Efficient Serial Episode Mining with Minimal Occurrences

Hideyuki Ohtani
Graduate School of IST,
Hokkaido University
Sapporo 060-0814, Japan
h-ohtani@ist.hokudai.ac.jp

Takuya Kida
Graduate School of IST,
Hokkaido University
Sapporo 060-0814, Japan
kida@ist.hokudai.ac.jp

Takeaki Uno
National Institute of
Informatics
Tokyo 101-8430, Japan
uno@nii.jp

Hiroki Arimura
Graduate School of IST,
Hokkaido University
Sapporo 060-0814, Japan
arim@ist.hokudai.ac.jp

ABSTRACT

Recently, knowledge discovery in large data increases its importance in various fields. Especially, data mining from time-series data gains much attention. This paper studies the problem of finding frequent episodes appearing in a sequence of events. We propose an efficient depth-first search algorithm for mining frequent serial episodes in a given event sequence using the notion of right-minimal occurrences. Then, we present some techniques for speeding up the algorithm, namely, occurrence-deliver and tail-redundancy pruning. Finally, we ran experiments on real datasets to evaluate the usefulness of the proposed methods.

Categories and Subject Descriptors

H2.8 [**Database Applications**]: Data mining—*sequence mining*; F2.2 [**Nonnumerical Algorithms and Problems**]: Pattern matching—*string algorithms*

General Terms

Algorithms, Performance, Experimentation

Keywords

frequent episode mining, closed sequences, depth-first algorithm

1. INTRODUCTION

1.1 Background

By the rapid increase of species and amount of electronic data on networks, data mining from *semi-structured data*, massive electronic data of new type, has been extensively studied [1]. Particularly, research on sequence mining is one of the most fundamental semi-structured data mining problem, and has attracted much attention for the last years [3, 4, 6, 5, 11]. In sequence mining, we can find characteristic sequences from biological sequences in bioinformatics, or find some useful marketing strategy from purchase histories in business data mining.

ICUIMC-09, January 15-16, 2009, Suwon, S. Korea

1.2 Frequent sequence mining problem

In this paper, we study the *frequent pattern discovery problem* for the class of simple sequence patterns, called *serial episodes* (or *episodes*, for short), which is the problem to find all *frequent patterns* appearing in a given sequence data with frequency more than a specified threshold, called *minsup*. For an alphabet Σ, an *input event sequence* is a sequence $S = S[1]\cdots S[n]$ of events. Given an window width w, a window in S is a consecutive subsequence $W = S[i]\cdots S[i+w-1]$ of S with length w, where $1 \le i \le n-w+1$. Then, a *serial episode* in S is a (non-consecutive) subsequence $\alpha = a_1 \cdots a_m$ $(m \le w)$ appearing in some window W, which means that all events $a_1, \ldots, a_m$ appear in the window in this order. We define the *frequency* of α by the number $fr(\alpha, S)$ of all windows containing α. Given a parameter, called *minimum frequency threshold* $1 \le \sigma \le n$, we say that an episode α is *frequent* if $fr(\alpha, S) \ge \sigma$.

From practical view, it is desirable that an algorithm efficiently finds all patterns in short time. However, since we know that there exist exponentially many distinct patterns in an input sequence of length n, we consider the output-sensitive computational complexity. Therefore, the goal of this paper is to present a mining algorithm that solve the frequent sequence mining problem efficiently in the sense of both running time and memory consumption.

1.3 The purpose of this paper

In this paper, we study the efficient algorithm for mining frequent serial episodes from sequence data. First, we present an depth-first episode mining algorithm DFS-MO, which is a basic version of computer program LCM_seq [9]. This algorithm DFS-MO uses depth-first search based on pattern-growth approach for enumeration of candidate episodes.

We then apply to the episode mining algorithm above the following three speed-up techniques, two of which were originally introduced to an efficient itemset mining algorithm LCM [8] and the last one is newly introduced in this paper:

- *Incremental update of the right-minimal occurrence intervals*: This technique enables simple and efficient update of frequencies of episodes.
- *Occurrence deliver*: This technique contributes to avoid redundant extensions by events that do not contribute to the growth of the current pattern.

- *Pruning by closed episode discovery*: This technique contributes to avoid enumeration of a subset of semantically equivalent episodes that have the same rightmost minimal occurrence intervals in an input sequence.

To evaluate the above speed-up techniques, we implemented the proposed algorithms and ran experiments on real world datasets. Experimental results show the followings. Firstly, the algorithm with depth-first search and incremental right-minimal occurrence interval showed good performance compared to the others. Secondly, occurrence-deliver is quite useful in reducing running time. Thirdly, the closed episode discovery efficiently finds more detailed patterns only in almost same time to frequent pattern discovery.

Although the occurrence deliver technique was originally introduced for frequent itemset mining, its usefulness for sequence mining has not been proved. By the experiments in this paper, we see that this technique is also useful in sequence mining. On the other hand, efficient mining algorithm for any subclasses of closed serial episodes have not known before. We show that it is possible to build efficient closed episode mining by using *reverse search* as in LCM [10].

An implementation, called LCMseq, of the proposed depth-first episode mining algorithm in C with various extensions is publicly available at the Website [9].

1.4 Related works

Mannila *et al.* [6] gave an efficient episode mining algorithm based on breadth-first search. Pei and Han [5] studied mining of frequent itemset sequences, and presented efficient algorithm PrefixSpan. Wang and Han [11] extended this result for closed sequence mining, and gave an algorithm called BIDE. Uchida *et al.* [7] considered frequent episode mining, and proposed efficient algorithms with depth-first search and minimal occurrence lists for the classes of serial episodes (with window) and serial episodes without windows (i.e., subsequence patterns). They also presented mining algorithm for episodes with approximate sequence matching [7]. Uno and Arimura [4] studied the frequent sequence mining for the class of closed flexible patterns, or VLDC patterns, which are serial episodes without window and with allowing consecutive constant letters. Uno and Arimura [8, 10] studied speed-up techniques for frequent itemset mining, and proposed the occurrence-deliver algorithms, which is used for episode mining in this paper.

1.5 Organization of this paper

This paper is organized as follows. In Section 2, we review basic definitions and concepts in episode mining. In Section 3, we present an efficient depth-first algorithm DFS-MO. In Section 4 and Section 5, we discuss speed-up by occurrence-deliver and closed pattern mining. In Section 6, we show experimental results, and in Section 7, we conclude.

2. PRELIMINARY

In this section, we define some terms and notations. We also denote the frequent episode mining problem.

2.1 Event sequences

In this paper, we assume that just one event occurs at a time. Please refer [6] for the general case.

Let Σ be an alphabet of symbols. We denote by Σ^* and $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ the set of all possibly empty strings and the set of all non-empty finite strings over Σ, respectively. An element $e \in \Sigma$ is called an *event*. An sequence $S \in \Sigma^+$ is called *event sequence*. We denote by $S[i]$ the i-th symbol of S, and denote by $S[i..j]$ the consecutive sequence from i-th to j-th of S, which is called a subsequence of S. We define here i and j are positive numbers, and $i < j$.

Consider an event sequence $S = S[1..n]$ of length n. Let be an integer $1 \leq win \leq n$. Then an *window* is a subsequence $W = S[i..i + win - 1]$ for any $0 \leq i \leq n - win + 1$, and the size of it is called the *width* of the window.

Let $\mathcal{W}(S, win) = \{ W_i = S[i..i + win - 1] : i = 1, \ldots, n - win + 1 \}$ be the set of all windows whose width is *win*. Note that the number of windows in $\mathcal{W}(S, win)$ is equal to $n - win + 1$.

An *interval* on S is a pair of positions $[i, j] \in \mathbf{N}^2$ $(i \leq j)$, and it represents the set of positions $\{i, i + 1, \ldots, j\} \subset \{1, \ldots, n\}$ on S. For two intervals $I = [i, j]$ and $I' = [i', j']$, we say I *is in* I' if $i' \leq i$ and $j \leq j'$, and denote by $I \subseteq I'$. We also denote $I \subset I'$ if $I \subseteq I'$ and $I' \nsubseteq I$.

2.2 Serial episodes and the frequent episode mining problem

An *serial episode* (*episode* for simplicity) is a shorter event sequence that occurs serially in a given longer event sequence, where some other events may occur within the range of the serial episode. Formally, a serial episode is a partial sequence of events $\alpha = a_1 \ldots a_m$ which occurs in a window. We call m the *size* of α, and denote by $m = |\alpha|$. We also denote by C the set of all episodes.

For a window $W_i = W[i..i + w - 1]$, we say the episode α *occurs* in W_i if there exists a sequence of positive integers $i \leq \phi(1) \leq \cdots \leq \phi(k) \leq i + w - 1$ and $a_k = S[\phi(k)]$ for any $k = 1, \ldots, m$.

Given a serial episode α, an event sequence S, and a window width *win*, we define the *number of occurrences* of α in S as

$$fr(\alpha, S, win) = |\{ W \in \mathcal{W}(S, win) : \alpha \sqsubseteq W \}|$$

. Namely, $fr(\alpha, S, win)$ is the number of windows in which α occurs. Consider a positive integer $\sigma > 0$. If $fr(\alpha, S, win) \geq \sigma$, then we say α is *frequent*, and call σ as *minimum frequency threshold*. The problem we have to solve here is to enumerate all the frequent serial episodes in S. We call the set of frequent episodes for S, *win* and σ, the *frequent episode set*, and denote by $\mathcal{F}(S, win, \sigma)$.

Definition 1. *Given event sequence S, window width win, and frequency threshold σ,* the frequent episode mining problem *is to enumerate all the frequent serial episodes $\mathcal{F}(S, win, \sigma)$, where each episodes has to be outputted just once.*

Example: Given $S = ABABBCAD$, $win = 4$, and $\sigma = 2$, $\alpha = ABB$ has two minimal occurrence intervals $[1, 4], [3, 5]$, and occurs in the windows $[1, 4]$, $[2, 5]$, $[3, 6]$ but $[4, 7]$, $[5, 8]$. Therefore α is frequent because $fr = 3$. Moreover the subsequences $\varepsilon, A, AB, B, BB$ of α are also frequent.

3. FREQUENT SERIAL EPISODE ENUMERATION ALGORITHM WITH THE MINIMAL OCCURRENCE

Algorithm DFS-MO(Σ, S, win, σ)

input: Event set Σ, event sequence $S = S[1] \cdots S[n] \in \Sigma^*$, window width win, frequency threshold σ.

output: All episodes whose window widths are w and frequencies $\geq \sigma$.

1: $MO(\varepsilon) = \{ [i, i] : 1 \leq i \leq n \}$;
2: Expand-MO(ε, $MO(\varepsilon)$, S);

Algorithm Expand-MO(α, $MO(\alpha)$, S)

input: Episode α, minimal occurrence set $MO(\alpha)$, event sequence S.

output: All episodes whose window widths are w and frequencies $\geq \sigma$.

1: **if** WinCount($MO(\alpha \cdot e), S) < \sigma$ **then** return;
2: Output the new episode $\alpha \cdot e$;
3: **for all** $e \in \Sigma$ **do**
4: $MO(\alpha \cdot e) = Update(MO(\alpha), e)$;
5: Expand-MO(α, $MO(\alpha \cdot e)$, S);
6: **end for**

Figure 1: Frequent episode mining algorithm with minimal occurrence list

In this section, we present algorithm DFS-MO which solves the frequent episode mining problem for any serial episode classes. The algorithm constructs episodes with *depth-first search* (DFS for short), and calculates their frequencies with *minimal occurrences* (MO for short).

3.1 Outline of the algorithm

We show in Figure 1 the basic frequent episode mining algorithm DFS-MO with DFS and MO. Given a event sequence S and a window width win, a minimal frequency threshold σ, the algorithm searches with DFS for all possible episodes by using a recursive procedure Expand-MO.

The algorithm starts with the empty sequence ϵ. Then suppose that Expand-MO is called with a frequent episode α (called the *parent episode*). Expand-MO builds a *child episode* $\beta = \alpha \cdot c$ by appending the frequent character $c \in \Sigma$ to α. Next it updates the occurrence lists of the parent episode, and newly calculates the occurrence lists of child episodes $MO(\beta, S)$ incrementally. To do fast calculation of occurrence lists, we use subprocedures Update and WinCount, which will be discussed later.

Let $n = |S|$, win, σ,and $k \leq |\Sigma|$ be a size of event sequence, a given window width, a given frequency threshold, and the number of frequent episodes in S, respectively. We call each episode which grows in the algorithm as a *episode pattern* (or pattern for short). Moreover, also let $m = |MO(\alpha, S)| \leq fr(\alpha, S, win) \leq |S|$ be the number of minimal occurrences for the pattern α if no confusion occurs.

3.2 Minimal occurrence list

Let $S = S[1..n] \in \Sigma^*$ be the given event sequence of length n. We say that α occurs in the interval $[s, e]$ if $\alpha \sqsubseteq W[s, e]$ for a window $W = S[s, e]$, and call $I = [s, e]$ the *occurrence* of α.

Mannila *et al.*[6] define the minimal occurrence as follows.

DEFINITION 2. **([6])** *The* minimal occurrence *of a serial episode α is an interval $I = [s, e]$ which satisfies the followings.*

- *α occurs in $W = S[s, e]$.*
- *There is no interval $[s', e']$ such that $[s', e'] \subset [s, e]$.*

We, moreover, propose right-minimal occurrence as follows.

DEFINITION 3. *The* right-minimal occurrence *of a serial episode α is an interval $I = [s, e]$ which satisfies the followings.*

- *α occurs in $W = S[s, e]$.*
- *There is no interval $[s', e']$ such that $s' = s$ and $e' < e$.*

LEMMA 1. *Any minimal occurrence $[s, e]$ of α is a right-minimal occurrence. However the converse proposition does not hold in general.*

LEMMA 2. *For any α, $|MO(\alpha, S)| \leq n = |S|$.*

In the serial pattern discovery algorithms such as PrefixSpan [5], the endpoint e is usually used to represent the occurrence position. In this paper, however, we use the interval $[s, e]$ as the occurrence position since the start position s is also related to the conditions of the right-minimal occurrence.

LEMMA 3. *For any episode α and any $i = 1, \ldots, n - w + 1$, the following two conditions are equivalent.*

1. *The episode α occurs in the i-th window $W_i = S[i..i+win-1]$.*
2. *For a right-minimal occurrence $[s, e]$ of α, $[s, e] \subseteq [i..i + win - 1]$ holds.*

From the above lemma, it is sufficient to store only the minimal occurrences of width not larger than win for calculating the window frequencies. We define *right-minimal occurrence list* of α as the set of intervals $MO(\alpha, S) = \{ [s, e] \mid [s, e]$ is a right-minimal occurrence of α in S and $e - s + 1 \leq win \}$. In the actual processing, all elements $[s, e] \in MO(\alpha, S)$ are sorted in ascending order at first, and stored so that we can access them sequentially.

3.3 Updating right-minimal occurrence list

In the recursive procedure Expand-MO in Figure 1, we discuss below the subprocedure Update in which $MO(\alpha \cdot c, S)$ for $\alpha \cdot c$ is updated.

Figure 2 shows the algorithm Update. Our basic idea of Update is as follows. From the definition of right-minimal occurrence, possible orders between two continuous right-minimal occurrences $MO1 = (s, e)$ and $MO2 = (s', e')$ are:

Algorithm Update($MO(\alpha), c, S, win$)

input: Right-minimal occurrence list $MO(\alpha$, appending event $c \in \Sigma$, event sequence S, window of width win.

output: Right-minimal occurrence list $MO(\alpha \cdot c, S)$ for new episode $\alpha \cdot c$.

```
1:  MO(α · c, S) = ∅;
2:  for all [s, e] ∈ MO(α) do
3:      for j = e + 1, ..., s + win − 1 do
4:          if (S[j] = c) then
5:              MO(α · c) ∪ {[s, j]};
6:              break the inner for-loop;
7:          end if
8:      end for
9:  end for
10: return MO(α · c);
```

Figure 2: Algorithm for updating right-minimal occurrence list

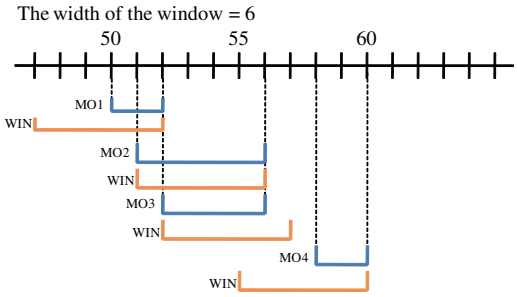

Figure 3: frequent interval

- $s < s'$, $s' < e$, and $e \leq e'$, or
- $s < s'$, $e \leq s'$, and $e < e'$.

For example, in Figure 3 each new events occur 52, 56, 56, 60, where each minimal interval is expanded to them.

For each $MO(\alpha) = \{ [s_i, e_i] : 1 \leq i \leq m \}$ for win, let (s_i, e_i) be the occurrence interval, where new event c is appended to α. Then c has to be searched from the position $e_i + 1$ to $s_i + win - 1$ in order to make a new episode $\alpha \cdot c$. If c is found, record the position into pos, and add (s_i, pos) to $MO(\alpha \cdot e)$. For example, we can see that $MO2$ in Figure 3 will never become a new frequent interval since $MO2$ becomes longer than win.

LEMMA 4. *The time complexity of the algorithm* Update *in Figure 2 is* $O(|MO(\alpha, S)| \cdot win)$.

3.4 Calculating window frequency from right-minimal occurrence list

Algorithm WinCount($MO(\alpha, CS), win$)

input: Right-minimal occurrence list $MO(\alpha, CS)$, window width win.

output: The number of windows where the episode occurs: $count = fr(\alpha, S, win)$.

```
1:  count = 0;
2:  for all mo ∈ MO(α, S) do
3:      w.s = mo.e − win + 1;
4:      w.s = max{w.s, last_mo.s + 1}
5:      while (w.s ≤ mo.s) do
6:          count = count + 1;
7:          w.s = w.s + 1;
8:      end while
9:      last_mo = mo;
10: end for
11: return count;
```

Figure 4: Algorithm for counting the number of windows where the episode occurs

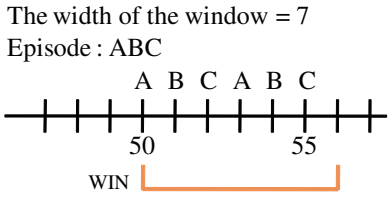

Figure 5: Example for the case of two $[s, e]$ in a window
The episode ABC is counted twice in this window.

In Expand-MO in Figure 1, we discuss a procedure WinCount below. It counts the number of windows where the episode occurs by using updated right-minimal occurrence lists. Figure 4 shows the algorithm WinCount.

Our basic idea is as follows. We first align the window to each right-minimal occurrence $[s, e]$ so that the end of the window corresponds to e. Then, move the window to right while the beginning of the window $w.s$ satisfies $w.s \leq s$. While the movements, increment $count$ because the episode occurs in the window. However, note that $count$ may be incremented many times in one window if there exists two or more episodes which are the same but occur at different positions. (see Figure 5). We need some efforts for preventing this situation.

First, let $MO(\alpha, S) = \{mo_1, \ldots, mo_m\}$. While w is moved, increment $count$ only if w satisfies the followings: (i) $mo_i \subseteq w$ and (ii) $mo_{i-1}.s < w.s$ with respect to current minimal occurrence mo_i. Note that $w.s$ is equal to the start position s of the interval $w = [s, e]$.

We obtain the following lemma since the start position only goes forward.

LEMMA 5. *The time complexity of the algorithm* WinCount *in Figure 4 is* $O(\min\{m \cdot win,\ n\}) = O(n)$, *where* $m = |MO(\alpha, S)|$ *is the number of minimal occurrences of* α.

Namely, the calculation of the window frequency runs in linear time to the input length. Moreover, it takes less time according to the growth of the pattern and decreasing of the number of minimal occurrences. When *win* is variable, it is efficient rather than the naive updating algorithm of $O(m \cdot win) \neq O(n)$ time.

It is summarized as follows.

LEMMA 6. *For a given event sequence* S *and a window width win, the algorithm of* DFS-MO *in Figure 1 can enumerate all frequent episodes and answer in* $O(km \cdot win)$ *time for each episode, where* $k \leq |\Sigma|$ *is the number of frequent events and* $m = |MO(\alpha, S)| \leq n$.

4. OCCURRENCE DELIVER TECHNIQUE

In this section, we present the *occurrence deliver technique*, which is used for speeding up DFS-MO presented in the previous section.

By the existing way [5], we expand frequent serial episode α to $\alpha \cdot e_i$ by adding a frequent symbol $e \in \Sigma$ to its tail. Then, we continue scanning if the expanded episode is frequent after calculating the frequency of it.

On the other hand, we use a hash table that stores pairs of the symbol c that can be added to *alpha* and the right-minimal occurrence lists for it. The hash table $Dict = \{ \langle c, MO_c \rangle : c \in \Sigma, MO_c = MO(\alpha \cdot c, S), |MO_c| \geq \sigma \}$ can be calculated by scanning the event sequence just once.

The above calculation is done in Extend-MO as follows.

Algorithm OccDeliver(α, $MO(\alpha, S), win, \sigma$)

1. Let α be the parent episode. For each minimal occurrences $mo \in MO(\alpha, S)$, scan the event sequence to right until the current scanning position is not over the window width *win* from the end position $mo.e$ of α. While scanning, refer the entry of $Dict[c]$ for a symbol c of each position and record the position to $MO(\alpha \cdot c, S)$. Note that, however, just one symbol can be recorded in a minimal occurrence.

2. After scanning for the parent's minimal occurrence list, read and pop up $MO(\alpha \cdot c, S) = Dict[c]$ for each key $c \in Dict.key$ which is inserted into the hash table. If the entry for c is frequent, that is, $|Dict[c]| \geq \sigma$, call the process Extend-MO recursively.

3. If it isn't frequent, repeat Step 2 for the next key c. If the above is done for all keys, return to process for the parent serial episode.

We have to scan k times to make the expanded episode by the naive way when there are k symbols which is added to the parent episode. By the occurrence deliver technique, we can do it by scanning once.

Let $k \leq |\Sigma|$ be the number of frequent symbols.

Although we see that the time complexity of DFS-MO is $O(k \cdot |MO(\alpha, S)| \cdot win)$ from the lemma 6, the occurrence deliver technique can accelerate it to k-times faster.

THEOREM 1. *For a given event sequence* S *and a window width win, the algorithm* DFS-MO *in Figure 1 can enumerate all frequent episodes and answer in* $O(m \cdot win)$ *time for each episode, where* $m = |MO(\alpha, S)| \leq n$.

Although this technique has an advantage when the number of different events is large, the scanning cannot be stopped when e_i is found while it can be done in the basic DFS-MO. However we can reduce the time for scanning by preserving **e** and jumping to the next minimal occurrence when all elements in **e** occurred.

5. MINING CLOSED SERIAL EPISODES

In this section, we describe the method for enumerating a special classes of frequent closed serial episodes.

We define the closed episode based on right-minimal occurrences as follow. Two episodes α and β are said to be *equivalent on* S w.r.t. right-minimal occurrences, denoted by $\alpha \equiv_S \beta$, if $MO(\alpha, S) = MO(\beta, S)$. Clearly, any equivalent episodes have the same window frequency for every window width *win*. An episode α is *redundant* w.r.t. right-minimal occurrences in S if there exists some longer episode β such that $\alpha \sqsubset \beta$ and $\alpha \equiv_S \beta$. If α is not redundant then it is called *closed*. At this time, we show the property as follow about the groups of frequent and closed episodes.

LEMMA 7. **(Reverse searching of the closed episode)** *For an arbitrary non-empty closed episodes* $\beta = \alpha c$ *for some* $c \in \Sigma$, *the episode* α *obtained by removing the last letter* $c \in \Sigma$ *from* β *is closed. Furthermore, if* β *is frequent then so is* α.

The proof for the above lemma is by the same idea in [4, 11]. From the above lemma, we know that an arbitrary frequent closed episodes are obtained by appending a letter to the tail of a smaller frequent closed episodes one by one.

Consequently, we can introduce some pruning rules for enumerate frequent closed episodes. The *complete redundancy pruning rule* for closed episodes stops the search by pruning the descendants of the current pattern if the current pattern α is *redundant*, i.e., there is some proper expansion β of α such that $\alpha \sqsubset \beta$ and $\alpha \equiv_S \beta$. To test if $\alpha = a_1 \cdots a_m$ is *redundant*, we can check if there exists some $1 \leq i \leq m$ and some $b \in \Sigma$ such that $\beta = a_1 \cdots a_{i-1} b a_i \cdots a_m$ and $\alpha \equiv_S \beta$ still holds. We can modify our algorithm DFS-MO by performing the test for redundancy at Line 1 of recursive procedure Expand-MO in Fig. 1, and prune the search whenever the test fails. From Lemma 7 above, we can show that the complete redundancy pruning rule for closed episodes is sound and complete for the class of frequent closed episodes.

On the other hand, the complete pruning is sometimes too costly to check in practice. Instead, we can use the following the *tail redundancy pruning*: to prune search if the current episode $\alpha = a_1 \cdots a_m$ is *tail redundant*, i.e., there exists some $b \in \Sigma$ such that $\beta = a_1 \cdots a_{m-1} b a_m$ and $\alpha \equiv_S \beta$ still holds. The tail redundancy pruning is *not* complete in the sense that it does not capture all redundant (i.e., non-closed) episodes, while it does not miss any

closed episodes. Therefore, it might be useful in reducing some of redundant episodes in search. The test for tail redundancy can be done in $O(m \cdot win)$ time. Details will be left to the full version of this paper.

6. EXPERIMENTS AND DISCUSSIONS

In this section, we show experimental results on real world datasets.

6.1 Data

As data on small alphabets, namely $\Sigma_{DNA} = \{a, t, c, g\}$, we use the data of DNA sequence `dna` from GenBank and `ecoli` from Calgary Corpus. As English text data, we use a text collection, called `paper` here, which is a collection of English research papers in the proceedings for some international conferences in computer science area. In what follows, we denote by *dataN* the data consisting of the initial N lines of the collection *data*. In Table 1, we show the characteristics of data sets.

Table 1: The property of each text data

file	line	byte	file	line	byte
paper1000	1000	49388	`paper9000`	9000	434092
paper2000	2000	92925	`paper10000`	10000	482426
paper3000	3000	143582	`paper11000`	11000	529908
paper4000	4000	194832	paper12000	12000	577833
paper5000	5000	246068	paper13000	13000	626807
paper6000	6000	292970	paper14000	14000	676669
paper7000	7000	342574	paper15000	15000	728683
paper8000	8000	389949	e_coli	1000	80000

6.2 Method

Experimental environment is as follows. All the algorithms are implemented in C++ and compiled with GNU g++. All the experiments are run on a PC (CPU Intel U2400 1.06GHz, RAM 1014MB, OS Windows Vista, Cygwin). We implement and compare the following algorithms and compare them in in Exp. 1 and Exp. 2, where the first two algorithms are based on breadth-first search of episodes (not described in this paper), and the next three algorithms are based on depth-first search (proposed in this paper):

- BFS-Scan: We scan the input entirely, check the occurrence in each window and calculate the frequency with Breadth-first-search.
- BFS-MO: Breadth-first-search with the right-minimal occurrence.
- DFS-MO: Depth-first-search with the right-minimal occurrence we propose in this paper (Sec. 3).
- DFS-MO-OD: DFS-MO with occurrence deliver method (Sec. 4).
- DFS-CLO: DFS-MO with closed episode discovery (pruning rule for tail redundant episodes) (Sec. 5).

6.3 Exp. 1: Scalability

In Fig. 6, we show the computation time and the number of the frequent episodes of DFS-MO-OD, varying the input size $N = 1000 \sim 15000$ line for `paper` data set. We set the width of the window $win = 6$, the minimal frequency threshold $\sigma = 5000$ (about XX%) which is fixed without input size. As a result, the relative frequency changes from 10.1% to 0.69%.

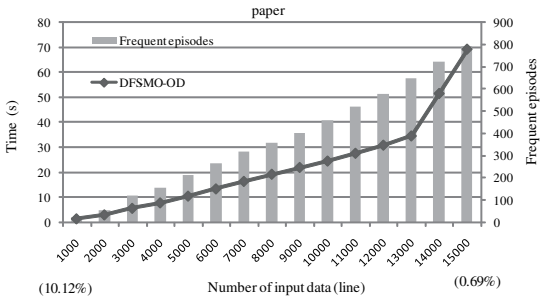

Figure 6: The change of the computation time on the size of input data

We see the number of discovered frequent episodes increases and the computation time increase while increasing in input size.

6.4 Exp. 2 : Running time

In Fig. 7, we show the comparison of DFS-MO, BFS-MO, BFS-Scan in computation time by varying the input size for $N = 1000 \sim 30000$ (letter) on `dna`. We set $win = 6$ and $\sigma = 5\%$.

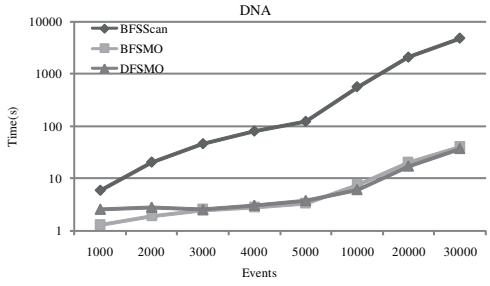

Figure 7: The comparison of the computation time on the size of input data

From the figure, we observed that DFS-MO and BFS-MO outperform the naive method BFS-Scan by saving the scanning time with incremental update of right-minimal occurrences. We did not see any difference between algorithms based on depth-first and breadth-first search in their speed.

6.5 Exp. 3 : Memory usage in DFS and BFS

In Fig. 8, we show the comparative result of the memory usage about DFS-MO, BFS-MO, BFS-Scan. The data sizes are $N = 1000, 10000$ (letter) and other parameters are same as Exp. 1.

We can easily see that DFS-MO outperforms BFS-MO in memory economy, while the running times are almost same each other as shown in Exp. 1.

6.6 Exp. 4 : Speed-up by occurrence deliver by window size

In Fig. 9, we show the computation time (the left y-axis) and the number of the frequent episodes (the right y-axis) of DFS-MO and DFS-MO-OD, by varying the width of the window *win* on `paper`. The data set is `paper` and we set $N = 10000$ (line) and $\sigma = 5000$ (1.0%).

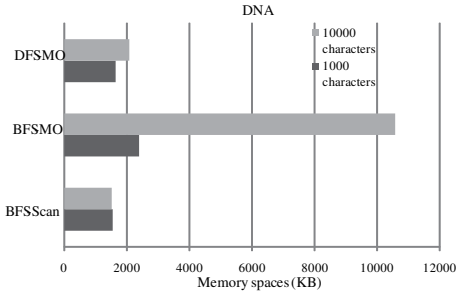

Figure 8: The comparison of the memory usage on the size of the input data

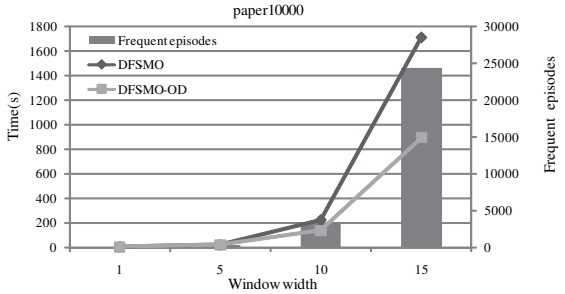

Figure 9: The comparison of the computation time on the occurrence deliver algorithm by varying the width of the window

From the figure, we can see that DFS-MO-OD is twice faster than DFS-MO regardless of the window width *win*.

6.7 Exp. 5 : Speed-up by occurrence deliver by alphabet sizes

We set $N = 1000$ and $win = 6$. In Fig. 10, we show an experiment on `paper` data set over a middle-size alphabet. the computation time (the left y-axis) and the number of the frequent episodes (the right y-axis) of DFS-MO and DFS-MO-OD, by varying $\sigma = 50 \sim 1000$ $(0.1 \sim 2.0\%)$ on `paper` data set. Similarly, in Fig. 11, we show same experiment on `ecoli` data set on $\Sigma_{DNA} = \{a, t, c, g\}$ by varying $\sigma = 50 \sim 1000$ $(0.06 \sim 1.25\%)$.

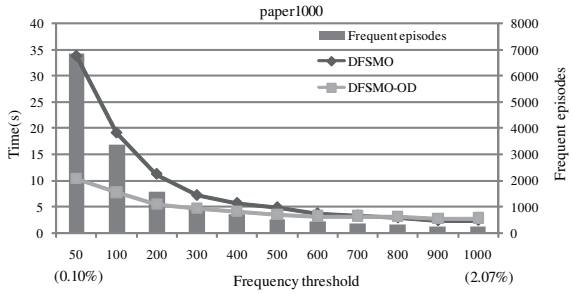

Figure 10: The comparison of the basic and the occurrence deliver algorithms on a middle-size alphabet (`paper` data set)

For the first experiment on a middle-size alphabet, we see that DFS-MO-OD with occurrence-deliver method is twice faster than DFS-MO even when the value of σ and the number of solutions change. For the second experiment on a middle-size alphabet, we see that

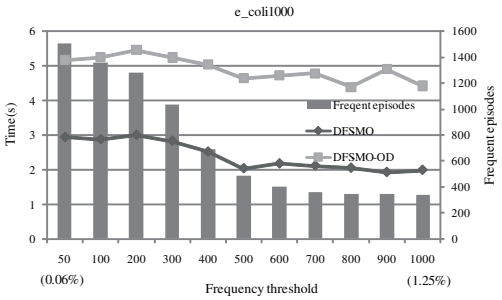

Figure 11: The comparison of the basic and the occurrence deliver algorithms on a small alphabet (`ecoli` data set)

simpler DFS-MO is faster. In summary, we see that the occurrence-deliver method is effective for data sets on large alphabets.

6.8 Exp. 6 : Pruning by tail redundancy

In this experiment, we evaluate the efficiency and utility of closed pattern discovery based on pruning by tail redundancy described in Sec. 5.

From Fig. 12 to Fig. 14, we show the computation time(the left y-axis) and the number of the pruned and unclosed episodes(the right y-axis) of DFS-MO-OD and DFS-MO-CLO, by varying parameters on `paper` data set. In Fig. 12, we change the input size $N = 1000 \sim 15000$ (line), $=0.8 \sim 0.05\%$, and we set $win = 6$ and $\sigma = 400$. In Fig. 13, we change the width of the window $win = 1 \sim 15$, and set $N = 3000$ (line) and $\sigma = 400$ (0.27%). In Fig. 14, we change the minimal frequency threshold $\sigma = 100 \sim 1000$ (0.06 ~ 0.6%), and set $N = 3000$ and $win = 6$.

From Fig. 12 to Fig. 14, the number of pruned episodes is at most 10% of all of frequent episodes in the experimental setting this time, and thus, the effect of speed-up by tail redundancy does not seem evident. Especially, curves are almost overlapping in Fig. 13 and Fig. 14. On the other hand, the overhead of removal of tail-redundant episodes is not so large in running time.

7. CONCLUSION

In this paper, we proposed an algorithm DFS-MO that finds frequent serial episodes from a time-series streaming data with a window, and we also presented several speeding-up techniques. The experiments showed that the occurrence deliver method works well when the minimal occurrence is small, and that the closed method can enumerate only closed episodes in detail.

Now we have provide the algorithm proposed in this paper as a public domain application, named LCM_seq [9]. It implements the algorithm that finds all serial episodes with window and without window, including the occurrence deliver method and the method of recursive database degeneration.

In this paper, we examined effectiveness of the occurrence deliver method and the closed pattern methods only, which are two of many speeding up techniques in [8]. To examine the efficiency of the anytime database reduction technique, and to apply to the mining problem on music databases, are our future works.

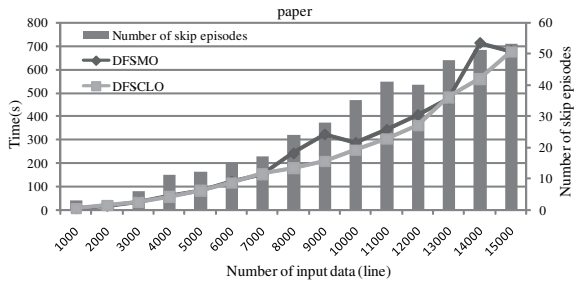

Figure 12: The comparison of the computation time varying the data size N

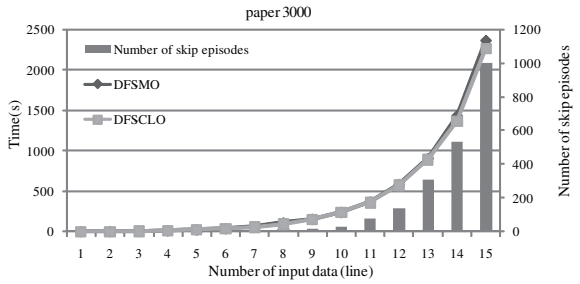

Figure 13: The comparison of the computation time varying the data size N

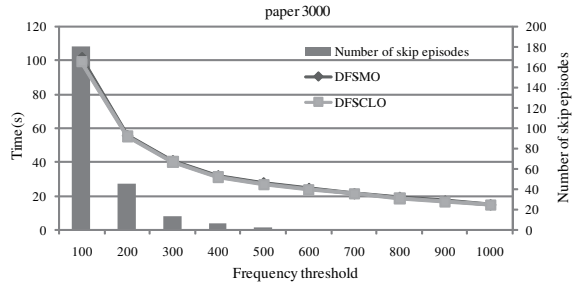

Figure 14: The comparison of the computation time varying the minimal frequency threshold σ

Acknowledgment

We would like to thank to Prof. Shin-ichi Minato, Prof. Thomas Zeugmann, Prof. Kimihito Ito, Prof. Akihiro Yamamoto, and the students of the IKN laboratory, Hokkaido University, for their valuable discussions and comments.

8. REFERENCES

[1] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufmann, 1999.

[2] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] H. Arimura, T. Uno, An Efficient Polynomial Space and Polynomial Delay Algorithm for Enumeration of Maximal Motifs in a Sequence, *J. Comb. Optim.*, Vol.13, 243-262, 2006.

[4] Hiroki Arimura and Takeaki Uno, Mining Maximal Flexible Patterns in a Sequence, Proc. LLLL2007, LNAI4914, Springer, 2008.

[5] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, M. Hsu, PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth, Proc. IEEE ICDE 2001, 215–224, 2001.

[6] Heikki Mannila, Hannu Toivonen, A. Inkeri Verkamo, Discovery of Frequent Episodes in Event Sequence, *Data Mining and Knowledge Discovery*, Vol. 1, 259–289, 1997.

[7] Y. Uchida, An efficient algorithm for enumerating episodes in the window and closed episodes, Kyushu Univ ISEE, master thesis, 2004.

[8] T. Uno, H. Arimura, Data intensive computing No.2 - An algorithm for enumerating frequent item sets -, Journal of the JSAE, Vol.17(2), 2007.

[9] Uno's Homepage, Nii, `http://research.nii.ac.jp/ uno/index-j.html`

[10] T. Uno, T. Asai, Y. Uchida, H. Arimura, An efficient algorithm for enumerating closed patterns in transaction databases, In *DS'04*, LNAI 3245, 16-30, 2004.

[11] J. Wang, J. Han, BIDE: Efficient Mining of Frequent Closed Sequences, Proc. IEEE ICDE 2004, 79–90, 2004.