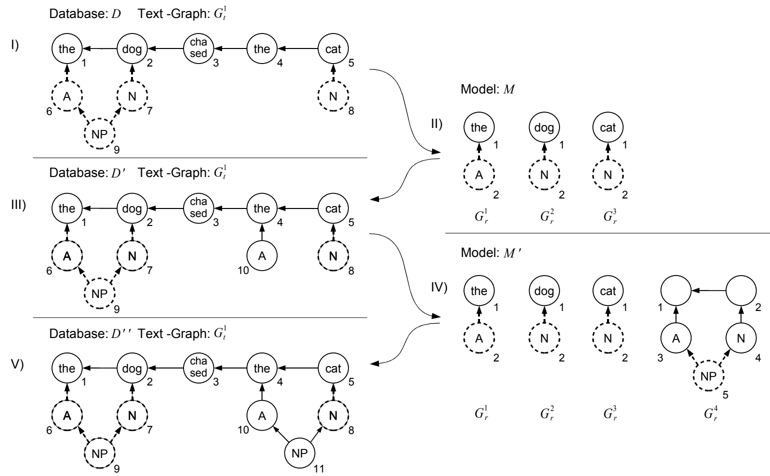


Fig. 2. The running example

Now, let us assume that the rule-graphs shown in model $\mathcal{M}$ meet the threshold demands and are therefore the results of the first mining round. These rules are then used in the inference step to create a first interpretation of the example sentence. This is done by applying all rules in the model on all occurrences of their bodies in the database. The interpretation, though not perfect yet, already delivers some additional information to the original sentence (see $\mathcal{D}^{'}$, G_t^1). These informations further the next mining round so that more interesting and complex rules can be found (see $\mathcal{M}^{'}$). The rule G_r^4 can not be found earlier because the vertices labeled A (article) and N (noun) have not been there before and the patterns "the dog" and "the cat" were too infrequent.

3 The Text- and Rule-Graphs

In this section, we define the text- and rule-graphs as well as some necessary notations related to them.

A *text-graph* G_t is a directed acyclic graph (DAG) consisting of a 5-tuple $(V_t, E_t, \Sigma, \lambda_t, \alpha_t)$, where $V_t = \{v_1, ..., v_n\}$ is a set of vertices, $E_t \subseteq V_t \times V_t$ is a set of edges, Σ is an alphabet, $\lambda_t : V_t \cup E_t \rightarrow \Sigma$ is a labeling function mapping the vertices and edges to Σ, and α_t is a function that assigns to each vertex and edge either $\{INPUT\}$, or $\{TARGET\}$, or $\{INPUT, TARGET\}$.

The target elements in text-graphs play an important role in that they determine the future structure and labeling of the learned rules. Therefore, they also determine the final processing results on new sentences.

An important feature of text-graphs is that during inference they can be enriched by additional information represented by new vertices and edges. In order to avoid redundancy (i.e. the problem that two vertices represent the same information) we define a merging scheme that allows to eliminate those redundant vertices. Such a merging scheme requires that all vertices except the initial ones can uniquely be identified by their labels and their outgoing edges.

Vertices of a text-graph can be interpreted in a dual way as objects and as predicates relating to other objects. Therefore, the merging scheme allows to uniquely identify objects even if multiple rule-graphs recognize the same object.

Let $G_t = (V_t, E_t, \Sigma, \lambda_t, \alpha_t)$ be a text-graph. Two vertices $v_i, v_j \in V_t$ are *mergeable* if and only if (iff) (i) both have the same label, i.e., $\lambda_t(v_i) = \lambda_t(v_j)$, (ii) there are edges $e_i = (v_i, v), e_j = (v_j, v) \in E_t$ pointing to some common vertex $v \in V_t$ such that $\lambda_t(e_i) = \lambda_t(e_j)$, (iii) there are no edges $e_i = (v_i, v_x), e_j = (v_j, v_y) \in E_t$ pointing to different vertices v_x and v_y such that $\lambda_t(e_i) = \lambda_t(e_j)$, and (iv) there is no edge (v_i, v_j) or (v_j, v_i) connecting these two vertices. To illustrate this, consider the vertices A_1 and A_2 in Fig.3. Here, A_1 and A_2 are mergeable in text-graph ii), but not in the text-graphs i), iii) and iv).

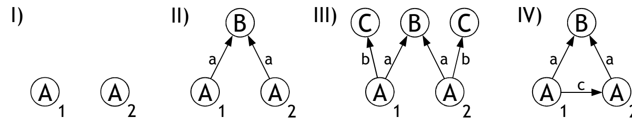


Fig. 3. Mergeable (ii) and non-mergeable (i,iii,iv) vertices.

Let $v_i, v_j \in V_t$ be two mergeable vertices of a text-graph $G_t = (V_t, E_t, \Sigma, \lambda_t, \alpha_t)$. The graph derived from G_t by merging the vertices $v_i, v_j \in V_t$, denoted $\beta(G_t, v_i, v_j)$, is a text-graph obtained from G_t containing a new vertex v_k, that replaces the vertices v_i and v_j. Edges that were either connected to v_i or v_j are now connected to v_k. The same applies to labels. The roles of v_i and v_j are combined by a set union.

A *canonical text-graph* $G_t = (V_t, E_t, \Sigma, \lambda_t, \alpha_t)$ is a text-graph containing no two vertices $v_i, v_j \in V_t$ that are mergeable.

Any non-canonical text-graph G_t can be transformed into a canonical text-graph by successively applying the merge operator β on all mergeable vertices in G_t. We note that the order of the merge operator leading to canonical text-graphs is arbitrary. This is stated by the following proposition.

Proposition 1 For each text-graph it holds that its canonical text-graph is unique.

A *rule-graph* G_r is a connected DAG consisting of a 5-tuple $(V_r, E_r, \Sigma, \lambda_r, \alpha_r)$, where $V_r = \{v_1, ..., v_n\}$ is a set of vertices, $E_r \subseteq V_r \times V_r$ is a set of edges, $\lambda_r : V_r \cup E_r \hookrightarrow \Sigma$ is a partial labeling function, and α_r is a function that assigns to each vertex and edge either $\{HEAD\}$ or $\{BODY\}$.

In Fig. 2:G_r^1,... ,G_r^4 we see that in a rule-graph all vertices and edges have a role label that assigns them a role as rule head (marked by a dotted line) or rule body (marked by a continuous line).

Given a rule-graph $G_r = (V_r, E_r, \Sigma, \lambda_r, \alpha_r)$, the *rule body* G_r^{body} of G_r is the graph $G_r^{body} = (\{v_r \in V_r \mid BODY \in \alpha_r(v_r)\}, \{e_r \in E_r \mid BODY \in \alpha_r(e_r)\}, \Sigma, \lambda_r, \{V_r \cup E_r \to \{BODY\}\})$ of G_r whose vertices and edges have the role $BODY$ assigned to them. In a similar way, the *rule head* G_r^{head} of G_r is the graph $G_r^{head} = (\{v \in V_r \mid \alpha_r(v) \subseteq \{HEAD\}\}, \{e \in E_r : \alpha_r(e) \subseteq \{HEAD\}\}, \Sigma, \lambda_r, \{V_r \cup E_r \to \{HEAD\}\})$ of G_r whose vertices and edges have only the role $HEAD$ assigned to them.

A *database* is a set of example text-graphs and a *model* a set of rule-graphs.

The *role mapping function* δ is a bijection, mapping the rule-graph roles onto the text-graph roles in the following way: $\{(\emptyset \mapsto \emptyset)$, $(\{BODY\} \mapsto \{INPUT\})$, $(\{HEAD\} \mapsto \{TARGET\})$, $(\{BODY, HEAD\} \mapsto \{INPUT, TARGET\})\}$

Let $G_t = (V_t, E_t, \Sigma, \lambda_t, \alpha_t)$ be a text-graph and $G_r = (V_r, E_r, \Sigma, \lambda_r, \alpha_r)$ be a rule-graph. An *isomorphic embedding* of G_r into G_t is a bijection $\varphi : V_r \to V_t$ preserving the edges, labels, and roles with respect to δ, i.e.,

- for every $v_i, v_j \in V_r, (v_i, v_j) \in E_r$ iff $(\varphi(v_i), \varphi(v_j)) \in E_t$,
- $\lambda_r(v) = \begin{cases} \lambda_t(\varphi(v)) & \text{if } \lambda_r(v) \text{ is defined} \\ \varnothing & \text{otherwise} \end{cases}$ for every $v \in V_r$,
- $\lambda_r((v_i, v_j)) = \begin{cases} \lambda_t((\varphi(v_i), \varphi(v_j))) & \text{if } \lambda_r((v_i, v_j)) \text{ is defined} \\ \varnothing & \text{otherwise} \end{cases}$
  for every $(v_i, v_j) \in E_r$,
- $\delta(\alpha_r(v)) \subseteq \alpha_t(\varphi(v))$ for every $v \in V_r$, and
- $\delta(\alpha_r((v_i, v_j))) \subseteq \alpha_t((\varphi(v_i), \varphi(v_j)))$ for every $(v_i, v_j) \in E_r$.

The isomorphic embedding describes the occurrence of either a rule body or a complete rule-graph, in a text-graph. Take for instance the rule-graph G_r^4 in model $\mathcal{M}'$ and the text-graph G_t^1 in database $\mathcal{D}'$ in our running example, then there exists an isomorphic embedding $\{(1 \mapsto 1), (2 \mapsto 2), (3 \mapsto 6), (4 \mapsto 7), (5 \mapsto 9)\}$.

4 The Graph Inference and the Graph Mining Problem

Before we explain the graph inference problem in detail we would like to give a more complex example (Fig.4) that shows how a syntactic disambiguation can be realized with graph inference. Given are two example sentences where the prepositional phrase (i.e. "with ...") relates either to the noun phrase "the rat" or to the verb "poisoned". These sentences have already been partially processed so that some additional information is already given. The example now shows how new information (marked by dotted lines) is added to the text-graphs by applying the rule-graphs c), d), and e). In the example, rule-graph c) adds another noun phrase vertex to the text-graph a), which refers to the phrase "the rat with white hair". This rule-graph is not a pure grammar rule, since it also takes the semantic information into account that a rat is an animal and that hair are a body part. Without these information the rule would not be able conclude that the noun phrase "the rat" together with the prepositional phrase "with white hair" form a larger noun phrase. It is obvious that the phrase "the rat with arsenic" in b) does not form a meaningful noun phrase. The next rule-graph adds the concept "C:poisoned" to both text-graphs[1] which represent the respective predicate of these sentences. The last rule-graph adds only a single edge, describing that arsenic has been used for poisoning.

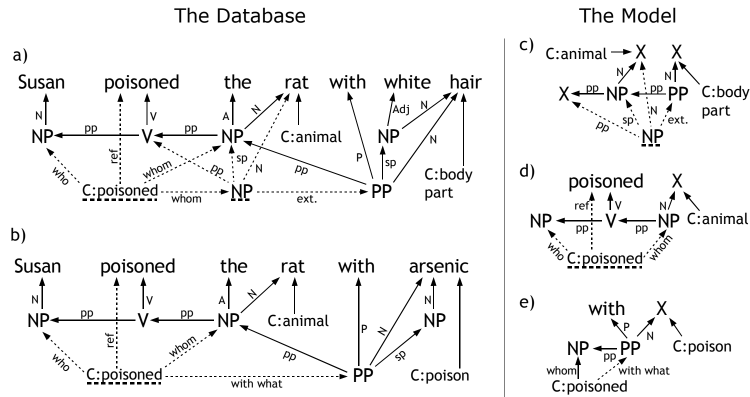


Fig. 4. A syntactic disambiguation example [2]

Now, we formally define the graph inference problem, i.e. the problem of how rule-graphs are used to derive interpretations from sentences.

[1] Note that rule-graph d) can be applied twice on text-graph a).

[2] X: unlabeled vertex; NP: noun phrase; PP: prepositional phrase; V: verb; pp: previous phrase; sp: subphrase; N: main noun; A: article. We note that edges determining the range of phrases and the relations between words have been omitted.

Let $G_t = (V_t, E_t, \Sigma, \lambda_t, \alpha_t)$ be a text-graph and $G_r = (V_r, E_r, \Sigma, \lambda_r, \alpha_r)$ be a rule-graph containing the subgraphs $G_r^{head} = (V_r^{head}, E_r^{head}, \Sigma, \lambda_r^{head}, \alpha_r^{head})$ and G_r^{body} such that G_r^{body} can be embedded into G_t via subgraph isomorphism. Let $V_t^{'}$ be a new set of vertices such that $V_t \cap V_t^{'} = \emptyset$ and $|V_r^{head}| = |V_t^{'}|$. Let further $\varphi^{body} : V_r^{body} \rightarrow V_t$ be a bijection corresponding to such an embedding of G_r^{body}, $\varphi^{head} : V_r^{head} \rightarrow V_t^{'}$ be a bijection, and $\varphi = \varphi^{head} \cup \varphi^{body}$ be the combined bijection from V_r to $V_t \cup V_t^{'}$. The graph derived from G_t by G_r and φ^{body}, denoted $\tau(G_r, G_t, \varphi^{body})$, is a text-graph $G_t^{\tau} = (V_t^{\tau}, E_t^{\tau}, \Sigma, \lambda_t^{\tau}, \alpha_t^{\tau})$, where

- $V_t^{\tau} = V_t \cup V_t^{'}$,
- $E_t^{\tau} = E_t \cup E_t^{'}$ such that $E_t^{'} = \{(\varphi(v_i), \varphi(v_j)) \mid \forall (v_i, v_j) \in E_r^{head}\}$,
- $\lambda_t^{\tau}(x) = \begin{cases} \lambda_t(x) & \text{if } x \in V_t \cup E_t \\ \lambda_r^{head}(\varphi^{-1}(x)) & \text{if } x \in V_t^{'} \\ \lambda_r^{head}((\varphi^{-1}(v_i), \varphi^{-1}(v_i))) & \text{if } x = (v_i, v_j) \in E_t^{'} \end{cases}$
- $\alpha_t^{\tau}(x) = \begin{cases} \alpha_t(x) & \text{if } x \in V_t \cup E_t \\ \{INPUT\} & \text{if } x \in V_t^{'} \cup E_t^{'} \end{cases}$

Given an occurrence of a rule body in a text-graph, the rule inference operator τ is used to apply this rule on the text-graph. That means, a copy of the head of this rule will be added to the text-graph. Head edges linking to the body of the rule will be transfered using the embedding of the rule body. To illustrate this, consider the text-graph G_t^1 in database $\mathcal{D}^{'}$ and the rule-graph G_r^4 in model $\mathcal{M}^{'}$ in our running example (Fig.2). Let $\{(1 \mapsto 4), (2 \mapsto 5), (3 \mapsto 10), (4 \mapsto 8)\}$ be an embedding of this rules body in the text-graph. The result of the inference operation is shown in v), that is, a new vertex labeled NP has been added.

Using the above notation of rule inference operator τ, we are now ready to define the *model inference operator*. This operator, denoted π is used to apply all rule-graphs of a model on a text-graph and to combine the results afterwards. Multiple additions of the same information are prevented by the text-graph merging scheme. For a canonical text-graph $G_t = (V_t, E_t, \Sigma, \lambda_t, \alpha_t)$ we define $G_t^{x,y} = \tau(G_r^x, G_t, \varphi_y^{body})$ for all $G_r^x \in \mathcal{M}$ [1] and all isomorphic embeddings φ_y^{body} of G_r^{body} into G_t. The text-graph $\pi(G_t, \mathcal{M})$ derived from the canonical text-graph G_t by $\mathcal{M}$ is $\pi(G_t, \mathcal{M}) = (V_t \cup V_t^{1,1} \cup V_t^{1,2} \cup ... \cup V_t^{n,m}, E_t \cup E_t^{1,1} \cup E_t^{1,2} \cup ... \cup E_t^{n,m}, \Sigma, \lambda_t \cup \lambda_t^{1,1} \cup \lambda_t^{1,2} \cup ... \cup \lambda_t^{n,m}, \alpha_t \cup \alpha_t^{1,1} \cup \alpha_t^{1,2} \cup ... \cup \alpha_t^{n,m})$.

Given a text-graph G_t and a model $\mathcal{M}$, the *graph inference problem* is to find a new text-graph $G_t^{'}$ such that $G_t^{'}$ can be derived by a sequence of model inference operator applications.

The basic assumption underlying the learning process in our approach is that predictive rules will occur often as positive embeddings in the training database and at the same time seldom as negative embeddings. In contrast to other supervised learning systems we use the notation of positive and negative embeddings instead of examples, because the target we are trying to predict is a subgraph within an example rather than a label for the whole example. Therefore, a single text-graph can contain several positive and negative embeddings of a rule-graph.

[1] Note that x here uniquely identifies a rule-graph of $\mathcal{M}$

Before formulating the graph-mining problem let us first define the *threshold function* $\mu(G_r)$. Given a rule-graph G_r and real numbers a and b, the threshold function is a linear function of the form $\mu(G_r) = |G_r| * a + b$.

Let $\mathcal{D}$ be a database of example text-graphs. Let $count^+(G_r)$ be the number of embeddings of the complete rule-graph G_r (i.e. HEAD + BODY). Let $count^-(G_r)$ be the number of negative embeddings of G_r, i.e. the number of rule body embeddings in those examples that contain the labels of all head vertices of G_r as labels of target vertices minus the number of complete rule embeddings in those examples. The *rule-graph mining problem* is to find a model $\mathcal{M} = \{G_r \mid count^+(G_r) \geq \mu(G_r) \wedge count^-(G_r) < freq_{min}\}$ containing all positively frequent rule-graphs that occur in $\mathcal{D}$.

5 Generalized Version Space Trees and the Canonical Form of Rule Graphs

As already mentioned in the introduction, we use the Generalized Version Space Tree (GVST) structure to efficiently store our rule-graphs in a model. The main idea behind the GVST is to decompose each rule-graph into a sequence of primitive graph refinements. The original rule-graph can then be reconstructed by following the path from the root node, and adding each refinement on the path. A rule-graph that is not in canonical form needs to be transformed to a canonical form prior to storing it in the GVST. This is necessary to prevent that duplicates of the same rule-graph are stored in the GVST.

Basically, the canonical form of a graph is a unique sequential ordering of its vertices. In general the vertices of any graph could be represented in $n!$ different sequences whereby n is the number of vertices. By using a canonical form we can prevent that we have to consider all those different orderings of what is essentially the same graph. Designing a canonical form definition, however, implies several constraints that we need to take care of. Firstly, it must be possible to sequentially construct a connected canonical graph without having to rely on unconnected intermediate graphs. Secondly, an order among the syntactic variants of a rule-graph must be defined. And thirdly, rule-graphs differ from usual graphs in that they consist of two parts, the head and the body. For our purposes, we need to calculate the embeddings not only for the entire rule-graph, but also for the body part only. For this reason it is necessary to separate the head and body vertices in two ranges. Therefore, any canonical rule-graph begins with vertices of the body part, followed by the vertices of the head part.

The following definitions describe the canonical form of rule-graphs by defining a set of interdependent compare operators. The comparison starts with the edges and their labels, goes over the vertices and finally compares whole rule-graphs. The least rule-graph according to this order is the canonical one.

Given two elements x_i and x_j, *the relation* $x_i \prec_R x_j$ is true iff: $(BODY \in \alpha_r(x_i) \wedge BODY \notin \alpha_r(x_j))$

Given two elements x_i and x_j, *the relation* $x_i \prec_L x_j$ is true iff the label $\lambda(x_i)$ of element x_i has a lexicographically lower value than the label $\lambda(x_j)$ of x_j.

Given two elements x_i and x_j, *the relation* $x_i \prec_N x_j$ is true iff: $((x_i = \varnothing) \wedge (x_j \neq \varnothing))$

In the following we use the notation $(a_1 \prec_{X_1} b_1) \rhd (a_2 \prec_{X_2} b_2) \rhd ... \rhd (a_n \prec_{X_n} b_n)$ as shorthand for:

---

if $(a_1 \prec_{X_1} b_1) \vee (b_1 \prec_{X_1} a_1)$ **then return** $(a_1 \prec_{X_1} b_1)$
else if $(a_2 \prec_{X_2} b_2) \vee (b_2 \prec_{X_2} a_2)$ **then return** $(a_2 \prec_{X_2} b_2)$
...
else if $(a_n \prec_{X_n} b_n) \vee (b_n \prec_{X_n} a_n)$ **then return** $(a_n \prec_{X_n} b_n)$

The $e^a \prec_E e^b$ *relation* defines an order on the edges $e^a = (v_i, v_x)$ and $e^b = (v_j, v_y)$ [1], where $x = y$. The $e^i \prec_E e^j$ *relation* is true iff: $(e^a \prec_R e^b) \rhd (i < j) \rhd (e^a \prec_L e^b)$

The $v_i \prec_{VE} v_j$ *relation* defines an order on the vertices v^i and v^j with regard to their adjacent edges. The edges of these vertices are represented by two sequences $T_i = (e^i_1, e^i_2, ..., e^i_m)$ and $T_j = (e^j_1, e^j_2, ..., e^j_n)$. $T_i = (e^i = (v_x, v_y) \in E_r \mid e^i \preceq_R v_i, (x < i \vee y < i), (x = i \vee y = i))$ These sequences are ordered according to $\prec_E$. The $v^i \prec_{VE} v^j$ relation is true iff: $(e^i_1 \prec_E e^j_1) \rhd (e^i_2 \prec_E e^j_2) \rhd ... \rhd (e^i_n \prec_E e^j_n) \rhd (|T_i| < |T_j|)$ where $n = min(|T_i|, |T_j|)$

Given two vertices v_i and v_j, *the relation* $v_i \prec_V v_j$ is true iff: $(v_i \prec_R v_j) \rhd (v_i \prec_{VE} v_j) \rhd (v_i \prec_L v_j)$

Let $v_i, v_j \in V_r$ be two vertices in a rule-graph $G_r = (V_r, E_r, \Sigma, \lambda_r, \alpha_r)$. The rule-graph G_r is *ordered* iff: $\forall v_i, v_j \in V_r : (v_i \prec_V v_j) \leftrightarrow (i \leq j)$

Let G^i_r and G^j_r be two rule-graphs that are ordered. Let further be v^i_x a vertex of rule-graph G^i_r and v^j_x be a vertex of rule-graph G^j_r respectively.

The $G^i_r \prec_G G^j_r$ *relation* is true iff: $((v^i_1 \prec_V v^j_1) \rhd (v^i_2 \prec_V v^j_2) \rhd ... \rhd (v^i_n \prec_V v^j_n))$ where $n = |V_r|$.

Let L be the set of all ordered syntactic variants (i.e. all possible sequential orderings of vertices in a graph that are ordered) $\{G^1_r, G^2_r, ..., G^n_r\}$ of a rule-graph G_r. The least entry in L according to $\prec_G$ is the *canonical form*[2] of rule-graph G_r. $G^{canonical}_r = \{G^i_r \in L \mid \forall G^j_r \in L : G^i_r \prec_G G^j_r\}$

6 The Algorithm

The main routine of the FRGM algorithm, consists of just two loops which alternately call the core routine in inference and mining mode (see Fig.1).

In rule inference mode the FRGM core routine takes a database $\mathcal{D}$ and a model $\mathcal{M}$ as input and computes a new database $\mathcal{D}'$ by applying all the rules in $\mathcal{M}$ on all text-graphs in $\mathcal{D}$. In rule mining mode the algorithm only takes a database $\mathcal{D}$ with example text-graphs as input and computes a new or improved model $\mathcal{M}$ containing all rules that meet the required frequency thresholds.

---

[1] Note that the vertices v_x and v_y may belong to two different variants of the structurally same rule-graph

[2] The implementation of the canonical verification algorithm can be found at: http://www.ais.fraunhofer.de/~lmolz

Input: $\mathcal{D}$: A database of canonical text-graphs
$\mathcal{M}$: The initial model
Output: $\mathcal{D}'$: The result database
$\mathcal{M}$: The result model

```
1:  procedure FRGM(D, D', M)
2:    do
3:      D' ← D, D'' ← D
4:      do
5:        FRGMCore(D', D'', Root(M), INFERE)
6:        D' ← Canonize all text-graphs in D''
7:      while D' changed and not max iter.
8:      FRGMCore(D', ∅, Root(M), MINE)
9:    while M changed and not max iter.
10: end procedure
```

The FRGM is designed on the basis of three data structures. The first one is the GVST which is used to efficiently store the rule-graphs and their associated refinements. The second one is the embedding tree. A root path in this embedding tree describes the embedding of a rule-graph in the same way as a root path in the GVST describes the rule-graph itself. Each rule-graph node in the GVST can have several associated embedding nodes describing all occurrences of this rule-graph in the database. The third data structure are the *refinement candidates* (RCs). A RC describes a possible extension to an embedded rule-graph and is stored in an ordered set associated with the respective embedding. A RC $\{INPUT, TARGET\} \times \{INPUT, TARGET\} \times V_r \times E_t \times V_t$ is determined by the vertex roles, the starting point within the rule-graph, and an edge and a vertex in the text-graph. The order $\prec_C$ on RCs is defined as follows. Given two RCs $c_i = (R_v^i, R_e^i, v_r^i, e_t^i, v_t^i)$ and $c_j = (R_v^j, R_e^j, v_r^j, e_t^j, v_t^j)$, $c_i \prec_C c_j$ is true iff:

if $(v_r^i = \varnothing) \vee (v_r^j = \varnothing)$ **return** $(R^i \prec_R R^j) \rhd (v_r^i \prec_N v_r^j) \rhd (v_t^i \prec_L v_t^j)$
else return $(R_v^i \prec_R R_v^j) \rhd (R_e^i \prec_R R_e^j) \rhd (v_r^i \prec_V v_r^j) \rhd (e_t^i \prec_L e_t^j) \rhd (v_t^i \prec_L v_t^j)$

By using RCs we avoid that refinements have to be considered that would lead to unordered (see definition in section 5) and therefore non-canonical rule-graphs. Of coarse, there may be more than one ordered variant of any given rule-graph. Thus, we still need to perform a complete canonical verification.

Input: $\mathcal{D}$: A database of text-graphs
G_r : The root rule of $\mathcal{M}$
$mode$: The mode $\{INFERE, MINE\}$
Inference Mode Output: $\mathcal{D}'$: The result database
Mining Mode Output: The improved model $\mathcal{M}$

```
1:  procedure FRGMCore(D, D', G_r, mode)
2:    if G_r is the root node of the GVST then
3:      Emb(G_r) ← {(∅, G_t, (∅, ∅)) | ∀G_t ∈ D}
4:    GenerateRCs(G_r)
5:    for all embedding nodes emb = (emb^parent, G_t, (v_r, v_t)) ∈ Emb(G_r) do
6:      for all refinement candidates c_it ∈ RefCand(emb)
7:        RefinementOperator(G_t, G_r, emb, c_it, mode)
8:    for all GVST nodes (ref = (v_r, v_r', l_v, l_e, r_v, r_e), G_r') ∈ Ref(G_r) do
9:      if (mode = MINE ∧ (count^+(G_r') ≥ μ(G_r')))∨
          (mode = INFERE ∧ ((count^+(G_r') > 0) ∨ (r_v = HEAD ∧ (v_r = ∅ ∨ r_e = HEAD)))) then
10:       if count^-(G_r') ≤ freq_min then activate G_r'
11:       Fill the refined rule-graph G_r' based on its parent G_r and the refinement ref.
12:       if G_r' is in its canonical form then FRGMCore(D, D', G_r', mode)
13:     end if
14:   end for
15:   if mode = INFERE and G_r is active then
16:     Apply the rule G_r on all embeddings in Emb(G_r) and add the results to D'.
17: end procedure
```

The FRGMCore routine starts with an initial database scan (ln:2-3) where for each text-graph in $\mathcal{D}$ an empty root embedding node is created. In the next step the algorithm calls the RC generation subroutine, where for all embeddings associated with the current rule-graph G_r the RCs are computed. These RCs are then converted to actual refinements (ln:5-7). The next section (ln:8-13) is the recursive step of the algorithm leading to the next level in the GVST. In line:9 the frequency threshold is checked. In inference mode a frequency of at least 1 is required while iterating the body parts of the rule-graphs. No embeddings are required while iterating the head parts of the rule-graphs, since those are the parts of the rules that we are going to add to the text-graphs. In inference mode the last step is to apply the rule-graph G_r on all occurrences in $\mathcal{D}$ (ln:14-15).

1: **procedure** GenerateRCs(G_r)
2: **if** $mode = INFERE$ **then** $R \leftarrow \{\{INPUT\}\}$ **else** $R \leftarrow \{\{INPUT\}, \{TARGET\}\}$
3: **for all** $(emb^{parent}, G_t, (v_r, v_t)) = emb \in Emb(G_r)$ **do**
4: **if** G_r is the root node in the GVST **then**
5: $C \leftarrow \{(R_v, \emptyset, \varnothing, \varnothing, v_t^{'}) \mid \forall v_t^{'} \in V_t(G_t), \forall R_v \in R : R_v \subseteq \alpha_t(v_t^{'})\}$
6: **else**
7: $C \leftarrow \{(R_v, R_e, v_r, e_t, v_t^{'}) \mid \forall e_t = (v_t, v_t^{'}) \in E_t(G_t), R_v \in R, R_e \in R : R_v \subseteq \alpha_t(v_t^{'}), R_e \subseteq \alpha_t(e_t)\}$
8: $C \leftarrow \{(R_v, R_e, v_r, e_t, v_t^{'}) \in C \mid \neg\exists e_r = (v_r, v_r^{'}) \in E_r(G_r) : v_t^{'} = \varphi(v_r^{'})\}$
9: **end if**
10: $RefCand(emb) \leftarrow RefCand(emb) \cup C$
11: **end for**
12: **end procedure**

The routine GenerateRCs(G_r) is responsible for the creation of RCs for the embeddings. For the root level only initial RCs are created that do not yet have an edge or a starting point in the rule. All later RCs start from an existing rule-graph and describe a possible extension by an edge and a vertex. RCs for edges that already exist in the rule-graph are filtered out (ln:8).

φ : Is an isomorphic embedding derived from the root path of *emb*. (See section 3)
δ : Is the role mapping function (See section 3)
$c_{it} = (R_v, R_e, v_r, e_t, v_t)$

1: **procedure** RefinementOperator($G_t, G_r, emb, c_{it}, mode$)
2: $v_r^{'} \leftarrow \varphi^{-1}(v_t)$
3: **if** $v_r^{'} = \varnothing$ **then** $v_r^{'} \leftarrow$ **new** *Vertex*
4: **else if** $(v_r \succ_V v_r^{'})$ **then return**
5: $ref \leftarrow (v_r, v_r^{'}, \lambda_t(v_t), \lambda_t(e_t), \delta^{-1}(R_v), \delta^{-1}(R_e))$
6: $G_r^{'} \leftarrow \{G_r^{'} \mid \exists(ref, G_r^{'}) \in Ref(G_r)\}$
7: **if** $(mode = MINE) \wedge (G_r^{'} = \varnothing)$ **then**
8: $G_r^{'} \leftarrow$ **new** *RuleGraph*
9: $Ref(G_r) \leftarrow Ref(G_r) \cup \{(ref, G_r^{'})\}$
10: **end if**
11: $emb^{'} \leftarrow (emb, G_t, (v_r^{'}, v_t))$
12: $Emb(G_r^{'}) \leftarrow Emb(G_r^{'}) \cup \{emb^{'}\}$
13: **if** $v_r \neq \varnothing$ **then** $RefCand(emb^{'}) \leftarrow \{c \in RefCand(emb) \mid c_{it} \prec_C c\}$
14: Count the embedding $emb^{'}$ as occurrence of $G_r^{'}$.
15: **end procedure**

The refinement operator converts the RCs into new refinements and embeddings for these refinements. Furthermore, the RCs are partially relayed (ln:13) to this next level of embedding nodes. That is, only those RCs are relayed that are greater than the current RC according to $\prec_C$ because all other RCs would lead to non-canonical rule-graphs, anyway.

7 Conclusions and Future Work

In this paper we have introduced text-graphs as a new way to represent text interpretations consisting of syntactic and semantic annotations as well as interrelating concepts. Furthermore, we have presented the FRGM algorithm which is able to extract knowledge from text-graphs and to apply it to new text-graphs. Our first experimental results suggest that the expressiveness of text- and rule-graphs allows to process effectively even complex linguistic tasks.

At the moment, we have only some preliminary experimental results. Using a generative grammar, in our first experiments we have created a test corpus consisting of 320 example datasets. On this corpus we were able to enumerate the 312378 frequent (given $\mu(G_r) = |G_r| * 2 + 6$) rule-graphs in 56 seconds (on an AMD Athlon XP 2000+). For this test we have limited the number of unlabeled rule-graph vertices to 3 because otherwise the number of frequent rule-graphs grows too large. We are going to perform experiments on large, real-world text datasets as well.

One interesting extension to the FRGM algorithm would be to introduce non-monotonic inference. A disadvantage of the threshold based rule-graph evaluation is that large rule-graphs require high frequencies and therefore many training examples. One way to overcome this problem could be the use of learning patterns. A learning pattern is a subgraph of both a text-graph and a candidate rule-graph which itself is embedded in the text-graph, too. Here, the learning pattern acts as a kind of template for this candidate rule-graph and allows to reduce the required minimum frequency threshold for the rule-graph.

Acknowledgments

I would like to thank Markus Ackermann, Thomas Gärtner, Tamas Horvath and Codrina Lauth, for their valuable comments and support.

References

1. C.L. Forgy. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, pages 17-37, 19, 1982.
2. I. Jonyer, L. Holder, and D. Cook. Concept formation using graph grammars. In *Proceedings of the KDD Workshop on, Multi-Relational Data Mining*, 2002.
3. J. Lyons (1968): Introduction to Theoretical Linguistics. Cambridge [dt. (1995): Einführung in die moderne Linguistik. 8. Aufl., München]
4. U. Rückert, S. Kramer: Generalized Version Space Trees. KDID 2003: 119-129
5. Wikipedia: http://en.wikipedia.org/wiki/Natural_language_processing
6. X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining* (ICDM 2002), pages 721-724, 2002.

Author Index

Borgelt, Christian, 1

de Graaf, Edgar H., 13
de Knijf, Jeroen, 89

Gürel, Tayfun, 25

Jacquemont, Steéphanie, 37
Jacquenet, Francois, 37
Jahn, Katharina, 77

Kersting, Kristian, 25
Kosters, Walter A., 13
Kramer, Stefan, 77

Last, Mark, 51

Markov, Alex, 51
Molzberger, Lukas, 103
Motoda, Hiroshi, 63

Ohara, Kouzou, 63

Rousset, Marie-Christine, 63

Sebag, Michèle, 63
Sebban, Marc, 37

Termier, Alexandre, 63

Washio, Takashi, 63