Online algorithms for mining semi-structured data stream

Asai, Tatsuya
Department of Informatics, Kyushu University

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

Abe, Kenji
Department of Informatics, Kyushu University

Kawasoe, Shinji
Department of Informatics, Kyushu University

他

出版情報：Proceedings of 2002 IEEE International Conference on Data Mining, pp.27-34, 2002. IEEE
バージョン：

Online Algorithms for Mining Semi-structured Data Stream

Tatsuya Asai Hiroki Arimura† Kenji Abe Shinji Kawasoe Setsuo Arikawa
Department of Informatics, Kyushu University, Fukuoka 812–8581, JAPAN
{t-asai,arim,k-abe,s-kawa,arikawa}@i.kyushu-u.ac.jp
† PRESTO, JST, JAPAN

Abstract

In this paper, we study an online data mining problem from streams of semi-structured data such as XML data. Modeling semi-structured data and patterns as labeled ordered trees, we present an online algorithm StreamT *that receives fragments of an unseen possibly infinite semi-structured data in the document order through a data stream, and can return the current set of frequent patterns immediately on request at any time. A crucial part of our algorithm is the incremental maintenance of the occurrences of possibly frequent patterns using a tree sweeping technique. We give modifications of the algorithm to other online mining model. We present theoretical and empirical analyses to evaluate the performance of the algorithm.*

1. Introduction

Recently, a new class of data-intensive applications such as network monitoring, web site management, sensor networks, and e-commerce emerged with the rapid growth of network and web technologies. In these applications, the data are modeled not as static collections but as transient *data streams*, where the data source is an unbounded stream of individual data items, e.g., transaction records or web page visits, which may arrive continuously in rapid, time-varying way [18].

Particularly in data communication through internet, it is becoming popular to use *semi-structured data*-based communication technologies [2], e.g., SOAP [19], to send heterogeneous and ill-structured data through networks. Since traditional database technologies are not directly applicable to such data streams, it is important to study efficient information extraction methods for semi-structured data streams.

In this paper, we model such semi-structured data streams by sequences of *labeled ordered trees*, and study the frequent pattern discovery problem in online setting. We model a semi-structured data stream as an infinite sequence of the nodes generated by the depth-first scanning of a possibly infinite data tree. An online algorithm has to continuously work on the data stream, and has to quickly answer queries on request based on the portion of the data received so far. This formulation captures typical situations for web applications reading a sequence of XML tags or SAX events element by element from a data stream. Since this is a finest-grained online model, the results of this paper can be easily generalized to coarser-grained models where, e.g., XML documents are processed page by page.

We present an online algorithm StreamT for discovering labeled ordered trees with frequency at least a given minimum threshold from an unbounded data stream. A difficulty lies in that we have to continuously work with unbounded data streams using only bounded resources. A key idea is a technique of sweeping a branch, called the *sweep branch*, over the whole virtual data tree to find all embeddings of candidate patterns intersecting it. Using this sweep branch as a synopsis data structure, we achieve incremental and continuous computation of all occurrences of patterns with bounded working space.

As another idea, we adopt a candidate management policy similar to Hidber [11] for online association mining to limit the number of candidate patterns as small as possible. We also use the enumeration technique for labeled ordered trees that we recently proposed in [4], a generalization of a technique by Bayardo [6]. Combining these ideas, our algorithm StreamT works efficiently in both time and space complexities in online manner. Furthermore, we extend our algorithm to the *forgetting model* of online data stream mining, where the effect of a past data item decays exponentially fast in its age. We also give theoretical analysis on the accuracy of the discovered patterns as well as an empirical analysis on the scalability of the algorithms.

The rest of this paper is organized as follows. In Section 2, we give basic definitions, and in Section 3, we present our online algorithm. In Section 4, we modify this algorithm in the forgetting model. In Section 5, we report experimental results, and in Section 6, we conclude. For proofs not found here, see [5].

1.1. Related Works

Emerging technologies of semi-structured data have attracted wide attention of networks, e-commerce, information retrieval and databases [2, 19]. In contrast, there

0-7695-1754-4/02 $17.00 © 2002 IEEE

have not been many studies on *semi-structured data mining* [1, 4, 7, 9, 13, 15, 16, 20, 22]. There are a body of researches on online data processing and mining [10, 14, 18]. Most related work is Hidber [11], who proposed a model of continuous pattern discovery from unbounded data stream, and presented adaptive online algorithm for mining association rules. Parthasarathy *et al.* [17] and Mannila *et al.* [14] studied mining of sequential patterns and episode patterns. Yamanishi *et al.* [21] presented an efficient online-outlier detection system *SmartSifter* with a forgetting mechanism.

Zaki [22] and Asai *et al.* [4] independently developed efficient pattern search techniques, called *rightmost expansion*, for semi-structured data, which is a generalization of the set-enumeration tree technique [6]. Although our algorithm partly uses this technique, its design principle is different from previous semi-structured data mining algorithms [4, 7, 9, 13, 15, 16, 20, 22]

2. Preliminaries

2.1. Model of Semi-Structured Data

Semi-structured data are heterogeneous collections of weakly structured data [2], which are typically encoded in a markup language such as XML [19]. We model semi-structured data and patterns [2] by labeled ordered trees. For the basic terminologies on trees, we refer to, e.g. [3].

Labeled Ordered Trees. We define labeled ordered trees according to [4, 12]. Let $\mathcal{L} = \{\ell, \ell_0, \ell_1, \ldots\}$ be a fixed alphabet of *labels*. Then, a *labeled ordered tree on* $\mathcal{L}$ (*tree*, for short) is a labeled, rooted, connected directed acyclic graph $T = (V, E, B, L, v_0)$ with the following properties [3]. Each node $v \in V$ of T is labeled by an element $L(v)$ of $\mathcal{L}$, and all node but the root v_0 have the unique parent by the child relation $E \subseteq V^2$. For each node v, its children are ordered from left to right by an indirect sibling relation $B \subseteq V^2$ [3]. Note that the term *ordered* means the order *not* on labels but on children.

The size of a tree T is the number of its nodes $|T| = |V|$. Throughout this paper, we assume that a tree of size k has the node set $V = \{1, \ldots, k\}$ and the nodes are ordered in the pre-order by the depth-first search order on T. We refer to the node i as the *i-th node* of T. This assumption is crucial in our discussion. By this assumption, the root and the rightmost leaf of T, denoted by $root(T)$ and $rml(T)$, are always 1 and k, respectively. For a tree T of size k, the *rightmost branch* of T, denoted by $RMB(T)$, is the path from the root 1 to the rightmost leaf k of T.

We denote by $\mathcal{T}$ the class of all labeled ordered trees on $\mathcal{L}$. We also refer to V, E, B and L as V_T, E_T, B_T and L_T, respectively, if it is clear from context.

Matching and Occurrences. Next, we define the notion of *matching* between two labeled ordered trees T and D. A *pattern tree* T *matches* a *data tree* D if T can be embedded in D with preserving the labels, the (direct) child relation, the (indirect) sibling relation by a non-collapsing mapping, that is, there exists some function $\varphi : V_T \to V_D$ that satisfies the following (i)–(iv) for any $v, v_1, v_2 \in V_T$:

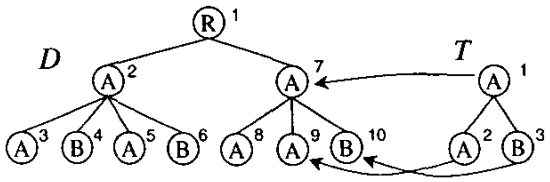

Figure 1. A data tree D and a pattern tree T on the set $\mathcal{L} = \{A, B\}$ of labels

(i) φ is one-to-one.

(ii) $L_T(v) = L_D(\varphi(v))$.

(iii) $(v_1, v_2) \in E_T$ iff $(\varphi(v_1), \varphi(v_2)) \in E_D$.

(iv) $(v_1, v_2) \in B_T$ iff $(\varphi(v_1), \varphi(v_2)) \in B_D$.

Then, we say that φ is a *matching function of* T *to* D, or T *occurs in* D. We assume the *empty tree* $\perp$ such that $|\perp| = 0$ and $\perp$ matches to any tree at any node. An *embedding* of T in D w.r.t. φ is the image $\varphi(T) \subseteq V_D$ of T into D, whose induced subgraph is isomorphic to T. We define the *root occurrence* and the *rightmost leaf occurrence* of T in D w.r.t. φ by the nodes $\varphi(1)$ and $\varphi(k)$ of D to which the root and the rightmost leaf of T map, respectively. If φ is not irrelevant then we simply omit φ.

For example, Fig. 1 shows examples of labeled ordered trees D and T on $\mathcal{L} = \{A, B\}$. We see that the pattern tree T matches the data tree D, where the matching is designated with a set of arrows from T to D. The root occurrences of T in D are 2 and 7, while the rightmost occurrences are 4, 6, and 10.

Semi-structured Data Streams. Let D be a labeled ordered tree, called a *data tree* with finite depth and possibly infinite width. Given a collection of trees as a data source, we always treat them as a single tree by combining trees with appending the imaginary common root. Recall that the nodes of D are numbered in the preorder traversal of D.

We introduce a convenient sequential representation of labeled ordered trees. The *depth* of node v of tree T is the number of edges on the path from the root to v. The *depth-label representation* of a node v of D is the pair $(d, \ell) \in \mathbb{N} \times \mathcal{L}$ of the depth d and the label ℓ of v. Then, a data tree D is encoded as the sequence $\pi = ((d_1, \ell_1), (d_2, \ell_2), \ldots)$ of depth-label pairs corresponding to the nodes on the preorder traversal of T. This depth-label representation π also linearly related to the *open-close parentheses* representation as in XML [19].

Conversely, we can uniquely decode a depth-label representation π into a labeled ordered tree as follows.

Definition 4 ([4, 22]) Let S be a tree of size $k \geq 1$. Then, a *rightmost expansion* of S is any tree T of size $k+1$ obtained from S by (i) attaching a new node w with a label in $\mathcal{L}$ as a child of a parent node p on the rightmost branch of S so that (ii) w is the rightmost sibling of p. Then, we say that T is a *successor* of S, or S is a *predecessor* of T. If the depth and the label of w is $d \in \mathbf{N}$ and $\ell \in \mathcal{L}$, resp., then T is called the (d, ℓ)-*expansion* of S. The $(0, \ell)$-expansion of $\perp$ is the single node tree with label ℓ.

Thus, the empty sequence ε transforms to the empty tree $\perp$, and if the sequence π transforms to a tree S, then the sequence $\pi \cdot (d, \ell)$ to the (d, ℓ)-expansion of S. The notion of depth-label representation is motivated by the tree expansion technique [4, 22], and plays an important role in the following discussion.

For example, in the previous example of Fig. 1, the data tree D transforms to the depth-label representation $\pi = (0, R), (1, A), (2, A), (2, B), (2, A), (2, B), (1, A), (2, A), (2, A), (2, B)$, and *vice versa*.

We model a semi-structured data stream as an infinite sequence of the nodes generated by the depth-first scanning of a possibly infinite data tree as follows. For a set A, we denote by A^∞ the sets of all infinite sequences on A. A *semi-structured data stream* for D is an infinite sequence $\mathcal{S} = (\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_i, \ldots) \in (\mathbf{N} \times \mathcal{L})^\infty$, where for every $i \geq 1$, the i-th element $\boldsymbol{v}_i = (d_i, \ell_i)$ is the depth-label representation of the i-th node $v_i = i$ of D. Then, $\boldsymbol{v}_i$ is called the i-th *node* of $\mathcal{S}$ and i is called the *time stamp*. The *i-th left-half tree*, denoted by D_i, is the labeled ordered tree that is the induced subgraph of D consisting of the first i nodes $(v_1, v_2, \ldots, v_i)$ of the traversal.

Online Data Mining Problems. Now, our online data mining problem is stated as follows. The definition of the frequency of a pattern T at time i will be specified later.

Definition 5 (Online Frequent Pattern Discovery from Semi-structured Data Streams) Let $0 \leq \sigma \leq 1$ be a non-negative number called the *minimum support*. In our online mining protocol, for stages $i = 1, 2, \ldots$, an *online mining algorithm* $\mathcal{A}$ iterates the following process: $\mathcal{A}$ receives the i-th node $\boldsymbol{v}_i$ from the stream $\mathcal{S}$, updates its internal state based on the first i nodes $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_i$ received so far, and then on request by a user $\mathcal{A}$ reports a set $\mathcal{F}_i$ of frequent patterns that appears in D_i with frequency no less than σ.

The goal of an online algorithm is to continuously work on unbounded stream for arbitrary long time with bounded resources, and to quickly answer user's queries at any time.

We define the models of the frequency of patterns as follows. Let $i \geq 1$ be any time. For every $1 \leq j \leq i$, we define the indicator function $hit_j^{(i)}(T) = 1$ if the pattern T has a root occurrence at the node v_j in D_i. Otherwise, we define $hit_j^{(i)}(T) = 0$. For a technical reason, we require not only $\varphi(1)$ but also the whole $\varphi(T)$ to be contained in D_i.

Definition 6 Let $\mathcal{S}$ be a given semi-structured data stream and $T \in \mathcal{T}$ be any pattern. Below, $count_i(T)$ and $freq_i(T)$ denote the *count* and the *frequency* of T at time i, resp.

- **Online Model (OL).** In this model motivated by Hidber [11], we count the number of distinct root occurrences of T in D_i. The *frequency* of T at time i is:

$$freq_i(T) = \tfrac{1}{i} count_i(T) = \tfrac{1}{i} \textstyle\sum_{j=1}^{i} hit_j^{(i)}(T)$$

- **Forgetting Model (FG).** In the forgetting model, e.g., [21], the contribution of the past event decays exponentially fast. For positive number $0 < \gamma < 1$ called a *forgetting factor*, the *frequency* of T is defined by:

$$freq_{\gamma,i}^{\mathrm{fg}}(T) = \tfrac{1}{Z_i} \textstyle\sum_{j=1}^{i} \gamma^{i-j} hit_j^{(i)}(T). \quad (1)$$

Although we used a simplified normalization factor $Z_i = i$ instead of a more precise one $Z_i = \sum_{j=1}^{i} \gamma^{i-j}$, most of the discussion in the later sections also holds.

A difference of above models is the speed of decay. Since the effect of a past event decays exponentially faster in FG than in OL, the former is more trend conscious than the latter. We can deal with the *sliding window model* in this framework in a similar manner. For details, see [5].

3. Online Mining Algorithms

In this section, we present an online algorithm StreamT for solving the online frequent pattern discovery problem from semi-structured data stream.

3.1. Overview of the Algorithm

In Fig. 2, we show our algorithm StreamT in the online model. Let $\mathcal{S} = (\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_i, \ldots) \in (\mathbf{N} \times \mathcal{L})^\infty$ be a possibly infinite data stream for a data tree D. Through the stages $i = 1, 2, \ldots$, StreamT receives the i-th node $\boldsymbol{v}_i = (d_i, \ell_i)$ from $\mathcal{S}$, updates a pool $\mathcal{C} \subseteq \mathcal{T}$ of candidate patterns and the internal state, and on request reports a set of frequent labeled ordered trees $\mathcal{F}_i \subseteq \mathcal{T}$ with frequency no less than a given threshold $0 \leq \sigma \leq 1$.

To continuously compute the set of frequent patterns on an unbounded stream, the algorithm uses a technique, similar to *plane sweeping* in computational geometry [8], to find all root occurrences of candidate patterns in D. A basic idea of our algorithm is explained as follows. To detect all embeddings of a set of patterns in D, we sweep a path from the root to the currently scanned node v_i, called the *sweep branch*, rightwards over the data tree D by increasing the stage $i = 1, 2, \ldots$. While we sweep the plane, we keep track of all embedding of patterns that intersect the current sweep branch.

Algorithm StreamT

Input: A set $\mathcal{L}$ of labels, a data stream $(v_1, v_2, \ldots, v_i, \ldots)$ of a data tree D, and a frequency threshold $0 \leq \sigma \leq 1$.

Output: A sequence $(\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_i, \ldots)$ of sets of frequent patterns, where $\mathcal{F}_i$ is the set of frequent patterns for every i.

Variables: The candidate pool $\mathcal{C} \subseteq \mathcal{T}$, and the bucket stack $B = (B[0], \ldots, B[Top])$.

Method:

1. $\mathcal{C}$:= the class of all single node patterns; $B := \emptyset$ and $Top = -1$; $i := 1$;
2. While there is the next node $v_i = (d, \ell)$, do the followings:
   (a) Update the bucket stack $B[0] \cdots B[d-1]$:
   $(B, EXP) :=$ UpdateB$(B, \mathcal{C}, (d, \ell), i)$;
   (b) Update the candidate pool $\mathcal{C}$ and the bucket $B[d]$:
   $(B, \mathcal{C}) :=$ UpdateC$(EXP, B, \mathcal{C}, (d, \ell), i)$;
   (c) Output the set $\mathcal{F}_i = \{ T \in \mathcal{C} \mid freq_i(T) \geq \sigma \}$ of frequent patterns; $i = i + 1$;

Figure 2. An online mining algorithm for semi-structured data stream

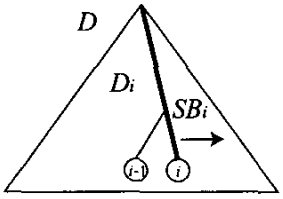

Figure 3. The i-th left-half tree D_i and i-th sweep branch SB_i for the data tree D

The algorithm incrementally maintains the following data structures during the computation.

- A set $\mathcal{C} \subseteq \mathcal{T}$ of patterns, called the *candidate pool*.
- A stack $B = (B[0], B[1], \ldots, B[Top])$ of buckets, called the *sweep branch stack* (*SB-stack*, for short).

For each candidate $T \in \mathcal{C}$, the following features are associated: A counter $count(T)$ of the root occurrences of T in D_i. A vector $Rto_T = (Rto_T[0], Rto_T[1], \ldots)$ of the latest root occurrences $Rto_T[d] = \rho$ of T with depth d.

3.2. Incremental Pattern Discovery Using Tree Sweeping

To keep track of all embeddings of candidate patterns, we do not need the whole information on them. Instead, we record the information on the intersections of these embedding and the current sweep branch at every stage.

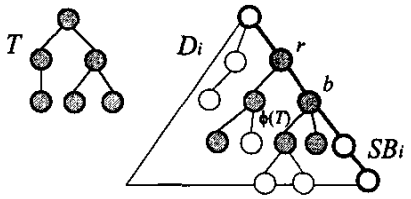

Figure 4. The root and the bottom occurrences r and b of pattern T on D_i w.r.t the sweep branch SB_i with matching ϕ.

Let $i \geq 1$ be any stage. In what follows, we denote by v_i, D_i and SB_i the present node, the left-half tree and the sweep branch at stage i. In other words, SB_i is the rightmost branch of D_i.

For pattern T, let $\varphi(T)$ be an embedding of T with some matching $\varphi : V_T \to V_D$ of T to D_i. Since an embedding of a tree is also an ordered tree in D_i, we can define the rightmost branch, denoted by $RMB(\varphi(T)) \subseteq V_D$, of $\varphi(T)$ in D_i. During the scan of D, the sweep branch $SB_i \subseteq V_D$ may have nonempty intersection $SB_i \cap RMB(\varphi(T))$ with $RMB(\varphi(T))$.

Lemma 1 *For any embedding $\varphi(T)$ of a pattern T and the sweep branch SB_i, the intersection $SB_i \cap RMB(\varphi(T))$ is a consecutive path in D.*

From the above lemma, we define the *root* and the *bottom occurrences* of T w.r.t. φ to be the highest and the lowest nodes in the intersection $SB_i \cap RMB(\varphi(T))$ (Fig. 4). We can easily see that if the intersection $SB_i \cap RMB(\varphi(T))$ is contained in SB_i then the corresponding bottom occurrence becomes the rightmost occurrence of T w.r.t. φ. The next lemma gives an incremental characterization of the rightmost occurrences, which enables us to detect all rightmost occurrences of candidate patterns by maintaining all bottom occurrences of their predecessors on the sweep branch using the SB-stack.

Lemma 2 *Let $T \in \mathcal{T}$ be any pattern of size $k > 1$. At every time $i \geq 1$, T has a rightmost occurrence at the current node v_i in D_i iff there exists some pattern S of size $(k-1)$ that has a bottom occurrence at the parent of v_i in D_i and such that T is the (d, ℓ)-expansion of S, where d is the depth of the rightmost leaf k of T from its root and $\ell = L(v_i)$ is the label of v_i. This is also true even if $k = 1$.*

To implement this idea, we use the sweep branch stack $B = (B[0], B[1], \ldots, B[Top])$ to record the intersections of embeddings of patterns with the current sweep branch SB_i. $Top \geq 0$ is the length of SB_i. Each bucket $B[b]$ $(0 \leq b \leq Top)$ contains a set of triples of the form $\tau = (T, r, b)$ such that pattern T has the root and the bottom occurrences

of the depths r and b, respectively, on SB_i. For each bucket $B[d]$, the time stamp $B[d].time \in \mathbf{N}$ of the last time is associated with the bucket.

For any stage $i \geq 1$, the SB-stack $B = (B[0], B[1], \ldots, B[Top])$ is said to be *up-to-date w.r.t.* the present sweep branch SB_i if Top is the length of SB_i, and for every $0 \leq b \leq Top$, the bucket $B[b]$ contains all triples $(T, r, b) \in \mathcal{T} \times \mathbf{N} \times \mathbf{N}$ for some $T \in \mathcal{C}$ and $r \in \mathbf{N}$ such that the pattern T appears in D_i and has the root and the bottom occurrences on SB_i of the depths r and b, respectively (Fig. 4). Then, we also say that each bucket $B[b]$ is up-to-date if no confusion arises. Note that the up-to-date stack is unique at time i. Now, we give a characterization of the contents of the current SB-stack B_i inductively.

Lemma 3 *Let $i \geq 1$ and $v_i = (d, \ell)$ be the current node of the data stream $\mathcal{S}$ for D. Let $B_k = (B_k[0], B_k[1], \ldots, B_k[Top_k])$ be the SB-stack at time $k \in \{i-1, i\}$. Suppose that both of the SB-stacks B_i and B_{i-1} are up-to-date. Then, the following 1 – 4 hold:*

1. *For any $0 \leq b < d-1$, $\tau \in B_i[b]$ if and only if $\tau \in B_{i-1}[b]$.*
2. *For $b = d-1$, $\tau \in B_i[d-1]$ if and only if either (i) or (ii) below holds:*
   - *(i) $\tau \in B_{i-1}[d-1]$.*
   - *(ii) τ is represented as $\tau = (T, r, d-1)$ for some tuple $(T, r, b) \in B_{i-1}[d] \cup \cdots \cup B_{i-1}[Top_{i-1}]$ such that $r \leq d \leq b$.*
3. *$\tau \in B_i[d]$ if and only if $\tau = (T, r, b)$ and either (i) or (ii) below holds:*
   - *(i) T is the single node tree with the label ℓ.*
   - *(ii) T is the $(d-r, \ell)$-expansion of S for some triple $(S, r, d-1) \in B_i[d-1]$.*
4. *For any $b > d$, $B_i[b]$ is undefined.*

Proof. Case 1, 2, 3(i) and 4 are obvious. For case 3(ii), suppose that E_T is an embedding of T in D and its right-branch embedding intersects SB_i with the bottom depth $b = d$. Then, T has the rightmost occurrence at the current node $v_i = i$. Let C_T is the tree obtained from E_T by removing v_i. Then, C_T is an embedding of the predecessor of T with the rightmost occurrence at the parent, say w, of $v_i = i$. Since the depth of the parent w is $d-1$, the corresponding triple $(S, r, d-1)$ for C_T belongs to $B_i[d-1]$ where T is the $(d-r, \ell)$-expansion of S. □

Fig. 5 illustrates how to update the sweep branch stack B_{i-1} based on Lemma 3. Suppose that we receive the i-th node (d, ℓ) from a data stream. Then, the triples in UNCHANGE buckets, i.e., $B[0] \cup \cdots \cup B[d-1]$, stay unchanged. The buckets in $B[d] \cup \cdots \cup B[Top]$ are partitioned into REMOVE and CHANGE. The triples in REMOVE buckets are discarded, and triples in CHANGE buckets move to the bucket $B[d-1]$. For all triples in $B[d-1]$, we apply the rightmost expansion and then insert obtained expansions into EXP.

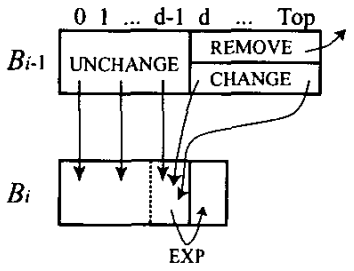

Figure 5. SB-stacks from time $i-1$ to i

Algorithm UpdateB$(B, C, (d, \ell), i)$

Input: A bucket stack $B = (B[0], B[1], \ldots, B[Top])$, a candidate pool C, a depth-label representation $v_i = (d, \ell) \in \mathbf{N} \times \mathcal{L}$ of the ith node of the data stream $\mathcal{S}$, and the current time i;

Output: A set $EXP \subseteq \mathcal{T} \times \mathbf{N} \times \mathbf{N}$ of triples ;

Method:

1. If $d \leq Top$, then do the followings:
   - $BELOW := B[d] \cup \ldots \cup B[Top]$;
   - */* Discard the triples below the branching point */*
     $REMOVE := \{ (T, r, b) \in BELOW \mid r \geq d \}$;
   - */* Collect the triples across the branching point */*
     $CHANGE := \{ (T, r, b) \in BELOW \mid r \leq d-1 \}$;
   - */* Change the bottom occurrences of the triples */*
     $B[d-1] := B[d-1] \cup \{ (T, r, d-1) \mid (T, r, b) \in CHANGE \}$;
2. $EXP := \{(T_\ell, d, d)\}$, where T_ℓ is the single node tree with the label ℓ;
3. For each $(S, r, d-1) \in B[d-1]$ do the followings:
   - T is the $(d-r, \ell)$-expansion of S;
   - $EXP := EXP \cup \{ (T, r, d) \}$;
4. Return (B, EXP);

Figure 6. Updating the SB-stack

In Fig. 6, we show the algorithm UpdateB for incrementally updating the SB-stack. At any stage $i \geq 1$, UpdateB updates the first $d-1$ buckets $B_i[0] \cdots B_i[d-1]$ of new one. The d-th bucket is not immediately updated, but the updated contents are separately returned as EXP for computing the bucket $B_i[d]$ in further processing. The following corollary is immediately from Lemma 3.

Corollary 4 *For every time invoked in the while loop in* StreamT *of Fig. 2 at time $i \geq 1$ with the current node $v_i = (d, l)$, the algorithm* UpdateB$(B, C, (d, \ell), i)$ *returns*

the followings: (i) The sequence $B[0], \ldots, B[d-1]$ of buckets that are up-to-date at time i up to depth $d-1$. (ii) The set EXP of all triples corresponding to the bottom occurrences on SB_i whose depth is d and predecessors belong to C.

3.3. Duplicate Detection for the Root Occurrences

From the observations above, the algorithm UpdateB (Fig. 6) detects all rightmost leaf occurrences of the patterns whose predecessors belong to C at Step 2 and Step 3.

Then, the next step is to compute the corresponding root occurrences from these rightmost occurrences. Let $b \in V_D$ be a rightmost occurrence of pattern T whose triple (T, r, b) is detected at Step 2 or Step 3 of UpdateB. Recall that a list $Rto_T = (Rto_T[0], Rto_T[1], \ldots)$ is associated with each $T \in C$ and it is initially empty. Then, we can obtain the root occurrence corresponding to the triple by the following procedure FindRootOcc:

FindRootOcc$(B, (T, r, b))$

- Let $t := B[r].time$ and $\rho := v_t$;
- If $Rto_T[r] = \rho$ then return UNDEF;
- Otherwise, $Rto_T[r] := \rho$ and return ρ as the root occurrence of T; (Duplicate check)

Figure 7. Finding root occurrences

It is easy to observe that FindRootOcc correctly finds the root occurrence as follows. If the sweep branch SB_i intersects an embedding of T w.r.t. a matching φ then it also intersects the root occurrence of T w.r.t. φ, and thus the component r of (T, r, b) correctly gives the depth of a root occurrence, say, $w \in V_D$. By definition, $B[r].time$ stores the time stamp, say t, of the node on SB_i whose depth is r when it is first encountered. Thus, $v_t = w$ gives the root occurrence corresponding to the triple.

Furthermore for a fixed r, any node w' occupies the bucket $B[r]$ in a consecutive period during the scanning of S. Thus, it is sufficient to record the last root occurrence of depth r for each depth $r \geq 0$ in order to check the duplication of the occurrences. Hence, we see that FindRootOcc correctly detect the root occurrence of candidate patterns without duplicates.

3.4. Candidate Management

The algorithm StreamT stores candidate patterns in a candidate pool $C \subseteq \mathcal{T}$. Fig. 8 shows an algorithm UpdateC for managing C by updating the frequency count of each patterns. A root occurrence has monotonicity that if pattern T is a rightmost most expansion of pattern S then the root count of S is greater than or equal to the root count of T. Based on this observation, the algorithm UpdateC uses a

---

Algorithm UpdateC$(EXP, B, C, (d, \ell), i)$

Input: A set EXP of triples, a bucket stack $B = (B[0], B[1], \ldots, B[Top])$, a candidate pool C, the i-th node $v_i = (d, \ell) \in \mathbf{N} \times \mathcal{L}$ of the data stream, and the time i;

Output: The updated pair (B, C);

Method:

1. */* Increment candidates */*
   For each triple $(T, r, b) \in EXP$, do:
   If $\rho :=$ FindRootOcc(B, (T, r, b)) is not UNDEF then
   - If $T \in C$ then $count(T) := count(T) + 1$;
   - If $T \notin C$ and the predecessor of T is frequent, then $count(T) := 1$ and $C := C \cup \{T\}$;
2. $B[d] := \emptyset$; $Top := d$; $B[d].time := i$;
   $freq(T) := count(T)/i$;
3. */* Delete candidates */*
   For each pattern $T \in C$ and the predecessor of T is infrequent at time i and frequent at time $i-1$,
   - $C = C \backslash \{T\}$;
4. */* Insert candidates in $B[d]$ */*
   For each triple $(T, r, b) \in EXP$,
   - If $T \in C$ then $B[d] := B[d] \cup \{(T, r, b)\}$;
5. Return (B, C);

Figure 8. Updating the candidate pool

candidate management policy similar to Hidber [11], which is summarized as follows.

- **Initialize.** We insert all single node patterns into C. This is done at Step 1 of the algorithm StreamT
- **Increment.** We increment $count(T)$ for all pattern trees $T \in C$ that has the rightmost occurrence at the current node v_i, i.e., $count(T) = count(T) + 1$.
- **Insert.** We insert a pattern of size more than one if its unique predecessor S is already contained in C and becomes frequent, i.e., $freq(S) \geq \sigma$ based on the monotonicity of $freq(S)$ w.r.t. rightmost expansion. This is an on-demand version of the insertion policy of [11]. If some pattern becomes frequent then insert *all* of its successors to the candidate pool.
- **Delete.** We delete a pattern T from C when its unique predecessor P becomes infrequent, i.e., $freq(T) < \sigma$. To be consistent to the initialization and the insertion policy, we do not delete any single nodes. As suggested in [11], we postpone the deletion of the patterns from C until the algorithm requires additional space.

As summary, our algorithm StreamT tries to maintain the *negative border* [17], the set of all patterns that are infrequent but whose predecessors are frequent.

IncFreq(T, i)

- If $hit_i(T) = 1$ then $Fr(T) := \frac{lt(T)}{i-1}\gamma^{i-lt(T)}\, fr(T) + \frac{1}{i}hit_i(T)$ and $lt(T) := i$;
- If $T \notin C$ then $ft(T) := i$;

GetFreq(T, i)

- If $hit_i(T) = 1$ then IncFreq(T, i) and return $Fr(T)$;
- Otherwise, return $\frac{lt(T)}{i-1}\gamma^{i-lt(T)}\, fr(T)$;

Figure 9. Updating and Computing the Frequency

3.5. Time Analysis

We give theoretical analysis of the update time of the algorithm StreamT at stage i. Let B_{i-1} be the previous SB-stack of top Top and $N_j = |B_{i-1}[j]|$ be the number of triples in the j-th bucket. If a node of depth d is received, then UpdateB updates the SB-stack in $O(\sum_{j=d-1}^{Top} N_j)$ time. Then, UpdateC updates pattern pool C in $O(kC + D)$ time, where k is the maximum pattern size, C is the number of candidates in C that occur at the current node v_i by the rightmost leaf occurrence, and D is the number of removed candidates in the stage.

4. Modification to the Forgetting Model

The algorithm StreamT in Fig. 2 is an online algorithm for the online frequent pattern discovery problem in the *online model* of Definition 6. Now we present modification of StreamT to the *forgetting model* also introduced in Definition 6.

Recall that in the forgetting model, the contribution of the past event decays exponentially fast. For a forgetting factor $0 < \gamma < 1$, the frequency of T at time i is given by Eq. 1 in Section 2. At first look, implementation of the forgetting model is as easy as the online model above because they only differ in the definition of the frequency. In the forgetting model, however, the frequency at time i depends on all of the weights γ^{i-j} of past events changing as $i \geq 1$ goes larger. Thus, it seems that an algorithm have to maintain all of these weights at every time. Fortunately, this is not true.

We abbreviate the frequency $freq^{\mathrm{fg}}_{\gamma,i}(T)$ and the event $hit^{(i)}_j(T)$, respectively, to fr_i and hit_j. Below, we give an incremental method to compute the frequency. Let $lt(i) = \max\{\, j \leq i \mid hit_j = 1 \,\}$ be the last time stamp at which T appeared. Then, we have the following lemma.

Lemma 5 *For every $T \in \mathcal{T}$ and every $i \geq 1$, we have the recurrence*

$$\begin{aligned} fr_0 &= 0, \\ fr_i &= \tfrac{lt(i)}{i-1}\gamma^{i-lt(i)}\, fr_{lt(i)} + \tfrac{1}{i}\, hit_i \quad (i > 0) \end{aligned} \tag{2}$$

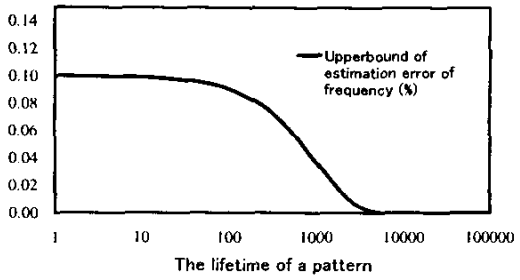

Figure 10. The upper bound of the frequency error against the life time with $\gamma = 0.999$

Proof. We first derive the recurrence for the consecutive steps. Then, derive the recurrence of the lemma by expanding fr_i using $lt(i)$. Since $hit_u = 0$ for any u with $lt(i) < u < i$, the claim immediately follows. □

Now, we present a modified version of StreamT in the forgetting model. We modify the algorithms StreamT and UpdateC (Fig. 8) as follows. StreamT maintains the three parameters $fr(T), ft(T), lt(T)$, the frequency, the first and the last time stamps of the updates for T. UpdateC uses IncFreq to update these parameters whenever a root occurrence of pattern T is detected at Step 1, and Stream uses GetFreq to compute the frequency of T whenever it receives a request at Step 2(c). We can see IncFreq(T, i) and GetFreq(T, i) can be executed in constant time when invoked by these procedures.

Then, we will present an upper bound of the error to estimate the true frequency of a pattern in this model. The *life time* of a pattern T is the period $\Delta i = i - ft(T) \geq 0$. The following theorem says that the error becomes exponentially smaller in the forgetting model as the life time Δi of T goes longer.

Theorem 6 *Let $i \geq 1$, $0 < \gamma < 1$, and $\varepsilon = 1 - \gamma$. For any $T \in C$ with the life time $\Delta i = i - ft(T)$ at time i, the following inequality holds:*

$$\mathsf{GetFreq}(T, i) \leq freq^{\mathrm{fg}}_{\gamma,i}(T) \leq \mathsf{GetFreq}(T, i) + \frac{1}{\varepsilon i}e^{-\varepsilon \Delta i}.$$

Proof. By standard arguments using elementary calculus, Eqs. $1 - \gamma^x \leq 1$ and $(1 - \varepsilon)^x \leq e^{-\varepsilon x}$ $(x \geq 0)$. □

In Fig. 10, we plot the upper bound of the frequency error given by Theorem 6 varying the life time from $\Delta i = 1$ to $\Delta i = 100,000$, where $\gamma = 0.999$ and $i = 1000,000$. A half-value period is about seven thousands for γ.

5. Experimental Results

In this section, we present preliminary experimental results on real-life datasets to evaluate the algorithm

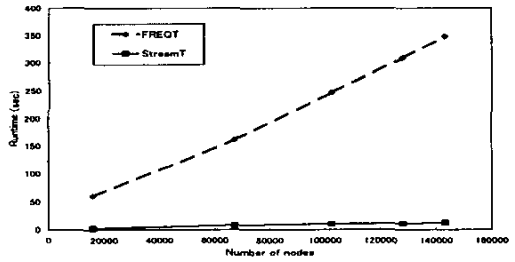

Figure 11. the online scale-up experiment

StreamT. We prepared a dataset *Citeseers* by collecting cgi-generated Web pages from an online bibliographic archive Citeseers [1]. This dataset consists of 189 HTML documents of 5.6MB and its data tree had 143,250 nodes. All experiments were performed on PC (PentiumIII 500MHz, 512MB RAM, Windows2000) and the algorithm was implemented in Java (SUN JDK1.3.1, JIT compiler).

We studied the scalability of StreamT. Fig. 11 shows the running times of the online version StreamT and the offline version FREQT [4] of frequent tree miners with the same frequency threshold $\sigma = 2(\%)$ on the data stream for the dataset *Citeseers* varying the data size from 16K(nodes) to 143K(nodes).

From this experiment, the proposed online algorithm StreamT seems to be much more efficient than the offline algorithm FREQT. However, this is possibly because our algorithm computes approximate answers due to candidate management policy in Sec. 3.4 due to [11] and two algorithms may generate different sets of patterns. Therefore, to compare the performance of those algorithms, we require further research.

6. Conclusion

In this paper, we studied an online data mining problem from unbounded semi-structured data stream. We presented efficient online algorithms that are continuously working on an unbounded stream of semi-structured data with bounded resources, and find a set of frequent ordered tree patterns from the stream on request at any time.

Our labeled ordered trees can be seen as a generalization of serial episodes of Mannila *et al.* [14], and of itemsets and sequential patterns with a pre-processing of data as used for encoding XML-attributes in [4]. Thus, it will be an interesting problem to study the relationship of our algorithms to other online algorithms for classes of patterns such as sequential patterns and episodes.

It is also a future problem to examine the online property of the proposed algorithms using long and trend-changing semi-structured data streams in the real world.

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

Acknowledgments

Hiroki Arimura is grateful to Masaru Kitsuregawa for his direction of the author's interests to online data processing. The authors would like to thank Hiroshi Motoda, Takeshi Tokuyama, Akihiro Yamamoto, Yoshiharu Ishikawa, and Mohammed Zaki for fruitful discussions on web mining.

References

[1] K. Abe, S. Kawasoe, T. Asai, H. Arimura, and S. Arikawa. Optimized Substructure Discovery for Semi-structured Data, In *Proc. PKDD'02*, 1–14, LNAI 2431, Springer, 2002.

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

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

[4] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient Substructure Discovery from Large Semi-structured Data. In *Proc. the 2nd SIAM Int'l Conf. on Data Mining (SDM2002)*, 158–174, 2002.

[5] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Online Algorithms for Mining Semi-structured Data Stream, *DOI Technical Report*, Dept. of Informatics, Kyushu Univ., DOI-TR 211, June 2002. `ftp://ftp.i.kyushu-u.ac.jp/pub/tr/trcs211.ps.gz`

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

[7] G. Cong, L. Yi, B. Liu, K. Wang, Discovering Frequent Substructures from Hierarchical Semi-structured Data, In *Proc. SDM2002*, 175–192, 2002.

[8] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry, Algorithms and Applications, Springer, 2000.

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

[10] P. B. Gibbons and Y. Matias, Synopsis Data Structures for Massive Data Sets, In *External Memory Algorithms*, DIMACS Series in Discr. Math. and Theor. Compt. Sci., Vol. 50, AMS, 39–70, 2000.

[11] C. Hidber, Online Association Rule Mining, In *Proc. SIGMOD'99*, 145–156, 1999.

[12] P. Kilpelainen, H. Mannila, Ordered and Unordered Tree Inclusion, SIAM J. Comput, 24(2), 340–356, 1995.

[13] M. Kuramochi, G. Karypis, Frequent Subgraph Discovery, In *Proc. ICDM'01*, 313–320, 2001.

[14] H. Mannila, H. Toivonen, and A. I. Verkamo, Discovering Frequent Episode in Sequences, In *Proc. KDD-95*, 210–215, AAAI, 1995.

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

[16] T. Miyahara, Y. Suzuki, T. Shoudai, T. Uchida, K. Takahashi, H. Ueda, Discovery of Frequent Tag Tree Patterns in Semistructured Web Documents. In *Proc. PAKDD-2002*, 341–355, 2002.

[17] S. Parthasarathy, M. J. Zaki, M. Ogihara, S. Dwarkadas, Incremental and Interactive Sequence Mining, In *CIKM'99*, 251–258, 1999.

[18] R. Rastogi, Single-Path Algorithms for Querying and Mining Data Streams, In *Proc. SDM2002 Workshop HDM'02*, 43–48, 2002.

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

[20] K. Wang, H. Liu, Discovering Structural Association of Semistructured Data, TKDE, 12(2), 353-371, 2000.

[21] K. Yamanishi, J. Takeuchi, A Unifying Framework for Detecting Outliers and Change Points from Non-Stationary Time Series Data, In *Proc. SIGKDD-2002*, ACM, 2002.

[22] M. J. Zaki. Efficiently mining frequent trees in a forest, In *Proc. SIGKDD-2002*, ACM, 2002.