Available online at www.sciencedirect.com
ScienceDirect
Information Systems 33 (2008) 96–114
www.elsevier.com/locate/infosys

Efficient mining of frequent episodes from complex sequences ☆

Kuo-Yu Huang, Chia-Hui Chang*

Department of Computer Science and Information Engineering, National Central University, No. 300, Jungda Rd, Chung-Li 320, Taiwan, ROC

Received 27 July 2005; received in revised form 12 May 2007; accepted 7 July 2007
Recommended by N. Koudas

Abstract

Discovering patterns with great significance is an important problem in data mining discipline. An episode is defined to be a partially ordered set of events for consecutive and fixed-time intervals in a sequence. Most of previous studies on episodes consider only frequent episodes in a sequence of events (called simple sequence). In real world, we may find a set of events at each time slot in terms of various intervals (hours, days, weeks, etc.). We refer to such sequences as complex sequences. Mining frequent episodes in complex sequences has more extensive applications than that in simple sequences. In this paper, we discuss the problem on mining frequent episodes in a complex sequence. We extend previous algorithm MINEPI to MINEPI+ for episode mining from complex sequences. Furthermore, a memory-anchored algorithm called EMMA is introduced for the mining task. Experimental evaluation on both real-world and synthetic data sets shows that EMMA is more efficient than MINEPI+.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Data mining; Frequent episodes; Temporal association

1. Introduction

Many data mining and machine learning techniques are adapted towards the analysis of unordered collections of data, e.g., transaction databases and sequence databases [1–7]. However, there are important application areas where the data to be analyzed are ordered, e.g., alarms in a telecommunication network, user interface actions, occurrences or recurrent illnesses. One basic problem in analyzing such a sequence is to find frequent *episodes*, i.e., collections of events occurring frequently together [8–10]. The goal of episode mining is to find relationships between events. Such relationships can then be used in an on-line analysis to better explain the problems that cause a particular event or predict future result. Episode mining has been of great interest in many applications, including internet anomaly intrusion detection [11,12], biomedical data analysis [13,14], stock trend prediction [15] and drought risk management in climatology data sets [16]. Take stock data as an example, we will find an episode rule R_1 like "When the price of stock Microsoft goes up for two consecutive days,

☆This paper was sponsored by National Science Council, Taiwan, under Grant NSC95-2524-S-008-002.
*Corresponding author. Tel.: +886 3 4227151x35302.
E-mail addresses: want@db.csie.ncu.edu.tw (K.-Y. Huang), chia@csie.ncu.edu.tw (C.-H. Chang).

0306-4379/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2007.07.003

the price of stock IBM will go up *within two days* with 80% probability.'' Besides, there are also studies on how to identify significant episodes from statistical model [17,18].

The task of mining frequent episodes was originally defined on ''a sequence of events'' where the events are sampled regularly as proposed by Mannila et al. [8]. Informally, an episode is a partially ordered collection of events occurring together. The user defines how close is close enough by giving the width of the time window *win*. The number of sliding windows with width *win* in a sequence is $t_e - t_s + win$, where t_s and t_e are called the starting interval and the ending interval, respectively. Take the sequence $S = A_3A_4B_5B_6$ (the subscript i represents the *ith* interval) and $win = 3$ as an example, there are $6 - 3 + 3 = 6$ sliding windows in S, e.g., $W_1 = A_3$, $W_2 = A_3A_4$, $W_3 = A_3A_4B_5$, $W_4 = A_4B_5B_6$, $W_5 = B_5B_6$ and $W_6 = B_6$. Mannila et al. introduced three classes of episodes. *Serial episodes* consider patterns of a total order in the sequence, while *parallel episodes* have no constraints on the relative order of event sets. The third class contains composite episodes like serial combination of parallel episodes. In a way, serial and parallel episodes can be captured by sequential patterns and frequent itemsets, respectively. Frequent itemsets for transaction databases are similar to parallel episodes, while sequential patterns for sequence databases are similar to serial episodes as defined in [8]. Therefore, we can mine parallel episodes by transforming an event sequence to a transaction database where each transaction is the union of events from sliding window W_i. Similarly, we can mine serial episodes by transforming an event sequence to a sequence database, where each sequence is the serial combination of events from W_i. However, such methods are not efficient, since the space requirement is *win* times the original database size. Finally, composite episodes can be mined from serial joins of parallel episodes.

Mannila et al. presented a framework for discovering frequent episodes through a level-wise algorithm, WINEPI [8], for finding parallel and serial episodes that are frequent enough. The algorithm was an Apriori-like algorithm with the ''anti-monotone'' property of episodes' support. Unfortunately, this support count has a defect, i.e., over-estimate of the occurrences of an episode. Take the sequence $S = A_3A_4B_5B_6$ and $win = 3$ as an example, the serial episode rule ''When event A occurs, then event B occurs within 3 time units'' should have probability or confidence $\frac{2}{2}$ in the sequence S since every occurrence of A is followed by B within 3 time units. However, since episode $\langle A\rangle$ is supported by four sliding windows ($W_1 = A_3$, $W_2 = A_3A_4$, $W_3 = A_3A_4B_5$, $W_4 = A_4B_5B_6$), and serial episode $\langle A, B\rangle$ is matched by two sliding windows (W_3 and W_4), the above rule will have confidence $\frac{2}{4}$.

Instead of counting the number of sliding windows that support an episode, Mannila et al. consider the number of minimal occurrences of an episode from another perspective. They presented MINEPI [9], an alternative approach to the discovery of frequent episodes from minimal occurrences (*mo*) of episodes. A minimal occurrence of an episode α is an interval such that no proper subwindow contains the episode α. For example, episode $\langle A\rangle$ has *mo* support 2 (interval $[3, 3]$ and $[4, 4]$) as the number of occurrences, while episode $\langle A, B\rangle$ has only *mo* support 1 from interval $[4, 5]$. Thus, the above rule will have confidence $\frac{1}{2}$. However, both measures are not natural for the calculating of an episode rule's confidence. Therefore, we need a measure that facilitates the calculation of such episode rules to replace the number of sliding windows or minimal occurrences. The problem has also been discussed in [19], but no algorithms are proposed.

In addition, we sometimes find several events occurring (multi-variables) at one time slot in terms of various intervals (e.g., hours, days and weeks). We refer to such sequences as *complex sequences*. Note that a temporal database is also a kind of complex sequence when temporal attributes are considered. Mining frequent episodes in a complex sequence has more extensive applications than that in a simple sequence. Therefore, we discuss the problem on mining frequent episodes over a complex sequence in this paper, where the support of an episode is modified carefully to count the exact occurrences of episodes. We propose two algorithms in mining frequent episodes in complex sequences, including MINEPI+ and EMMA. MINEPI+ is modified from previous vertical-based MINEPI [9] for mining episodes in a complex sequence. MINEPI+ employs depth-first enumeration to generate the frequent episodes by *equalJoin* and *temporalJoin*. To further reduce the search space in pattern generation, we propose a brand new algorithm, EMMA (Episodes Mining using Memory Anchor), which utilizes memory anchors to accelerate the mining task. Experimental evaluation on

both real-world and synthetic data sets shows that EMMA is more efficient than MINEPI+.

The rest of this paper is organized as follows. We define the problem of frequent serial episode mining in Section 2. Section 3 reviews related work in sequence mining. Section 4 presents our serial episode mining algorithms, including MINEPI+ and EMMA. Experiments on both synthetic and real-world data sets are reported in Section 5. Finally, conclusions are made in Section 6.

2. Problem definition

In this section, we first define the problem of frequent serial episode mining (parallel episode mining is discussed in the Appendix A). Let E be a set of all events. An event set is a nonempty subset of E. An input sequence can be represented as $(X_1, X_2, \ldots, X_O)$ where X_i is an event set that occurs in ith time interval or empty. The input sequence can also be described using a more general concept like a temporal database, where each tuple (t_j, X_{t_j}) records the time interval (in terms of various units, hours, days, etc.) t_j for each event set X_{t_j} (nonempty). We use time intervals because event set sequences (or temporal databases) are formed due to the merging of several records in the same time interval. Fig. 1(a) shows an input sequence with this horizontal format. Let N be the number of tuples in the temporal database TDB. We say that TDB has length N in O observation time intervals. We say that an event set Y is supported by a record (t_i, X_{t_i}) if and only if $Y \subseteq X_{t_i}$. An event set with k events is called a k-event set. Let *maxwin* be the maximum window bound. When mining episode rules, only the rules with span less than or equal to *maxwin* intervals will be mined. Users can thus use this mining parameter to avoid mining rules that span across too many intervals.

Definition 2.1 (*Sliding window*). A *sliding window* W_i in a temporal database TDB is a block of *maxwin* continuous records along the time interval, starting from interval t_i (where TDB contains an event set at t_ith time interval). Each interval t_{i_j} in W_i is called a subwindow of W_i denoted as $W_i[j]$, where $j = t_{i_j} - t_i$. Therefore, TDB with length N can be divided into N sliding windows, such as $W_1 = (X_{t_1}, X_{t_1+1}, \ldots, X_{t_1+maxwin-1})$, $W_2 = (X_{t_2}, X_{t_2+1}, \ldots, X_{t_2+maxwin-1}), \ldots, W_N = (X_{t_N})$.

Example 2.1. Fig. 1(a) shows a temporal database TDB with 14 ($N = 14$) transactions located at intervals 1–16 except 2 and 10. Assume a value of 3 is set for maximum window *maxwin*. According to the definition, the number of sliding windows in Fig. 1(c) is 14, from $W_1, W_2, \ldots, W_{14}$. This will form a sequence database of size 14, which is different from the 18 sliding windows defined in [8] where empty intervals are considered. Thus, window W_1 has three subwindows $W_1[0] = \{A, C, F\}$, $W_1[1] = \emptyset$ and $W_1[2] = \{B, D\}$. As another example, window W_2 has also three subwindows $W_2[0] = \{B, D\}$, $W_2[1] = \{A, C, F\}$, and $W_2[2] = \{D, E\}$. Fig. 1(b) shows the transaction database where each transaction is the union of the events in all subwindows of a window W_i, while Fig. 1(c) shows the sequence database where each sequence represents a sliding window (with subscript denoting the time interval).

a

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Events	A C F		B D	A C F	D E	B D	A F	A C F	B D		A C F	B D	D E	A C F	C E	B D

b

TID	Itemset
1	$\{A, B, C, D, F\}$
2	$\{A, B, C, D, E, F\}$
	..
13	$\{B, C, D, E\}$
14	$\{B, D\}$

c

WID	Sequence
1	$\{A, C, F\}_1\{B, D\}_3$
2	$\{B, D\}_3\{A, C, F\}_4\{D, E\}_5$
	..
13	$\{C, E\}_{15}\{B, D\}_{16}$
14	$\{B, D\}_{16}$

Fig. 1. Temporal database TDB: (a) a temporal database TDB, (b) transactional database, (c) sequence database.

Definition 2.2 (*Serial episode*). A *serial episode* is a nonempty partial ordered set of events $P = \langle p_1, p_2, \ldots, p_k \rangle$ where each p_i is a nonempty event set and p_i occurs before p_j for $i<j$. P is a k-tuple serial episode or has length k.

Definition 2.3 (*Super/Sub sequence*). Given two sequences $S = \langle s_1, s_2, \ldots, s_n \rangle$ and $S' = \langle s'_1, s'_2, \ldots, s'_m \rangle$, we say that S is a *super-sequence* of S' (i.e., S' is a *subsequence* of P) if and only if, each s'_j can be mapped by s_{i_j} ($s'_j \subseteq s_{i_j}$) and preserve its order $(1 \leqslant i_1 \leqslant i_2 \leqslant \cdots \leqslant i_m \leqslant n)$.

For example, sequence $S_1 = \langle \{A, B\}, \{C\}, \{D, E\} \rangle$ is a super-sequence of sequence $S_2 = \langle \{A\}, \{D\} \rangle$, since the pattern $\{A\}$ ($\{D\}$, resp.) is a subset of $\{A, B\}$ ($\{D, E\}$, resp.). On the contrary, $S_3 = \langle \{A\}, \{C, D\} \rangle$ is not a subsequence of S_1, since the pattern $\{C, D\}$ cannot be mapped to any itemset in S_1.

Usually, a match of serial episode P is defined when P is a subsequence of a sliding window. However, it causes duplicate counting of the occurrence of an episode. To overcome this problem, the match of a serial episode in this paper is defined as following.

Definition 2.4 (*Match*). Given a serial episode $P = \langle p_1, p_2, \ldots, p_k \rangle$ and window bound w, we say that a sliding window $W_i = (X_{t_i}, X_{t_i+1}, \ldots, X_{t_i+w-1})$ in *TDB supports* P if and only if, $p_1 \subseteq X_{t_i}$ and $\langle p_2, \ldots, p_k \rangle$ is a subsequence of $(X_{t_i+1}, \ldots, X_{t_i+w-1})$. W_i is also called a *match* of the serial episode P. The number of sliding windows that match episode P is called the support count of P in temporal database *TDB*.

Let us return to the previous example in Fig. 1(a) and assume $maxwin = 3$, the serial episode $\langle \{A\}, \{D\} \rangle$ is matched by sliding windows starting from time slots 1, 4, 7, 8, 11 and 14. Therefore, there are six matches with respect to serial episode $\langle \{A\}, \{D\} \rangle$. Note that the sequence corresponding to window $W_2 = \langle \{B, D\}, \{A, C, F\}, \{D, E\} \rangle$, although contains $\langle \{A\}, \{D\} \rangle$, does not support this episode because the first subwindow does not contain $\{A\}$.

This definition of match is sufficient to capture the true support of an episode without overestimate. However, the problem is that a serial episode of length 1 may has support counts less than its super episode since the overlapping of sliding windows make most subwindows counted *maxwin* times (except for the first $maxwin - 1$ subwindows). Take sequence $S = A_1A_2B_3B_4C_5$, $maxwin = 3$ for example. Although episode $\langle C \rangle$ has support 1, episode $\langle B, C \rangle$ has support 2 since it appears in W_3 and W_4. Note that the matches of serial episode are equally over-counted, therefore this problem does not exist in previous work. Despite this issue, anti-monotone property still holds for the supports of most serial episodes. In fact, we are more interested in frequent serial episodes whose sub episodes are frequent as well. Thus, the frequent episodes in this paper are defined as follows.

Definition 2.5 (*Frequent episode*). An episode $P = \langle p_1, \ldots, p_k \rangle$ is *frequent* if and only if the supports of P and all subsequences p_i $(1 \leqslant i \leqslant k)$ in P are at least the required user-specified minimum supports (i.e., *minsup*).

Definition 2.6 (*Episode concatenation*). The *concatenation* of two serial episodes $P = \langle p_1, \ldots, p_{l_1} \rangle$ and $Q = \langle q_1, \ldots, q_{l_2} \rangle$ is defined as $P \cdot Q = \langle p_1, \ldots, p_{l_1}, q_1, \ldots, q_{l_2} \rangle$. P is called a *prefix* of $P \cdot Q$.

Definition 2.7 (*Episode rule*). An *episode rule* is an implication of the form $X \Rightarrow Y$, where

(1) X, Y are episodes with length l_1 and l_2, respectively.
(2) The concatenation $X \cdot Y$ is an episode with length $l_1 + l_2$.

Similar to the studies in mining typical association rules, episode rules are governed by two interestingness measures: support and confidence.

Definition 2.8 (*Support and confidence*). Let N be the number of transactions in the temporal database *TDB*. Let $Match(X \cdot Y)$ be the number of support counts with respect to episode $X \cdot Y$ and $Match(X)$ be the number of support counts with respect to episode X. Then, the *support* and *confidence* of an episode rule $X \Rightarrow Y$ are defined as

$$Support = \frac{Matches(X \cdot Y)}{N},$$
$$Confidence = \frac{Matches(X \cdot Y)}{Matches(X)}. \quad (1)$$

Example 2.2. Let the user-specified threshold minimum support *minsup* and minimum confidence *minconf* be 30% and 100%, respectively. An example of a serial episode rule with maximum time window bound $maxwin = 3$ from the temporal database in Fig. 1(a) will be

$\langle \{A, C, F\} \rangle \Rightarrow \langle \{B, D\} \rangle$.

This rule "event set $\{B, D\}$ occurs within two interval after event set $\{A, C, F\}$" holds in the temporal database TDB with $support = 35.7\%(\frac{5}{14})$ and $confidence = 100\%(\frac{5}{5})$.

As in classical association rule mining, when frequent episodes and their support are known, the episode rule generation is straightforward. Hence, the problem of mining episode rules is reduced to the problem of determining frequent episodes and their supports. Therefore, the problem is formulated as follows: given a minimum support level *minsup* and a maximum window bound *maxwin*, our task is to mine all frequent episodes from the temporal database with support greater than *minsup* and window bound less than *maxwin*.

3. Related works

Mining significant patterns in sequence(s) is an important and fundamental issue in knowledge discovery area. For example, sequential patterns [1–3,5–7] consider the problem on discovering repeated subsequences in a database of sequences. On the other hand, some mining tasks focus on mining repeated subsequences in a sequence, e.g., frequent episodes [8–10], frequent continuities [20–22] and periodic patterns [23–27]. In this section, we distinguish various sequence mining tasks including sequential patterns, periodic patterns and frequent continuities which are related to frequent episodes. We also make an overall comparison between frequent itemsets and the four mining tasks.

The problem of mining sequential patterns was introduced in [1]. This problem is formulated as "Given a set of sequences, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified *minsup* threshold, sequential pattern mining is to find all frequent subsequences, whose occurrence frequency in the set of sequences is no less than *minsup*." The main difference between frequent itemsets and sequential patterns is that a sequential pattern considers the order between items, whereas frequent itemset does not specify the order. Srikant et al. proposed an Apriori-based algorithm, GSP (Generalized Sequential Pattern) [6] to the mining of sequential patterns. However, in situations with prolific frequent patterns, long patterns, or quite low *minsup* thresholds, an Apriori-like algorithm may suffer from a huge number of candidate sets and multiple database scans. To overcome these drawbacks, Han et al. extend the concept of FP-tree [28] and proposed the PrefixSpan algorithm by prefix-projected pattern growth [5] for sequential pattern mining. In addition to algorithms based on horizontal formats, Zaki proposed a vertical-based algorithm SPADE [7]. SPADE utilizes combinatorial properties to decompose the original problem into smaller sub-problems that can be independently solved in main memory using efficient lattice search techniques and simple join operations. SPAM [29] employs a vertical bitmap representation, which makes it more efficient than PrefixSpan and SPADE. However, it consumes more memory space for vertical bitmap maintenance.

Periodic pattern, as suggested by its name, consider regularly appear events where the exact positions of events in the period are fixed [30,27,26]. To form periodicity, a list of k disjoint matches is required to form a contiguous subsequence where k satisfying some predefined minimum repetition threshold. For example, in Fig. 1, pattern (A,*,B) is a periodic pattern that matches [1, 3], [4, 6], and [7, 9], three contiguous and disjoint matches, where event {A} (resp. {B}) occurs at the first (resp. third) position of each match. The character "$*$" is a "do not care" character, which can match any single set of events. Note that [14, 16] is not part of the pattern because it is not located contiguously with the previous matches. To specify the occurrence, we use a 4-tuple (P, l, rep, pos) to denote a valid segment of pattern P with period l starting from position *pos* for *rep* times. In this case, the segment can be represented by ((A,*,B), 3, 3, 1). Algorithms for mining periodic patterns also fall into two categories, horizontal-based algorithms, LSI [26], and vertical-based algorithms, SMCA [30,27].

A continuity pattern is similar to a periodic pattern, but without the constraint on the contiguous and disjoint matches. For example, pattern [A,*,B] is a continuity with four matches [1, 3], [4, 6], and [7, 9], and [14, 16] in Fig. 1. The term continuity pattern was coined by Huang et al. in [21] to replace the general term inter-transaction association defined by Tung et al. in [22], since episodes and periodic patterns are also a kind of inter-transaction associations in the conceptual level. In comparison, frequent episodes are a loose kind of frequent continuities since they consider only the partial order between events, while periodic patterns are a strict kind of frequent continuities with constraints on the subsequent matches. In a word, frequent

episodes are a general case of the frequent continuity, and periodic patterns are a special case of the frequent continuity. Two algorithms have been proposed for this task, including FITI and PROWL. FITI [22] is an Apriori-based algorithm which uses breadth-first enumeration for candidate generation and scans the horizontal-layout database. The PROWL algorithm [21], on the other hand, generates frequent continuities using depth-first enumeration and relies on the use of both horizontal and vertical-layout databases.

Table 1 shows the comparison of the above mining tasks with frequent itemsets. The column "Order" represents whether the discovered pattern specify order; the column "Temporal" indicates whether the task is defined for a temporal database. According to the input database, frequent itemsets and sequential patterns are similar since they are defined on databases where the order among transactions/sequences is not considered; whereas episodes, continuities, and periodic patterns are similar for they are defined on sequences of events that are usually sampled regularly. Frequent itemsets and sequential patterns are defined for a set of transactions and a set of sequences, respectively. Frequent itemsets show contemporal relationships, i.e., the associations among items within the same transaction; whereas sequential patterns present temporal/causal relationships among items within transactions of customer sequences.

Among the mining tasks from a single sequence, periodic patterns have the most restrictions, while episodes have the least restrictions. Episode mining discovers more patterns than periodic patterns and continuity patterns, thus it requires more computation time. There are also other kinds of patterns defined over sequences. For example, MAGIIC-pro employs both intra- and inter-block gap constraints to discover functional long motifs that are interleaved by several large irregular gaps [31]. Whether episode patterns are more useful than periodic patterns or continuity patterns depends on the applications, which is not the main topic of this paper. In this paper, we should focus on how to discover serial episodes (which are similar to sequential patterns) more efficiently.

4. Mining serial episodes

In this section, we propose two algorithms for serial episode mining. Note that MINEPI outperforms WINEPI and overcome some drawbacks of WINEPI. Therefore, we first show how to extend existing algorithm MINEPI to find the support counts instead of minimal occurrences in a complex sequence. Then, a new algorithm EMMA is proposed for more efficient mining of serial episodes from complex sequences. The comparison of the two algorithms is given in Section 4.3.

4.1. MINEPI+

MINEPI is an iteration-based algorithm which adopts breadth-first manner to enumerate longer serial episodes from shorter ones. However, instead of scanning the temporal database (in horizontal format) for support counting, MINEPI computes the minimal occurrences *mo* of each candidate episode from the *mo* of its subepisode by temporal joins. For example, we want to find all frequent serial episodes from a simple sequence $S = A_1A_2B_3A_4B_5$ with $maxwin = 4$ and $minsup = 2$. MINEPI first finds frequent 1-episode and records the respective minimal occurrence, i.e., $mo(A) = \{[1,1],[2,2],[4,4]\}$, $mo(B) = \{[3,3],[5,5]\}$. (We call this representation of the temporal sequence as vertical format.) Using temporal join, we get intervals $[1,3]$, $[2,3]$, $[2,5]$ and $[4,5]$ for candidate

Table 1
Comparison of various pattern mining

	Notation	Order	Temporal	Input	Constraint
Frequent itemset	$I = \{i_1, \ldots, i_n\}$	N	N	A transaction DB	
Sequential pattern	$S = I_1, \ldots, I_n$	Y	N	A sequence DB	
Serial episode	$SEP = \langle I_1, \ldots, I_n \rangle$	Y	Y	A sequence	
Parallel episode	$PEP = \{I_1, \ldots, I_n\}$	N	Y	A sequence	
Frequent continuity	$C = [I_1, \ldots, I_n]$	Y	Y	A sequence	[a]
Periodic pattern	$P = (I_1, \ldots, I_n)$	Y	Y	A sequence	[a,b]

[a]Fixed interval between I_i and I_{i+1}.
[b]Contiguous match.

2-tuple episode $\langle A, B\rangle$. Since $[1, 3]$ and $[2, 5]$ are not minimal, the minimal occurrences of $\langle A, B\rangle$ will be $\{[2, 3], [4, 5]\}$.

To extend MINEPI for our problem, where the support of an episode is defined by the number of sliding windows that match serial episode $\langle A, B\rangle$, the support count for serial episode $\langle A, B\rangle$ is 3, including intervals $[1, 3]$, $[2, 3]$, and $[4, 5]$ since $[2, 3]$ and $[2, 5]$ denote the same sliding window. We will use these intervals or bounds to compute the right support count for the problem.

Definition 4.1 (*Bound list for episode*). Given the maximum window boundary *maxwin*, the *bound list* of a serial episode $P = \langle p_1, \ldots, p_k\rangle$ is the set of intervals $[ts_i, te_i]$ $(te_i - ts_i < maxwin)$ such that $p_1 \subseteq X_{ts_i}$, $p_k \subseteq X_{te_i}$ and $[X_{ts_i+1}, X_{ts_i+2}, \ldots, X_{te_i-1}]$ is a super-sequence of $\langle p_2, \ldots, p_{k-1}\rangle$. Each interval $[ts_i, te_i]$ is called a matching bound of P. By definition, the bound list of an event Y is the set of intervals $[t_i, t_i]$ such that X_{t_i} supports Y.

Given a serial episode $P = \langle p_1, \ldots, p_k\rangle$ and a frequent 1-pattern f and their matching bound lists, e.g., $P.boundlist = \{[ts_1, te_1], \ldots, [ts_n, te_n]\}$ and $f.boundlist = \{[ts'_1, ts'_1], \ldots, [ts'_m, ts'_m]\}$. The operation *temporal join* (concatenation) of P and f (denoted by $P \cdot f$) which computes the bound list for new serial episode $P_1 = \langle p1, \ldots, p_k, f\rangle$ is defined as the set of intervals $[ts_i, te'_j]$ such that $te'_j - ts_i < maxwin$, and $te'_j > te_i$ for some j $(1 \leqslant j \leqslant m)$.

To deal with complex sequences, we also need equal join in addition to temporal join. The operation *equal join* of P and f which computes the bound list for a new serial episode $P_2 = \langle p_1, \ldots, p_k \cup f\rangle$ (denoted by $P \odot f$) is defined as the set of intervals $[ts_i, te_i]$ such that $te_i = ts'_j$ for some j $(1 \leqslant j \leqslant m)$. This concept of temporal joins and equal joins in episodes is similar to the *sequence-extension* and *item-extension* in sequential patterns.

Example 4.1. Let $maxwin = 4$, we use the matching bound lists $\langle A\rangle.boundlist = \{[1, 1], [4, 4], [7, 7], [8, 8], [11, 11], [14, 14]\}$, $\langle B\rangle.boundlist = \{[3, 3], [6, 6], [9, 9], [12, 12], [16, 16]\}$, and $\langle C\rangle.boundlist = \{[1, 1], [4, 4], [8, 8], [11, 11], [14, 14], [15, 15]\}$ as an example. The matching bound list of equal join $(\langle A\rangle \odot \langle C\rangle)$ and temporal join $(\langle B\rangle \cdot \langle A\rangle)$ are $\langle AC\rangle.boundlist = \{[1, 1], [4, 4], [8, 8], [11, 11], [14, 14]\}$ and $\langle B, A\rangle.boundlist = \{[3, 4], [6, 7], [6, 8], [9, 11], [12, 14]\}$, respectively.

With the boundlist for each episode, it is easy to find the support count. Continuing the above example, the serial episode $\langle B, A\rangle$ is matched by four sliding windows since $[6, 7]$ and $[6, 8]$ refer to the same sliding window.

Lemma 4.1. *The support count of a serial episode P equals the number of distinct starting positions of the bound list for P, denoted by* EntityCount(*P.boundlist*).

Different from MINEPI, we apply depth-first enumeration to pattern generation for memory saving. This is because breadth-first enumeration must keep track of records for all episodes in two consecutive levels, while depth-first enumeration needs only to keep intermediate records for episodes generated along a single path. Fig. 2 outlines the proposed MINEPI+ algorithm. We call our algorithm as MINEPI+ since the vertical-based operation in MINEPI+ is similar to MINEPI. The input to MINEPI+ is a temporal database, minimum support threshold *minsup* and maximum window bound *maxwin*. According to Definition 2.5, the frequent episode is generated by frequent itemsets. Therefore, before applying depth-first enumeration, we scan the temporal database *TDB* once, find frequent 1-items F_1 and the boundlists (line 1). Frequent episodes are then generated by joining the boundlists of an existing episode (lines 2–3) and an f_j in F_1 (line 4) through procedure call to *SerialJoins*. To avoid duplicate enumeration for equal joins, we define an order (e.g., alphanumerical order) in the events E. If the order of f_j is greater than the order of the *lastItem* in the episode, we apply equal join (lines 5–6) and check if the new serial episode $\alpha \cdot f_j$ is frequent or not (line 7, where EntityCount returns the number of distinct starting positions in the bound list). If the result is true, all frequent episodes which have prefix $\alpha \odot f_j$ (line 8) will be enumerated by recursive call to subprocedure *SerialJoins*. Similarly, we apply temporal join to the existing serial episode and the f_js in F_1 to get $\alpha \cdot f_j$ in lines 10–12. We illustrate the MINEPI+ algorithm using the following example.

Example 4.2. Given $minsup = 30\%$ ($\lceil 16 * 30\% \rceil = 5$ times) and $maxwin = 4$, the frequent 1-items F_1 for Fig. 1(a) include $\langle A\rangle$, $\langle B\rangle$, $\langle C\rangle$, $\langle D\rangle$ and $\langle F\rangle$. Owing space limitation, we only use $F_1 = \{\langle A\rangle, \langle C\rangle\}$ as an example. The execution process is shown in Fig. 3. In the beginning, we try to join $\langle A\rangle$ with $\langle A\rangle$, the first element in F_1. Since they have the same order, we only apply the temporal join for them. The entity count for $temporalJoin(\langle A\rangle, \langle A\rangle) = \{[1, 4], [4, 7], [7, 8], [8, 11], [11, 14]\}$ is 5. Thus, we

Procedure of **MINEPI+(temporal database** TDB**,** $minsup$**,** $maxwin$**)**
1. **Scan** TD **once, find frequent 1-items** F_1 **and the** $boundlists$**;**
2. **for each** f_i **in** F_1 **do**
3. **SerialJoins(**f_i**,** $f_i.boundlist$**,** f_i**);**

Subprocedure of **SerialJoins(**α**,** $boundlist$**,** $lastItem$**)**
4. **for each** f_j **in** F_1 **do**
5. **if (** $f_j > lastItem$ **) then**
6. $tempBoundlist = equalJoin(\alpha, f_j)$;
7. **if (**$EntityCount(tempBoundlist) \geq minsup * |TDB|$**) then**
8. **SerialJoins(**$\alpha \odot f_j, tempBoundlist, f_j$**);**
9. **end if**
10. $tempBoundlist = temporalJoin(\alpha, f_j)$;
11. **if (**$EntityCount(tempBoundlist) \geq minsup * |TDB|$**) then**
12. **SerialJoins(**$\alpha \cdot f_j, tempBoundlist, f_j$**);**

Fig. 2. MINEPI+: Vertical-based Frequent Serial Episode Mining Algorithm.

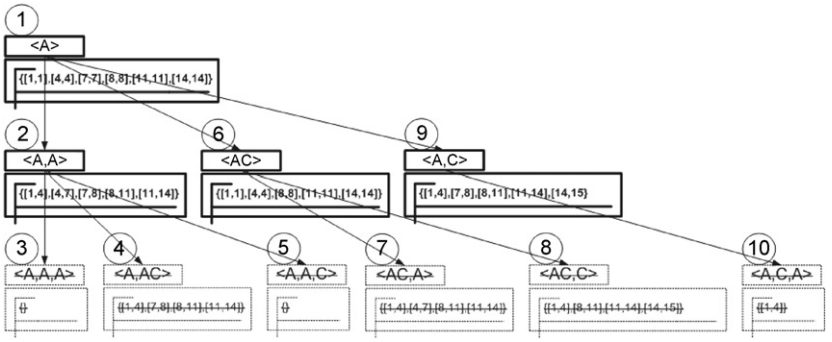

Fig. 3. Flowchart for prefix $\langle A \rangle$ (only events A and C are demonstrated).

recursively call *SerialJoins* for generating serial episodes which have prefix $\langle A, A \rangle$. Next, we check serial episodes $\langle A, A, A \rangle$, $\langle A, \{A, C\} \rangle$ and $\langle A, A, C \rangle$. In this layer, all of them are infrequent. The recursive call stops and returns to the prior procedure. Next, we compute the matching bound list for serial episode $\langle \{A, C\} \rangle$ by *equalJoin*, which returns the matching bounds of $\langle \{A, C\} \rangle$ as $\{[1, 1], [4, 4], [8, 8], [11, 11], [14, 14]\}$. The recursive call of *SerialJoins* then enumerates $\langle \{A, C\}, A \rangle$ and $\langle \{A, C\}, C \rangle$ (see Fig. 3). Finally, only five serial episodes $\langle A \rangle$, $\langle A, A \rangle$, $\langle A, C \rangle$, and $\langle A, C \rangle$ are outputted in this example.

Though the extension of MINEPI discover all frequent serial episodes, MINEPI+ has the following drawbacks.

- *A huge amount of combinations/computations*: Let $|I|$ be the number of frequent 1-episodes, WINEPI + needs $|I|^2$ and $(|I|^2 - |I|)/2$ checking for temporal joins and equal joins, respectively. For example, if there are 1000 frequent 1-episodes, there are approximately 1.5 million combinations. Moreover, when the number of matching bounds increases, MINEPI+ requires more time in computation.
- *Unnecessary joins*: Since long episodes are generated from shorter ones, sometimes MINEPI+ makes some unnecessary checking. Take the bound list of serial episode $\langle A, A \rangle$ in Example 4.2 as an example. In this case, only the time bound $[7, 8]$ can be extended by temporal join to generate long episode since other bounds already reach the limits of maximum windows. Since the number of the extendable matching bounds for serial episode $\langle A, A \rangle$ is less than $minsup * |TDB|$, we can skip all temporal joins for this prefix. We will discuss a pruning strategy in the following section.

- *Duplicate joins*: Furthermore, MINEPI+ also performs some duplicate checking. For example, to find serial episode $\langle ABCD, ABCD, ABCD\rangle$, MINEPI+ needs nine of equal joins (e.g., $equalJoin(\langle A\rangle, \langle B\rangle)$, $equalJoin(\langle AB\rangle, \langle C\rangle)$ and $equalJoin(\langle ABC\rangle, \langle D\rangle)$, etc.) and two temporal joins (e.g., $temporalJoin(\langle ABCD\rangle, \langle A\rangle)$ and $temporalJoin(\langle ABCD, ABCD\rangle, \langle A\rangle)$). However, if we maintain the bound list for $\langle ABCD\rangle$, we only needs two temporal joins.

4.2. EMMA

In this section, we propose an algorithm, EMMA (**E**pisode **M**ining using **M**emory **A**nchor), that overcomes the drawbacks of the MINEPI+ algorithm described in the previous section. First, to avoid duplicate equal joins or intra-transaction joins, we can find all frequent itemsets first and then use temporal joins for inter-transaction patterns. The idea comes from FITI (First-Intra-Then-Inter) for frequent continuity mining [22]. The set of frequent itemsets are actually length 1 frequent serial episodes. With the boundlists for all frequent 1-tuple episode, we can use temporal joins to discover longer serial episodes. This approach has shown to perform well for frequent continuity mining in efficiency improvement. However, the problem is that the set of all frequent itemsets is much larger than the set of frequent items. If we apply traditional candidate-generation-and-test procedure as MINEPI+, it will cost a lot of computation. Luckily, there is already known solution to mining sequential patterns without candidate generations [32]. The idea is to grow a frequent pattern by adding locally frequent items from its projected database (vs. from global frequent items F_1). Thus, the cost will depend on how we implement the projected database. In this paper, we should see the combination of vertical-based data representation with horizontal-based data representation to achieve pseudoprojection, where vertical-based format records the anchors to horizontal data representation. We will apply this technique both in frequent itemset mining as well as serial episode mining. In summary, EMMA is divided into the following three phases.

(1) *Frequent itemset mining*: Mining frequent itemsets in the complex sequence.
(2) *Database encoding*: Encode each frequent itemset with a unique ID and construct them into an encoded horizontal database.
(3) *Frequent serial episode mining*: Mining frequent serial episodes in the encoded database.

4.2.1. Frequent itemset mining

There are already a lot of frequent itemset mining algorithms. Since the third phase of serial episode mining requires the time lists of each frequent itemset, we prefer using a vertical-based mining algorithm, e.g., Eclat [33]. However, similar to the drawbacks of MINEPI+, unnecessary candidates are generated in the computation of Eclat. Therefore, we devise a more efficient algorithm FIMA (**F**requent **I**temset mining using **M**emory **A**nchor) which validates local frequent items to reduce unnecessary combinations of existing frequent itemsets with nonlocal frequent items. To accelerate the validation of local frequent items, a flat format of the database is introduced, which records the items as well as their Tids.

As shown in Fig. 4(a), the flat data representation of frequent items is an array of 2-tuples, (Tid, I), which is consistent with the relational data representation of the database. To implement pseudo-projection of a frequent item, we record the indexes of the frequent items in the array, as shown in Fig. 4(b). For example, frequent item A occurs at 1, 6, 12, 14, 19, 25. With this LocList, the projected database can be easily known since the flat representation is sorted by transaction ID and item. For example, the projected database for pattern A includes those tuples at $\{2, 3, 7, 8, 13, 15, 16, 20, 21, 26, 27\}$ since these tuples have the same *Tid* as A. By examining the items in these tuples, we can find local frequent items, e.g., C occurs at $\{2, 7, 15, 20, 26\}$, and F occurs at $\{3, 8, 13, 16, 21, 27\}$, which are exactly the LocList for frequent itemset $\{A, C\}$ and $\{A, F\}$, respectively. Formally, the projected database of a pattern is defined as follows.

Definition 4.2 (*Projected location list*). Given the location list (vertical format) of an itemset I, $I.LocList = \{t_1, t_2, \ldots, t_n\}$ in the flat (horizontal) database *IndexDB* (see Fig. 4), the *projected List* (*PList*) of I is defined as $I.PList = \{t'_1, t'_2, \ldots, t'_m\}$, where $t'_j.TID = t_i.TID$ for some t_i and $t_i < t_j \leqslant t_{|IndexTD|}$.

The main frame of the FIMA is outlined in Fig. 5. First of all, we scan database once and find frequent 1-items F_1 (line 1). Next, we transform the database into an array of 2-tuple (*Item*, *Tid*) sorted by *Tid* and then *Item*. Then, we maintain the *LocList* of

a

Loc	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tid	1	1	1	3	3	4	4	4	5	6	6	7	7	8	8	8
Item	A	C	F	B	D	A	C	F	D	B	D	A	F	A	C	F

Loc	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Tid	9	9	11	11	11	12	12	13	14	14	14	15	16	16
Item	B	D	A	C	F	B	D	D	A	C	F	C	B	D

b

Item	LocList
$\{A\}$	{1, 6, 12, 14, 19, 25}
$\{B\}$	{4, 10, 17, 22, 29}
$\{C\}$	{2, 7, 15, 20, 26, 28}
$\{D\}$	{5, 9, 11, 18, 23, 24, 28, 30}
$\{F\}$	{3, 8, 13, 16, 21, 27}

Fig. 4. Bi-format implementation of pseudoprojection for Fig. 1(a): a flat format sorted by Tid and Item (horizontal format), (b) the location lists of frequent items (vertical format).

Procedure of **FIMA(temporal database** TDB**,** $minsup$**)**
1. **Scan** TDB**, find frequent 1-item** F_1**;**
2. **Remove nonfrequent items and transform** TDB **into flat database** $IndexDB$**;**
   meanwhile maintain the locations of all F_1 **in the flat database;**
3. **for each** f_i **in** F_1 **do**
4. **Output** f_i **and its TidList;**
5. **fimajoin**($f_i, f_i.LocList$)**;**

Subprocedure of **fimajoin**(FP**,** $LocationList$)
6. **LFI = local frequent items in the projected location list of** FP **(i.e.** $FP.PList$)**;**
7. **for each** lf_j **in** LFI **do**
8. **Output** $FP \cup lf_j$ **and its TidList;**
9. **fimajoin**($FP \cup lf_j, lf_j.LocList$)**;**

Fig. 5. FIMA: Frequent Itemset mining using Memory Anchor.

each item in F_1 as searching anchors. For each f_i in F_1, we call subprocedure *fimajoin* to extend longer itemsets with prefix f_i (lines 3–4). In the subprocedure *fimajoin*, we find all local frequent 1-items lf_j by examining the projected list of input pattern (line 5). As an example, if we want to extend input pattern $\{A, C\}$ with LocList $\{2, 7, 15, 20, 26\}$, we will examine those tuples at $\{3, 8, 16, 21, 27\}$ since these tuples have the same *Tid* as $\{A, C\}$. The local frequent 1-items in this list (called the projected list) are $\{F\}$ with counts 5. Thus, new frequent itemsets are generated by uniting $\{A, C\}$ with $\{F\}$ (lines 6–7), and its LocList are exactly $\{3, 8, 16, 21, 27\}$ where F occurs.

The subprocedure *fimajoin* is applied recursively to enumerate all frequent itemsets with known frequent itemsets as their prefixes. For example, the projected list for pattern $\{A, C, F\}$ is $\emptyset$. The recursive call stops when no more frequent itemsets are generated. Finally, we output the TidList for each frequent itemsets (instead of LocList) by examining the locations recorded in locList. (We show the LocList for each frequent itemsets in Fig. 6(a) as a reference.) With local frequent items, we reduce a lot of unnecessary joins of the existing frequent itemset with any frequent 1-items. The gain in time is a tradeoff of the cost in space as in many algorithms. We will see how such tradeoffs are

a

ID	Item	LocList	boundlist
#1	$\{A\}$	{1, 6, 12, 14, 19, 25}	{[1,1],[4,4],[7,7],[8,8],[11,11],[14,14]}
#2	$\{B\}$	{4, 10, 17, 22, 29}	{[3,3],[6,6],[9,9],[12,12],[16,16]}
#3	$\{C\}$	{2, 7, 15, 20, 26, 28}	{[1,1],[4,4],[8,8],[11,11],[14,14],[15,15]}
#4	$\{D\}$	{5, 9, 11, 18, 23, 24, 30}	{[3,3],[5,5],[6,6],[9,9],[12,12],[13,13],[16,16]}
#5	$\{F\}$	{3, 8, 13, 16, 21, 27}	{[1,1],[4,4],[7,7],[8,8],[11,11],[14,14]}
#6	$\{A, C\}$	{2, 7, 15, 20, 26}	{[1,1],[4,4],[8,8],[11,11],[14,14]}
#7	$\{A, C, F\}$	{3, 8, 16, 21, 27}	{[1,1],[4,4],[8,8],[11,11],[14,14]}
#8	$\{A, F\}$	{3, 8, 13, 16, 21, 27}	{[1,1],[4,4],[7,7],[8,8],[11,11],[14,14]}
#9	$\{B, D\}$	{5, 11, 18, 23, 30}	{[3,3],[6,6],[9,9],[12,12],[16,16]}

b

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	#1		#2	#1	#4	#2	#1	#1	#2		#1	#2	#4	#1	#3	#2
	#3		#4	#3		#4	#5	#3	#4		#3	#4		#3		#4
	#5		#9	#5		#9	#8	#5	#9		#5	#9		#5		#9
ID	#6			#6				#6			#6			#6		
	#7			#7				#7			#7			#7		
	#8			#8				#8			#8			#8		

Fig. 6. Location list and encoding table: (a) encoding table of the frequent itemsets for Fig. 1(a), (b) encoded horizontal database *EDB*.

applied in serial episode mining in the following sections.

4.2.2. Encoded database construction

In the second phase, we associate each frequent itemset with a unique *ID* and construct a horizontal database *EDB* composed of these IDs. As shown in Fig. 6(b), *EDB* records the set of frequent itemsets (IDs) that occur at each time slot. Recall that we have output the TidList for each frequent itemset at Phase I, which can be easily transformed to boundLists for each frequent 1-tuple episode (see Fig. 6(a)). Using these the boundLists of frequent 1-tuple episode and the encoded database constructed from Phase II, pseudoproject can be easily implemented as presented below.

4.2.3. Frequent serial episode mining

Different from sequential pattern mining where the projected database is defined as the collection of suffixes of sequences with regards to a prefix α, there is no such concepts as sequence suffixes in serial episode mining. Instead, the mining of frequent serial episodes is guarded by the maximum window size *maxwin*, which is what we can use to define the projected database for an episode pattern.

Definition 4.3 (*Projected bound list*). Given the bound list of a serial episode P, $P.boundlist = \{[ts_1, te_1], \ldots, [ts_n, te_n]\}$ in the encoded database ED, the *projected bound list* (*PBL*) of P is defined as $P.PBL = \{[ts'_1, te'_1], \ldots, [ts'_n, te'_n]\}$ where $ts'_i = te_i + 1$ and $te'_i = min(ts_i + maxwin - 1, |TDB|)$. Discard the last bound ts'_n if it is greater than $|TDB|$.

For example, given 1-episode $\langle \#7 \rangle = \langle \{A, C, F\} \rangle$ with boundlist {[1,1], [4,4], [8,8], [11,11], [14,14]} and $maxwin = 4$, the projected database is a set of bounds including $[2, 4]$, $[5, 7]$, $[9, 11]$, $[12, 14]$, and $[15, 16]$. As another example, 1-episode $\langle \#9 \rangle = \langle \{B, D\} \rangle$ with boundlist $\{[3, 3], [6, 6], [9, 9], [12, 12], [16, 16]\}$ has the projected bound list $[4, 7]$, $[7, 9]$, $[10, 12]$, $[13, 15]$, $[17, 16]$, where the last bound should be discarded.

With projected database, we should be able to find local frequent IDs by counting the number of bounds an ID occurs. For example, in the projected list of episode $\langle \#7 \rangle$, #2, #4 and #9 occur at all five bounds, #6 and #7 occur at only three bounds, while others occur at four bounds. Given $minsup = 5$, only #2, #4 and #9 are frequent. These local frequent IDs, with respective boundlists, form new episode

by temporal joins with episode ⟨#7⟩. For example, the boundlist for ID #2 is {[3, 3], [6, 6], [9, 9], [12, 12], [16, 16]}, while the boundlist for ID #4 is {[3, 3], [5, 5], [6, 6], [9, 9], [12, 12], [13, 13], [16, 16]}. By temporal joins with the boundlist of episode ⟨#7⟩, we have the boundlist for episode ⟨#7, #2⟩ as {[1, 3], [4, 6], [8, 9], [11, 12], [14, 16]}, and the boundlist for episode ⟨#7, #4⟩ as {[1, 3], [4, 5], [4, 6], [8, 9], [11, 12], [11, 13], [14, 16]}. The bound list is the same as the temporal join of #7 and #4 in MINEPI+, where the support count is defined as the number of distinct bounds. Although we count the number of bounds for all IDs in this example, the number of IDs in an episode's projected boundlist is much less in real situations.

The complete algorithm of EMMA is outlined in Fig. 7. Lines 1 and 2 represent Phases I and II, respectively. Similar to FIMA, it adopts depth-first enumeration to generate longer serial episodes (lines 4–6, 9–12) by joining an existing serial episode with local frequent IDs (line 9). This is accomplished by examining those transactions following the matching bounds of current serial episodes, i.e., the projected database as described above. The procedure *emmajoin* is called recursively until no more new serial episodes can be extended.

As with sequential pattern mining, if the number of sequences in the projected database is less than the minimum support, it is impossible to grow patterns. Similarly, if the number of bounds in the projected database (called extendable bounds) for a serial episode P is less than the minimum support, then we can skip all extensions of the prefix P (line 5 and 11). For example, 1-episode ⟨#9⟩ with boundlist {[3, 3], [6, 6], [9, 9], [12, 12], [16, 16]} has only four bounds in its projected boundlist, i.e., {[4, 6], [7, 9], [10, 12], [13, 15]}, thus, there is no need to extend this prefix if the *minsup* is 5. This strategy further avoids some unnecessary checking spent in MINEPI+.

Definition 4.4 (*Extendable bounds*). The *extendable bounds* is defined as the number of bounds in the projected bound list of a serial episode P, denoted by $ExtCount(P)$.

Example 4.3. Let *minsup* and *maxwin* be 5 and 4 as before. We should illustrate the operation of Phase III for EMMA by completing the example with episode ⟨#7⟩. By examining the transactions in its projected boundlist, we find ID #2, #4, and #7 to be frequent as described above. For each local frequent ID, we use temporal join (line 10) to compute the boundlist for new episode ⟨#7, #2⟩, which is {[1, 3], [4, 6], [8, 9], [11, 12], [14, 16]}. Since the number of extendable bounds is four (including [4, 4], [7, 7], [10, 12], [13, 14]), there is no need to extend this prefix. We then proceed with next ID #4, which has boundlist {[1, 3], [4, 5], [4, 6], [8, 9], [11, 12], [11, 13], [14, 16]}. With extendable counts 6, we recursively call *emmejoin* to extend prefix ⟨#7, #4⟩. However, there is no frequent IDs in ⟨#7, #4⟩.*PBL* ({[4, 4], [6, 7], [7, 7], [10, 11], [13, 14], [14, 14]}). Thus, no new episodes are generated. Finally, we compute the boundlist for episode ⟨#7, #9⟩ as {[1, 3], [4, 6], [8, 9], [11, 12], [14, 16]}, which has extendable counts less than 5 (since the projected boundlist is {[4, 4], [7, 7], [10, 11], [13, 14]}), thus, complete the procedure call. Other episodes can be mined by applying the above process recursively to each 1-tuple episode, ⟨#1⟩, ⟨#2⟩, etc.

Procedure of **EMMA(temporal database** TDB**,** $minsup$**,** $maxwin$**)**
1. **Call FIMA(**TDB**,** $minsup$**) to find all frequent itemsets** FP_1 **and their TidLists;**
2. **Associate each itemset with an** ID **and construct the encoded database** EDB **;**
3. **for each** fid_i **in frequent** ID**s** FP_1 **do**
4. **Output** fid_i**;**
5. **if** $(ExtCount(fid_i.boundlist) \geq minsup * |TDB|)$
6. **emmajoin(**$fid_i, fid_i.boundlist$**);**

Procedure of **emmajoin(**$Episode$**,** $boundlist$**)**
7. LFP **= local frequent IDs in the projected bound list of** $Episode$ **(i.e.** $Episode.PBL$**);**
8. **for each** lf_i **in** LFP **do**
9. **Output** $Episode \cdot lf_i$**;**
10. $tempBoundlist = temporalJoin(boundlist, f_i.boundlist)$**;**
11. **if** $(ExtCount(tempBoundlist) \geq minsup * |TDB|)$
12. **emmajoin(**$Episode \cdot lf_i, tempBoundlist$**);**

Fig. 7. EMMA: Frequent Serial Episode Mining Using Memory Anchor.

4.3. Discussion

One of the reason that pattern growth is designed is to avoid generating candidate patterns for checking, which can also be applied to other mining tasks. However, the pseduoprojection actually requires both a horizontal-based encoded database and vertical-based boundlists for each frequent itemset. Therefore, the memory requirement for EMMA is greater than that for MINEPI+. When the number of frequent itemsets grows too large, it is unrealistic to maintain all patterns in the main memory. Fortunately, there is solution to memory shortage. First, we can maintain the boundlists of frequent itemsets in disk, then read them sequentially for episode extension. Theoretically, the disk-based EMMA can reduce half the memory requirement than the original EMMA. On the other hand, if the horizontal-based encoded database is still too large to fit the memory, then partition can be used to divide the problem to find local frequent episodes, with a final scan to the whole horizontal database for validation.

5. Experiments

In this section, we report the performance study of the proposed algorithms on both synthetic data and real-world data. All the experiments are performed on a 3.2 GHz Pentium PC, running Microsoft Windows XP. All the programs are written in Microsoft/Visual C++ 6.0.

5.1. Synthetic data

For performance evaluation, we use synthetically generated temporal data, D, consisting of N distinct symbols and $|D|$ time instants. A set of candidate patterns, C, is generated as follows. First, we decide the window length using geometrical distribution with mean W. This motivation is based on the observation that the shorter episode pattern is more frequent than longer. Again, the number of episode pattern increases as length decreases sharply. Therefore, we use geometrical distribution to simulate this observation. Then L $(1<L<W)$ positions are chosen for nonempty event sets. The average number of frequent events for each time slot is set to I. The number of occurrences of a candidate episode follows a geometrical distribution with mean $Sup * |D|$. A total of $|C|$ candidate patterns are generated. Next, we assign events to each time slot in D. The number of events in each time instant is picked from a Poisson distribution with mean T. For each time instant, if the number of events at this time instant is less than T, the insufficient events are picked randomly from the symbol set N. Table 2 shows the notations used and their default values.

Fig. 8 depicts the comparison results between MINEPI+ and EMMA for synthetic data with default parameter $minsup = 4\%$ and $maxwin = 5$. From Fig. 8(a), we can see that when the data size increases, the gap between MINEPI+ and EMMA in the running time becomes more substantial. EMMA is faster than MINEPI+ (by a magnitude of 150 for $|D| = 250$K). However, EMMA requires more memory as shown in Fig. 8(b). We also record the memory requirement of EMMA at Phase I, denoted by EMMA(I). If the timelists of frequent itemsets are maintained in the disk, the memory requirement will be $EMMA - EMMA(I)$. Therefore, EMMA(disk) needs approximately 27 MB at $|D| = 250$K.

The runtime of MINEPI+ and EMMA on the default data set with varying minimum support threshold, *minsup*, from 2% to 6% is shown in Fig. 8(c). Clearly, EMMA is faster and more scalable than MINEPI+, since the number of combinations in MINEPI+ grows rapidly as the *minsup* decreases, while EMMA only considers the local frequent patterns in the projected bound lists. Again, the memory requirement for EMMA increases as *minsup* decreases, since the number of frequent itemsets increases as *minsup* decreases (see Fig. 8(d)).

Fig. 8(e) shows the scalability of the algorithms with varying maximum window. Both curves in Fig. 8(e) go upwards because the number of frequent episodes increases exponentially as *maxwin* increases. However, EMMA still outperforms MINEPI+ with varying *maxwin*. In Fig. 8(f), the

Table 2
Meanings of symbols

Sym	Definition	Default
$\|D\|$	# of time instants	100K
N	# of events	1000
T	Average transaction size	6
$\|C\|$	# of candidate pattern	2
L	Average episode tuple size	3
I	Average itemset length	3
W	Average window length	5
Sup	Average support	4%

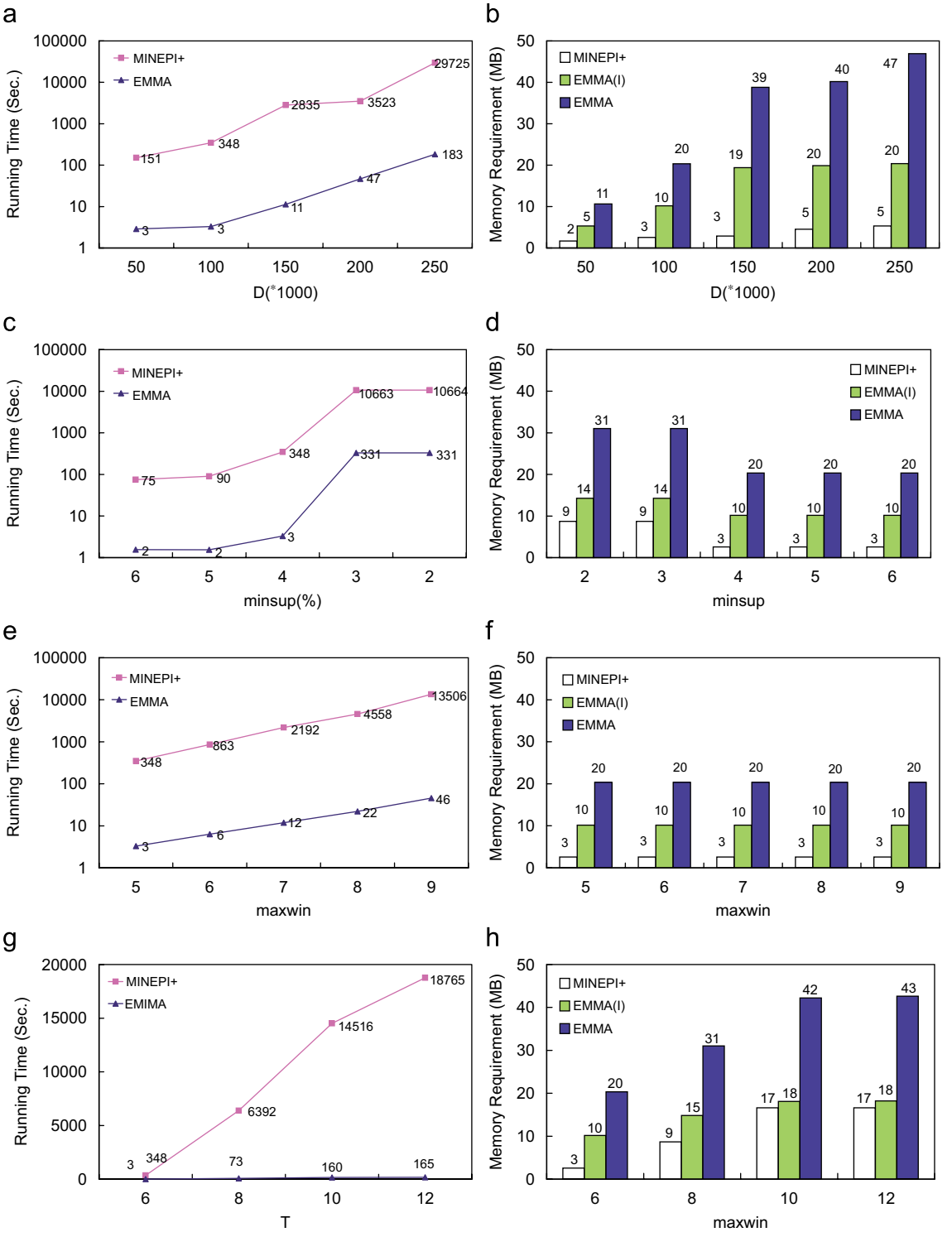

Fig. 8. Performance comparison using synthetic data: (a) running time vs. data size, (b) memory requirement vs. data size, (c) running time vs. *minsup*, (d) memory requirement vs. *minsup*, (e) running time vs. *maxwin*, (f) memory requirement vs. *maxwin*, (g) running time vs. *T*, (h) memory requirement vs. *T*.

memory requirement is steady for both MINEPI+ and EMMA. Thus, the maximum window threshold does not affect the memory requirement a lot. In Fig. 8(g), the total running time for MINEPI+ and EMMA are linear to the average transaction size T. However, for large transaction size, MINEPI+ requires significantly more time in equal join. In short, the performance study shows that the EMMA algorithm is efficient and scalable for frequent episode mining, and is about an order of magnitude faster than MINEPI+. However, MINEPI+ requires smaller and stable memory space than EMMA.

5.2. Real-world data

We also run our algorithms on a variety of different real-world data sets to get a better view of the usefulness of frequent episodes in a sequence.

5.2.1. Stock data

First, we apply MINEPI+ and EMMA to a data set composed of 10 stocks (electronics industry) in the Taiwan Stock Exchange Daily Official list for 2618 trading days from September 5, 1994 to June 21, 2004. We discretize the stock price of go-up/go-down into five levels: upward-high (UH): $\geqslant 3.5\%$, upward-low (UL): $<3.5\%$ and $>0\%$, changeless (CL): 0%, downward-low (DL): $>-3.5\%$ and $<0\%$, downward-high (DH): $\leqslant -3.5\%$. In this case, the number of events in each time slot is 10, and the number of events is 50 (10*5). Fig. 9(a) shows the running time with an increasing support threshold, *minsup*, from 10% to 30%. Fig. 9(c) shows the same measures with varying *maxwin*. As the *maxwin*/*minsup* threshold increases/decreases, the gap between MINEPI+ and EMMA in the running time becomes more substantial. Figs. 9(b) and (d) show the memory requirements and the number of frequent episodes with varying *minsup* and *maxwin*. As the *maxwin* threshold increases or *minsup* threshold decreases, the number of frequent episodes also increases. The memory requirement in MINEPI+ is steady. However, EMMA needs to maintain more frequent itemsets as the *minsup* decreases; whereas the

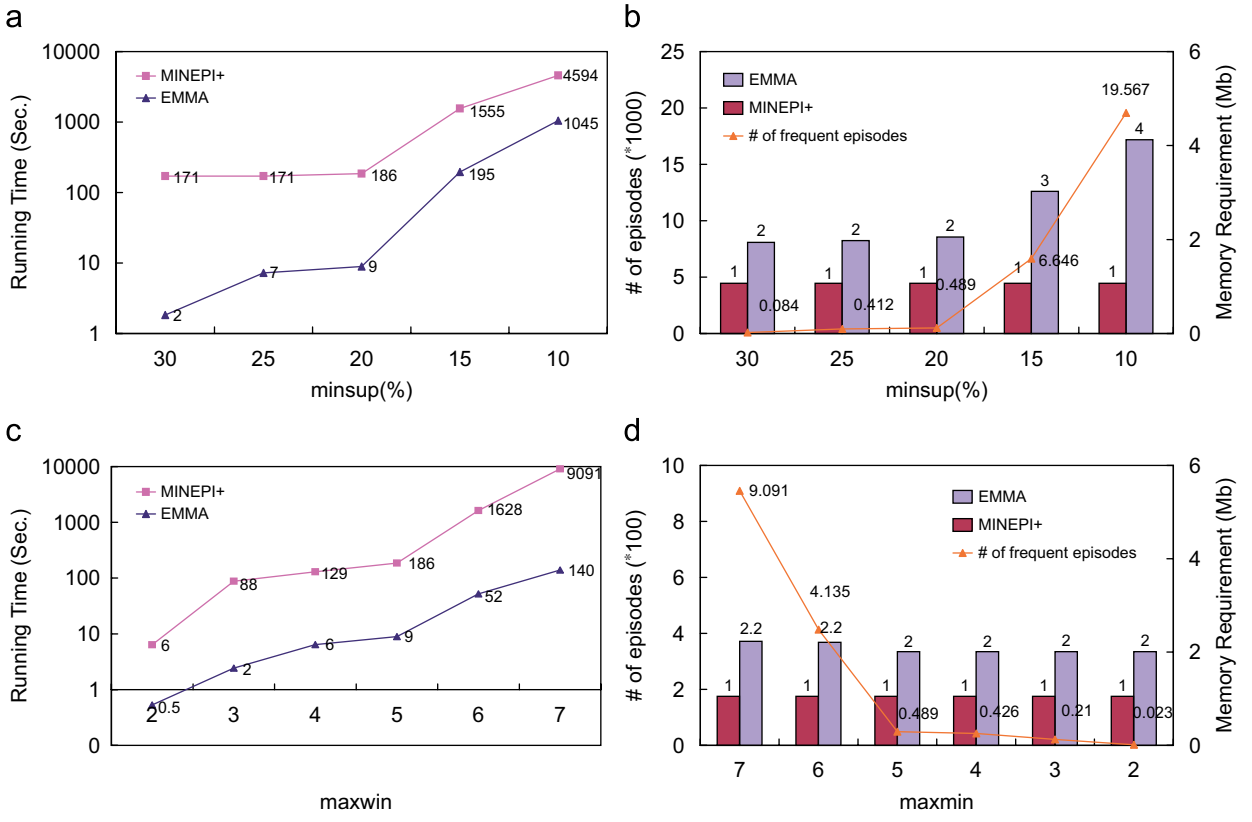

Fig. 9. Performance comparison using real data: (a) running time vs. *minsup*, (b) memory usage vs. *minsup*, (c) running time vs. *maxwin*, (d) memory usage vs. *maxwin*.

memory requirement with varying *maxwin* in EMMA is changed slightly. MINEPI+ is better than EMMA in memory saving (by a magnitude of 4 for $minsup = 10\%$).

We list some of the episode rules regarding the temporal relationship between TSMC (http://www.tsmc.com) and UMC (http://www.umc.com) under $minsup = 5\%$ and $maxwin = 3$ (sorted by confidence, then support) in Table 3. Since these two companies are both semiconductor manufacturing companies, there should have a close relationship among them. As shown in the table, top episodes rules usually have less constraints, thus it is less possible to use them for prediction. However, precedents and consequents often change in the same direction. Precedents with more constraints are much easier to understand. Whenever one of the companies goes down in a row, the other either goes down or rebounds.

5.2.2. UNIX log

Next, we employ our algorithms to mine the user's behaviors from usage logs. The data sets are taken from the UNIX user usage logs in UCI Machine Learning Database Repository. The UNIX user usage logs contains nine subsets of sanitized user data drawn from the command histories of eight UNIX computer users at Purdue over the course of up to 2 years. USER0 and USER1 were generated by the same person, working on different platforms and different projects. The description of the data used and the running time of our algorithms are presented in Table 4. Since the User6 and User8 have a lot of events, they cause a huge of combinations in MINEPI+. Besides, by mining frequent serial episodes in the usage logs, we can discover some users' usage behaviors. For example, we find an interesting serial episode $\langle$**SOF**, elm, exit, **EOF**$\rangle$ in USER0's usage log. It means that USER0 always login only for sending/receiveing e-mail.

5.2.3. Protein sequence

Finally, we apply our algorithms on protein sequences to discover repeated subsequence, which is an important problem in bioinformatics. We used data from the PROSITE database of the ExPASy Molecular Biology Server (http://www.expasy.org/). The purpose of our experiment is to evaluate frequent serial episode can find all previous known repeated subsequences. We selected a protein sequence P13813 (110K_PLAKN) with a known tandem repeats $\langle E, E, T, Q, K, T, V, E, P, E, Q, T\rangle$, $minsup = 10$ and $maxwin = 12$. The total length of this sequence is 296 events, with an alphabet of 22 event types. As expected, several frequent serial episodes which are related to the known tandem repeat are discovered. With this problem, we are only interested in the maximal frequent serial episodes. For example, we found six frequent serial

Table 3
Some episode rules for UMC and TSMC (two semiconductor manufacturing companies)

ID	Rule	Support (%)	Confidence (%)
1	⟨UMC-CL⟩ ⟶ ⟨TSMC-UL⟩	7.91	67.0
2	⟨TSMC-CL⟩ ⟶ ⟨UMC-UL⟩	6.04	64.5
3	⟨TSMC-CL⟩ ⟶ ⟨TSMC-UL⟩	6.04	64.5
4	⟨UMC-DL⟩ ⟶ ⟨UMC-UL⟩	20.9	64.1
5	⟨TSMC-DL⟩ ⟶ ⟨TSMC-UL⟩	21.9	63.7
6	⟨UMC-DH⟩ ⟶ ⟨UMC-UL⟩	6.68	63.6
7	⟨{UMC-DL,TSMC-DL}⟩ ⟶ ⟨TSMC-UL⟩	12.6	63.4
8	⟨{UMC-DL,TSMC-DL}⟩ ⟶ ⟨UMC-UL⟩	12.6	63.2
	…		
13	⟨{UMC-UL,TSMC-UL}⟩ ⟶ ⟨UMC-UL⟩	14.63	61.4
15	⟨{UMC-UL,TSMC-UL}⟩ ⟶ ⟨TSMC-UL⟩	14.40	60.4
20	⟨{UMC-UL,TSMC-UL}⟩ ⟶ ⟨TSMC-DL⟩	13.87	58.2
30	⟨{UMC-DL,TSMC-DL}⟩ ⟶ ⟨UMC-DL⟩	11.27	56.5
34	⟨{UMC-DL,TSMC-DL}⟩ ⟶ ⟨TSMC-DL⟩	11.08	55.6
37	⟨{UMC-UL,TSMC-UL}⟩ ⟶ ⟨UMC-DL⟩	12.41	52.1
40	⟨TSMC-UL,TSMC-UL⟩ ⟶ ⟨TSMC-UL⟩	6.34	27.9
	…		

Table 4
UNIX user usage log ($min = 5\%$ and $maxwin = 10$)

Data set	Events	Event types	# of serial episodes	MINEPI+ (s)	EMMA (s)
UNIX User0	8974	197	361	6.5	0.3
UNIX User1	19881	288	464	93.7	1.2
UNIX User2	18738	310	385	43.3	0.6
UNIX User3	16966	273	295	13.2	0.4
UNIX User4	37817	479	368	165.3	1.3
UNIX User5	34821	563	245	4.8	0.3
UNIX User6	64152	609	482	**2853.3**	14.6
UNIX User7	17329	449	192	0.6	0.2
UNIX User8	54042	706	246	**1362.3**	9.8

episodes with length 12. However, only one serial episode $\langle E, E, T, Q, K, T, V, E, P, E, Q, T\rangle$ is the tandem repeats, the others are tandem repeats with some varying shifts. This indicates that frequent episode mining can be used in finding repeated subsequences over biological sequence.

6. Conclusion and future work

In this paper, we discuss the problem of mining frequent episodes in a complex sequence and propose two algorithms to solve this problem. First, we modify previous vertical-based MINEPI [9] to MINEPI+ as the baseline for mining episodes in a complex sequence. To avoid the huge amount of combinations/computations and unnecessary/duplicate checking, we utilize memory to propose a brand-new memory-anchored algorithm, EMMA. (The extensions of the algorithms for parallel episode mining are given in the Appendix A.) The experiments show that EMMA is more efficient than MINEPI+ for both synthetic and real data set.

For future work, we can mine closed frequent episodes instead of all frequent episodes since the number of closed frequent itemsets is usually less than the number of frequent itemsets. We can mine frequent itemsets in Phase I and generate compressed frequent episodes or devise new algorithms for mining closed frequent episodes as suggested in [20]. So far we have only discussed serial and parallel episodes. The combination of serial and parallel episodes remains to be solved. As suggested in [10], the recognition of an arbitrary episode can be reduced to the recognition of a hierarchical combination of serial and parallel episodes. However, there are some complications one has to take into account. Thus, further researches are required.

Appendix A

In this section, we define the problem of frequent parallel episodes mining and discuss how to modify our algorithms for this problem.

Definition A.1. A *parallel episode* $I = \{i_1, \ldots, i_k\}$ ($i_j \in E$) is a set of events that occur within a window with length less than *maxwin*. We say that parallel episode I is also a k-event parallel episode.

Definition A.2. Given a parallel episode $I = \{i_1, \ldots, i_k\}$ and the window bound *win*, we say that the sliding window $W_i = (X_{t_i}, X_{t_i+1}, \ldots, X_{t_i+win-1})$ in *TDB* supports I if and only if, $I \subseteq U_i$ where

$$U_i = \bigcup_{j=0}^{win-1} X_{t_i+j}.$$

The number of sliding windows that match episode I is called the window count of I in the temporal database *TDB*.

Take parallel episode $I_1 = \{A, D\}$ in Fig. 1(a) for example. We find that I_1 is supported by 12 sliding windows (from W_1 to W_{12}). Thus, the parallel episode I_1 has 12 matches. Note that the number of windows that support an event can be as large as *win* times the number of occurrences for the event, since every event, except for those in the last $win - 1$ intervals, is counted by *win* windows. For example, E, although has only three appearances in Fig. 1(a), is supported by eight sliding windows. Thus, the minimum support for window counts should not be set too low, or there will be too many frequent 1-event parallel episodes.

Definition A.3. Given a *minsup*, we say an event x is *window frequent* if and only if it occurs more that *minsup* sliding windows.

Procedure of **MINEPI2(temporal database** TDB**,** $minsup$**,** $maxwin$**)**
1. **Scan** TDB **once, find window frequent items** WF_1 **and their matching** $windowlists$**;**
2. **for each** wf_i **in** WF_1 **do**
3. **ParallelJoins(**wf_i**,** $wf_i.windowlist$**,** wf_i**);**

Subprocedure of **ParallelJoins(**α**,** $windowlist$**,** $lastItem$**)**
4. **for each** $wf_j > lastItem$ **in** WF_1 **do**
5. $tempWindowlist$ = windowJoin(α, wf_j);
6. **if (**$|tempWindowlist| \geq minsup * |TDB|$**) then**
7. **ParallelJoins(**$\alpha \bigcup wf_j$**,** $tempWindowlist$**,** wf_j**);**

Fig. 10. MINEPI2: Vertical-based Frequent Parallel Episode Mining Algorithm.

Procedure of **EMMA2(temporal database** TDB**,** $minsup$**,** $maxwin$**)**
1. **Find all window frequent items** WF_1 **and their** $windowlists$**;**
2. **for each** wf_i **in** WF_1 **do**
4. **WJoin(**wf_i**,** $wf_i.windowlist$**,** wf_i**);**

Procedure of **WJoin(**β**,** $windowlist$**,**$lastitem$**)**
5. LF_1 **= local window frequent items in the projected** $windowlist$**;**
6. **for each** $lf_i > lastItem$ **in** LF_1 **do**
7. **if** $(|lf_i.windowlist| \geq minsup * |TDB|)$
8. **WJoin(**$\beta \bigcup lf_i$**,** $lf_i.windowlist$**,** lf_i**);**

Fig. 11. EMMA2: Frequent Parallel Episode Mining Using Memory Anchor.

The problem of frequent parallel episode mining is defined as discovering all parallel episodes that have at least *minsup* support count within the maximum window bound *win*. Using vertical format representation, we shall maintain the sliding windows that support each window frequent parallel episodes (called *matching window lists*). For example, $\{A\}.windowlist = \{1, 4, 7, 8, 11, 14\}$. Since we do not consider the order of events within a sliding window, we only need to check the common parts (i.e., via intersection) of two known window lists when extending a short frequent episode.

The modified MINEPI+ for frequent parallel episode mining proceeds as follows. Given a parallel episode $I = \{i_1, \ldots, i_k\}$, a window frequent 1-pattern *wf* and their matching window lists, e.g., $I.windowlist = \{IW_1, \ldots, IW_n\}$ and $wf.windowlist = \{FW_1, \ldots, FW_m\}$. The operation *windowJoin* of I and f which computes the window list for a new parallel episode $I' = \{i_1, \ldots, i_k, f\}$ (denoted by $I \cup f$) is defined as the intersection of the two window lists, $I.windowlist \cap wf.windowlist$. The modified MINEPI+ for parallel episodes is illustrated in Fig. 10. To avoid the duplicate enumeration, we use alphabetical order to generate long parallel episodes (line 4). Starting from each window frequent event wf_i, all frequent parallel episodes with prefix wf_i can be enumerated by recursive calls to *ParallelJoins*.

As for the parallel episode version of EMMA, the detailed modified algorithm is illustrated in Fig. 11. We shall see more clearly how EMMA differs from MINEPI in parallel episode mining. Instead of doing windowJoin directly in MINEPI, we will examine local frequent items from the memory anchors, i.e., the window list of the current episode. Conceptually, this algorithm is similar to the combination of FIMA algorithm and EMMA algorithm. However, instead of examine those transactions in FIMA, we need to check in those windows instead. More importantly, the support of an item is counted in terms of windows instead of transactions. For example, in the window list of parallel episode $\{A\}$, B has six occurrences instead of five. As usual, we shall need more memory space to facilitate quick checking as discussed in Section 4.3.

References

[1] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proceedings of the 11th International Conference on Data Engineering (ICDE'95), 1995, pp. 3–14.

[2] M. Garofalakis, R. Rastogi, K. Shim, Spirit: sequential pattern mining with regular expression of constraints, IEEE Trans. Knowl. Data Eng. 14 (3) (2002) 530–552.
[3] J. Han, J. Pei, Mining frequent patterns by pattern-growth: methodology and implications, ACM SIGKDD Explor. (special issue on Scalable Data Mining Algorithms) 2 (2) (2000) 14–20.
[4] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, M. Hsu, Freespan: frequent pattern-projected sequential pattern mining, in: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00), 2000, pp. 355–359.
[5] J. Pei, J. Han, B.M. Asl, H. Pinto, Q. Chen, U. Dayal, M. Hsu, Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth, in: Proceedings of the International Conference on Data Engineering (ICDE'01), 2001, pp. 215–226.
[6] R. Srikant, R. Agrawal, Mining sequential patterns: generalizations and performance improvements, in: Proceedings of the Fifth International Conference on Extending Database Technology (EDBT'96), 1996, pp. 3–17.
[7] M. Zaki, Spade: an efficient algorithm for mining frequent sequences, Mach. Learning 42 (1/2) (2001) 31–60.
[8] H. Mannila, H. Toivonen, A.I. Verkamo, Discovering frequent episodes in sequences, in: Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95), 1995, pp. 210–215.
[9] H. Mannila, H. Toivonen, Discovering generalized episodes using minimal occurrences, in: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96), 1996, pp. 146–151.
[10] H. Mannila, H. Toivonen, A.I. Verkamo, Discovering frequent episodes in event sequences, Data Mining Knowl. Discovery 1 (3) (1997) 259–289.
[11] J. Luo, S. Bridges, Mining fuzzy association rules and fuzzy frequency episodes for intrusion detection, Int. J. Intell. Syst. 15 (8) (2000) 687–704.
[12] M. Qin, K. Hwang, Frequent episode rules for internet anomaly detection, in: Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA), 2004.
[13] G.C. Garriga, Discovering unbounded episodes in sequential data, in: Proceedings of the Seventh European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD), 2003.
[14] N. Meger, N.L.C. Leschi, C. Rigotti, Mining episode rules in stulong dataset, in: ECML/PKDD 2004 Discovery Challenge (PKDD), 2004.
[15] A. Ng, A. Fu, Mining frequent episodes for relating financial events and stock trends, in: Proceedings of the Seventh Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), 2003.
[16] S. Harms, J. Deogun, J. Saquer, T. Tadesse, Discovering representative episodal association rules from event sequences, in: Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01), 2001.
[17] M. Atallah, R. Gwadera, W. Szpankowski, Detection of significant sets of episodes in event sequences, in: Proceedings of the Third IEEE International Conference on Data Mining (ICDM), 2004.
[18] R. Gwadera, M. Atallah, W. Szpankowski, Reliable detection of episodes in event sequences, in: Proceedings of the Third IEEE International Conference on Data Mining (ICDM), 2003.
[19] K. Iwanuma, Y. Takano, H. Nabeshima, On anti-monotone frequency measures for extracting sequential patterns from a single very-large data sequence, in: Proceedings of the First International Workshop on Knowledge Discovery in Data Streams, 2004.
[20] K. Huang, C. Chang, K. Lin, Cocoa: an efficient algorithm for mining inter-transaction associations for temporal database, in: Proceedings of the Eighth European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'04), 2004, pp. 509–511.
[21] K. Huang, C. Chang, K. Lin, Prowl: an efficient frequent continuity mining algorithm on event sequences, in: Proceedings of the Sixth International Conference on Data Warehousing and Knowledge Discovery (DaWak'04), 2004, pp. 351–360.
[22] A. Tung, H. Lu, J. Han, L. Feng, Efficient mining of intertransaction association rules, IEEE Trans. Knowl. Data Eng. 15 (1) (2003) 43–56.
[23] J. Han, G. Dong, Y. Yin, Efficient mining paritial periodic patterns in time series database, in: Proceedings of the 15th International Conference on Data Engineering (ICDE'99), 1999, pp. 106–115.
[24] S. Ma, J. Hellerstein, Mining partially periodic event patterns with unknown periods, in: Proceedings of the International Conference on Data Engineering (ICDE'01), 2001, pp. 205–214.
[25] B. Ozden, S. Ramaswamy, A. Silberschatz, Cyclic association rules, in: Proceedings of the 14th International Conference on Data Engineering (ICDE'98), 1998, pp. 412–421.
[26] J. Yang, W. Wang, P. Yu, Mining asynchronous periodic patterns in time series data, IEEE Trans. Knowl. Data Eng. 15 (3) (2003) 613–628.
[27] K. Huang, C. Chang, Smca: a general model for mining asynchronous periodic pattern in temporal database, IEEE Trans. Knowl. Date Eng. 17 (6) (2005) 774–785.
[28] J. Han, J. Pei, Y. Yin, R. Mao, Mining frequent patterns without candidate generation: a frequent-pattern tree approach, Data Mining Knowl. Discovery: Int. J. 8 (1) (2004) 53–87.
[29] J. Ayres, J. Flannick, J. Gehrke, T. Yiu, Sequential pattern mining using a bitmap representation, in: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02), 2002, pp. 429–435.
[30] K. Huang, C. Chang, Asynchronous periodic pattern mining in transactional databases, in: Proceedings of the IASTED International Conference on Databases and Applications (DBA'04), 2004, pp. 17–19.
[31] C.-M. Hsu, C.-Y. Chen, C.-C. Hsu, B.-J. Liu, Efficient discovery of structural motifs from protein sequences with combination of flexible intra- and inter-block gap constraints, in: Proceedings of the 10th Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2006, pp. 530–539.
[32] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, M. Hsu, Mining sequential patterns by pattern-growth: the prefixspan approach, IEEE Trans. Knowl. Data Eng. 16 (11) (2004) 1424–1440.
[33] M. Zaki, Scalable algorithms for association mining, IEEE Trans. Knowl. Data Eng. 12 (3) (2000) 372–390.