"Efficient discovery of frequent unordered trees"

Nijssen, Siegfried ; Kok, Joost

Document type : *Communication à un colloque (Conference Paper)*

Efficient Discovery of Frequent Unordered Trees

Siegfried Nijssen and Joost N. Kok

Leiden Institute of Advanced Computer Science,
Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands
snijssen@liacs.nl

Abstract. Recently, an algorithm called FREQT was introduced which enumerates all frequent induced subtrees in an *ordered* data tree. We propose a new algorithm for mining *un*ordered frequent induced subtrees. We show that the complexity of enumerating unordered trees is not higher than the complexity of enumerating ordered trees; a strategy for determining the frequency of unordered trees is introduced.

1 Introduction

Recently, several exploratory data mining algorithms for structural databases have been proposed which search for *frequent* structures in such databases. These frequent pattern mining algorithms use the same principles as APRIORI [2], the well-known frequent item set mining algorithm. A structure may be a sequence, a tree or a graph. In this paper, we focus on frequent tree mining. In general, the frequent tree discovery task is the task of discovering all trees — referred to as the *pattern* trees — that occur frequently in some large tree called a *data tree*. Within this general setup, there are several blanks to be filled in:

- What kind of trees are considered? Is the given database an ordered tree, a node labeled tree or an edge labeled tree?
- What kind of occurrence relation is used? Does a pattern tree occur in a data tree when the pattern tree is an *induced* subtree (which means that parental relations between vertices in the data tree must be the same as in the pattern tree) or when it is an *embedded* subtree (where a parent in a pattern tree may be an ancestor in the data tree)?
- How are tree occurrences counted? Is each occurrence in the data tree counted, or is the data tree partitioned into several separate trees, and is only the occurrence of a tree in a sufficient number of partitions interesting?

Previous publications have dealt with several of these possibilities:

- Wang and Liu [5] developed an algorithm for discovering both ordered and unordered induced edge-labeled subtrees. A pattern tree is frequent when its root can be mapped to the *root* of a sufficient number of partitions in the data tree. To determine the inclusion relation, for each pair of pattern and partition, an $O(nm^{1.5})$ algorithm is used, where n is the size of the pattern tree and m is the size of the partition. The algorithm features a strategy for enumerating unordered trees which requires some redundant trees to be generated.

- Asai et al. [1] developed an algorithm called FREQT for discovering induced subtrees in a node labeled, ordered data tree. The algorithm counts each occurrence in a data tree, but can easily also be used for partition counting. It uses an efficient scheme for generating and counting ordered trees.
- Zaki [6] developed an algorithm for discovering embedded subtrees in a node labeled, ordered data tree. The algorithm counts occurrences in partitions, but can easily also be used to count occurrences separately. It enumerates trees in a similar way as FREQT, but uses a different evaluation technique based on *scope lists*.

The algorithms of Asai et al. [1] and Zaki [6] share their efficient enumeration technique for *ordered* trees; both algorithms define their own evaluation technique for such trees. As also indicated by Wang et al. [5], in some structured databases a child order is of minor importance or even unavailable; for such databases, it is more interesting to search for patterns that do not take the order into account. In this paper, we will propose an algorithm for discovering *unordered* frequent induced subtrees.

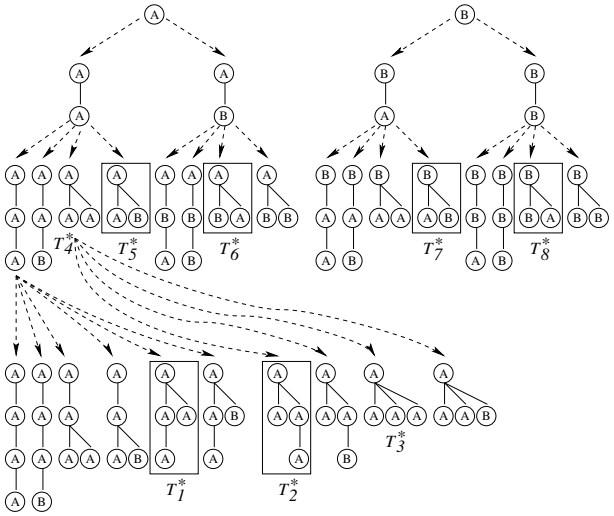
In section 2 we will extend the enumeration technique of [1] and [6] to efficiently enumerate unordered trees. We will define one ordered tree to be the normal form of the unordered trees. As a consequence of the absence of order, a new evaluation technique is required to compute the frequency of trees. In section 3 we will therefore define a new bottom-up algorithm for this task. This algorithm promises to be more efficient than the approach of [5] as it reuses the matchings of previous trees to compute the frequency of new trees.

2 Unordered Tree Enumeration

For an efficient algorithm, it is of major importance that all possible pattern trees are enumerated efficiently. An efficient enumeration technique is a technique that enumerates each unordered tree exactly once. We will first show why techniques from [1] and [6] cannot straightforwardly be applied to unordered trees.

The algorithms of [1] and [6] use rightmost path expansions. Starting with pattern trees with only one node, nodes are added only to the rightmost path to generate new pattern trees. The technique is illustrated in Fig. 1. As can be seen in this figure, several ordered pattern trees may be constructed by this technique which represent the same unordered pattern trees. Ordered trees which represent the same unordered pattern trees are considered to be *equivalent*. In an efficient enumeration technique, no two equivalent pattern trees are constructed. We will propose an efficient technique here; the technique is novel to the best of our knowledge.

In our technique, we define one of the equivalent ordered trees to be the normal form of the corresponding unordered tree. We first introduce some notation. Given a node v in an ordered tree T, firstchild(v) denotes the first child of v, lastchild(v) is the last child, nextsibling(v) is the next node in the ordered child list of v's parent and prevsibling(v) is the previous node in that list. With subtree(v) we denote the subtree in T of which v is the root.



Tree T_i^*	String $l(T_i^*)$
T_1^*	$(0,A)(1,A)(2,A)(1,A)$
T_2^*	$(0,A)(1,A)(1,A)(2,A)$
T_3^*	$(0,A)(1,A)(1,A)(1,A)$
T_4^*	$(0,A)(1,A)(1,A)$
T_5^*	$(0,A)(1,A)(1,B)$
T_6^*	$(0,A)(1,B)(1,A)$
T_7^*	$(0,B)(1,A)(1,B)$
T_8^*	$(0,B)(1,B)(1,A)$

Fig. 1. Enumeration of pattern trees in the alphabet $\{A, B\}$ using rightmost path expansions. Only a selected number of trees is expanded after 2 steps. Dotted lines indicate an expansion. By conceiving pattern trees as nodes and expansions as edges, an enumeration tree [1] is obtained which relates pattern trees to each other. Pattern trees T_1^* and T_2^*, trees T_5^* and T_6^* and trees T_7^* and T_8^* are equivalent. Of a selected number of trees, the pre-order label is given.

Given an ordered tree T, we define the following pre-order string $l(T)$ for this tree: in a depth-first tree traversal, add a tuple label $l(v)$ =(depth(v),label(v)) for each node v to an initially empty string when this node v is visited for the first time. Some examples of this pre-order notation are also given in Fig. 1.

Note that each of these strings corresponds to exactly one tree. In a string, the order of the tuples exactly matches the order of rightmost path expansions; a pre-order string of a tree can therefore also be read as a series of subsequent tree expansions that leads to this tree.

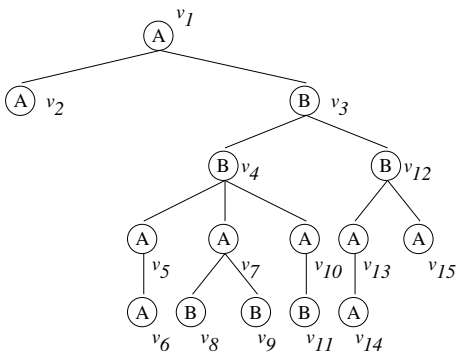
A tree T_1 is called a prefix of a tree T_2 if $l(T_1)$ is a prefix of $l(T_2)$. Tree T_1 is an immediate prefix of T_2 if T_1 is a prefix of T_2 and $|T_1| + 1 = |T_2|$. A suffix is defined analogously.

Given an order on the labels, we define the following order on tuples: $(d_1, l_1) < (d_2, l_2)$ iff $d_1 > d_2$ (this may sound counterintuitive, but will become clear later) or $l_1 < l_2$ if $d_1 = d_2$. Other (in)equalities are derived from this order.

Given two trees T_1 and T_2, we define that $l(T_1) < l(T_2)$ iff:

- either, T_2 is a prefix of T_1,
- or, at the leftmost position i at which $l(T_1)$ and $l(T_2)$ differ, $(d_1, l_1) < (d_2, l_2)$ for the tuples $(d_1, l_1) \in T_1$ and $(d_2, l_2) \in T_2$ occurring at that position.

In the example, $l(T_1^*) < l(T_2^*) < l(T_3^*) < l(T_4^*) < l(T_5^*) < l(T_6^*) < l(T_7^*) < l(T_8^*)$. The enumeration tree of Fig. 1 was obtained by expanding the rightmost path bottom-up; the order of the pattern trees is obtained by performing a post-order walk in the enumeration tree.



v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
$(0, A)$	$(1, A)$	$(1, B)$	$(2, B)$	$(3, A)$	$(4, A)$	$(3, A)$	$(4, B)$	$(4, B)$	$(3, A)$	$(4, B)$	$(2, B)$	$(3, A)$	$(4, A)$	$(3, A)$

Fig. 2. A tree in ordered normal form and its pre-order string. The nodes are numbered in the order of expansion.

We define that a tree T_1 is in (ordered) normal form if no equivalent tree T_2 exists with $l(T_2) < l(T_1)$.

We will now illustrate some observations with respect to this normal form.

Lemma 1. *Given an ordered tree T, T is in normal form if and only if for each node $v \in T$, $l(subtree(v)) \leq l(subtree(v'))$, for each next sibling v' of v.*

Proof. Omitted here; see [3]. □

We will illustrate this lemma on the example of Fig. 2. Under the assumption that this tree is in normal form, we consider two node order changes: an exchange of nodes v_5 and v_7 (exchange 1) and an exchange of nodes v_7 and v_{10} (exchange 2). In the pre-order string, these exchanges correspond to exchanges of subtrings:

$$(0,A)(1,A)(1,B)(2,B)|\underline{(3,A)(4,A)}|\underline{(3,A)(4,B)(4,B)}|(3,A)(4,B)(2,B)(3,A)(4,A)(3,A)$$
$$\Rightarrow \text{Exchange 1} \Rightarrow$$
$$(0,A)(1,A)(1,B)(2,B)|\underline{(3,A)(4,B)(4,B)}|\underline{(3,A)(4,A)}|(3,A)(4,B)(2,B)(3,A)(4,A)(3,A)$$

$$(0,A)(1,A)(1,B)(2,B)(3,A)(4,A)|\underline{(3,A)(4,B)(4,B)}|\underline{(3,A)(4,B)}|(2,B)(3,A)(4,A)(3,A)$$
$$\Rightarrow \text{Exchange 2} \Rightarrow$$
$$(0,A)(1,A)(1,B)(2,B)(3,A)(4,A)|\underline{(3,A)(4,B)}|\underline{(3,A)(4,B)(4,B)}|(2,B)(3,A)(4,A)(3,A)$$

As the tree was in normal form, both exchanges should yield an equivalent tree with a higher pre-order string. Indeed, in case of exchange 1, $(4, B) > (4, A)$ due to $l(\text{subtree}(v_7)) = (3, A)(4, B)(4, B) > (3, A)(4, A) = l(\text{subtree}(v_5))$; in case of exchange 2, $(3, A) > (4, B)$ as $l(\text{subtree}(v_{10}))$ is a prefix of $l(\text{subtree}(v_7))$.

Lemma 2. *Let T be a pattern tree in normal form. Then every prefix of T is also in normal form.*

Proof. This follows from the previous lemma. □

According to this lemma it is sufficient to only generate rightmost path expansions that immediately yield trees in normal form; even then one still enumerates a normal form for each possible tree: the enumeration is *complete*. We will now consider such expansion techniques in more detail. How can valid expansions be characterized?

By a rightmost path expansion only subtrees of some nodes on the rightmost path are modified. Only for these subtrees one has to check again that they are higher than their previous sibling subtree. In the example of Fig. 2, expansions which lead to a tree in normal form, are:

- $(4, l)$, with $l \geq l(v_8) = B$. If $l < B$: subtree(v_{12}) < subtree(v_4), which is not allowed; if $l = B$: subtree(v_{12}) is still a prefix of subtree(v_4), and therefore higher. If $l > B$: subtree(v_{12}) > subtree(v_4). From subtree(v_7) > subtree(v_5) follows that subtree(v_{15}) > subtree(v_{13}) if $l \geq B$. Finally, before expansion, already subtree(v_3) > subtree(v_2), while subtree(v_3) was not a prefix. This shows that all subtrees are still higher than their previous sibling;
- $(3, l)$, with $l \geq l(v_{15}) = A$; obviously, in this case the new node is higher than or equal to subtree(v_{15}), and subtree(v_3) > subtree(v_2). Also subtree(v_{12}) > subtree(v_4), as the new node in subtree(v_{12}) is at a higher level in the tree than the 'next' node v_8 in subtree(v_4);
- $(2, l)$ with $l \geq l(v_{12}) = B$: the new node is higher than the previous sibling subtree(v_{12}); subtree(v_3) was already higher than subtree(v_1);
- $(1, l)$ with $l \geq l(v_3) = B$: the new node is higher than its previous sibling.

The example shows the importance of knowing the largest suffix subtree (subtree(v_{12})) which is a prefix of its previous sibling subtree (subtree(v_4)). This previous sibling restricts the level and the label of new nodes. One can show the following:

Lemma 3. *Given a tree T in normal form, the* lowest prefix node *v is the node on the rightmost path for which the size of subtree(v) is maximized and subtree(v) is a prefix of subtree(prevsibling(v)). A tree may not have a lowest prefix node, in which case the lowest prefix node is undefined. The* next prefix node *(d', l') is the node in l(subtree(prevsibling(v))) immediately after l(subtree(v)), if a lowest prefix node v is defined. An expansion (d, l) yields a tree in normal form iff: $(d, l) \geq p(d)$ and $(d, l) \geq (d', l')$ (if the lowest prefix node is defined). Here $p(d)$ is the label of the node at depth d on the rightmost path of T.*

Proof. Omitted here; see [3]. □

Furthermore, one can also show the following:

Lemma 4. *Given is a tree T in normal form which is normally expanded with a node (d, l). Then the location of the lowest prefix node either:*

- *does not change if (d, l) equals the next prefix node (d', l') (when defined);*
- *or, otherwise, becomes (d, l) if l equals the label of $p(d)$;*
- *or, otherwise, becomes undefined.*

Proof. Omitted here; see [3]. □

Algorithm Enumerate
Input: a tree T in normal form, its representation as a string $l(T)$, and an index t which is either undefined or points to a position in $l(T)$.
Output: a print of each tree that can be obtained by expanding this tree to a new tree in normal form.

```
1. print(T)
2. if t is defined then
3.     Increase t
4.     Let (d, l) be the tuple at position t in l(T).
5.     Enumerate ( T expanded with (d, l), l(T) · (d, l), t ).
6.     for each rightmost expansion (d', l') > (d, l), l' ≥ l(p(d')) do
7.         if l' = l(p(d')) then
8.             Enumerate ( T expanded with (d', l'), l(T) · (d', l'), position of p(d') in l(T) );
9.         else
10.            Enumerate ( T expanded with (d', l'), l(T) · (d', l'), undefined );
11.    Decrease t
12. else
13.    for each rightmost expansion (d', l'), l' ≥ l(p(d')) do
14.        if l' = l(p(d')) then
15.            Enumerate ( T expanded with (d', l'), l(T) · (d', l'), position of p(d') in l(T) );
16.        else
17.            Enumerate ( T expanded with (d', l'), l(T) · (d', l'), undefined );
```

Fig. 3. An algorithm for enumerating all trees in normal form.

If the example tree is expanded with $(4, B)$, v_{12} remains the lowest prefix node. If the example tree is expanded with $(4, C)$, the tree no longer has a lowest prefix node.

All these observations can be used to construct an efficient enumeration algorithm, as given in Fig. 3. With $l(T) \cdot (d, l)$ we denote the concatenation of $l(T)$ and (d, l). Index t points to the next prefix node. By increasing t in line 3., we either obtain the next prefix node (which should be added to the tree to maintain the prefix), or we walk out of the lowest prefix node's sibling subtree. In this latter case, the complete previous tree was copied, and we may continue copying the next tree. In lines 8., 10., 15. and 17., we redefine the value of t as indicated by our observations.

Theorem 1. *Given an alphabet of symbols, Algorithm Enumerate enumerates exactly one ordered normal form for each unordered node-labeled tree that can be constructed using this alphabet.*

Proof. This follows from the lemmas. □

The overhead of this procedure is small. A datastructure is needed which efficiently stores the pre-order string and allows for a quick lookup of the rightmost path in the tree that is represented by the pre-order string. The additional constraints on rightmost path expansions can be checked in constant time. This shows that the problem of enumerating unordered trees is not much more complex than the problem of enumerating ordered trees.

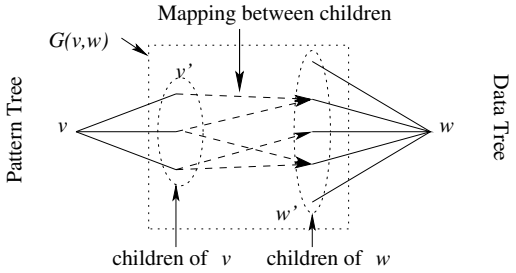


Fig. 4. To make sure children are mapped injectively, a bipartite matching problem has to be solved between the sets of children. If $G(v, w)$ is solvable, we store pointers between the mapping (v, w) and the mappings between the children of v and w.

3 Tree counting

In the previous section we introduced an algorithm which enumerates all unordered pattern trees. In practice, this is impossible as the number of unordered trees is infinite given an alphabet of labels. A frequency criterion is used to stop further expansion of a tree. If a tree occurs less frequently than a certain predefined threshold in a large data tree, it is not expanded further, as every tree that could be constructed subsequently can only be less frequent. As the overhead of the enumeration technique is minimal, the performance of the algorithm largely depends on the speed with which the (in)frequency of a tree is determined.

We will first define the frequency of a pattern tree T. A node v in a pattern tree T can be mapped to a node w in a data tree iff v has the same label as w and there is an injective mapping from the children of v to the children of w. The frequency of a tree T is the number of nodes in the data tree to which the root of T can be mapped. Other frequency criteria built on partitioning can be determined in similar ways.

An easy — but not very efficient way — of determining the frequency is to determine for each pattern tree anew how many times it can be mapped to a data tree. An $O(nm^{1.5})$ algorithm (with n the number of nodes in the pattern tree and m the number of nodes in the data tree) exists for this task, as given in [4]. We take this algorithm as starting point for our counting strategy.

The first step of our algorithm is to determine for each label all locations in the database at which this label occurs. Those labels which fail to meet the predefined frequency criterion, are removed from further consideration.

An important task of the tree mapping algorithm is to determine that there is an *injective* mapping from 'pattern children' to 'database children'. Assume that a node v in a pattern tree and a node w with the same label in a data tree are given, and that each child of v can be mapped to one or more children of w, then the algorithm still has to make sure that an injective mapping can be obtained. The situation is clarified in Fig. 4. The mappings between children of v and children of w constitute a bipartite graph; to determine whether there is an injective mapping is a problem known as the *maximum bipartite matching problem*. The most efficient algorithm for this task has complexity $O(|E|\sqrt{|V|})$, where E is the set of edges in the bipartite graph and V is the set of vertices.

Algorithm Update
Input: a tree T in normal form, its associated mappings, and an expansion (d, l).
Output: an expanded tree T in which the mappings have been updated.

1. Let v be the node at depth $d-1$ on the rightmost path of the pattern tree.
2. Add a node v' at depth d with label l
3. **for all** $m \in \text{Map}(v)$ **do**
4. Let w be the node in the data tree to which m maps
5. $k_1, k_2 :=$ number of children of w (respectively v) with label l
6. **if** $k_1 - k_2 \geq 0$ **then**
7. **for all** children w' of w with label l **do**
8. Add to Map(v') a mapping from v' to w'
9. **else** Remove Mappings(m)

Procedure Remove Mappings
Input: a mapping m from a node v in T to a node w in a data tree
Output: a tree in which m is removed and the mappings of all nodes which are a child of the rightmost path are updated accordingly.

10. Let m' be parent(m), if v is not the root
11. Remove Mappings Below(m)
12. **if** v is not the root of T **then**
13. Let G be the (new) bipartite graph matching problem associated with m'
14. **if** G has no bipartite matching **then** Remove Mappings(m')

Procedure Remove Mappings Below
Input: a mapping m from a node v in T to a node w in a data tree
Output: a tree in which m is removed and the mappings of all nodes which are below v and are child of the rightmost path are updated accordingly.

15. **for all** children v' of v not on the rightmost path of T **do**
16. **for all** $m' \in \text{Map}(v', m)$ **do** Remove m' from Map(v')
17. Let v' be the child of v on the rightmost path of T
18. **for all** $m' \in \text{Map}(v', m)$ **do** Remove Mappings Below (m')
19. Remove m from Map(v)

Fig. 5. An algorithm for updating the datastructure that is associated to a pattern tree T.

With $G(v, w)$ we denote the bipartite graph that is involved in the determination of the injective child mapping of v to w. This graph contains all children of v and w, as well as all the mappings between these children. A node v can be mapped to a node w in the data tree iff there is an associated $G(v, w)$ for which a bipartite matching can be computed that maps each child of v to a different child of w.

The datastructure that is used by our counting algorithm has the following invariant property. Given a pattern tree, with each node v' that is a child of a node v on the rightmost path, we store exactly those mappings that are included in some solvable bipartite graph $G(v, w)$ that belongs to a mapping stored in the parent v; furthermore, we maintain pointers between each mapping of v' and the mapping in the parent to which this mapping is associated. For the root node we store all possible mappings to the data tree.

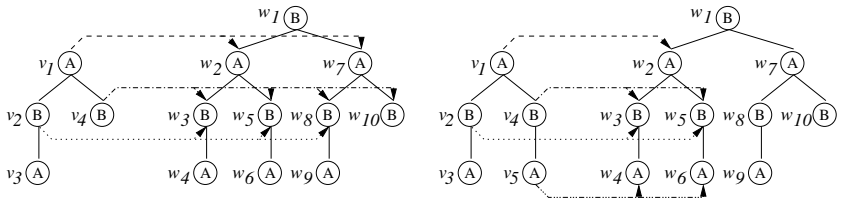


Fig. 6. Example of tree mappings, before and after expansion with $v_5 = (2, A)$.

The algorithm in Fig. 5 describes how the invariant is maintained when a tree T is expanded with a node (d, l). From the set of mappings of the parent, after this update, we can determine the frequency of the expanded tree.

In this algorithm, we use the following notation. With parent(v') we denote the parent of a node v' in a tree; Map(v) denotes all mappings stored for a given node v in the pattern tree. Given a node v' and a mapping $m \in$ Map(parent(v')), with Map(v',m) we denote those mappings in Map(v') which have a pointer to mapping m. If m is a mapping from v' to w', with parent(m) we denote the mapping from the parent of v' to the parent of w'.

We will briefly discuss some elements of the algorithm and illustrate these using the example of Fig. 6. In line 5.-6. we use the observation that a new node v' (in the example, v_5) can be mapped to every node which has the same label l (in the example, w_4, w_6 and w_9). In general, given a mapping $(v \to w) \in$ Map(parent(v')), the siblings of the new node could be mapped injectively to the children of w in the old situation. If the number of children of w with label l is larger than the number of children of v with label l, the added node can always be mapped to one of those additional nodes to solve the bipartite matching $G(v, w)$. If the number of children in the data tree is insufficient, such as in the example for $v_4 \to w_{10}$, the bipartite matching problem $G(v, w)$ can no longer be solved and the mapping associated to that matching problem must be removed. This mapping is in its turn part of some other matching problem of its parent node (in the example, $G(v_1, w_7)$). The parent's bipartite matching may no longer be solvable either. Therefore, it is necessary to recursively check that the matching problem for that ancestor node can still be solved (line 9.).

If some bipartite matching problem can no longer be solved, the corresponding mapping is removed. This mapping may have pointers to some mappings in the children ($v_1 \to w_7$ has pointers to $v_2 \to w_8$, $v_4 \to w_8$ and $v_4 \to w_{10}$); according to the definition of the invariant, these child mappings should also be removed, which is done by the Remove Mapping Below procedure.

The advantage of the invariant is that the number of (active) child mappings is kept very small. Still all those mappings are updated which are later required to determine the frequency of expanded trees. For this reason the procedure is also restricted to the rightmost path; the bipartite matching of other subtrees is not required to recompute all matchings after rightmost path expansion.

We propose to compute the frequent trees by traversing the enumeration tree in a depth-first fashion. The disadvantage of such a strategy is that it is difficult to apply some pruning strategies that are frequently used in Apriori-like

algorithms. The advantage is that the memory demand is much smaller; it is sufficient to store the mappings of the current tree only, together with information for undoing the removal of mappings when the enumeration backtracks over an expansion. For the latter purpose, with every expansion, we store a list of all the mappings that have been removed by that expansion. As every mapping can only be removed once, one can easily see that once a mapping is added to the pattern tree in line 8., information about this mapping is not removed from memory before the enumeration backtracks over the expansion to which this mapping belongs. The memory requirement of the algorithm is therefore still quite large: if n is the number of nodes in the largest frequent pattern tree, and m is the length of longest mapping list, the memory demand is of order $O(nm)$. In practice, of course, the memory footprint is much lower as large trees are not very frequent in most databases.

4 Conclusions and future research

In this paper we introduced an algorithm for mining frequent unordered induced subtrees. It extends the enumeration technique that was introduced in [1] and [6] for ordered subtrees. We showed that the enumeration of unordered subtrees is not much more difficult than the enumeration of ordered trees. The evaluation of unordered trees turns out to be more complex, from space as well as time complexity point of view. We propose to reduce the memory demand by a depth-first enumeration strategy, but this turns some pruning strategies difficult.

We envision many directions for future research. First of all, the validity of our approach should be verified experimentally. One can imagine several small optimizations to our algorithm which have not been discussed here; for example, one could also compute all solutions to a bipartite matching problem in stead of only computing one. Furthermore, we are also interested in ways to efficiently combine unordered trees with ordered trees to obtain a similar algorithm as the algorithm of Wang and Liu [5]; we are also interested in an algorithm for mining frequent unordered embedded subtrees.

References

1. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient Substructure Discovery from Large Semi-structured Data. In: Proceedings of the 2nd Annual SIAM Symposium on Data Mining. (2002).
2. Agrawal, R., Manilla, H., Srikant, R., Toivonen, H., Verkamo, A.: Fast Discovery of Association Rules. In: U.M. Fayyad et al. (eds). Advances in Knowledge Discovery and Datamining. AAAI/MIT Press. (1996).
3. Nijssen, S., Kok, J.N.: Efficient Discovery of Frequent Unordered Trees: Proofs. Technical Report 2003-01, Leiden Institute of Advanced Computer Science. (2003).
4. Reyner, S.W.: An analysis of a good algorithm for the subtree problem. SIAM Journal of Computing, Vol. 6, No. 4. (1977).
5. Wang, K., Liu, H.: Discovering Structural Association of Semistructured Data. (1999).
6. Zaki, J.: Efficiently Mining Frequent Trees in a Forest. In: Proceedings of the SIGKDD'02, Edmonton, Canada (2002).