Appeared in Proc. Second SIAM International Conference on Data Mining 2002 (SDM'02), 158-174, SIAM, 2002

Efficient Substructure Discovery from Large Semi-structured Data

Tatsuya Asai[*], *Kenji Abe*[*], *Shinji Kawasoe*[*], *Hiroki Arimura*[*,†], *Hiroshi Sakamoto*[*], *and Setsuo Arikawa*[*]

1 Introduction

By rapid progress of network and storage technologies, a huge amount of electronic data such as Web pages and XML data [23] has been available on intra and internet. These electronic data are heterogeneous collection of ill-structured data that have no rigid structures, and often called *semi-structured data* [1]. Hence, there have been increasing demands for automatic methods for extracting useful information, particularly, for discovering rules or patterns from large collections of semi-structured data, namely, *semi-structured data mining* [7, 11, 18, 19, 21, 25].

In this paper, we model such semi-structured data and patterns by *labeled ordered trees*, and study the problem of discovering all frequent tree-like patterns that have at least a *minsup* support in a given collection of semi-structured data. We present an efficient pattern mining algorithm FREQT for discovering all frequent tree patterns from a large collection of labeled ordered trees.

Previous algorithms for finding tree-like patterns basically adopted a straightforward generate-and-test strategy [19, 24]. In contrast, our algorithm FREQT is an incremental algorithm that simultaneously constructs the set of frequent patterns and their occurrences level by level. For the purpose, we devise an efficient enumeration technique for ordered trees by generalizing the itemset enumeration tree by Bayardo [10].

The key of our method is the notion of the *rightmost expansion*, a technique to grow a tree by attaching new nodes only on the rightmost branch of the tree. Furthermore, we show that it is sufficient to maintain only the occurrences of the

[*]Department of Infomatics, Kyushu University, Japan, e-mail: t-asai@i.kyushu-u.ac.jp
[†]PRESTO, JST, Japan, e-mail: arim@i.kyushu-u.ac.jp

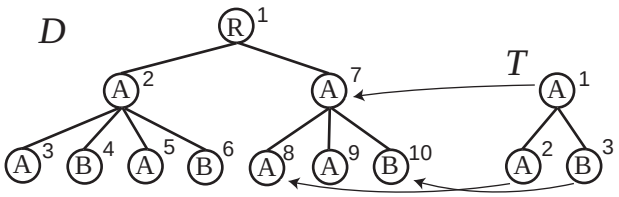


Figure 1. *A data tree D and a pattern tree T on the set $\mathcal{L} = \{A, B\}$ of labels*

rightmost leaf to efficiently implement incremental computation of frequency.

Combining the above techniques, we show that our algorithm scales almost linearly in the total size of maximal tree patterns contained in an input collection slightly depending on the size of the longest pattern. We also developed a pruning technique that speeds-up the search. Experiments on real-world datasets show that our algorithm runs efficiently on real-life datasets in a wide range of parameters.

1.1 Related Works

Only a small number of researches have been done on data mining from semi-structured data [11, 18, 19, 24, 25, 26] unlike studies on semi-structured databases [3, 23]. Most related researches would be [14, 19, 25, 26] that study the frequent tree discovery. Wang and Liu [25] considered mining of collections of paths in ordered trees with Apriori-style technique. Miyahara *et al.* [19] developed a straightforward generate-test algorithm for discovering ordered trees in a similar setting as ours.

Dehaspe *et al.* [11] developed a mining algorithm WARMR for the first-order models and graph structures. Since a tree is a special case of graph structures, we can apply WARMR to frequent tree discovery. A difference to our setting is that the trees are unordered and a matching is not necessarily one-to-one in this case. Also, since their enumeration technique is more powerful than ours, it has a drawback of generating dupulicated patterns. [14, 17, 18, 24] study graph mining.

Among the studies on association rule discovery, most closely related ones are [10, 13, 22]. In discovery of *long* itemsets, Bayardo [10] proposed an efficient enumeration technique based on *set-enumeration tree* for generating itemsets without repetition to overcome the inefficiency of itemset lattice-based enumeration of Apriori [4, 5]. Sese and Morishita [22] combined a merge-based counting technique and the set-enumeration tree to cope with discovery with low frequency thresholds. To cope with the problems of long patterns and the low frequency common in, say, Web mining, we extend the above techniques for mining ordered trees.

Very recently, Zaki [26] independently proposed efficient algorithms for the frequent pattern discovery problem for ordered trees. He adopted an efficient enumeration technique essentially same to our rightmost expansion using a special string representation of ordered trees. Combining this enumeration technique with depth-first search and the vertical decomposition of trees, he presented an efficient algorithm TREEMINERV that achieves linear scaleup with data size.

1.2 Organization

The rest of this paper is organized as follows. In Section 2, we prepare basic notions and definitions. In Section 3, we present our algorithm for solving the frequent pattern discovery problem for labeled ordered trees using the techniques of rightmost expansion and incremental occurrence update. In Section 4, we run experiments on real datasets to evaluate the proposed mining algorithm. In Section 5, we conclude this paper.

2 Preliminaries

In this section, we introduce basic notions and definitions on semi-structured data and our data mining problems according to [1, 2, 6, 20].

2.1 Labeled Ordered Trees

As the models of semi-structured databases and patterns such as XML [23] and OEM model [3], we adopt the class of labeled ordered trees defined as follows. For a set A, $\#A$ denotes the cardinality of A. Let $\mathcal{L} = \{\ell, \ell_0, \ell_1, \ldots\}$ be a finite alphabet of labels, which correspond to attributes in semi-structured data or tags in tagged texts.

A *labeled ordered tree on* $\mathcal{L}$ (an ordered tree, for short) is a 6-tuple $T = (V, E, \mathcal{L}, L, v_0, \preceq)$ satisfying the following properties. $G = (V, E, v_0)$ is a tree with the root $v_0 \in V$. If $(u, v) \in E$ then we say that u is a *parent* of v or that v is a *child* of u. The mapping $L : V \to \mathcal{L}$, called a *labeling function*, assigns a label $L(v)$ to each node $v \in V$. The binary relation $\preceq \subseteq V^2$ represents a *sibling relation* for the ordered tree T such that u and v are children of the same parent and $u \preceq v$ iff u is an elder brother of v. In the above definition, an ordered tree is *not ranked*, that is, a node can have arbitrary many children regardless its label. In what follows, given an ordered tree $T = (V, E, \mathcal{L}, L, v_0, \preceq)$, we refer to V, E, L, and $\preceq$, respectively, as V_T, E_T, L_T, and $\preceq_T$ if it is clear from context.

Let T be a labeled ordered tree. The *size* of T is defined by the number of its nodes $|T| = \#V_T$. The *length* of a path of T is defined by the number of its nodes. If there is a path from a node u to a node v then we say that u is an *ancestor* of v or v is a *descendant* of u. For every $p \geq 0$ and a node v, the *p-th parent* of v, denoted by $\pi_T^p(v)$, is the unique ancestor u of v such that the length of the path from u to v has length exactly $p + 1$. By definition, $\pi_T^0(v)$ is v itself and $\pi_T^1(v)$ is the parent of v. The *depth* of a node v of T, denoted by $depth(v)$, is defined by the length d of the path $x_0 = v_0, x_1, \ldots, x_{d-1} = v$ from the root v_0 of T to the node v. The *depth* of T is the length of the longest path from the root to some leaf in T.

In Fig. 1, we show examples of labeled ordered trees, say D and T, on the alphabet $\mathcal{L} = \{A, B\}$, where a circle with the number, say v, at its upper right corner indicates the node v, and the symbol appearing in a circle indicates its label $L(v)$. We also see that the nodes of these trees are numbered consecutively by the preorder.

We introduce the canonical representation of labeled ordered trees as follows.

A labeled ordered tree T of size $k \geq 1$ is said to be *of normal form* if the set of the nodes of T is $V_T = \{1, \ldots, k\}$ and all elements in V_T are numbered by the preorder traversal [6] of T. For instance, trees D and T in the previous example of Fig. 1 are of normal form. The following lemma is useful.

Lemma 1. *Let T be any labeled ordered tree with k nodes. If T is of normal form then the root is $v_0 = 1$ and the rightmost leaf is $v_{k-1} = k$ in T.*

2.2 Matching of trees

Let T and D be ordered trees on an alphabet $\mathcal{L}$, which are called the *pattern tree* and the *data tree* (or the text tree), respectively. Furthermore, we assume that T is an order tree of normal form with $k \geq 0$ nodes. We call such a tree T as a k-pattern tree on $\mathcal{L}$ (or k-pattern, for short), and denote by $\mathcal{T}_k$ the sets of all k-patterns on $\mathcal{L}$. Then, we define the set of patterns on $\mathcal{L}$ by $\mathcal{T} = \bigcup_{k \geq 0} \mathcal{T}_k$.

First, we define the notion of matching functions as follows. A one-to-one function $\varphi : V_T \to V_D$ from nodes of P to nodes of D is called a *matching function of T into D* if it satisfies the following conditions for any $v, v_1, v_2 \in V_T$:

- φ preserves the parent relation, i.e., $(v_1, v_2) \in E_T$ iff $(\varphi(v_1), \varphi(v_2)) \in E_D$.
- φ preserves the sibling relation, i.e., $v_1 \preceq_T v_2$ iff $\varphi(v_1) \preceq_D \varphi(v_2)$.
- φ preserves the labels, i.e., $L_T(v) = L_D(\varphi(v))$.

A pattern tree T *matches* a data tree D, or T *occurs in* D, if there exists some matching function φ of T into D. Then, the *total occurrence of T in D w.r.t. φ* is the the list $Total(\varphi) = (\varphi(1), \ldots, \varphi(k)) \in (V_D)^k$ of the nodes that the nodes of T map, and the *root occurrence of T in D w.r.t. φ* is the node $Root(\varphi) = \varphi(1) \in V_D$ of D that the root of T maps, where $k = |T|$. Note that $Total(\varphi)$ is another representation of the mapping φ itself assuming that T is of normal form. For a pattern T, we define $Occ(T) = \{Root(\varphi) \mid \varphi \text{ is a matching function of } T \text{ into } D\}$, that is, the set of the root-occurrences of T in D [1]. Then, the *frequency* (or *support*) of the pattern T in D, denoted by $freq_D(T)$, is defined by the fraction of the number of the distinct root occurrences to the total number of nodes in D, that is, $freq_D(T) = \#Occ(T)/|D|$. For a positive number $0 < \sigma \leq 1$, a pattern T is *σ-frequent* in D if $freq_D(T) \geq \sigma$.

For example, consider the previous example in Fig. 1. In this figure, a matching function, say φ_1, of the pattern T with three nodes into the data tree D with ten nodes is indicated by a set of arrows from the nodes of P. The total- and the root-occurrences corresponding to φ_1 are $Total(\varphi) = (7, 8, 10)$ and $Root(\varphi) = 7$, respectively. Furthermore, there are two root-occurrences of T in D, namely 2 and 7, while there are five total occurrences of T in D (or distinct matching functions of T into D), namely $(2, 3, 4), (2, 3, 6), (2, 5, 6), (7, 8, 10)$ and $(7, 9, 10)$. Hence, the support of T in D is $freq_D(T) = \#Occ(T)/|D| = 2/10$.

[1] Although $Total(\varphi)$ is a natural candidate for the notion of occurrences for a pattern T in D, it has a drawback that it is diffcult to define $freq_D(T)$ as a number in the interval $[0, 1]$.

Algorithm FREQT

Input: A set $\mathcal{L}$ of labels, a data tree D on $\mathcal{L}$, and a *minimum support* $0 < \sigma \leq 1$.

Output: The set $\mathcal{F}$ of all σ-frequent patterns in D.

1. Compute the set $\mathcal{C}_1 := \mathcal{F}_1$ of σ-frequent 1-patterns and the set RMO_1 of their rightmost occurrences by scanning D; Set $k := 2$;
2. While $\mathcal{F}_{k-1} \neq \emptyset$, do:
   2 (a) $\langle \mathcal{C}_k, RMO_k \rangle :=$ Expand-Trees$(\mathcal{F}_{k-1}, RMO_{k-1})$; Set $\mathcal{F}_k := \emptyset$.
   2 (b) For each pattern $T \in \mathcal{C}_k$, do the followings: Compute $freq_D(T)$ from $RMO_k(T)$, and then, if $freq_D(T) \geq \sigma$, then $\mathcal{F}_k := \mathcal{F}_k \cup \{T\}$.
3. Return $\mathcal{F} = \mathcal{F}_1 \cup \cdots \cup \mathcal{F}_{k-1}$.

Figure 2. *An algorithm for discovering all frequent ordered tree patterns*

2.3 Problem Statement

Now, we state our data mining problem, called the frequent pattern discovery problem, which is a generalization of the frequent itemset discovery problem in association rule mining [4], as follows.

> **Frequent Pattern Discovery Problem**
> Given a set of labels $\mathcal{L}$, a data tree D on $\mathcal{L}$, and a positive number $0 < \sigma \leq 1$, called the *minimum support* (or *minsup*, for short), find all σ-frequent ordered trees $T \in \mathcal{T}$ such that $freq_D(T) \geq \sigma$.

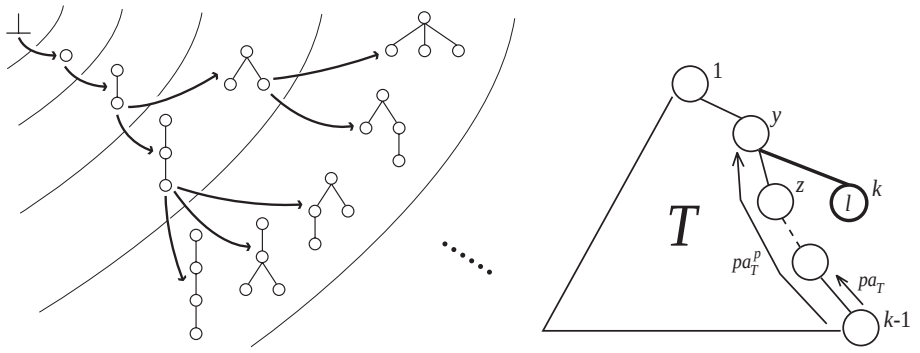
The frequent pattern discovery problem seems too simple to apply real world problems. However, an efficient solusion for this problem can be a base of algorithms for more complicated data mining problems such as the *frequent pattern discovery problem with document count* [25], where the input is collections of trees, and the *optimal pattern discovery problem* [8, 12, 20, 22], whose goal is to find the patterns that optimize a given statistic measure such as the information entropy on a data set.

Throughout this paper, we assume the standard *leftmost-child and right-sibling representation* for ordered trees (e.g., [6]), where a node is represented by a pair of pointers to its first child, *child*(), and the next sibling, *next*(), as well as its node label and the parent pointer, *parent*().

3 Mining Algorithms

In this section, we present an efficient algorithm for solving the frequent pattern discovery problem for ordered trees that scales almost linearly in the total size of the maximal frequent patterns.

In Fig. 2, we present our algorithm FREQT for discovering all frequent ordered tree patterns with the frequency at least a given minimum support $0 < \sigma \leq 1$ in a



(a) A search graph for ordered trees (b) The (p, ℓ)-expansion of tree T

Figure 3. *The rightmost expansion for ordered trees*

data tree D. As the basic design of the algorithm, we adopted the levelwise search strategy as in [4] and the search space similar to the enumeration tree of [10].

In the first pass, FREQT simply creates the set $\mathcal{F}_1$ of all 1-patterns and stores their occurrences in RMO_1 by traversing the data tree D. In the subsequent pass $k \geq 2$, FREQT incrementally computes a set $\mathcal{C}_k$ of all *candidate* k-patterns and the set RMO_k of the rightmost occurrence lists for the trees in $\mathcal{C}_k$ simultaneously from the sets $\mathcal{F}_{k-1}$ and RMO_{k-1} computed in the last stage by using the rightmost expansion and the rightmost occurrence technique using the sub-procedure Expand-Trees. Repeating this process until no more frequent patterns are generated, the algorithm computes all σ-frequent patterns in D. In the rest of this section, we will describe the details of the algorith.

3.1 Efficient Enumeration of Ordered Trees

In this subsection, we present an enumeration technique for generating all ordered trees of normal form without duplicates by incrementally expanding them from smaller to larger. This algorithm is a generalization of the itemset enumeration technique of [10], called the *set-enumeration tree*.

Rightmost expansion. A basic idea of our enumeration algorithm will be illustrated in Fig. 3(a). In the search, starting with the set of trees consisting of single nodes, for every $k \geq 2$ the enumeration algorithm expands a given ordered tree of size $k-1$ by attaching a new node only with a node on the rightmost branch of the tree to yield a larger tree of size k.

Let $k \geq 0$ be an integer and $\mathcal{L}$ be an alphabet. Let T be any $(k-1)$-pattern over $\mathcal{L}$ and $rml(T)$ be the rightmost leaf of T. Since T is of normal form by definition, $rml(T) = k$. Then, the *rightmost branch* of S is the unique path starting from the root to $rml(T)$.

A rightmost expansion of T is any k-pattern obtained from T by expanding a node on the rightmost branch of T by attaching a new rightmost child to v. More

precisely, it is defined as follows. Let $p \geq 0$ be any nonnegative integer no more than the depth of $rml(T)$ and $\ell \in \mathcal{L}$ be any label. Then, the (p, ℓ)-*expansion* of T is the labeled ordered tree S of normal form obtained from T by attaching a new node, namely k, to the node $y = \pi_T{}^p(x)$ as the rightmost child of y. The label of k is ℓ (See Fig. 3(b)). A *rightmost expansion* of an ordered tree T is the (p, ℓ)-expansion of T for some integer $p \geq 0$ and some label $\ell \in \mathcal{L}$. Then, we say that either T is the *predecessor* of S or S is a *successor* of T. A pattern T is *maximal w.r.t. rightmost expansion* (or *maximal*, for short) if T has no successor.

Lemma 2. *For every $k \geq 2$, if S is a $(k-1)$-pattern then any rightmost expansion T of S is also a k-pattern. Furthermore, if T is a k-pattern then there exists the unique $(k-1)$-pattern S such that T is a rightmost expansion of S.*

Proof. If T is the rightmost expansion of S, then trivially $|T| = k$. Since the new node $v_k = k$ is attached to a node on the rightmost branch of S, it should be the last node in the preorder traversal of T. This shows that T is an ordered tree of normal form with k nodes. On the other hand, suppose that T is obtained from some $(k-1)$-pattern S by attaching the node k as a rightmost leaf. Then, we see that removing the attached leaf k from T is the only way to build any predecessor of T. Since this choice of k as the rml of T is unique, the predecessor of T is also unique. □

The *enumeration graph* for $\mathcal{T}$ is a directed acyclic graph G, where each node is an ordered tree in $\mathcal{T} \cup \{\perp\}$, and a node has an edge to another node T iff T is a successor of S. Every single node trees in $\mathcal{T}_1$ is assumed to be a successor of the empty tree $\perp$ of size 0. From the above lemma, the enumeration graph for $\mathcal{T}$ is acutually a tree with root $\perp$. Hence, we can enumerate all trees in $\mathcal{T}$ by traversing in either the breadth-first or the depth-first manner.

3.2 Updating Occurrence Lists

The key of our algorithm is how to efficiently store the information of a matching φ of each pattern T into the data tree D. Instead of recording the full information $\langle \varphi(1), \ldots, \varphi(k) \rangle$ of φ, our algorithm maintains only the partial information on φ called the rightmost occurrences defined as follows.

Rightmost occurrences. Let $k \geq 0$ be any integer. Let T be any k-pattern and $\varphi : V_T \to V_D$ be any matching function of T into D. Then, the *rightmost occurrence of T in D w.r.t. φ* is the node $Rmo(\varphi) = \varphi(k)$ of D that the rightmost leaf k of T maps. For every T, we define $RMO(T) = \{Rmo(\varphi) \,|\, \varphi$ is a matching function of T into $D\}$, the set of the rightmost occurrences of T in D. For example, consider the data tree D in Fig. 1. Then, the pattern tree T has three rightmost occurrences $4, 6$ and 10 in D. The root-occurrences 2 and 7 of T can be easily computed by taking the parents of $4, 6$ and 10 in D.

Now, we will give an inductive characterization of the rightmost occurrences. Let x be any node in D. For every positive integer $p \geq 0$, we define the *p-th head* of x, denoted by $head^{(p)}(x)$, as follows: If $p = 0$ then $head^{(p)}(x)$ is a child of x;

Algorithm Update-RMO(RMO, p, ℓ)

1. Set $RMO_{\rm new}$ to be the empty list ε and $check := null$.
2. For each element $x \in RMO$, do:
   (a) If $p = 0$, let y be the leftmost child of x.
   (b) Otherwise, $p \geq 1$. Then, do:
      - If $check = {\pi_D}^p(x)$ then skip x and go to the beginning of Step 2 (Duplicate-Detection).
      - Else, let y be the next sibling of ${\pi_D}^{p-1}(x)$ (the $(p-1)$st parent of x in D) and set $check := {\pi_D}^p(x)$.
   (c) While $y \neq null$, do the following:
      - If $L(y) = \ell$, then $RMO_{\rm new} := RMO_{\rm new} \cdot (y)$; /* Append */
      - $y := next(y)$; /* the next sibling */
3. Return $RMO_{\rm new}$.

Figure 4. *The incremental algorithm for updating the rightmost occurrence list of the (p, ℓ)-expansion of a given pattern T from that of T*

Otherwise, $p > 0$, and $head^{(p)}(x)$ is the next sibling of the $(p-1)$st parent h of x, i.e., the leftmost (oldest) node y such that ${\pi_D}^{p-1}(x) \prec y$.

Lemma 4. *Let S be a $(k-1)$-pattern occuring in a data tree D and $\varphi : V_S \to V_D$ be a matching function of S into D. Let T be a (p, ℓ)-expansion of S and $\psi : V_T \to V_D$ be any extension of φ, i.e., $\psi(i) = \varphi(i)$ holds for every $i = 1, \ldots, k-1$. Then, ψ is a matching function of T into D iff ψ satisfies the following (1) and (2):*

(1) *$\psi(k)$ is either the p-th head of $\varphi(k-1)$ or one of its right (younger) siblings, i.e., $head^{(p)}(\varphi(k-1)) \preceq \psi(k)$.*
(2) *$L_D(\psi(k)) = \ell$.*

For every $p \geq 0$ and $\ell \in \mathcal{L}$, we define the binary relation $HEAD^{(p,\ell)} = \{ (x, y) \,|\, x, y \in V_D, head^{(p)}(x) \preceq y,\ L(y) = \ell \} \subseteq V_D \times V_D$. Let U be any set. For a set $A \subseteq U$ and a binary relation $R \subseteq U \times U$, we define the set of images $R(A) = \{ b \,|\, a \in A, (a, b) \in R \}$. The following lemma follows from Lemma 4.

Lemma 5. *Let $p \geq 0$ and $\ell \in \mathcal{L}$. If T is the (p, ℓ)-expansion of a pattern S, then $RMO(T) = HEAD^{(p,\ell)}(RMO(S))$ holds.*

Algorithm Update-RMO. Following Lemma 5, the algorithm Update-RMO of Fig. 4 exactly generates the elements in $RMO(T)$ for the (p, ℓ)-expansion T of a pattern S without duplicates from the rightmost occurrence list $RMO(S)$. Since the algorithm computes the elements of $HEAD^{(p,\ell)}(RMO(S))$, the correctness of the algorithm is immediate from Lemma 4.

However, the straightforward implementation of Lemma 5 often scans the same nodes more than once if $p \geq 1$ and then the computed list of the elements in $RMO(T)$ may contain some duplicates. To avoid this problem, we introduce the **Duplicate-Detection** technique as follows. When $p \geq 1$ holds, the algorithm checks if the value of *check* is equivalent to the p-th parent $\pi_D^p(x)$ for each $x \in RMO(T)$ before scanning its siblings. If the answer is yes then it skips the current element $x \in RMO(T)$ and goes to the next element in $RMO(T)$. Otherwise the algorithm update the variable *check* with $\pi_D^p(v)$. Later in the experiments of Section 4, we will see that the Duplicate-Detection technique greatly improves the efficiency of the algorithm FREQT.

Now, we show the correctness of the algorithm Update-RMO. Let A be a set, $s = a_1 \cdots a_n \in A^*$ be a sequence over A, and $\leq_A$ be a binary relation on A. Let $1 \leq i, j \leq n$. Then, s is said to be *monotonic (noncrossing) w.r.t.* $\leq_A$ if $a_i <_A a_j$ ($a_i \leq_A a_j$) holds for any pair $i < j$, For a sequence $s = a_1 \cdots a_n \in A^*$ and a function $\xi : A \to A^*$, we assume that $\xi(s) = \xi(a_1) \cdots \xi(a_n)$. In what follows, $RMO(T)$ means the *list* of the rightmost occurrences of T but not the *set*.

Lemma 6. *The algorithm* Update-RMO *(with the Duplicate-Detection technique) enumerates all the nodes of any rightmost occurrence list without repetition, if the following two conditions are satisfied: (i) all the elements of $RMO(T)$ are ordered in the preorder of D for any 1-pattern T, and (ii) the algorithm scans all the nodes of $RMO(T)$ in the order of $RMO(T)$.*

Proof. We prove the lemma by showing that a sequence $RMO(T)$ of nodes of D preserves a monotonicity w.r.t. a particular relation on V_D, namely a *k-th preorder relation*, at any stage k. First, the *k-th preorder sequence* of D is a sequence Π_k of nodes of D, recursively defined as follows: (1) Π_1 is the sequence of all the nodes in D ordered in the preorder of D, and (2) $\Pi_k = C_D(\Pi_{k-1})$, where $C_D : V_D \to (V_D)^*$ is the function that returns for an input $v \in V_D$ the sequence of all the children of v whose node occurs at once in the order of the sibling relation $\preceq_D$. Then, a k-th preorder relation $\leq_k$ is defined as follows: $v_1 \leq_k v_2$ iff v_1 occurs in front of v_2 or equals to v_2 in the k-th preorder sequence Π_k, for any $v_1, v_2 \in V_D$. From this definition, the assertion that a sequence s of nodes of D is monotonic w.r.t. $\leq_k$ is equivalent to the assertion that s is a subsequence of Π_k. For the relation $\leq_k$, it is easy to prove that $v_1 \leq_k v_2$ implies ${\pi_D}^p(v_1) \leq_{k-p} {\pi_D}^p(v_2)$. Suppose that $RMO(S)$ is monotonic w.r.t. the d-th preorder relation $\leq_d$, where S is any pattern and $d = \mathit{depth}(\mathit{rml}(S))$, then the sequence ${\pi_D}^p(RMO(S))$ is noncrossing w.r.t. $\leq_{d-p}$ because of the above statement. Moreover, the algorithm Update-RMO skips manipulation about the duplicate occurrences in ${\pi_D}^p(RMO(S))$ using the Duplicate-Detection technique. Therefore, the output $RMO(T)$ of the algorithm for the input $\langle RMO(T), p, \ell \rangle$ is monotonic w.r.t. $\leq_{d-p+1}$ where T is the (p,l)-expansion of S, and $d - p + 1 = \mathit{depth}(\mathit{rml}(T))$ holds at this time. Now we have shown that $RMO(T)$ computed by the algorithm is monotonic w.r.t. $\leq_d$ for any pattern T where $d = \mathit{depth}(\mathit{rml}(T))$, and also the lemma. □

Algorithm Expand-Trees($\mathcal{F}, RMO$)

1. $\mathcal{C} := \emptyset$; $RMO_{\rm new} := \emptyset$;
2. For each tree $S \in \mathcal{F}$, do:
   - For each $(p, \ell) \in \{1, \ldots, d\} \times \mathcal{L}$, do the followings, where d is the depth of the rightmost leaf of S:
     - Compute the (p, ℓ)-expansion T of S;
     - $RMO_{\rm new}(T) :=$ Update-RMO($RMO(S), p, \ell$);
     - $\mathcal{C} = \mathcal{C} \cup \{T\}$;
3. Return $\langle \mathcal{C}, RMO_{\rm new} \rangle$;

Figure 5. *The algorithm for computing the set of rightmost expansions and their rightmost occurrence lists.*

3.3 Analysis of the Algorithm

We go back to the computation of the candidate set $\mathcal{C}_k$. In Fig. 5, we present the algorithm Expand-Trees that computes the set $\mathcal{C}$ and the corresponding set RMO_k of the rightmost occurrence lists. The set RMO_k is implemented by a hash table such that for each tree $T \in \mathcal{C}$, $RMO_k(T)$ is the list of the rightmost occurrences of T in D. We show the correctness of the algorithm FREQT of Fig. 2.

Theorem 7. *Let $\mathcal{L}$ be a label set, D be a data tree on $\mathcal{L}$, and $0 < \sigma \leq 1$ be a minimum support. The algorithm* FREQT *correctly computes all σ-frequent patterns in $\mathcal{T}$ without duplicates.*

The running time of the algorithm is bounded by $O(k^2 bL\,N)$, where k is the maximum size of the frequent patterns, b is the maximum branching factor of D, $L = \#\mathcal{L}$, and N is the sum of the lengths of the rightmost occurrences lists of frequent patterns. Furthermore, FREQT generates at most $O(kLM)$ patterns during the computation, where M is the sum of the sizes of the maximal σ-frequent patterns, while a straightforward extension of Apriori [4] to tree patterns may generate exponentially many patterns in k.

3.4 Pruning by Node-Skip and Edge-Skip

In this subsection, we describe some improvements for FREQT.

Node-Skip. This pruning technique skips useless nodes with infrequent labels using the information of $\mathcal{F}_1$. Suppose that a 1-pattern with label $\ell \in \mathcal{L}$ is not σ-frequent in D. Then obviously, the label ℓ does not appear in any σ-frequent pattern $\mathcal{F}$. Hence, we can skip the call of Update-RMO(RMO, p, ℓ) when ℓ is infrequent label.

Edge-Skip. This pruning technique removes useless edges with infrequent pairs of labels by using the information of $\mathcal{F}_2$. Similarly to Node-Skip above, we

observe that if (ℓ_1, ℓ_2) is a pair of the labels of infrequent 2-pattern $E \notin \mathcal{F}_2$, then the pair (ℓ_1, ℓ_2) does not appear in any σ-frequent pattern in $\mathcal{F}$. Hence, we can skip the call of Update-RMO(RMO, p, ℓ_2) when ℓ_1 is the label of the nodes in RMO and the pair (ℓ_2, ℓ_2) is infrequent.

3.5 An Example

Consider the data tree D in Fig. 1 of size $|D| = 10$ and assume that the minimum support is $\sigma = 0.2$ and $\mathcal{L} = \{A, B\}$. This value of σ implies that a σ-frequent pattern requires two nodes as the number of the minimum root occurrences. In Fig. 6, we show the patterns generated by FREQT during the computation.

First, the algorithm computes the set $\mathcal{F}_1$ of the frequent 1-patterns T_1 and T_2 in stage 1 by traversing the data tree D and records their rightmost occurrences in RMO_1. In stage 2, calling Expand-Trees with $\mathcal{F}_1$ and RMO_1 gives the candidate set $\mathcal{C}_2$ and the set of their rml-occurrence lists RMO_2. In Fig. 6, we see that $\mathcal{C}_2 = \{T_{11}, T_{12}, T_{21}, T_{22}\}$, and that they are obrained from their predecessor T_1 or T_2 by attaching a new leaf with label A or label B. The rightmost occurrence list of T_{11} is $\{3, 5, 8, 9\}$ and one of T_{12} is $\{4, 6, 10\}$, then the root occurrence lists of T_{11} and T_{12} are both $\{2, 7\}$ and thus T_{11} and T_{12} are σ-frequent. On the other hand, the patterns T_{21} and T_{22} have frequency $0 < \sigma = 0.2$, and thus, it is discarded from $\mathcal{F}_2$.

By repeating these processes after stage 3, the algorithm terminates at stage 5 and returns $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4 = \{T_1, T_2, T_{11}, T_{12}, T_{113}, T_{114}, T_{1134}\}$ as the answer.

4 Experimental Results

In this section, we present experimental results on real-life datasets to evaluate the performance and the robustness of our algorithm on a range of datasets and parameters. The task considered here is substructure discovery from HTML/XML pages, which is to discover a set of frequent substructures as patterns in a large document tree generated from a collection of Web pages gathered from Internet. The goals of these experiments are: (i) How the running time scales with the size of the data trees and the value of minimum frequecy thresholds; (ii) Performance comparison of the versions of our algorithm equipped with the speed-up methods described in Section 3.

4.1 Implementation and Experimental Setup

We implemented our prototype system of the effcient tree mining algorithm FREQT described in Section 3 (Fig. 2) in Java (SUN JDK1.3.1 JIT) with a DOM library (OpenXML), a standard API for manupulating document trees of XML/HTML data. The experiments were run on a PC (Pentium III 600MHz) with 512 megabytes of main memory running Linux 2.2.14.

In the experiments, we implemented and compared the following three algorithms. In what follows, the parameters n and σ denote the data size as the number

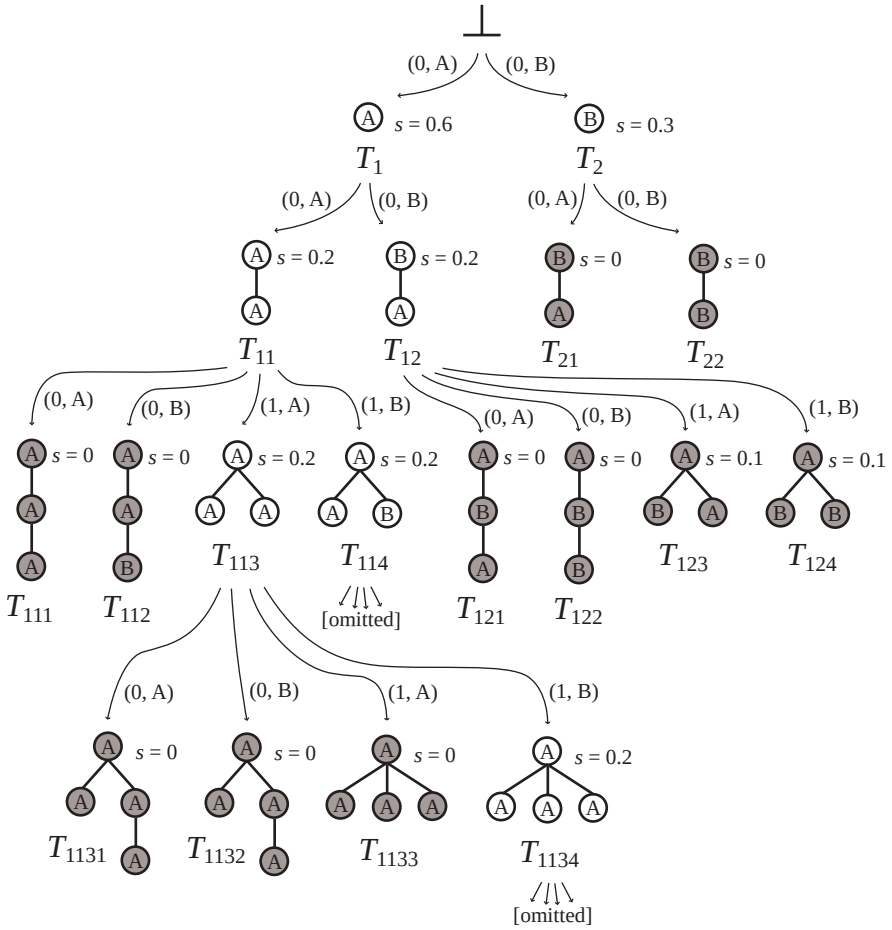
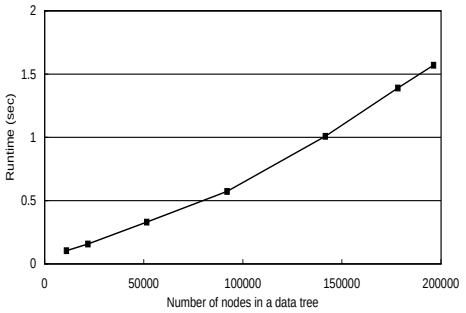


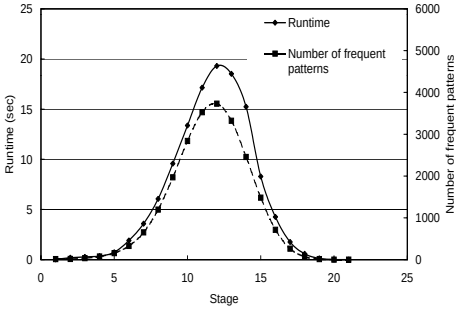
Figure 6. *The enumeration tree for patterns on the data tree D in Fig. 1, where the minsup $\sigma = 0.2$. The pair (p, ℓ) attached to an arrow from a pattern S to a pattern T indicates that T is tha (p, ℓ)-expansion of S. A white pattern represents a frequent pattern, and a shadowed pattern represents an infrequent pattern. For each pattern, the number s attached it represents its frequency.*

of nodes and the minimum support.

- **FREQT without the Duplicate-Detection**: This version uses explicit duplicate check instead of the Duplicate-Detection technique.
- **FREQT**: This is a straightfoward implementation of the FREQT algorithm of Fig. 2 with the Duplicate-Detection technique in Section 3.
- **FREQT with Node-Edge-Skip**: This version is FREQT with the Node-Skip and the Edge-Skip techniques of Section 3.4 as well as the Duplicate-Detection.



(a) The scalability with the size of a data tree.



(b) The running time and the number of frequent patterns.

Figure 7. *The running time of the FREQT algorithm*

4.2 Data

We prepared two datasets *Citeseers* (5.6MB) and *Allsite* (3.6MB) by collecting Web pages from search engines and Web databases. The *Citeseers* dataset was a collection of Web pages from an online bibliographic archive Citeseers[2], and the *Allsite* dataset was a mixture of Web pages from 30 major Web search engine sites listed in [15]. Both datasets consisted of a few hundreds of HTML pages. After collecting pages, datasets were parsed to create DOM trees, and then attribute-value pairs in a DOM tree were converted into a set of nodes as follows. If a node v has an attribute-value pair $(Attr, Val)$ then we create a two-node tree consisting of the root and a child labeled with $Attr$ and Val, resp. Then, we attach such two-node trees represening pairs as a subtree of v in the lexicographic order on attribute-value pairs. After preprocessing, the data trees for *Citeseers* had 196,247 with 7,125 unique tags, and *Allsites* had 192,468 nodes with 9,696 unique tags.

4.3 Scalability

In the first experiment, we studied the scalability of the algorithm *FREQT*. Fig. 7(a) showed the running time of *FREQT* with $\sigma = 3\%$ on the *Citeseers* data as the size n of the data tree is increased from 316KB to 5.62MB as HTML page size. The total number of discovered maximal frequent patterns is around ten for all data size while the number of the rightmost occurrences scanned linearly increases from 12,844 nodes to 223,003 nodes as the data size is increased. As a result, the running time scales almost linearly with size n of the data tree, but slightly above linear. This nonlinearlity probably comes from that the average length of sibling lists may slowly increase as n grows.

[2] `http://citeseer.nj.nec.com/`

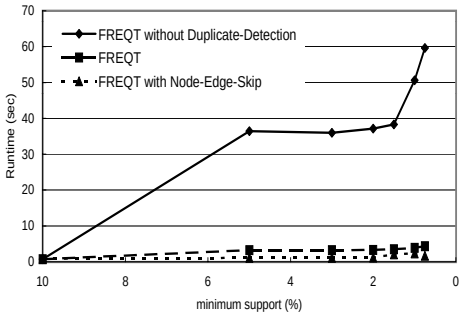


Figure 8. *Comparison of algorithms*

4.4 Comparison of algorithms

Secondly, we show the performance comparison among three algorithms with different update technique for the rightmost occurrence lists, namely *FREQT without the Duplicate-Detection, FREQT*, and *FREQT with Node-Edge-Skip*. Fig. 8 shows the running time on the *Allsite* data as the minimum support σ is decreased from 10% to 1.75%. For the largest value of $\sigma = 10\%$, only 1-patterns can be frequent and, thus, three algorithms coincide each other. For smaller values of $\sigma < 10\%$, *FREQT* was more than ten-times faster than the slowest algorithm *FREQT without the Duplicate-Detection*, and *FREQT with Node-Edge-Skip* was three-times faster than *FREQT*. For instance, the running times of the three algorithms with $\sigma = 2\%$, in this order, were 37.15 (sec), 3.285 (sec), and 1.151 (sec), respectively.

4.5 Running time and tree statistics against stages

In the third experiment, we studied the behavior of the algorithm *FREQT* in more detail when the number of frequent patterns is quite large. Fig. 7(b) shows the running time of *FREQT* with $\sigma = 2\%$ on a subset of *Citeseers* of 1.49MB as well as the total number of the discovered frequent patterns as the stage proceeds. In association rule mining [4], it is often reported that the number of frequent patterns has a peak at stage 2 and decreases afterwards. Unlike this, it is interesting that in Fig. 7(b) both the running time and the number of frequent patterns have a peak at the middle stages. From this, a special optimization strategy may be required for tree mining.

4.6 Examples of discovered patterns

In Fig. 9, we show examples of frequent patterns, in HTML format, discovered by the *FREQT* algorithm from the HTML pages in the *Citeseers* data (89,128 nodes) with the minimum support $\sigma = 1.17\%$. By inspection on the data, we observe that the algorithm correctly captured repeated substructure of bibliographic entries; One is for an HTML link at the header of an entry and another is for the body of an entry, where `#text_1` corresponds to the fixed label "Correct," `#text_2` stand for

No. 68, Size 6, Hit 1162, Freq 1.30%

```
<a href=_> <font color="#6F6F6F"> #text_1 </font> </a>
```

No. 10104, Size 20, Hit 1039, Freq 1.17%

```
<p> #text_2
<b> #text_3 <!-- CITE --> <font color="green"> #text_4 </font>
#text_5 </b> #text_6 <br /> <br />
<font color="#999999"> #text_7 <i> #text_8 </i> #text_9 </font> </p>
```

Figure 9. *Examples of discovered frequent patterns*

the title of an article, `#text_3` to `#text_6` for texts in its abstract, `#text_7` to `#text_9` for the title of a citation below the abstract, the color codes `"#6F6F6F"` and `"#999999"` stand for dark gray and light gray, respectively. These two trees have sizes 6 and 20 (nodes), and appeared in 1.30% (1162 times) and 1.17% (1039 times) of nodes in the data tree.

5 Conclusion

In this paper, we studied a data mining problem for semi-structured data by modeling semi-structured data as labeled ordered trees. We presented an efficient algorithm for finding all frequent ordered tree patterns from a collection of semi-structured data, which scales almost linearly in the total size of maximal patterns. We run experiments on real-life Web data to evaluate the proposed algorithms.

From the experiment on Web data, our algorithm is useful to extract regular substructures in a large collection of Web pages, and thus, may have applications in information extraction from Web and query modification in semi-structured database languages [15, 21].

To deal with more realistic applications, we need to expand our algorithm to deal with more complex components such as attributes and texts in semi-structured data. Extension for graph structures [14, 17, 18] and first-order models [11] is another future work. Studies on the computational complexity of learning tree structured patterns [9, 15] may give some insights on mining such complex structures. Moreover, heuristic approaches like the algorithm Cupid [16] computing a score that represents a similarity of two labeled trees, would be promising.

Acknowledgments

The authors would like to thank Shinichi Morishita, Satoru Miyano, Akihiro Yamamoto, Masayuki Takeda, Ayumi Shinohara, and Shinichi Shimozono for the valuable dicussions and comments. Hiroki Arimura would like to express his sincere thanks to Heikki Mannila and Esko Ukkonen to direct his attention to this area.

Bibliography

[1] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web, Morgan Kaufmann, 2000.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, S. Arikawa, *Efficient Substructure Discovery from Large Semi-structured Data*, Department of Informatics, Kyushu Univ., DOI Technical Report DOI-TR-200, Oct. 2001. `ftp://ftp.i.kyushu-u.ac.jp/pub/tr/trcs200.ps.gz`

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, *The Lorel Query Language for Semistructured Data*, Intl. J. on Digital Libraries, 1(1), 1997, pp. 68–88.

[4] R. Agrawal, R. Srikant, *Fast Algorithms for Mining Assoiciation Rules*, In Proc. the 20th VLDB, 1994, pp. 487–499.

[5] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo, *Fast Discovery of Association Rules*, Advances in Knowledge Discovery and Data Mining, Chapter 12, AAAI Press / The MIT Press, 1996.

[6] Aho, A. V., Hopcroft, J. E., Ullman, J. D., Data Structures and Algorithms, Addison-Wesley, 1983.

[7] H. Arimura, *Efficient Learning of Semi-structured Data from Queries*, In Proc. the 12th International Conference on Algorithmic Learning Theory (ALT'01), LNAI 2225, 2001, pp. 315–331.

[8] H. Arimura, S. Arikawa, S. Shimozono, *Efficient discovery of optimal word-association patterns in large text databases*, New Generation Computing, 18, 2000, pp. 49–60.

[9] H. Arimura, H. Ishizaka, T. Shinohara, *Learning Unions of Tree Patterns Using Queries*, Theoretical Computer Science, 185, 1997, pp. 47-62.

[10] R. J. Bayardo Jr., *Efficiently Mining Long Patterns from Databases*, In Proc. SIGMOD98, 1998, pp. 85–93.

[11] L. Dehaspe, H. Toivonen, R. D. King, *Finding Frequent Substructures in Chemical Compounds*, In Proc. KDD-98, 1998, pp. 30–36.

[12] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama, *Data mining using two-dimensional optimized association rules*, In Proc. SIGMOD'96, 1996, pp. 13–23.

[13] J. Han, J. Pei, T. Yin, *Mining frequent patterns without candidate generation*, In Proc. SIGMOD 2000, ACM, 2000, pp. 1–11.

[14] A. Inokuchi, T. Washio, H. Motoda, *An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data,* In Proc. PKDD 2000, LNAI 1910, 2000, pp. 13-23.

[15] N. Kushmerick, *Wrapper induction: efficiency and expressiveness*, Artificial Intelligence, Vol.118, 2000, pp. 15–68.

[16] J. Madhavan, P. A. Bernstein, E. Rahm, *Generic Schema Matching with Cupid*, In Proc. 27th VLDB Conference, 2001, pp. 49–58.

[17] H. Mannila, C. Meek, *Global Partial Orders from Sequential Data*, In Proc. KDD2000, 2000, pp. 161–168.

[18] T. Matsuda, T. Horiuchi, H. Motoda, T. Washio, K. Kumazawa, N. Arai, *Graph-Based Induction for General Graph Structured Data*, In Proc. DS'99, 1999, pp. 340–342.

[19] T. Miyahara, T. Shoudai, T. Uchida, K. Takahashi, H. Ueda, *Discovery of Frequent Tree Structured Patterns in Semistructured Web Documents*, In Proc. PAKDD-2001, 2001, pp. 47–52.

[20] S. Morishita, *On classification and regression*, In Proc. DS'98, LNAI 1532, 1998, pp. 49–59.

[21] K. Taniguchi, H. Sakamoto, H. Arimura, S. Shimozono and S. Arikawa, *Mining Semi-Structured Data by Path Expressions*, In Proc. the 4th Int'l Conf. on Discovery Science, LNAI 2226, 2001, pp. 387–388. (To appear)

[22] J. Sese, S. Morishita, In Proc. the second JSSST Workshop on Data Mining, JSSST, 2001, pp. 38–47. (In Japanese)

[23] W3C, Extensibe Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 06 October 2000.
`http://www.w3.org/TR/REC-xml`

[24] J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, C.-Y. Chang, *Automated Discovery of Active Motifs in Multiple RNA Seconary Structures*, In Proc. KDD-96, 1996, pp. 70–75.

[25] K. Wang, H. Liu, *Schema Discovery for Semistructured Data*, In Proc. KDD'97, 1997, pp. 271–274.

[26] M. J. Zaki, *Efficiently Mining Frequent Trees in a Forest*, Computer Science Department, Rensselaer Ploytechnic Institute, PRI-TR01-7-2001, 2001.
`http://www.cs.rpi.edu/~zaki/PS/TR01-7.ps.gz`