Efficient Substructure Discovery from Large Semi-structed Data

Asai, Tatsuya
Department of Informatics Kyushu University

Abe, Kenji
Department of Informatics Kyushu University

Kawasoe, Shinji
Department of Informatics Kyushu University

Arimura, Hiroki
PRESTO, JST | Department of Informatics Kyushu University

他

http://hdl.handle.net/2324/3048

出版情報：DOI Technical Report. 200, 2001-10. 九州大学大学院システム情報科学研究院情報理学部門
バージョン：
権利関係：

Efficient Substructure Discovery from Large Semi-structured Data

(Submitting to the 2nd Annual SIAM Symposium on Data Mining, SDM2002)

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

[†]Department of Informatics, Kyushu University, Japan
[‡]PRESTO, JST, Japan
{t-asai, k-abe, s-kawa, arim, sakamoto, arikawa}@i.kyushu-u.ac.jp

Abstract

In this paper, we consider a data mining problem for semi-structured data. Modeling semi-structured data as labeled ordered trees, we present an efficient algorithm for discovering frequent substructures from a large collection of semi-structured data. By extending the enumeration technique developed by Bayardo (SIGMOD'98) for discovering long itemsets, our algorithm scales almost linearly in the total size of maximal tree patterns contained in an input collection depending mildly on the size of the longest pattern. We also developed several pruning techniques that significantly speed-up the search. Experiments on Web data show that the our algorithm runs efficiently on real-life datasets combined with proposed pruning techniques in the wide range of parameters.

keywords: Web mining, semi-structured data, association rule mining, itemset enumeration tree, labeled ordered trees, data mining algorithms

Correspondence:
Tatsuya Asai
Department of Informatics, Kyushu University
Fukuoka 812-8581, Japan
EMAIL: t-asai@i.kyushu-u.ac.jp
TEL: +81-92-642-2688, FAX: +81-92-642-2698

By rapid progress of network and storage technologies, a huge amount of electronic data such as Web pages and XML data 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, 16, 17, 19, 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 Find-Freq-Trees 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 [17, 24]. In contrast, our algorithm Find-Freq-Trees 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 [9].

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 rightmost leaves to efficiently implement incremental computation.

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 significantly speeds-up the search.

Experiments on real-world datasets show that the our algorithm runs efficiently on real-life datasets combined with a proposed pruning technique in the wide range of parameters.

```
{  person:  { name:Alan,tel:7786,tel:2133 },
   person:  {
      name:  {first:Sara, last:Green },
      tel:6877 },
   person:  { name:Fred, tel:6312, age:33 }
}
```

Figure 1: semi-structured data

```
   <person>
      <name>Alan</name>
      <tel>7786</tel>
      <tel>2133</tel>
   </person>
   <person>
      <name>
         <first>Sara</first>
         <last>Green</last>
      </name>
      <tel>6877</tel>
   </person>
   <person>
      <name>Fred</name>
      <tel>6312</tel>
      <age>33</age>
   </person>
```

Figure 2: a XML document

1.1 Related Works

Semi-structured data is such data represented as a tree-like structure as shown in Fig. 1. There are a number of studies on a semi-structured database. For example, a considerable amount of studies on query languages and index structures are executed [2, 10, 22]. On the other hand, the classes of tagged texts such as HTML and XML [21], as shown in Fig. 2 can also be regarded as semi-structured data. There are standardization activities of query languages and data structures of XML[21, 23].

But, there are still not many researches about data mining for semi-structured data. Wang and Liu[25] considered a schema discovery problem for semi-structured data, and presented the algorithm for discovering association rules where an item is a path. Miyahara et al.[17] also considered the same problem based on the special tree-like patterns, called

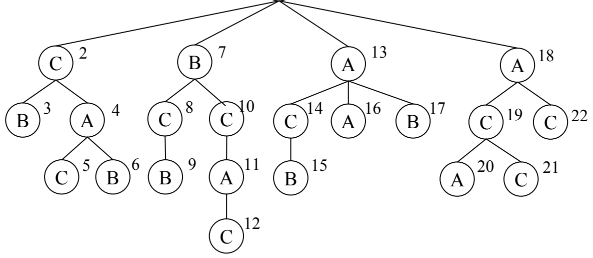

Figure 3: a database

tag tree patterns.

Several mining methods for a tree or a graph structure are proposed although they are not directly applicable to semi-structured data mining. Wang, Shapiro, Shasha et al.[24] devised the algorithm for discovering approximately common subtree, and applied it to a motif discovery in multiple RNA secondary structures in genomics. Dehaspe et al.[11] presented the efficient algorithm solving the frequent substructure discovery problem for labeled graphs, and applied it to the problem of function prediction of chemical compounds. Matsuda and Motoda et al.[16] presented the algorithm for extracting typical patterns from a directed graph. Their algorithm uses a method called the graph-based induction.

In the association rule discovery problem, Agrawal et al.[4, 5] developed an algorithm, called Apriori, which is a popular data mining problem. Their algorithm discovers frequent itemsets efficiently by using a subset lattice of an itemset. Actually the algorithms [11, 25] described above are based on Apriori. But it is said that the efficiency of Apriori slowdowns if a database contains long itemsets.

To cope with this problem, Bayardo[9] proposed the algorithm discovering long itemsets efficiently. The algorithm is based on the itemset enumeration technique, called *set-enumeration tree*, which enumerates all the frequent itemsets without repetition. Sese and Morishita[20] presented the algorithm discovering optimal itemsets, based on the set-enumeration tree and a method of merging occurrence lists of each itemset. In addition to these works, there are many works that complement Apriori algorithm [3, 4, 5, 9, 14, 20] Our results generalizes the techniques of [9] and [20] above, and thus can be regarded as a tree-counter part of the second

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 describe our search strategy used for enumerating all labeled ordered tree without duplicates. In Section 3, we present our algorithm for discovering all frequent patterns from a collection of labeled ordered trees using incremental update technique of the occurrence information combined with the enumeration technique shown in the previous section. In Section 4, we describe how we can speed-up the search by using pruning techniques. In Section 5, we run experiments on real datasets and show the efficiency and the scale-up properties of our mining algorithm. In Section 6, we conclude and give future works.

2 Preliminaries

In this section, we introduce basic notions and definitions on semi-structured data and our data mining problems.

2.1 Labeled Ordered Trees

For a set A, $\#A$ denotes the cardinality of A. Let $\mathcal{L} = \{l, l_0, l_1, \ldots\}$ be a finite set of labels corresponding to attributes in semi-structured data or tags in tagged texts. We model semi-structured databases and patterns over them with labeled ordered trees [6] defined as follows.

Definition 1 A *labeled ordered tree on* $\mathcal{L}$ (an ordered tree, for short) is a 6-tuple $T = (V, E, \mathcal{L}, L, v_0, \preceq)$, where

- $G = (V, E, v_0)$ is a tree with a root v_0, where V is a finite set of nodes and $E \subseteq V^2$ is the set of edges. For a node $v \in V$, we denote by $pa_T(v)$ the parent node of v, and by $ch_T(v)$ the set of all the child nodes of v.
- $L : V \to \mathcal{L}$ is a labeling function that assigns a label $L(v)$ to each node $v \in V$.
- $\preceq \subseteq V^2$ is a binary relation, called the sibling relation for T, satisfying the followings: (i) If

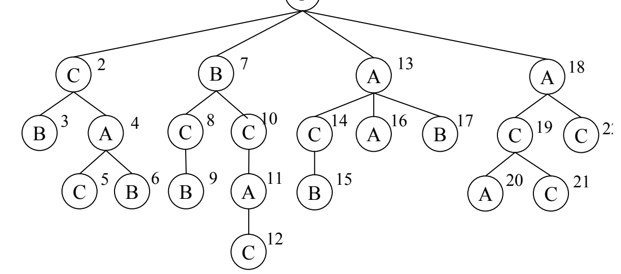

Figure 4: a database

nodes $v_1, v_2 \in V$ have a same parent then v v_2 or $v_2 \preceq v_1$, (ii) otherwise neither $v_1 \preceq v_2$ $v_2 \preceq v_1$. For an internal node, its children ordered from left to right as $u_1 \prec \cdots \prec u_n$ some $n \geq 0$, where $u \prec v$ iff $u \preceq v$ but $v \not\preceq u$.

In Fig. 4 and Fig. 5, we show examples of labeled ordered trees, where a circle indicates a node and a symbol appears in a circle indicates a label.

Let T be a labeled ordered tree. The *size* of T is defined by the number of its nodes $|T| = \#V$. For a path $\Pi = (x_0, \ldots, x_{n-1})$ $(n \geq 0)$ in T, we define the *length* of the path to be $|\Pi| = n$. For a node $v \in V$, the *depth* of v, denoted by $depth(v)$, is defined as the length of the path from the root v_0 to v. The *depth* of a labeled ordered tree T is the length of the longest path from the root to some leaf in T. For an ordered tree $T = (V, E, \mathcal{L}, L, v_0, \preceq)$, we refer to $V, E, L, and \preceq$ as $V_T, E_T, L_T, and \preceq_T$ respectively, if it is clear from context.

2.2 Problem Statement

A *semi-structured database* (a *dataset*, for short) is a triple $\mathcal{D} = \langle D, \Delta, \delta \rangle$, where

- D is an ordered tree on $\mathcal{L}$, and is called a *data tree*. We assume that the root v_0 of D, has a special label that does not belong to $\mathcal{L}$.
- $\Delta = \{d_1, \ldots, d_m\}$ is a set of document names.
- $\delta : V_D \backslash \{v_0\} \rightarrow \Delta$ is a *document name function* defined as follows. Let D_i be a subtree whose root is the i-th child of v_0. Then, δ is defined as the function that satisfies $\delta(v) = d_i$ for all $v \in V_{D_i}$.

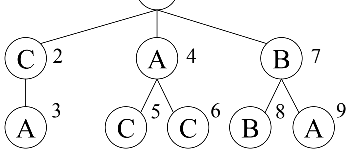

Figure 5: an ordered tree of normal form

In Fig. 4, we show an example of semi-structured databases. Intuitively, D is a labeled tree obtained by merging a collection of labeled trees $D_1, \ldots, D_m$, where each subtree D_i represents a Web page or an XML documents. Thus, we refer to the maximal subtree of D whose nodes are labeled with the same document name as a *document*. We define the *size* of $\mathcal{D}$ by the total number $||\mathcal{D}|| = \#V_D$ of its nodes.

A labeled ordered tree T of size $k \geq 1$ is said to be *of normal form* if it satisfies the following conditions:

- The set of the nodes of T is $V_T = \{1, \ldots, k\}$.
- All elements in V_T are numbered by preorder traversal [6] of T.

The following lemma immediately follows from the definition.

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.*

Fig. 5 shows an ordered tree of normal form, where a symbol in a circle represents a label, and a number attached to the right of a circle represents a node-name. The size of the tree is 9, and we see that the root is 1 and the rightmost leaf is 9.

For a nonnegative integer $k \geq 1$, a *substructure pattern* of size k (a *k-pattern*, for short) is an ordered tree T on $\mathcal{L}$ of normal form and $|T| = k$. For every $k \geq 1$, we denote the set of all k-patterns by $\mathcal{T}_k$ and the set of all patterns by $\mathcal{T} := \bigcup_k \mathcal{T}_k$. We also define $\mathcal{T}^{\leq n} := \bigcup_{k=1}^{n} \mathcal{T}_k$ for every positive integer $n \geq 1$.

To define our data mining problem, we need the notions of the occurrences of a pattern in a dataset, which is not straightforward in the case for ordered trees unlike the case for itemsets. We first start with defining the notion of matching functions.

a *matching function from T to D* is any function $\varphi : V_T \to V_D$ that satisfies the following conditions (i)–(iv) for any $v, v_1, v_2 \in V_T$.

(i) φ is a *one-to-one mapping*. That is, if $v_1 \neq v_2$ then $\varphi(v_1) \neq \varphi(v_2)$.

(ii) φ *preserves the parent-child relation*. That is, $(v_1, v_2) \in E_T$ iff $(\varphi(v_1), \varphi(v_2)) \in E_D$.

(iii) φ *preserves the sibling relation*. That is, $v_1 \preceq_T v_2$ iff $\varphi(v_1) \preceq_D \varphi(v_2)$.

(iv) φ *preserves the label of each node*. That is, $L_T(v) = L_D(\varphi(v))$.

We say that T *occurs* in D if there exists some matching function from T to D.

The above definition of matching functions is a variant of those widely used in the community of string/tree pattern matching [15]. Note that $\varphi(v_1)$ and $\varphi(v_2)$ need not be adjacent even if v_1 and v_2 are adjacent in the condition (iii) of our definition. Equivalently, a tree T matches another tree D iff T is obtained from a subtree D' of D by repeatedly removing leaves and incident edges from D'.

Unlike the case for itemsets or strings, the matching problem for labeled ordered tree patterns is not easy to solve [24]. A straightforward algorithm decide the matching, if T matches D, in $O(mn)$ time and this is improved to $O(m^{0.75}n)$ time in [15], while it is not known if it is possible to solve this problem in $O(m + n)$, where $m = |T|$ and $n = |D|$.

Most straightforward notion of an occurrence of a k-pattern T in a dataset will be a matching function $\varphi : V_T \to V_D$ from T to D itself. However, this definition of occurrences has a drawback that a pattern has unnecessarily many occurrences. In reality, we can see that a k-pattern may have exponentially many occurrences in the size k of the pattern on a dataset of total size n.

Instead, we will define the occurrences of a pattern based on the set of nodes of the data tree D to which a designated node in the pattern maps. Let $\mathcal{D} = \langle D, \Delta, \delta \rangle$ be a dataset, and k be a positive integer. Recall that for any k-pattern T, it follows from Lemma 1 that the nodes 1 and $k \in V_T$ are the root k-pattern $T \in \mathcal{T}_k$ and a matching function φ from T to D, we define the *root occurrence* (the occurrence or Occ, for short), the *rightmost occurrence* (the rml-occurrence or Roc for short), and the *document occurrence* (the doc-occurrence or Doc, for short) of T w.r.t. φ to be $\varphi(1)$, $\varphi(k)$, and $\delta(\varphi(1))$, respectively.

Based on the above notions of the occurrences, we define the *root-occurrence list* ($Occ(T)$), the *rml-occurrence list* ($Roc(T)$), and the *document occurrence list* ($Doc(T)$) as follows.

- $Occ(T) =$
  $\{\varphi(1) \mid \varphi : V_T \to V_D$ is a matching function$\}$.
- $Roc(T) =$
  $\{\varphi(k) \mid \varphi : V_T \to V_D$ is a matching function$\}$.
- $Doc(T) = \{\delta(v) \mid v \in Occ(T)\}$.

Though the definition of the rml-occurrence may appear to be strange at first glance and it is not used in the definitions below, the discussion in Section 3 and Section 4 will reveal that it possesses a number of good properties and will play a key role in our algorithms.

Given a dataset $\mathcal{D} = \langle D, \Delta, \delta \rangle$ and a k-pattern T, we define the occurrence-frequency (the occ-frequency, for short) and the document-frequency (the doc-frequency, for short) of as follows.

- $freq_{\mathcal{D}}(T) = \frac{\#Occ(T)}{||\mathcal{D}||}$.
- $docfreq_{\mathcal{D}}(T) = \frac{\#Doc(T)}{\#\Delta}$.

Let $0 < \sigma \leq 1$ be a positive number. A k-pattern T is *σ-frequent w.r.t. the occurrence frequency* in $\mathcal{D}$ if $freq_{\mathcal{D}}(T) \geq \sigma$ and is *σ-frequent w.r.t. the document frequency* in $\mathcal{D}$ if $docfreq_{\mathcal{D}}(T) \geq \sigma$.

An instance of our data mining problem is a triplet $\langle \mathcal{L}, \mathcal{D}, \sigma \rangle$, a set of labels $\mathcal{L}$, a dataset $\mathcal{D} = \langle D, \Delta, \delta \rangle$, and a positive number $0 < \sigma \leq 1$, called the *minimum support* (*minsup*, for short). We now state our data mining problem, called the frequent pattern discovery problem, as follows.

Frequent Pattern Discovery Problem

frequent w.r.t. the *occurrence frequency* in $\mathcal{D}$, i.e., $freq_{\mathcal{D}}(T) \geq \sigma$.

This problem is a generalization of the frequent itemset discovery problem in association rule mining [4]. The following problem is a natural variation of the above problem in Web mining applications.

Doc-Frequent Pattern Discovery Problem

Problem: Find all patterns $T \in \mathcal{T}$ that are σ-frequent w.r.t. the *document frequency* in $\mathcal{D}$, i.e., $docfreq_{\mathcal{D}}(T) \geq \sigma$.

2.3 Representation of Ordered Trees

In this paper, we assume that ordered trees are represented by the *first-child, right-sibling representation* [6]. In this representation, each node $v \in V_T$ of a ordered tree T has two informations, the first child $child(v)$ and the right sibling $next(v)$. The first child of v is a leftmost child of v, and the right sibling is a next sibling of v. If v does not have the first child (resp., the right sibling) then we assume that $child(v) = NIL$ (resp., $next(v) = NIL$). This representation is also adopted in a standard of data format and application interface of XML, called DOM [1].

3 A Basic Mining Algorithm

In this section, we present an efficient algorithm for discovering all frequent tree patterns that scales almost linearly in the total size of the maximal patterns embedded in a dataset. In the rest of this section, we describe the efficient enumeration of labeled ordered tree patterns in Subsection 3.2 and the incremental computation of the rml-occurrence lists in Subsection 3.3.

3.1 Overview of the Algorithm

In Fig. 6, we present our algorithm Find-Freq-Trees for discovering all frequent ordered tree patterns that have the frequency at least the given minimum support threshold $0 \leq \sigma < 1$ in a dataset $\mathcal{D}$. As the basic design of the algorithm, we adopted the levelwise search as in the Apriori algorithm [4] combined with efficient enumeration technique [9].

```
Algorithm Find-Freq-Trees
Input: A dataset $\mathcal{D} = \langle D, \Delta, \delta \rangle$, a set $\mathcal{L}$ of labels,
      and a minsup $0 < \sigma \leq 1$.
Output: A set $\mathcal{F}$ of all $\sigma$-frequent patterns in $\mathcal{D}$.
Method:
$\mathcal{F}_1$ := the set of $\sigma$-frequent 1-patterns, and
compute the set $Roc_1$ of their rml-occurrences;
$k := 2$;
while $\mathcal{F}_{k-1} \neq \emptyset$ do begin
   $\langle \mathcal{C}_k, Roc_k \rangle :=$ Expand-Trees($\mathcal{F}_{k-1}, Roc_{k-1}$);
   foreach pattern $T \in \mathcal{C}_k$ do
      Compute $freq_{\mathcal{D}}(T)$ from $Roc_k(T)$;
      if $freq_{\mathcal{D}}(T) \geq \sigma$ then $\mathcal{F}_k = \mathcal{F}_k \cup \{T\}$;
   $k := k + 1$;
end /* while-loop */
return $\mathcal{F} = \mathcal{F}_1 \cup \cdots \cup \mathcal{F}_{k-1}$;
```

Figure 6: The algorithm for discovering all frequent ordered tree patterns in a dataset $\mathcal{D}$, where for every $k \geq 1$, $\mathcal{F}_k$ is the set of all σ-frequent patterns in $\mathcal{D}$ and Roc_k is the set of their rightmost occurrence lists.

In the first pass, the algorithm simply counts the number and the positions of the occurrences of the distinct labels by traversing the data tree D. The results are stored in the set $\mathcal{F}_1$ and Roc_1, respectively. In the subsequent pass $k \geq 2$, the algorithm computes the set $\mathcal{F}_k$ of all frequent k-patterns and the set Roc_k of their rightmost occurrence lists simultaneously by using the sub-procedure Expand-Trees shown in Fig. 9. Repeating this process until no more frequent patterns are generated, the algorithm computes all σ-frequent patterns in $\mathcal{D}$.

3.2 Efficient Enumeration of Ordered Trees

Our enumeration algorithm uses a similar technique to the set enumeration tree search of Bayardo [9]. A basic idea of our enumeration algorithm will be illustrated in Fig. 7. In the search, starting with a set of trees consisting of single nodes, the enumeration algorithm expands a given ordered tree of size $k-1$ by attaching a new node at a leaf position to yield larger tree of size k. We can easily see that this search strategy yields all k-patterns in $\mathcal{T}_k$ by at most k applications of expansion.

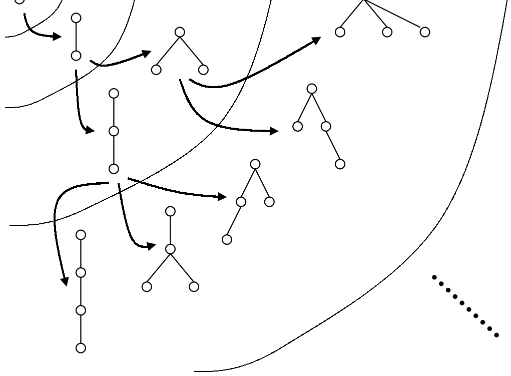

Figure 7: A search graph for (unlabeled) orde trees

However, when expansion at arbitrary positions may cause that some ordered tree may have more than one predecessor. This causes the duplicated enumeration and makes the Apriori-like algorithm [4] behave poorly on the datasets containing long frequent patterns. To overcome this problem, we give the restricted form of expansion, called a rightmost expansion, as follows. We denote the rightmost leaf of an ordered tree T by $rml(T)$.

Definition 3 For a labeled ordered tree T, the *rightmost branch* of T is the unique path from the root to the rightmost leaf in T.

Equivalently, the rightmost branch is a path Π in T starting from the root such that every $x \in \Pi$ is maximal in $\preceq$. Recall that for any k-pattern T, $rml(T) = k$ by Lemma 1.

Let A be a set, $f : A \to A$ be a function, and $n \geq 0$ be a nonnegative integer. Then, f^n is a function defined as $f^0(x) = x$ and $f^k(x) = f(f^{k-1}(x))$ for any $x \in A$ and $k = 1, \ldots, n$.

Definition 4 Let $T \in \mathcal{T}$ be patterns over $\mathcal{L}$, $0 \leq p < depth(rml(T))$ be any integer, and $l \in \mathcal{L}$ be any label. Suppose that $x = rml(T)$ is the rightmost leaf of T, and $y = {pa_T}^p(x)$ is the p-th parent of x. Then, the *(p, l)-expansion* of T is the labeled ordered tree S obtained by attaching a new node k with the label l to the node y so that the attached node is the rightmost child of y. (See Fig. 8).

A *rightmost expansion* of an ordered tree T is the (p, l)-expansion S of T for some integer $p \geq 0$ and some label $l \in \mathcal{L}$. Then, we say that either S is the *predecessor* of T or T is a *successor* of S. For convention, we assume a special labeled ordered tree, called the *empty tree*, $\bot$ such that $|\bot| = 0$ and any single node tree is a successor of $\bot$.

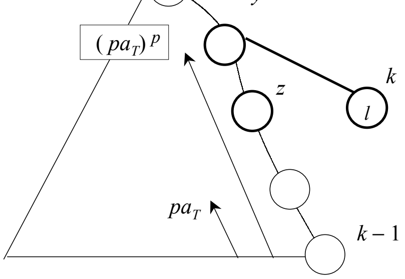

Figure 8: (p, l)-expansion

Definition 5 Let $\mathcal{L}$ be the set of labels and $\mathcal{T}$ be the set of ordered trees on $\mathcal{L}$. The *enumeration dag* for $\mathcal{T}$ based on the rightmost expansion is a directed graph G defined as follows. There is the unique root $\bot$. Each node of the tree G is an ordered tree on $\mathcal{L}$. There exists an out-going edge from a node S to another node T iff T is a rightmost expansion of S.

In Fig. 7, we show an example of the enumeration dag for the unlabeled ordered trees. The dag of Fig. 7 is actually a tree in this case. We will formally show this claim in the last part of this section. The *level* of a node S is the minimum length of the paths from the root to S. For every $k \geq 0$, the k-th layer of G is the set of all nodes of level k. We see that G is actually acyclic because an application of the expansion increases the size of a tree by one. We have the following properties on the rightmost expansion. Recall that a k-pattern is a labeled ordered tree of size k in normal form.

Lemma 2 *For every $k \geq 2$, if T is a $(k-1)$-pattern then any rightmost expansion of T is also a k-pattern.*

Proof. If T' is the rightmost expansion of T, then trivially $|T'| = k$. Since the new node $v_k = k$ is attached to a node on the rightmost branch of T, it should be the last node in the preorder traversal of T'. This shows the lemma. □

Lemma 3 *For every $k \geq 2$, if T is a k-pattern then*

T is a rightmost expansion of T'.

Proof. Suppose that T is obtained from some $(k-1)$-pattern T' 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. □

From Lemma 2 and Lemma 3 above, we see that the enumeration graph has good properties for the search of $\mathcal{T}$ as follows.

Theorem 4 *Let G be the enumeration dag for $\mathcal{T}$ based on the rightmost expansion. Then, for every $k \geq 0$, the k-th layer exactly contains the members of $\mathcal{T}_k$, and the whole G exactly contains the members of $\mathcal{T}$. Furthermore, G forms a tree, i.e., all nodes but the root have the unique predecessor.*

By the above theorem, we call the dag G the *enumeration tree* for $\mathcal{T}$ in what follows. Using the enumeration dag for $\mathcal{T}$, all labeled ordered tree in $\mathcal{T}$ can be enumerated.

In Fig. 9, we present the algorithm EXPAND-TREES that computes $\mathcal{F}_k$ and the corresponding set Roc_k of the rml-occurrence lists. Given a set $\mathcal{F}_{\text{old}} \subseteq \mathcal{T}$ and the set Roc_{old} of the corresponding rml-occurrence lists, the algorithm EXPAND-TREES computes the set $\mathcal{F}$ of the successors of those trees in $\mathcal{F}_{\text{old}}$ by using the rightmost expansion technique shown above. The computation of the set Roc of rml-occurrence lists in step 6 and the correctness of the whole algorithm will be discussed in detail in the next subsection.

3.3 Updating Occurence Lists

To compute the frequency of each enumerated pattern $T \in \mathcal{T}$, it is sufficient to compute the rml-occurrence list $Roc(T)$ of T since either $Doc(T)$ or $Occ(T)$ is easily computable from $Roc(T)$. Thus, we concentrate on the problem of computing $Roc(T)$ for all candidates $T \in \mathcal{C}_k$.

In this subsection, we give the algorithm UPDATE-ROC for incrementally updates the rml-occurence list $Roc(T)$ of a k-pattern T obtained by the rightmost expansion. (Fig. 10.)

```
Algorithm EXPAND-TREES($\mathcal{F}_{\text{old}}$, $Roc_{\text{old}}$)
Input: A set $\mathcal{F}_{\text{old}}$ of patterns, and the indexed set
   $Roc_{\text{old}}$ of their rml-occurrences indexed by trees
   in $\mathcal{F}_{\text{old}}$.
Output: The set $\mathcal{F}$ of the rightmost expansions of
   trees in $\mathcal{F}_{\text{old}}$ and the indexed set $Roc_{\text{old}}$ of
   their rml-occurrences indexed by trees in $\mathcal{F}$.
Method:
1 $\mathcal{F} := \emptyset$; $Roc := \emptyset$;
2 foreach tree $T \in \mathcal{F}_{\text{old}}$ do
3   foreach position $0 \leq p < depth(rml(T))$ do
4     foreach label $l \in \mathcal{L}$ do begin
5       Compute the $(p, l)$-expansion $S$ of $T$;
6       $Roc(S)$ := UPDATE-ROC($Roc_{\text{old}}(T), p, l$);
7       $\mathcal{F} = \mathcal{F} \cup \{S\}$;
8     end
9 return $\langle \mathcal{F}, Roc \rangle$;
```

Figure 9: The algorithm for computing all rightmost expansions of a set of trees. The sub-procedure UPDATE-ROC will be later described in Subsection 3.3.

The key of the algorithm UPDATE-ROC is how to efficiently store the information of the matching $\varphi : V_T \rightarrow V_D$ from each pattern $T \in \mathcal{C}_k$ to the data tree D. Instead of recording the full information $\langle \varphi(1), \ldots, \varphi(k) \rangle$ of the matching functions $\varphi : V_T \rightarrow V_D$, the algorithm maintains only the information of the rml-occurrences $\varphi(k)$, where $k = rml(T)$ is the rightmost leaf.

The following lemma is useful to incrementally compute $Roc(T)$ from $Roc(S)$ of its predecessor S.

Lemma 5 *Suppose a $(k-1)$-pattern T occurs in a data tree D, and $\varphi : V_T \rightarrow V_D$ is a matching function from T to D. Let T' be a (p, l)-expansion of T, then the function $\psi : V_{T'} \rightarrow V_D$ satisfying the following three conditions is a matching function from T' to D.*

(1) *ψ is an extension of φ as a mapping. That is, $\psi(i) = \varphi(i)$ holds for any $i = 1, \ldots, k-1$.*

(2) *$\psi(k)$ is a child of $pa_D(i)$ for any $p \geq 0$, and $\psi(k) \prec i$ if $p \neq 0$, where $i = {pa_D}^{p-1}(\varphi(k-1))$.*

(3) *$L_D(\psi(k)) = l$ holds.*

The next lemma immediately follows from Lemma 5.

```
Algorithm UPDATE-ROC($OldRoc, p, l$)
Input: A node list $OldRoc$, a nonnegative
  integer $p \geq 0$, and a label $l \in \mathcal{L}$.
Output: A node list $newlist$.
Method:
$NewRoc := \phi$;
for each $last \in OldRoc$ do
  if $p = 0$ then
    $v = child(last)$; /* the first child of $last$ */
  else then
    $cur := {pa_D}^{p-1}(last)$;
    $v := next(cur)$; /* the right sibling of $cur$ */
  while $v \neq NIL$ do begin
    if $L_D(v) = l$ then
      $NewRoc = NewRoc \cup \{v\}$;
    $v = next(v)$;
  end /* while-loop */
end /* for-loop */
return $NewRoc$;
```

Figure 10: An incremental algorithm for updating the rml-occurence list of a given k-pattern

Lemma 6 *Let $T \in \mathcal{T}_k$, $0 \leq p < depth(rml(T)) \leq k$, and $l \in \mathcal{L}$ be any label. The algorithm* UPDATE-ROC *computes, given an input $\langle Roc(T), p, l\rangle$, the rml-occurrence list $Roc(S)$ of the (p,l)-expansion S of T in time $O(kbn)$, where b is the maximum branching of D and $n = \#Roc(T)$.*

A straightforward algorithm computes $Roc(T)$ in time $O(km)$ from scratch, where $k = |T|$ and $m = ||\mathcal{D}||$. Therefore, the proposed algorithm above runs faster than the straightforward algorithm when the product bn is much smaller than $||\mathcal{D}||$, that is, the frequency of T in $\mathcal{D}$ is low.

Consider the algorithm EXPAND-TREES of Fig. 9. The algorithm maintains two data $\mathcal{F}$ and Roc. $\mathcal{F}$ is a set of successors enumerated by the rightmost expansion technique. $Roc \subseteq \mathcal{T} \times (V_D)^*$ is an indexed set of occurrence lists such that for every $T \in \mathcal{T}$, $Roc(T)$ denotes the list of occurrences corresponding to T. For $\mathcal{F} \subseteq \mathcal{T}$, we define $Roc(\mathcal{F}) = \{(T, Roc(T)) : T \in \mathcal{F}\}$. We define the size of Roc by $||Roc|| = \sum_{T \in \mathcal{F}} Roc(T)$.

Now, we show the correctness of the algorithm EXPAND-TREES of Fig. 9 and the main algorithm

Lemma 6, we have the next corollary.

Corollary 7 *Let $\mathcal{F}_{\text{old}} \subseteq \mathcal{T}$ and $Roc_{\text{old}} = Roc(\mathcal{F}_{\text{old}})$. Given an input $\langle \mathcal{F}_{\text{old}}, Roc_{\text{old}}\rangle$, The algorithm* EXPAND-TREES *computes the pair $\langle \mathcal{F}, Roc(\mathcal{F})\rangle$ in time $O(k^2 \ell b N)$, where k is the maximum size of patterns in $\mathcal{F}_{\text{old}}$, $\ell = \#\mathcal{L}$, b is the maximum branching of D and $N = ||Roc_{\text{old}}|| = \sum_{T \in \mathcal{F}_{\text{old}}} \#Roc(T)$.*

We prepare some notations. In the following, $\mathcal{D}$ is an input dataset on $\mathcal{L}$ and $0 < \sigma \leq 1$ is a real number, $\ell = \#\mathcal{L}$, b is the maximum branching of D. Let $\mathcal{F}_\sigma$ be the set of all σ-frequent patterns in $\mathcal{D}$. Then, k denotes the maximum size of the frequent patterns in $\mathcal{F}_\sigma$. A frequent pattern is *maximal* if any rightmost expansion of the pattern is no longer frequent on $\mathcal{D}$. From the next theorem, we can expect that the algorithm runs with reasonable time complexity in practice since both $\mathcal{F}_\sigma$ and $||Roc(\mathcal{F}_\sigma)||$ will not be extremely large.

Theorem 8 *Let $\mathcal{D}$ be a database, $\mathcal{L}$ be a label set, and $0 < \sigma \leq 1$ be a real number. Then, the algorithm* FIND-FREQ-TREES *of Fig. 6 finds all σ-frequent patterns in $\mathcal{T}$ in time $O(||\mathcal{D}|| + k^2 b \ell N)$, where $N = ||Roc(\mathcal{F}_\sigma)||$ is the total occurrences of the frequent patterns.*

Furthermore, the next theorem indicates that the algorithm scales almost linearly in the sum of the sizes of the maximal frequent patterns as in MaxMiner [9] when the maximum size k and $\ell = \#\mathcal{L}$ grows slowly. Thus, the algorithm is efficient for datasets with long patterns.

Theorem 9 *The algorithm* FIND-FREQ-TREES *of Fig. 6 enumerates at most $O(k\ell M)$ patterns during the computation, where M is the sum of the sizes of the maximal σ-frequent patterns.*

3.4 An Example

Consider the dataset $\mathcal{D}$ consisting of $|\Delta| = 4$ documents in Fig. 4 and assume that the minimum support is $\sigma = 0.5$ and $\mathcal{L} = \{A, B, C\}$. This value of σ implies that the minimum doc-occurence is 2 documents. We show the patterns generated by the sub-procedure EXPAND-TREES of Fig. 9 in Fig. 11 and the corresponding rml-occurence lists computed

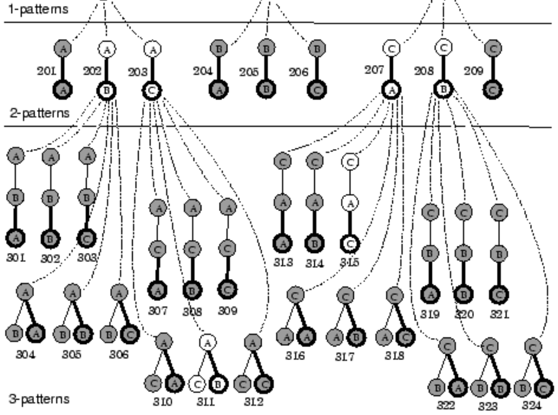

Figure 11: an example of FIND-FREQ-TREES: a white pattern represents a frequent pattern, and a shadowed pattern represents an infrequent pattern. The number attached to each pattern represents the ID of the pattern.

by the sub-procedure UPDATE-ROC of Fig. 10 in Table 1. In Fig. 11, each dotted line indicates the generation of a pattern of size $k \geq 1$ from its predecessor of size $k-1$ by the rightmost expansion.

First, the algorithm computes the set $\mathcal{F}_1$ of the frequent 1-patterns in step 1 by traversing the data tree D and records their occurrences in Roc_1. Calling EXPAND-TREES with $\mathcal{F}_1$ and Roc_1 gives the candidate set $\mathcal{C}_2$ and the set of their rml-occurence lists Roc_2. In Table 1, we see that $\mathcal{C}_2$ contains patterns 207, 208, 209. In the entry for 207, we see that the pattern 207 is obtained from its predecessor 103 with label C by attaching a new leaf with label A. The rml-occurence list of 207 is $\{4, 11, 20\}$, and the doc-occurence list is $\{1, 2, 4\}$, corresponding to the first, second and the fourth documents of D in Fig. 4. Then, the pattern 209 has frequency $0.25 < \sigma = 0.5$, and thus, it is discarded from $\mathcal{F}_2$.

Calling EXPAND-TREES with $\mathcal{F}_2$ and Roc_2 gives $\mathcal{C}_3$ and Roc_3. In the entry for $\mathcal{C}_3$ in Table 1, we see that the successors of 208, say 319–324, have doc-frequency less than $\sigma = 0.5$. Thus, they are removed from $\mathcal{F}_3$. Only one successor of 207, namely 315, has doc-frequency no less than 0.5 and are frequent. Finally, since none of successor of 315 in $\mathcal{C}_4$ appear in the dataset, the algorithm terminates and returns $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3$.

```
Algorithm SCANNING-SIBLINGS($OldRoc, p$)
Input: A node list $OldRoc$, and a nonnegative
  integer $p \geq 0$.
Output: A set of rml-occurence lists
  $\{NewRoc(l) \mid l \in \mathcal{L}\}$.
Method:
for each $l \in \mathcal{L}$ do
  $NewRoc(l) := \emptyset$;
for each $last \in OldRoc$ do
  if $p = 0$ then
    $v = child(last)$; /* the first child of $last$ */
  else then
    $cur := pa_D{}^{p-1}(last)$;
    $v := next(cur)$; /* the right sibling of $cur$ */
  while $v \neq NIL$ do begin
    $\lambda = L_D(v)$ ;
    $NewRoc(\lambda) = NewRoc(\lambda) \cup \{v\}$;
    $v = next(v)$;
  end /* while-loop */
end /* for-loop */
return $\{NewRoc(l) \mid l \in \mathcal{L}\}$;
```

Figure 12: An improved algorithm of UPDATE-ROC

4 Improvement

In this section, we consider some improvements of the proposal algorithms.

4.1 The Algorithm SCANNING-SIBLINGS

The algorithm SCANNING-SIBLINGS, shown in Fig. 12, is an improvement of the algorithm UPDATE-ROC of Fig. 10. Let T be a pattern, then SCANNING-SIBLINGS calculates all the rml-occurence lists w.r.t. each (p, l)-expansion of T for a given nonnegative integer p and a given rml-occurence list *OldRoc*. The algorithm performs as follows. First the algorithm creates the empty lists *NewRoc*(l) for every $l \in \mathcal{L}$. Then, for each $v \in$ *OldRoc* it adds v to *NewRoc*(λ) where λ is a label of v. Note that SCANNING-SIBLINGS scans each node $v \in Roc(T)$ only at once, while UPDATE-ROC scans them $\#\mathcal{L}$ times for calculating all the rml-occurence lists w.r.t. each (p, l)-expansion of T for a fixed $p \geq 0$. Therefore SCANNING-SIBLINGS is more efficient than UPDATE-ROC by factor of $O(\#\mathcal{L})$.

Table 1: $Roc(T)$, $Doc(T)$ and $docfreq_{\mathcal{D}}(T)$ of the patterns in Fig. 11

stage k	$\mathcal{C}_k$	predecessor	$\mathcal{F}_k$	$Roc(T)$	$Doc(T)$	$docfreq_{\mathcal{D}}(T)$
1	–	–	101	{4,11,13,16,18,20}	{1,2,3,4}	1.0
	–	–	102	{3,6,7,9,15,17}	{1,2,3}	0.75
	–	–	103	{2,5,8,10,12,14,19,21,22}	{1,2,3,4}	1.0
2	207	103	207	{4,11,20}	{1,2,4}	0.75
	208	103	208	{3,9,15}	{1,2,3}	0.75
	209	103	–	{21}	{4}	0.25
	...	...	...	...	...	...
3	313	207	–	$\emptyset$	$\emptyset$	0
	314	207	–	{6}	{1}	0.25
	315	207	315	{5,12}	{1,2}	0.5
	316	207	–	$\emptyset$	$\emptyset$	0
	317	207	–	$\emptyset$	$\emptyset$	0
	318	207	–	{21}	{4}	0.25
	319	208	–	$\emptyset$	$\emptyset$	0
	320	208	–	$\emptyset$	$\emptyset$	0
	321	208	–	$\emptyset$	$\emptyset$	0
	322	208	–	{4}	{1}	0.25
	323	208	–	$\emptyset$	$\emptyset$	0
	324	208	–	$\emptyset$	$\emptyset$	0
	...	...	...	...	...	...

4.2 The Pruning Using Frequent 1-patterns

Let T be a σ-in frequent 1-pattern and the label of the unique node of T be l. Then, the following obviously holds: The label l does not appear in any σ-frequent pattern $T' \in \mathcal{T}^+$.

Suppose the set of the labels evaluated to all the σ-frequent 1-patterns is called the *frequent label set* and is denoted $\mathcal{L}^+$. Then the rightmost expansions of any pattern is generated for each $l \in \mathcal{L}^+$ instead of $\mathcal{L}$ because of the fact.

4.3 The Pruning Using Frequent 2-patterns

Let T be a σ-frequent 2-pattern with the labels of the root and the leaf of T be l_1 and l_2. Then, the following assertion holds: For any σ-frequent pattern $T' \in \mathcal{T}^+$ and nodes v_1, v_2 of T', if $L'(v_1) = l_1$ and $L'(v_2) = l_2$ holds then $(v_1, v_2) \notin E'$ also holds.

The algorithm with the pruning method based on this assertion are developed as follows. First the algorithm records the pairs of labels for every σ-frequent 2-pattern. Then it checks the pair of labels of a new node and the node attached it to, on generating a rightmost expansion. If the checked pair of labels is not σ-frequent, then the algorithm prune this rightmost expansion since it is not σ-frequent.

4.4 The Algorithm Duplicate-Detection

This optimization is shown to be most effective in the proposed techniques in this section from experiments of Section 5. Let T be a σ-frequent pattern and T' be a rightmost expansion of T. When the algorithm Update-Roc or the algorithm Scanning-Siblings computes the rml-occurence list $Roc(T')$, it scans all the nodes of $ch_D(v)$ if $p = 0$, and on all the nodes that are to the right of $pa_D^{p-1}(v)$ if $p \geq 1$, for any $v \in Roc(T)$. But this method often scans the same nodes more than once if $p \geq 1$.

This problem is solved by improving the algorithm Scanning-Siblings and this yields an optimization technique Duplicate-Detection, as follows. First the algorithm initializes a variable *last* to NIL.

Table 2: the URL of the corresponding search engine of each dataset

datasets	URL
allsites	(mixture of 30 sites)
altavista	`http://www.altavista.com/`
google	`http://www.google.com/`
WebCrawler	`http://www.webcrawler.com/`
lycos	`http://www.lycos.com/`
fast	`http://www.fastsearch.com/`
argos	`http://argos.evansville.edu/`
citeseers	`http://citeseer.nj.nec.com/`

Next, for each $v \in Roc(T)$ it checks the value of *last* before scanning on the nodes of D. If $last = pa_D^p(v)$ then it skips the manipulation for $v \in Roc(T)$ and goes to the next nodes of v in $Roc(T)$. Otherwise the algorithm substitutes the variable *last* for $pa_D^p(v)$.

Duplicate-Detection enumerates all the nodes of rml-occurence list without repetition, if the following two conditions are satisfied: (i) all the elements of $Roc(T)$ are ordered in the preorder of D for any 1-pattern T, and (ii) the algorithm picks up all the nodes of $Roc(T)$ in the order of $Roc(T)$. Details are omitted here and will be given in the full paper.

5 Experimental Results

To evaluate the effectiveness of our algorithm and the pruning techniques, we run several experiments on real-life datasets.

We describe the datasets used in the experiments. The datasets are collections of HTML pages from several Web search site. We listed the URL of the sites in Table 2. We collected HTML pages by giving a set of keywords such as “Honda” or “NP optimization problems” to a search engine and simply collected the cgi-generated HTML pages returned by the search engine.

The dataset *allsites* is a collection of cgi-generated 288 HTML pages of 3.6MB from 30 Web databases and search engine sites. This dataset attempt to model information extraction from heterogeneous sources in internet. This dataset was used for the performance comparison among versions of mining algorithms.

were also collections of cgi-generated HTML pages, but each dataset consists of HTML pages obtained from the same search engines. Thus, these datasets are homogeneous and may contains evident regularities embedded in the HTML pages. These datasets include *altavista, google, WebCrawler, lycos, fast, argos*. These five datasets contains the list of the search results with title, summary, URL, related pages and so on. These datasets were used for the performance evaluation varying the minimum support.

The last dataset, *citeseers*, is the largest collection of 5.6MB and contains the list of bibliographic data in HTML pages for online research papers consisting of title, authors, abstracts, sources, citations, and other bibliographic informations. This dataset was used for the scale-up experiments.

Table 3 shows the parameters of these datasets. The first four entries show the size of the HTML pages, the number of pages, the number of nodes in the corresponding data tree $\mathcal{D}$, the number of distinct labels/HTML tags. The next three entries show the average, the variance, and the maximum of the branching factor of the data tree. The last three entries show the average, the variance, and the maximum of the depth of the data tree.

All experiments were performed on PC (PentiumIII 600MHz, 512 megabytes RAM, linux 2.2.14). The algorithms were implemented in Java (SUN JDK1.3.1, JIT compiler) with a DOM library (OpenXML).

We implemented the following three variations of the basic algorithm Find-Freq-Trees of Fig. 6 combined with the pruning methods described in Section 4.

- *basic* : Find-Freq-Trees + Scanning-Siblings
- *middle* : *basic* + the pruning using 1-patterns and 2-patterns
- *full* : *middle* + Duplicate-Detection

The algorithm *basic* is the base algorithm equipped with the optimization Scanning-Siblings of Fig. 12 in Subsection 4.1. The algorithm *middle* is *basic* equipped with pruning techniques with

Table 3: datasets

dataset $\mathcal{D}$	size			tag	branching			depth		
	pagesize (byte)	$\#\Delta$	$\|\|\mathcal{D}\|\|$	$\#\mathcal{L}$	ave.	var.	max.	ave.	var.	max.
allsites	3,633,255	288	72,317	53	2.10	11.18	571	9.10	6.37	96
altavista	2,336,474	100	69,726	32	2.31	4.47	100	8.67	1.29	12
google	1,264,134	100	51,742	25	2.33	3.24	100	8.99	2.34	18
WebCrawler	1,478,728	100	36,623	29	1.91	2.38	100	10.78	2.01	18
lycos	2,399,679	100	61,591	43	1.96	4.80	601	10.30	2.48	17
fast	1,269,455	100	25,750	29	1.99	3.84	200	11.32	2.94	16
argos	1,312,475	100	45,440	24	2.17	4.80	100	8.69	1.48	14
citeseers	5,614,548	180	144,356	22	–	–	–	–	–	–

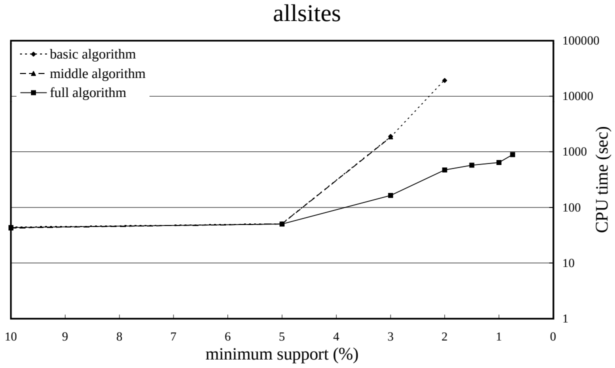

Figure 13: the efficiency comparison of the three algorithms (the case $\sigma = 2.0(\%)$ of *middle* are not recorded.)

1-patterns and 2-patterns in Subsection 4.2 and Subsection 4.3. Finally, the third algorithm *full* will be the fastest one equipped with all optimization, namely, with Duplicate-Detection in Section 4.4.

5.1 Basic Algorithm vs. Improved Versions

Fig. 13 compare the performance of the basic algorithm *basic* and its variants *middle* and *full* on the heterogeneous dataset *allsites*. The y-axis is logarithmically scaled. At the points of $\sigma \leq 5\%$, there is no distinction on the performance is observed. This is because up to this point, the maximum stage of the computation is at most two and thus the optimization does not make any effect. On the other hand, at some data points with σ larger than 5%, the fully optimized version *full* is over order of magnitude faster than *basic* and *middle*. At these points, the maximum stages were from 5 to over 10, and duplicate detection works well. At the data points with *sigma* larger than 2%, we stopped to run *basic* and *middle* since the running time exceeds several hours.

Consequently, the duplicate detection technique in Subsection 4.4 is quite useful on this dataset, which mimics heterogeneous sources. The speed-up of *middle* by pruning with 1-pattern and 2-patterns against *basic* is not observed.

5.2 Performance Comparison

Fig. 14 shows the execution times for the six real-world datasets from Web by varying the values of minimum support from 5% to 0.25% at minimum, where each dataset contains a homogeneous collection of cgi-generated Web pages collected from the same search engine site. We used the full optimized version *full* as a mining algorithm.

We can see trends of our algorithm on a variety of datasets. As in the case for itemset discovery [4], the execution time increases as the minimum support decreases. Although these six datasets have similar characteristics in sizes, the number of nodes, the branching factor, and the depth seen in Table 3, the detailed trends were quite different.

To indicates more detailed characteristics of the datasets, in each plot, we also show the number of

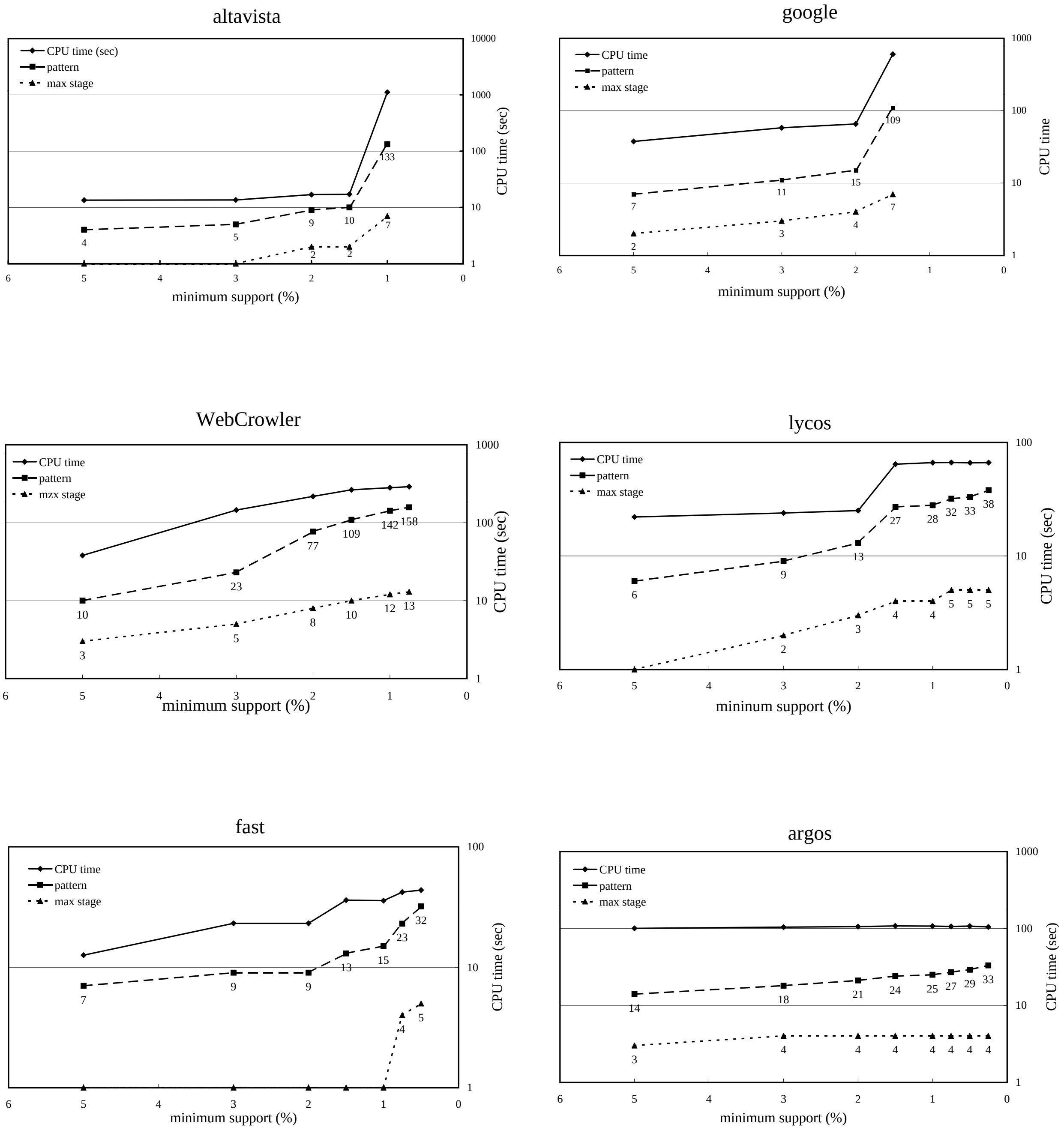

Figure 14: Main experiment: the execution time of each dataset

the maximum stage, this equals the maximum size of the frequent patterns, (dotted lines). Then, we observed that the number of frequent patterns and the cpu time were mainly dominated by the maximum stage or the maximum size of the frequent patterns in these datasets.

5.3 Performances for Each Stage

To give more detailed look on the execution of our algorithm, we show in Fig. 15 the trends of the cpu time, the number of frequent patterns, and the average number of the rightmost occurrences per pattern as the stage proceeds. The datasets were *allsites* with $\sigma = 1.5\%$ used in the experiment in Subsection 5.1, *google* with $\sigma = 1.5\%$ and *WebCrawler* with $\sigma = 0.75\%$. In allsites and WebCrawler, the number of frequent patterns decreases as the stage proceeds, while in google the number slightly increases and falls before the algorithm terminates. The cpu times are monotonically decreasing in all datasets.

5.4 Scale-up Experiment

Fig. 16 shows the scale-up property of the FIND-FREQ-TREES with all optimization technique (*full*). The dataset was *citeseers*, which is the collections of HTML pages containing bibliographic records of online papers of 5.6MB. We increases the number of HTML pages from 5 pages (316KB) to 180 pages (5,615KB) with the fixed minimum support $\sigma = 2\%$. We can observe that the execution time scales almost linearly. Similar scale-up properties were observed for other datasets.

For example, at the data point of 100 pages (4,000KB), the data tree contains 102,392 nodes. Then, the algorithm found 34 frequent patterns with maximum size 6 (nodes) in about 30 minutes. Below, we show three patterns among 34 patterns found by the algorithm, where the label P, B, FONT and #TEXT stand for the tags for paragraphs, bold fonts, font specification (color in this case) and text nodes, respectively.

- {P : { B, B : {FONT : {#TEXT}}}}
- {P : {#TEXT, B, #TEXT, #TEXT, #TEXT}}

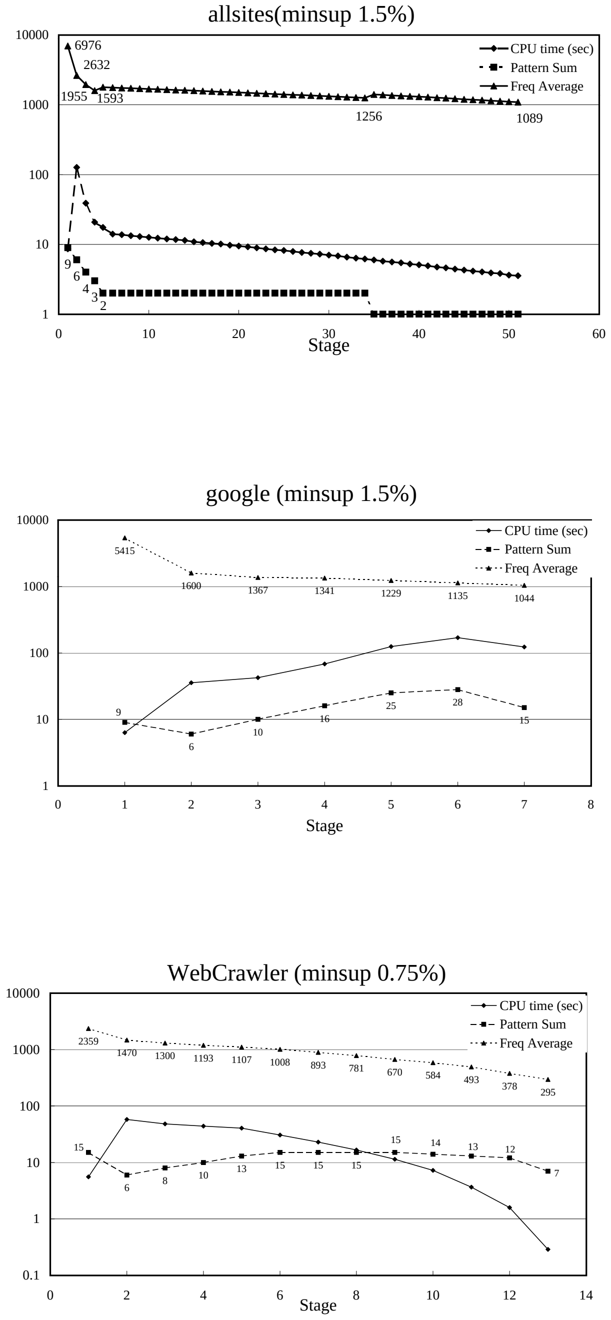

Figure 15: Performance at each stage

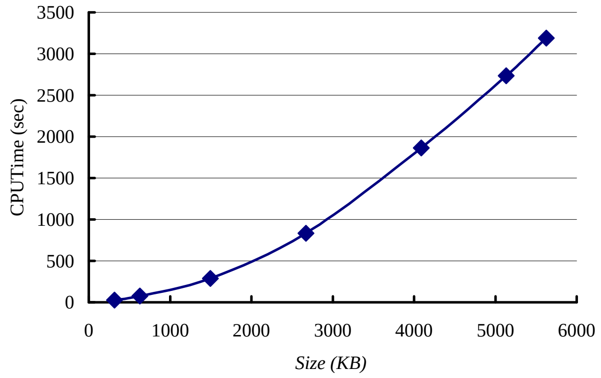

Figure 16: the scale-up experiment

- {P : {#TEXT, #TEXT, B, #TEXT, #TEXT}}

6 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.

Optimized pattern mining is to find those patterns that optimize a given statistical measure and attracting much attention in both data mining and machine learning communities [13, 18]. We have devised fast and robust mining algorithm for finding simple text patterns [8, 12] and it was shown that the optimized pattern discovery is effective in text mining in ill defined environment. Thus, it is our future problem to develop optimized pattern discovery algorithm for tree-structured data by extending our framework.

In this paper, we consider the mining problem from semi-structured data in a simplest model, and ignore the complex components such as attributes and texts. It will be interesting to extend the proposal algorithm to deal with texts and attributes of Web pages or XML documents.

Hiroki Arimura would like to express his sincere thanks to Heikki Mannila and Esko Ukkonen to direct his attention and introduce to this area of research as his stay in Helsinki in 1996. He also would like to thank Shinichi Morishita for his warm encouragements and fruitful discussion on Web Mining with him. The authors would like to thank Saturu Miyano, Akihiro Yamamoto, Masayuki Takeda, Ayumi Shinohara, and Shinichi Shimozono for the comments and advice to this work.

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

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

[3] R. C. Agarwal, C. C. Aggarwal, V. V. V. Prasad, Depth first generation of Long Patterns, In *Proc. SIGKDD'00*, 108–118, ACM, 2000.

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

[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, Proc. the 12th International Conference on Algorithmic Learning Theory (ALT'01), LNAI, Washington, D.C., Nov. 2001. (To appear)

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

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

[10] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, a Query Language and Optimization Techniques for Unstructured Data, In *Proc. SIGMOD'96*, pp.505–516, 1996.

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

[12] R. Fujino, H. Arimura, S. Arikawa, Discovering unordered and ordered phrase association pat-

LNAI 1805, 281–293, 2000.

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

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

[15] Kosaraju, S. R., Efficient tree pattern matching, In *Proc. 30th FOCS*, pp.178–183, 1989.

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

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

[18] S. Morishita, On classification and regression, In *Proc. Discovery Science '98*, LNAI 1532, 49–59, 1998.

[19] 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, 2001. (To appear)

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

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

[22] W3C, XQuery 1.0: An XML Query Language, *W3C Working Draft*, 07 June 2001.
`http://www.w3.org/TR/xquery`

[23] W3C, XML schema Part0: Primer, *W3C Recommendation*, 02 May 2001.
`http://www.w3.org/TR/xmlschema-0`

[24] J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, C.-Y. Chang, Automated Discov-

Structures, In *Proc. KDD-96*, pp.70–75, 1996.

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