Regular Paper

Mining Sequential Patterns More Efficiently by Reducing the Cost of Scanning Sequence Databases

Jiahong Wang,[†] Yoshiaki Asanuma,[†,☆] Eiichiro Kodama,[†]
Toyoo Takata[†] and Jie Li[††]

Sequential pattern mining is a useful technique used to discover frequent subsequences as patterns in a sequence database. Depending on the application, sequence databases vary by number of sequences, number of individual items, average length of sequences, and average length of potential patterns. In addition, to discover the necessary patterns in a sequence database, the support threshold may be set to different values. Thus, for a sequential pattern-mining algorithm, responsiveness should be achieved for all of these factors. For that purpose, we propose a candidate-driven pattern-growth sequential pattern-mining algorithm called FSPM (Fast Sequential Pattern Mining). A useful property of FSPM is that the sequential patterns concerning a user-specified item can be mined directly. Extensive experimental results show that, in most cases FSPM outperforms existing algorithms. An analytical performance study shows that it is the inherent potentiality of FSPM that makes it more effective.

1. Introduction

Sequential pattern mining is a technique used to discover frequent subsequences as patterns in a sequence database. Given a set of sequences called a *sequence database*, where each sequence is a list of items, and given a user-specified support threshold *MinSup*, sequential pattern mining's task is to find all the *frequent subsequences*, i.e., the subsequences whose occurrence frequency in the sequence database is not less than *MinSup* [2]. Note that if something is said to be frequent, we mean its occurrence frequency $\geq$ *MinSup*. Sequential pattern mining algorithms are necessary for numerous applications. A running example found in Ref. 10) is given below. A similar application on the Web can be found in Ref. 12).

Example 1 (Running Example). *The CPU, memory, and hard disk are essential components of a computer. The hard disk is far slower than the CPU and memory, and has become a bottleneck in modern computers. A solution to the bottleneck problem is to discover the frequently pursued sequences of access requests (i.e., the access patterns) and to implement a prefetching cache based on those patterns. To do this, sequential pattern mining is necessary: One user's access request log for one session is represented as a sequence of block numbers; logs from all users form a sequence database; users' access patterns are mined in this sequence database.*

Most earlier algorithms for sequential pattern mining are based on the Apriori property [1] that any subsequence of a sequential pattern must be frequent. Based on this heuristic, a series of Apriori-based algorithms such as AprioriAll [2], GSP [15], PSP [11], Spirit [7], and SPADE [20] was proposed. This series of algorithms is characterized by the *candidate generation-and-tests* that first generates a set of candidates and then tests whether each candidate is sufficiently supported. Another series of sequential pattern mining algorithms are *pattern growth* algorithms such as FreeSpan [9], WAPTree-based Web mining [6),12], PrefixSpan [13),14], and SPAM [3),19]. PrefixSpan is a representative one that uses three strategies for efficiently finding sequential patterns [5]: candidate sequence pruning, database partitioning, and customer sequence reducing. With PrefixSpan there is no need for candidate generation, and only recursive projects of databases according to their prefixes are created. Pattern growth algorithms have been shown to outperform the Apriori-based ones [14].

One challenge in sequential pattern mining is to reduce the cost of scanning a sequence database. Although PrefixSpan is one of the fastest algorithms, we found that it suffers from the cost of redundant scanning of the same

† Iwate Prefectural University
†† University of Tsukuba
☆ Presently with Internet Initiative Japan Inc.

data. For example, assume that the set of frequent items is $\{a, b, c, d, e\}$, and an access sequence $S = bafdecg$. PrefixSpan first divides the database into five subsets according to the five items: (1) ones with prefix a, (2) ones with prefix b, ..., and (5) ones with prefix e. Then it generates patterns. While generating the patterns with prefix a, starting from a, S is scanned once; and while generating the patterns with prefix b, starting from b, S is scanned again. The subsequences containing a are processed twice. Considering that in the same time the patterns with prefix a are being generated, we could also generate patterns containing a, there is really a lot of wasted work in PrefixSpan that actually can be avoided.

Motivated by the above observation, we propose an algorithm called FSPM (Fast Sequential Pattern Mining). The basic idea of FSPM is as follows. Given a sequence database *SD*, we can mine all the patterns in *SD* by mining the patterns containing each frequent item in *SD* separately. For each frequent item, we can first find the set of all such items that head the patterns containing that frequent item, and then grow patterns from short ones level by level, starting from that set.

FSPM is a candidate-driven pattern-growth algorithm. Utilizing the concepts of candidate set and search space greatly reduces the cost of growing patterns. FSPM is more efficient than PrefixSpan. On the one hand, FSPM uses a two-level problem partitioning strategy to reduce the cost incurred by PrefixSpan. For the above example, FSPM first generates all patterns containing a, and thereafter a need not be considered. Similarly, after those containing b have been generated, b need not be considered, and so on. Thus the wasted work is eliminated. On the other hand, FSPM uses a level-by-level pattern-growing strategy to reduce the cost of recursively projecting databases and make it possible to generate as many patterns as possible by scanning the sequence database only once. This strategy also assures that FSPM uses memory resources more reasonably. We found in our experiments that for PrefixSpan, to recursively project databases costs too much, and no matter how large a memory a computer has, it uses only a small part of it, both of which greatly degrade efficiency. Note that the previous version of PrefixSpan published in Ref. 13) introduces a technique called *bilevel projection*, which shares FSPM's objective of reducing the number of projections. However, since PrefixSpan with the bilevel projection is also a PrefixSpan algorithm, the differences between PrefixSpan and FSPM as stated above still hold.

A useful property of FSPM is that it can be used to directly discover the patterns that contain a user-specified item. Algorithms such as PrefixSpan and SPAM mine a full set of the patterns that reach a support threshold. Algorithms in Refs. 10), 16), 18) mine a full set of the patterns containing no subpatterns with the same support. Algorithms in Refs. 3), 14) mine the patterns beginning with a user-specified item. But for the running example, in addition to all the patterns, the patterns that contain a specific item are required. To the best of our knowledge, there have been no efficient methods developed for this purpose.

Extensive experiments comparing FSPM with PrefixSpan have been conducted. Performance was evaluated in terms of five factors, which were chosen based on the following consideration. On the one hand, sequence databases vary mainly in four dimensions. (1) Depending on, e.g., the size of a sliding window [4),10),15)], sequence databases vary by the number of sequences. (2) Depending on the type of application, sequence databases vary by the number of individual items. For the running example, the number of individual items is equal to the number of data blocks on the hard disk. (3)(4) Depending on the characteristics of users, sequence databases vary by the average length of sequences and the average length of potential patterns. On the other hand, given a sequence database, a support threshold is necessary so as to extract only the necessary patterns. There are the five factors. The experimental results demonstrate that FSPM is more effective than PrefixSpan. An analytical performance study shows that it is the inherent potentiality of FSPM that makes it effective.

The remainder of the paper is organized as follows: In Section 2, the sequential pattern-mining problem is defined. In Section 3, FSPM is stated and formally proved. The experimental results and performance analysis are reported in Section 4. The paper is concluded in Section 5.

2. Problem Statement

Let $I = \{e_1, e_2, \cdots, e_m\}$ be a set of all items. A sequence S is an ordered list of items, denoted by $< e_1^{'} e_2^{'} \cdots e_n^{'} >$, where $e_i^{'} \in I$ $(i \leq n)$.

The number of instances of individual items in a sequence is called the length of the sequence. A sequence with length l is called a *length-l sequence*.

A sequence $\alpha = a_1 a_2 \cdots a_{l_\alpha}$ is called a subsequence of another sequence $\beta = b_1 b_2 \cdots b_{l_\beta}$, denoted as $\alpha \subseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < \cdots < j_{l_\alpha} \leq l_\beta$, such that $a_i = b_{j_i}$ for $i \leq l_\alpha$.

A sequence database, denoted by *SD*, is a set of tuples <*sid*, *S*>, where *S* is a sequence, and *sid* is its identifier. The support count of a sequence α, denoted by $support_{SD}(\alpha)$, is the number of tuples containing α in *SD*, defined as $|\{<sid, S> \mid <sid, S> \in SD \wedge \alpha \subseteq S\}|$. The support of a sequence α, denoted by $sup_{SD}(\alpha)$, is defined as $support_{SD}(\alpha) / |SD|$.

Given a real number *MinSup* ($0 \leq MinSup \leq 1$) as support threshold, a sequence α is called a *sequential pattern* in *SD* if $sup_{SD}(\alpha) \geq MinSup$. In this case, α is called a *MinSup-pattern*, or a *frequent pattern*, or simply, the *pattern*.

Problem Statement 1. *The problem of sequential pattern mining in this paper is as follows. Given a sequence database SD and a support threshold MinSup, mine the complete set of MinSup-patterns of SD.*

Example 2. *Continuing with the running example, let us consider the following hard disk access log data. Each record in the log is of the format <UserId, BlkNo>. For simplicity, the set of BlkNos is assumed to be {a, b, c, d, e, f, g, h, i}, and the set of UserIds is assumed to be {100, 200, 300, 400, 500}:*
<100, f> <200, c> <200, i> <100, e> <200, a> <500, b> <300, f> <200, f> <300, c> <300, b> <100, b> <300, e> <300, a> <300, d> <400, b> <400, h> <200, e> <400, e> <200, d> <100, a> <400, c> <100, d> <400, a> <400, f> <400, d> <500, c> <500, f>

We can divide the log data into access sequences based on UserId. The resulting access sequence database is shown in **Table 1**. *There is a total of five access sequences. The access sequence of user 100, febad, is a length-5 sequence, while fd and fe are two subsequences of it. fe is a 60%-pattern because it gets support from the three access sequences of user 100, 200, and 300.*

Table 1 A sequence database SD.

UserId	Hard disk access sequences
100	f e b a d
200	c i a f e d
300	f c b e a d
400	b h e c a f d
500	b c f

3. Mining Patterns by FSPM

In this section, accompanied first by explanations using Example 2 (assume $MinSup = 0.6$), FSPM is systematically stated. Then, an example is given to illustrate it. Finally, considerations for its implementation are presented.

3.1 The Algorithm FSPM

FSPM consists of three main operations: partitioning problems into two levels, collecting support counts in projected databases, and generating patterns.

3.1.1 Problem partitioning

Suppose that we are mining frequent sequential patterns in the environment given in Example 2 (hereafter, for simplicity we will use "pattern" instead of "frequent sequential pattern"). Support counts are first collected: {*a*:4, *c*:4, *d*:4, *e*:4, *b*:4, *f*:5, *h*:1, *i*:1}; note that the order of items is not important. Extracting the frequent items we have list *FIList* = <*a*, *c*, *d*, *e*, *b*, *f*>, each element of which is also a length-1 pattern. According to *FIList*, we can divide the set of patterns in *SD* into six disjoint subsets: 1) the ones that contain *a*; 2) the ones that contain *c* but do not contain *a*; ....; 6) the ones that contain *f*, but do not contain *a*-*b*. Accordingly, the patterns related to the six subsets can be mined by constructing six new databases derived from *SD*: 1) the sequences containing *a*; 2) the sequences containing *c*, but with item *a* removed; ...; 6) the sequences containing *f*, but with all items before *f* removed. Motivated by this idea, the problem of mining patterns can be divided into a set of subproblems as follows:

Lemma 1 (Level 1 problem partitioning). *Given a sequence database SD. Let FIList = $\{x_1, x_2, \cdots, x_n\}$ be a complete list of the frequent items in SD. The set of patterns (recall that we use the word "pattern" in place of the phrase "frequent sequential pattern") in SD can be divided into n disjoint subsets. The ith ($i \leq n$) subset, denoted by PSetT(x_i), includes the patterns that contain x_i, but do not contain x_j ($j < i$).*

Proof. By the Apriori heuristic, patterns in *SD* are solely composed of items in *FIList*. On the other hand, for any $x \in FIList$, there exists at least one pattern containing x: x itself. Thus the complete set of patterns in *SD* can be partitioned into such n disjoint subsets that the ith ($i \leq n$) subset includes patterns that contain

x_i, but do not contain x_j ($j < i$). So we have the lemma. □

Corollary 1 (Identifying patterns in SD). *Given a sequence database SD with FIList = $\{x_1, x_2, \cdots, x_n\}$. The complete set of patterns in SD can be mined by identifying PSetT(x_i) ($i \leq n$) separately and in order.*

For Example 2, the patterns can be mined by identifying *PSetT(a)* = {*a, ad, ca, ea, ba, cad, ead, bad*}, *PSetT(c)* = {*c, cd, cf*}, *PSetT(d)* = {*d, ed, fd, fed, bd*}, *PSetT(e)* = {*e, fe*}, *PSetT(b)* = {*b*}, and *PSetT(f)* = {*f*} separately and in order. Next we define some terms related to identifying patterns.

Definition 1 (Target item, seed, domain). *Given a sequence database SD with FIList = $\{x_1, x_2, \cdots, x_n\}$. Assume we are identifying PSetT(x_i) ($i \leq n$). Item x_i is called the target item. The complete set of such sequences in SD that contain x_i, but with x_j ($j < i$) removed, is called x_i's domain and denoted by Domain(x_i). $\{x_i, x_{i+1}, \cdots, x_n\}$ is called a seed set and is denoted by Seed(x_i).*

For Example 2, we have six domains: *Domain(a)* = {*febad, ciafed, fcbead, bhecafd*}, *Domain(c)* = {*cifed, fcbed, bhecfd, bcf*}, *Domain(d)* = {*febd, ifed, fbed, bhefd*}, *Domain(e)* = {*feb, ife, fbe, bhef*}, *Domain(b)* = {*fb, fb, bhf, bf*}, and *Domain(f)* = {*f, if, f, hf, f*}. Given a target item, say item *d*, *PSetT(d)* can be mined in the *Domain(d)*. At that time, we need to know the complete set of such items that head patterns in *PSetT(d)*, which is called the *FIRST* item set with regard to item *d*. For defining and calculating the *FIRST* we introduce the concepts of prefix and maximal prefix.

Definition 2 (Prefix and maximal prefix). *Given a sequence $\alpha = e_1 e_2 \cdots e_{l_\alpha}$. $e_{k'}$ ($k \leq l_\alpha$) is an item in α. A sequence $\beta = e'_1 e_2 \cdots e'_k$ is called a prefix of α with regard to e_k, and is denoted by $\alpha[1:k]$ or $\alpha[e_k]$ for simplicity, if and only if $e'_i = e_i$ for $i \leq k$. In the case of $e_j \neq e_k$ ($k < j \leq l_\alpha$, or $k = l_\alpha$), $\alpha[e_k]$ is called the maximal prefix, denoted by $\alpha[e_k]_{max}$.*

For example, given the item *c* and a sequence α = *abcdecfg*, both *abc* and *abcdec* are its prefixes, and are represented by $\alpha[1:3]$ and $\alpha[1:6]$, respectively. The latter is the maximal prefix, denoted by $\alpha[c]_{max}$.

Definition 3 (FIRST set). *Given a target item x. Let PSetT(x) = $\{\alpha_1, \alpha_2, \cdots, \alpha_m\}$. Item set $\{f_1, f_2, \cdots, f_n\}$ is called the FIRST item set with regard to x, denoted by FIRST(x), if and only if (1) for any $i \leq n$, there exists $j \leq m$, $f_i = \alpha_j[1:1]$; (2) for any $i \leq m$, there exists $j \leq n$, $\alpha_i[1:1] = f_j$.*

For Example 2 we have *FIRST(a)* = {*a, b, c, e*}, *FIRST(c)* = {*c*}, *FIRST(d)* = {*b, d, e, f*}, *FIRST(e)* = {*e, f*}, *FIRST(b)* = {*b*}, and *FIRST(f)* = {*f*}.

Lemma 2 (Calculate FIRST). *Let x be the target item, and Domain(x) = $\{\gamma_1, \gamma_2, \cdots, \gamma_p\}$. FIRST(x) is equal to the complete set of frequent items in the maximal prefix set $\{\gamma_1[x]_{max}, \gamma_2[x]_{max}, \cdots, \gamma_p[x]_{max}\}$.*

Proof. (1) Let $e \in$ *FIRST(x)*. By Definition 3 and Lemma 1, $e \in$ *PSetT(x)* if $e = x$, and $ex \in$ *PSetT(x)* if $e \neq x$. Thus, by the definition of maximal prefix, *e* must be frequent in the maximal prefix set. (2) Let *e* be frequent in the maximal prefix set. By Lemma 1, *ex* ($e \neq x$) or *e* ($e = x$) $\in$ *PSetT(x)*. By Definition 3 we know that $e \in$ *FIRST(x)*. So we have the lemma. □

Applying this lemma to Example 2, taking *a* as a target item, the maximal prefix set {*feba, cia, fcbea, bheca*} is found first, then sequences in it are scanned, and finally, *FIRST(a)* = {*a, b, c, e*} is obtained. When we have obtained the *FIRST*, using elements in it we can partition the pattern mining problem further by Lemma 3 stated below.

Lemma 3 (Level 2 problem partitioning). *Let x be a target item, and FIRST(x) = $\{f_1, f_2, \cdots, f_m\}$. PSetT(x) can be divided into m disjoint subsets. The ith subset ($i \leq m$) contains the patterns with prefix f_i, denoted by PSetF(f_i).*

Proof sketch. It follows from Definition 3. □

For Example 2, taking *d* as the target item, we have *FIRST(d)* = {*b, d, e, f*}. Accordingly, *PSetT(d)* can be divided into four subsets: *PSetF(b)* = {*bd*}, *PSetF(d)* = {*d*}, *PSetF(e)* = {*ed*}, and *PSetF(f)* = {*fd, fed*}.

In summary, the problem of mining sequential patterns can be divided into two levels. For Example 2, the first level is to take each item in *FIList* = {*a, c, d, e, b, f*} as a target item to generate the corresponding *PSetT* separately. Given a target item, its *PSetT* is generated by calculating the *FIRST* set first, and then, in the second level, generating *PSetF* for each item in *FIRST*.

3.1.2 Collecting Support Counts in Projected Databases

PSetF can be generated by generating length-(l+1) patterns from a length-l pattern that has already been mined. The items that

qualify for attachment to the length-l pattern, called the *qualified items*, are the items that are sufficiently supported in the projected database of length-l pattern.

Definition 4 (Projected database). *Let α be a pattern in Domain(x). The α-projected database, denoted by $Domain(x)|_\alpha$, is the set of suffixes of sequences in Domain(x) with regard to α. The term suffix is defined as follows. Given a sequence $\varphi = e_1e_2\cdots e_n$. Let $\beta = a_1a_2\cdots a_m$ $(m \le n)$ be a subsequence of φ, and $a_m = e_{m'}$ $(m \le m')$. Sequence $\gamma = e_{m'+1}e_{m'+2}\cdots e_n$ is called the suffix of φ with regard to β, and is denoted by $\gamma = \varphi/\beta$.*

For example, given $\varphi = abcdecfg$ and $\beta = abc$, $decfg$ is the suffix of φ with regard to β, as represented by $abcdecfg/abc$.

For Example 2, take d as target item, and f as α, then $Domain(d)|_f = \{ebd, ed, bed, d\}$.

Definition 5 (Support count). *Let x be a target item, α be a pattern in Domain(x), and β be a sequence with prefix α. The support count of β is the number of sequences γ in $Domain(x)|_\alpha$ such that $\beta \subseteq \alpha\gamma$, denoted by $support_{Domain(x)|_\alpha}(\beta)$.*

Following the above example, let $\alpha = f$, $\beta = fe$. By definition, $support_{Domain(d)|_\alpha}(\beta) = 3$.

Lemma 4 (Collect support counts). *Let x be a target item, α and β be two patterns in Domain(x) such that α is a prefix of β. Then $support_{Domain(x)}(\beta) = support_{Domain(x)|_\alpha}(\beta)$.*

Proof sketch [14]. This lemma states that to collect the support count of a sequence β, only the sequences in *Domain(x)* sharing the same prefix α should be considered. Furthermore, only those suffixes with the prefix being a supersequence of β should be counted. The claim follows from the related definitions. □

Applying this to Example 2, by collecting the support count of the item e in $Domain(d)|_f$, we know that e is frequent in $Domain(d)|_f$. Therefore we can generate pattern fe from f. Similarly we can generate pattern fed from fe by collecting the support count of d in $Domain(d)|_{fe}$. These are justified by Lemma 4.

Note that Definition 5 and Lemma 4 are the same as those given in Ref. 14), except that here we deal with a target item's domain, rather than with the whole database.

3.1.3 Generating Patterns in PSetF

To make sure each generated pattern includes the target item, and also to be efficient, we introduce the concepts of *candidate set* and *search space*.

Lemma 5 (Candidate set, search space). *Given a sequence database SD with FIList = $\{x_1, x_2, \cdots, x_n\}$. Let x_i $(i \le n)$ be the target item, and α a length-l pattern in $Domain(x_i)$. $\alpha^+ = \{\beta_1, \beta_2, \cdots, \beta_m\}$, the complete set of length-(l+1) patterns with prefix α, is to be generated. The qualified items, $QIS(\alpha) = \{\beta_1/\alpha, \beta_2/\alpha, \cdots, \beta_m/\alpha\}$, need to be determined. The set of subsequences where $QIS(\alpha)$ is searched is called the search space and is denoted by $SS(\alpha)$. The set of items that can be members of $QIS(\alpha)$ is called the candidate set and is denoted by $CS(\alpha)$. Candidate set and search space are determined in four cases (called "states"):*

1. *Find-FIRST (α = Null): In this state, the $FIRST(x_i)$ is determined. $SS(Null) = \{\gamma_1[x_i]_{max}, \gamma_2[x_i]_{max}, \cdots, \gamma_p[x_i]_{max}\}$, where $\{\gamma_1, \gamma_2, \cdots, \gamma_p\} = Domain(x_i)$. $CS(Null) = Seed(x_i)$.*
2. *Pre-Target ($x_i \not\subseteq \alpha$): $SS(\alpha) = SS(Null)|_\alpha$. $CS(\alpha) = QIS(\alpha^-)$, where α^- denotes the length-(l-1) prefix of α.*
3. *Is-Target ($x_i \not\subseteq \alpha[1:l-1] \wedge \alpha[l:l] = x_i$): $SS(\alpha) = Domain(x_i)|_\alpha$, and $CS(\alpha) = Seed(x_i)$.*
4. *Post-Target ($x_i \subseteq \alpha[1:l-1]$): $SS(\alpha) = Domain(x_i)|_\alpha$. $CS(\alpha) = QIS(\alpha^-)$.*

Proof. We give a proof for each state.

1. Find-FIRST: $SS(\alpha)$ and $CS(\alpha)$ follow from Definition 3 and Lemma 2.

2. Pre-Target: Since any pattern has to include the target item, any item that is not frequent in the maximal prefix set cannot be a candidate in this state. So we have the $SS(\alpha)$. On the other hand, assume $e \in QIS(\alpha)$, but $e \notin QIS(\alpha^-)$. This means that αe is a pattern, but $\alpha[1:l-1]e$ is not. With the Apriori heuristic, this is impossible. So we have the $CS(\alpha)$.

3. Is-Target: Since in this state the target item has been included, any item in $Domain(x_i)|_\alpha$ can be a candidate to generate α^+. So we have the $SS(\alpha)$. Since the QIS needs to be renewed, we have the $CS(\alpha)$.

4. Post-Target: $SS(\alpha)$ follows from derivation in State 3, $CS(\alpha)$ in State 2.

So we have the Lemma. □

For Example 2, take c as the target item, $Domain(c) = \{cifed, fcbed, bhecfd, bcf\}$. When generating the $FIRST(c)$, we have $SS(Null) = \{c, fc, bhec, bc\}$ and $CS(Null) = \{c, d, e, f, b\}$. Searching $SS(Null)$ we find $FIRST(c) = \{c\}$. Since c is the target item, we enter state 3, and have $SS(c) = \{ifed, bed, fd, f\}$ and $CS(c) =$

CS(Null). Searching *SS(c)* we obtain length-2 patterns {*cd, cf*}. Here we have used the short pattern *c* to generate long ones *cd* and *cf*. The pattern growth method is justified as follows.

Lemma 6 (Generate PSetF). *Let f be a member of a given FIRST item set. A pattern with prefix f belongs to PSetF(f) if and only if it is generated by the following two steps and it is in either Is-Target or Post-Target state. (1) f is the length-1 pattern. (2) Let α be a length-l pattern ($l \geq 1$), and QIS(α) = $\{q_1, q_2, \cdots, q_p\}$. Then $\{\alpha q_1, \alpha q_2, \cdots, \alpha q_p\}$ is the complete set of length-(l+1) patterns with the prefix α.*

Proof sketch. (Direction if). The two steps generate a pattern, according to the Apriori heuristic. The generated pattern has the prefix f, according to Step 1. Since it is in either *Is-Target* or *Post-Target* state, it contains the target item. By this we show that a pattern is in *PSetF(f)* if it is generated by the two steps, and is in either *Is-Target* or *Post-Target* state. (Direction only-if) Let $\beta = b_1 b_2 \cdots b_n$ be any pattern in *PSetF(f)*. Since β contains the target item, according to Lemma 5, it must be in either *Is-Target* or *Post-Target* state. Next we prove that β can be generated by the two steps. Step 1 assures that subpattern $\beta[1:1]$, which is the item f, can be generated. Assume $\beta[1:l]$ ($1 \leq l < n$) is the subpattern that has been generated. By the Apriori heuristic, b_{l+1} is frequent. Thus $b_{l+1} \in QIS(\beta[1:l])$, and thus Step 2 can generate $\beta[1:l+1]$. By this we show that if a pattern is in *PSetF(f)*, then it is in either *Is-Target* or *Post-Target* state, and is generated by the two steps. Thus we have the lemma. □

3.1.4 Algorithm FSPM

The algorithm FSPM is given in the next page, and its correctness and completeness can be justified on the basis of Theorem 1. We have also compared the results of FSPM with those of PrefixSpan using datasets, and found that they have the same results.

Theorem 1 (FSPM). *Given a sequence database SD, a sequence α in SD is a sequential pattern if and only if FSPM identifies it as a sequential pattern.*

Proof sketch. (Direction if). Given a target item x, a length-l sequence α ($l \geq 1$) is identified as a pattern by FSPM if and only if α is a pattern in the projected database of its length-(l-1) prefix α^- in *Domain(x)*. When l = 1, the length-0 prefix of α is α^- = Null, and the projected database is *Domain(x)* itself. So, α is a pattern in *Domain(x)*. When $l >$ 1, according to Lemma 4, $support_{Domain(x)}(\alpha) = support_{Domain(x)|_{\alpha^-}}(\alpha)$. Therefore, if α is a sequential pattern in $Domain(x)|_{\alpha^-}$, it is also a sequential pattern in *Domain(x)*. Considering that the above is conducted for every target item, we have shown that a sequence α is a sequential pattern if FSPM identifies it as a sequential pattern.

(Direction only-if). Corollary 1, Lemma 3, and Lemma 6 guarantee that FSPM identifies the complete set of patterns in *SD*.

So we have the theorem. □

3.2 An Example for Illustrating FSPM

In this subsection we use Example 2 to illustrate FSPM. For the same database *SD* given in Table 1 and *MinSup* = 0.6, FSPM behaves as follows.

1. Scan *SD* to find all the frequent items, and we have *FIList* = <*a*:4, *c*:4, *d*:4, *e*:4, *b*:4, *f*:5>. Each item in *FIList* is a length-1 pattern.
2. Remove from *SD* the items that are not in *FIList*.
3. First take the item *a* as the target item. Searching the maximal prefix set {*feba, ca, fcbea, beca*} we have *FIRST(a)* = {*a, b, c, e*}, which is used as the *Level(1)*, as shown in **Fig. 1**.
4. Stamp each pattern with a state. The state of item *a* is stamped with "$\alpha[1:1]$ is the target", and others with "Target has not been included".
5. Grow patterns. For pattern *a*, the level 2 items (i.e., the items that qualify to be attached to *a* to form length-2 patterns with prefix *a*) are searched for among {*d, fed, d, fd*}, and the candidates are in *FIList* = <*a, c, d, e, b, f*>. *d* is found to be the level 2 item, and it is stamped with the state "Target has been included". For pattern *b*, the level 2 items are searched for among {*a, ea, eca*}, and the candidate set is {*a, b, c, e*}. The level 2 item is found to be *a*, and it is stamped with the state "$\alpha[2:2]$ is the

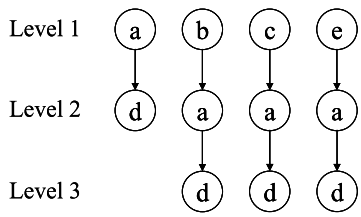


Fig. 1 Mining sequential patterns level by level.

Algorithm 1: FSPM – Fast Sequential Pattern Mining algorithm

```
input  : (1) SD: a sequence database
         (2) MinSup: a support threshold
output: All the sequential patterns with support not less than MinSup

FSPM ( SD, MinSup ) begin
1   scan SD once to calculate FIList;
2   remove from SD the items that are not in FIList;
3   for each item x (i.e., the target item) in FIList do
4     calculate FIRST(x) by Lemma 2 and the Find-FIRST of Lemma 5;
5     l ← 1, Level(l) ← FIRST(x);
6     for each length-1 pattern α in Level(1) do
        if it is the target item then
          set its state to "α[l : l] is the target";
        else
          set its state to "Target has not been included";
7     repeat
8       forall patterns α in Level(l) do
9         switch the state of α do
10          case Target has not been included
              find all the frequent items by the Pre-Target of Lemma 5;
              for each frequent item do
                append the item to α to form a sequential pattern α';
                if this item is the target item then
                  set its state to "α[l + 1 : l + 1] is the target";
                  output α';
                enter α' into Level(l+1);
11          case α[l : l] is the target
              find all the frequent items by the Is-Target of Lemma 5;
              for each frequent item do
                append the item to α to form a sequential pattern α';
                set its state to "Target has been included";
                output α';
                enter α' into Level(l+1);
12          case Target has been included
              find all the frequent items by the Post-Target of Lemma 5;
              for each frequent item do
                append the item to α to form a sequential pattern α';
                set its state to "Target has been included";
                output α';
                enter α' into Level(l+1);
13      l = l + 1;
      until Further growing is impossible ;
end
```

target". Similarly, for both pattern *c* and *e*, the level 2 item is *a*, and each is stamped with the state "$\alpha[2:2]$ is the target".

6. Go to level 2. For each level 2 item, FSPM does the same as stated in Step 5. Then the length-3 patterns are generated, as shown in Fig. 1.
7. Finally we have the results for target *a*: {*a, ad, ba, bad, ca, cad, ea, ead*}.
8. For the remaining target items *c*, *d*, *e*, *b*, and *f*, the above operations are repeated. The results are {*c, cd, cf*}, {*d, ed, fd, fed, bd*}, {*e, fe*}, {*b*}, {*f*}.

Note that PrefixSpan gives the results {*a, ad*}, {*b, ba, bd, bad*}, {*c, ca, cd, cf, cad*}, {*d*}, {*e, ea, ed, ead*}, and {*f, fd, fe, fed*}. We have the same pattern set. But they are output in different order.

3.3 Considerations for Implementation

One implementation consideration concerns how to determine the domain of a target item. Obviously it is inefficient to search the whole database whenever a new target item is to be processed. A solution we have used is the data structure shown in **Fig. 2**. All frequent items and their support counts are held in the Header, one record per frequent item. Starting from the *Link* field, all the instances of a frequent item are linked into a list. Note that this is not a special data structure like the FP-tree [8], but a very simple and classic one.

Another consideration concerns how to make the projected databases fit into the memory. Given a target item, starting from its *FIRST* item set, patterns grow level by level. Since one short pattern may grow into multiple long ones, for a large sequence database, the new level may become too wide to fit into the memory. A solution we have used is as follows. When the number of items on a level is larger than a predefined threshold, the items are divided into several groups, and patterns grow in the unit of the group. We found that the size of set *FIRST* is a suitable candidate for the threshold. Note that second-level problem partitioning gives us the flexibility to divide a level into groups.

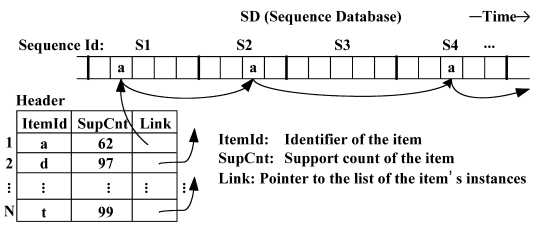


Fig. 2 Data structure used in our implementation.

The last consideration concerns how to make the sequence database fit into the memory. One technique is to process one target item at a time, which would take only the necessary sequences into the memory. In addition, the virtual memory provided by operating systems can be utilized. Here, a paging file on a hard disk is used as an extension of the memory, which can be as large as the hard disk can provide. But as our experimental results in the next section suggest, at that time the bottleneck would then be the CPU rather than the memory.

4. Performance Study

We have conducted experimental and analytical performance studies. In this section we discuss the results of the performance study.

4.1 Analyzing the Efficiency of FSPM

FSPM's efficiency can be examined based on its properties.

4.1.1 Level-by-level Pattern Growing

On the one hand, FSPM adopts the pattern-growing strategy as proposed in Ref. 14). Therefore, as proved in Ref. 14), FSPM has the potential to be more efficient than the Apriori-like algorithms. On the other hand, FSPM grows patterns in the unit of the level; a level generally contains a lot of prefixes. What is the merit of the level-by-level strategy? PrefixSpan grows patterns from one prefix at a time. As the average length of sequences or potential patterns in a sequence database increases, or as the support threshold decreases, PrefixSpan tends to recursively project databases with too high a frequency. As shown in the next section, this would seriously degrade PrefixSpan's performance. By contrast, FSPM tends to be less affected.

As stated in Section 1, technique *bilevel projection* [13] can be used to reduce the number of projections. Unfortunately, its authors found that this technique is ineffective [14]: "Based on our recent substantial performance study, the role of bilevel projection in performance improvement is only marginal in certain cases, but can barely offset its overhead in many other cases. Thus, this technique is dropped from the PrefixSpan options and also from the performance study." They finally proposed PrefixSpan with pseudo projection. Therefore we will also not consider the bilevel projection any further and will use PrefixSpan with the pseudo

projection in our performance study.

4.1.2 Two-level Problem Partitioning

FSPM's two-level problem partitioning strategy helps greatly to reduce the redundant scanning of sequence data. For sequential pattern mining, a sequence may be scanned several times, which would take too much time, and seriously degrade the efficiency of mining algorithms. A measure of the efficiency would be how many items on average need to be scanned to generate one pattern. With FSPM, a target item need not be considered after it is processed. This implies that FSPM generates as many patterns as possible in one scanning pass, with the result that FSPM is more efficient than PrefixSpan. Results given in Section 3.2 support this assertion. Taking the patterns with subsequence *ad* as an example, PrefixSpan scans a sequence database four times, while FSPM scans it only once.

4.1.3 Candidate-driven Pattern Mining

The candidate-driven property of FSPM makes its pattern-growing efficient. When extending a short pattern to its next level, the items that are qualified to be appended to that pattern have to be found by scanning its projected database. For FSPM, since the candidate set is known beforehand and its size is expected to be small, an efficient solution for searching for the items is possible, especially if the candidate set keeps shrinking. This is because (1) the seed set, which is the initial candidate set for a given target item, shrinks as we process items in *FIList* one by one, and (2) long patterns generally tend to be rare [14], implying that as patterns become longer, qualified items become rarer, and the candidate set becomes smaller.

The search spaces keep shrinking too, making the candidate-driven property more effective. As with PrefixSpan, as patterns grow, sequences in a projected database become shorter, and the search space shrinks. In addition, since the candidate set keeps shrinking, the items outside the candidate set are removed from the domain of a target item, and the search space shrinks.

4.1.4 Reasonable Memory Resource Usage

Due to its level-by-level pattern-growing property, compared with PrefixSpan's generations of long patterns from one short pattern each time, FSPM has the potential to use memory resources more reasonably. For PrefixSpan and other algorithms, minimizing the usage of memory resources is a major objective. We think that minimizing memory use just wastes the memory resources. In fact, we advocate full use of the memory resources a computer can provide. Since sequential pattern mining algorithms are all CPU-bound [3], we believe that reasonable use of memory resources is essential. As shown in the next section, this reasonable use contributes to high performance.

The above analysis indicates why FSPM's performance is excellent: it is not because of implementation tricks; it is inherent in the algorithm itself. Based on the above analysis we are convinced that FSPM has the potential to be an effective algorithm.

4.2 Experimental Comparison of FSPM and PrefixSpan

The main reason we conducted experiments that compared FSPM with PrefixSpan is its significant influenence as a sequential pattern-mining algorithm.

4.2.1 Experimental Environment

The synthetic sequence databases we used were generated by the IBM data set generator at the IBM Almaden Research center (http://www.almaden.ibm.com/software/quest/Resources/index.shtml). This generator has been used in most data mining studies.

A real sequence database downloaded from http://tds.cs.byu.edu/tds/, which is a disk I/O trace on Redhat Linux OS, collected over 15 consecutive days using a text editor, a compiler, a browser, email, and a desktop, was also used to examine performance in terms of the number of individual items. FSPM functioned as usual, but PrefixSpan did not function at all. For PrefixSpan, as explained below, this is because the number of individual blocks was too large, as suggested by a piece of a hard disk access sequence: "14402355 8460075 491549".

In the experiments, we evaluated performance in terms of (1) support threshold, (2) size of a sequence database, (3) average length of sequences, (4) average length of potential patterns, and (5) number of individual items. Accordingly, we used and varied experimental parameters as follows. The number of sequences in a sequence database, denoted by letter D, varied from 100 K to 500 K in steps of 100 K. The average length of sequences in a sequence database, denoted by letter S, varied from 5 to 25 items in steps of 5. The aver-

age length of potential patterns in a sequence database, denoted by letter I, varied from 2 to 10 items in steps of 2. The number of individual items in a sequence database, denoted by letter N, were 5 K, 10 K, 20 K, 30 K, 40 K, and 50 K. Combining these parameters we generated a lot of sequence databases, and got a lot of results. Some representative ones are discussed below.

We implemented FSPM using g++. For PrefixSpan, we used the pseudo-projection technique as suggested in Ref. 14), which makes PrefixSpan faster than SPAM [17]. The program for PrefixSpan was obtained directly from its authors' homepage (http://www-sal.cs.uiuc.edu/hanj/pubs/software.htm). Note that PrefixSpan has been demonstrated to have superior performance compared to earlier algorithms such as those found in Refs. 2), 15), 20).

All experiments were conducted on a 1.5 GHz Pentium PC with 1 GB memory, running Cygwin in Microsoft Windows XP. To be fair, for all experiments the results were obtained by excluding the effect of the cache in the operating system.

4.2.2 Experimental Results

The *execution time*, defined as the interval from the time the mining process is started until the time it finished, was used as a performance metric.

Experiment 1 (Varying support thresholds): **Figure 3** shows execution time as the support threshold increases from 0.05% to 0.25%. Two experimental environments are considered: a relatively light one resulting in the first figure and a relatively heavy one resulting in the second figure. For the light one, corresponding to each support threshold, <maximum length of patterns, total number of patterns>s are <1, 1154>, <3, 1725>, <6, 3354>, <10, 10660>, and <12, 50302>, respectively. For the heavy one, they are <2, 901>, <7, 2940>, <11, 7179>, <11, 33804>, and <14, 355292>, respectively. As shown in these two figures, FSPM outperforms PrefixSpan in both environments. Of particular note is that the smaller the support threshold was, the larger the performance difference was.

The maximum memory usages of PrefixSpan and FSPM are 6.7 MB and 234 MB, respectively, for the light environment, and 29 MB and 611 MB, respectively, for the heavy one. These results demonstrate two facts. (1) FSPM utilizes memory more reasonably than PrefixSpan. This is simply because we have a memory of

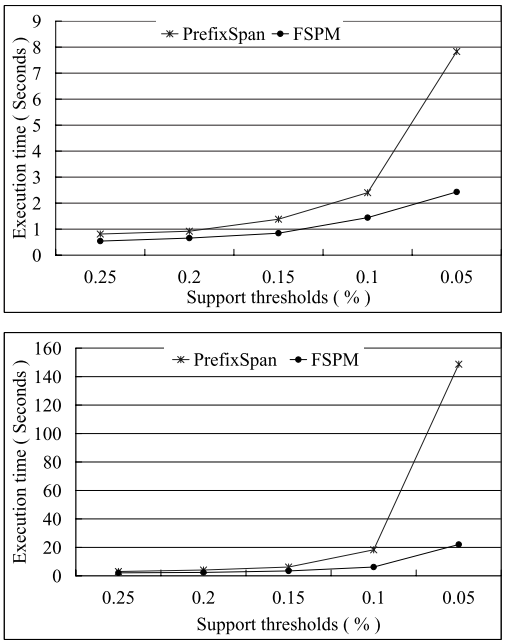


Fig. 3 Comparisons on support thresholds. Database: S10I4D100KN10000 (the 1st figure) and S15I6D300KN30000 (the 2nd figure).

1 GB, and we have used it with no waste. (2) FSPM is less efficient in utilizing memory than PrefixSpan. However, we don't think this will limit FSPM very much, because modern computers generally have a lot of memory resources. If memory resources really become a problem, solutions discussed in Section 3.3 can be used. This was done in Experiment 4, where a paging file was used. Note that because the explanation of experimental results for memory usage given here still holds for the remaining experiments, hereafter the results for memory usage will be given without any explanation.

Experiment 2 (Varying database sizes): **Figure 4** shows execution time as database size grows from 100 K to 500 K sequences. As above, the first figure corresponds to a light environment, and the second to a heavy one. As shown in the first figure, PrefixSpan slightly outperforms FSPM in scalability in this case, since as a database grows, FSPM's execution time increases slightly faster than PrefixSpan's. The main reason is as follows. In this case, the pattern numbers corresponding to each database size are 50302, 49668, 49440, 49427, and 49378, respectively, and remain almost unchanged. Thus, generating patterns would take approximately the same time for both algorithms. However, for FSPM there is a preprocessing

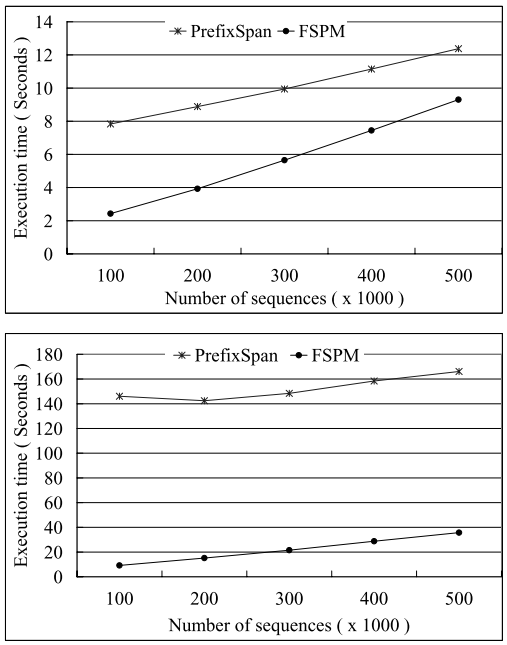


Fig. 4 Comparisons on number of sequences. Support: 0.05%. Database: S10I4D100-500KN10000 (the 1st figure) and S15I6D100-500KN30000 (the 2nd figure).

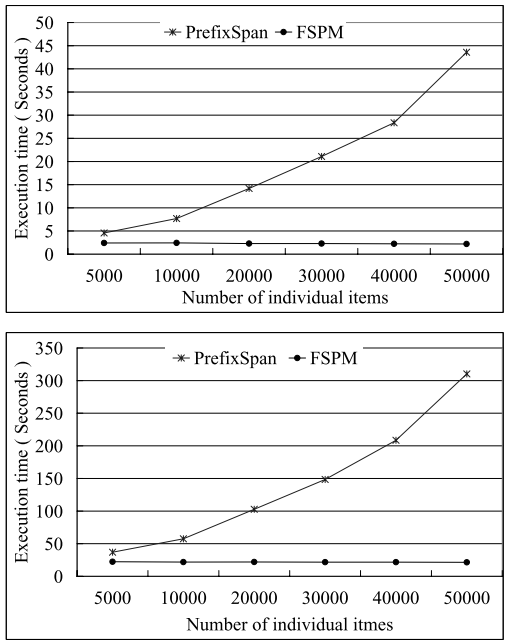


Fig. 5 Comparisons on number of individual items. Support: 0.05%. Database: S10I4D100KN5K, 10K-50K (the 1st figure) and S15I6D300KN5K, 10K-50K (the 2nd figure).

stage where the data structure is configured, as shown in Fig. 2. PrefixSpan counts items and creates projected databases, which is only a part of FSPM's preprocessing. Since FSPM's preprocessing cost is higher than PrefixSpan's, we have the above result. The performance difference, however, was not very large, since configuring the data structure would not take much time. For example, in the heavy environment, the pattern numbers corresponding to each database size are 373345, 353045, 355292, 364366, and 370481, respectively, which change more than in the previous case. Therefore, the preprocessing cost is negligible, and FSPM and PrefixSpan have similar scalability with respect to the database size.

The maximum memory usages of PrefixSpan and FSPM are 33.1 MB and 344 MB, respectively, for the light environment, and 47.2 MB and 675 MB, respectively, for the heavy one.

Experiment 3 (Varying the number of individual items): **Figure 5** shows execution time with the number of individual items ranging from 5 K to 50 K. As shown in these two figures, PrefixSpan's execution time increases as the number of individual items increases, while the increase has a negligible effect on FSPM's. In fact, given the same pattern distribution, FSPM is designed so that the number of individual items in a sequence database does not affect it. This is because a one-to-one mapping table, as shown in Fig. 2, which maps the set of the frequent items onto a so-called large item set $\{1, 2, \cdots,$ number of frequent items$\}$, was used, and only the large item set is used in the mining operation.

The maximum memory usages of PrefixSpan and FSPM are 5.5 MB and 361 MB, respectively, for the light environment, and 26.5 MB and 660 MB, respectively, for the heavy one.

Experiment 4 (Varying the average length of sequences): **Figure 6** shows execution time as the average length of sequences grows from 5 to 25 items. For the first figure, corresponding to each average length value, <maximum length of patterns, total number of patterns>s are <12, 19692>, <12, 50302>, <13, 95038>, <13, 125465>, and <13, 153366>, respectively. For the second, they are <11, 27947>, <13, 126301>, <14, 355292>, <14, 575315>, and <15, 797078>, respectively. As shown in these two figures, for both PrefixSpan and FSPM, the longer the average length of sequences, the longer the execution time. The effect on FSPM, however, is far smaller than that on PrefixSpan.

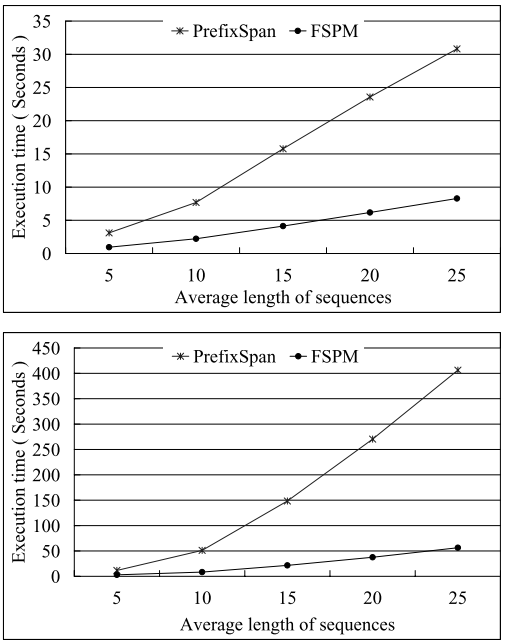


Fig. 6 Comparisons on average length of sequences. Support: 0.05%. Database: S5-25I4D100K N10000 (the 1st figure) and S5-25I6D300K N30000 (the 2nd figure).

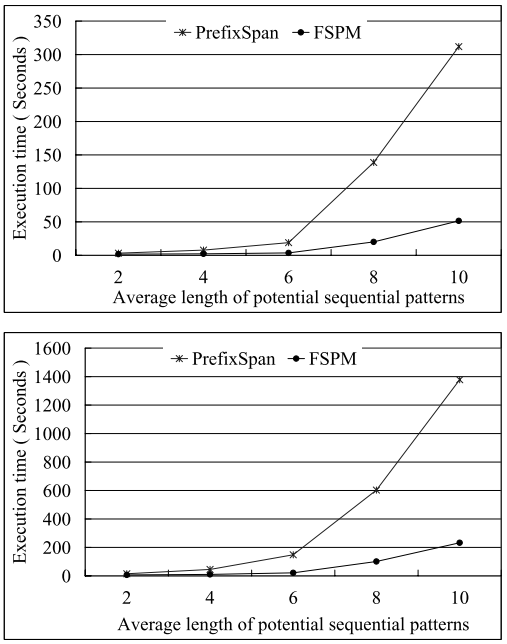


Fig. 7 Comparisons on average length of potential sequential patterns. Support: 0.05%. Database: S10I2-10D100KN10000 (the 1st figure) and S15I2-10D300KN30000 (the 2nd figure).

The maximum memory usages of PrefixSpan and FSPM are 18.3 MB and 356 MB, respectively, for the light environment, and 52.4 MB and 857 MB, respectively, for the heavy one.

Experiment 5 (Varying the average length of sequential patterns): **Figure 7** shows execution time as the average length of potential patterns grows from 2 to 10 items. For the first figure, corresponding to each pattern length value, <maximum length of patterns, total number of patterns>s are <8, 13628>, <12, 50302>, <12, 137296>, <18, 1043548>, and <19, 2368640>, respectively. For the second, they are <8, 20024>, <12, 88911>, <14, 355292>, <19, 1539450>, and <19, 3584710>, respectively. As shown in these two figures, the execution times of both PrefixSpan and FSPM increase as average length of sequential patterns increases. Again, however, FSPM is far less affected than PrefixSpan.

The maximum memory usages of PrefixSpan and FSPM are 8.7 MB and 296 MB, respectively, for the light environment, and 25.9 MB and 699 MB, respectively, for the heavy one.

Experiment 6 (Performance with small sequence databases): Based on Experiment 2, it seems that FSPM's performance is inferior to PrefixSpan's when the sequence database is small. To verify this result, the same experiments were done using small databases. The results are given in **Fig. 8**, **Fig. 9**, **Fig. 10**, and **Fig. 11**. First, as shown in Fig. 8, it was found that FSPM still has shorter execution times as the support threshold grows from 0.05% to 0.25%. Second, as shown in Fig. 9, in this case PrefixSpan really does have higher scalability with respect to the database size than FSPM. As discussed with regard to Experiment 2, this is mainly due to FSPM's preprocessing cost. Third, as shown in Fig. 10, and as expected, FSPM's execution times are shorter except when the number of individual items is

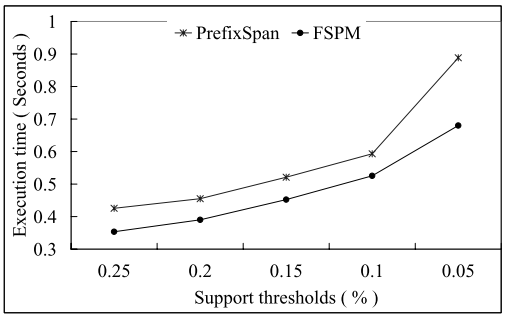


Fig. 8 Comparisons on support thresholds. Database: S5I2D100KN5000.

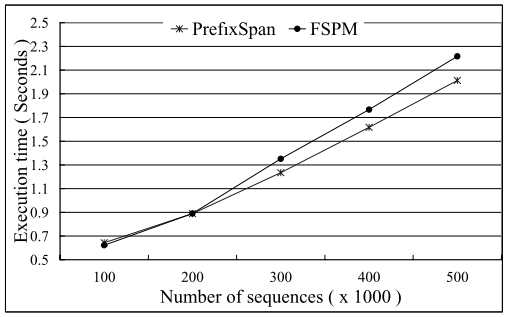


Fig. 9 Comparisons on number of sequences. Support: 0.1%. Database: S5I2D100-500KN5000.

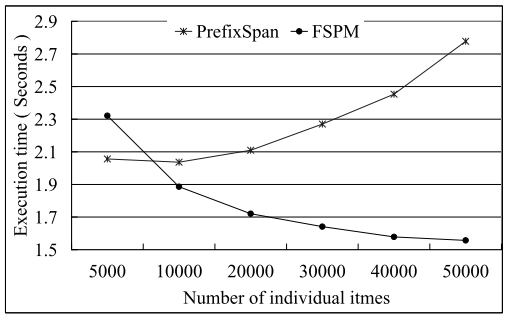


Fig. 10 Comparisons on number of individual items. Support: 0.1%. Database: S5I2D500KN5000, 10000-50000.

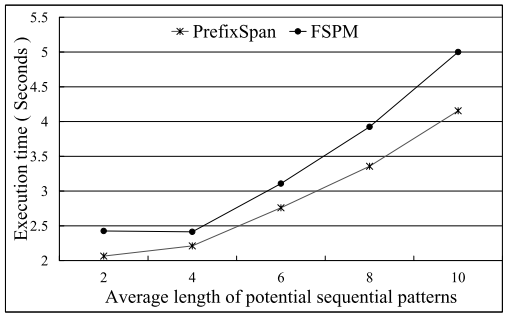


Fig. 11 Comparisons on average length of sequential patterns. Support: 0.1%. Database: S5I210D100KN5000.

5000. FSPM's execution time decreases as the number of individual items increases. This is because the pattern numbers corresponding to each database size are 2429, 2461, 2179, 1945, 1839, and 1733, respectively, which is a decrease. Finally, as shown in Fig. 11, PrefixSpan has shorter execution times with this small database.

The above results show that for small sequence databases, PrefixSpan does outperform FSPM in some cases.

For Fig. 8 – Fig. 11, the maximum memory usages of PrefixSpan are 3.1 MB, 13.1 MB, 13.1 MB, and 32.5 MB, and the maximum memory usages of FSPM are 107 MB, 153 MB, 153 MB, and 129 MB.

5. Conclusion

In this paper we addressed the subject of mining sequential patterns in a sequence database and proposed an effective algorithm called FSPM. FSPM is characterized by high performance in five essential dimensions of sequential pattern mining: the number of sequences; the number of individual items; the average length of sequences; the average length of potential sequential patterns; and the support threshold.

FSPM has some interesting and helpful properties: (1) It is a level-by-level pattern-growth algorithm. (2) It uses a two-level problem-partitioning strategy. (3) It conducts candidate-driven sequential pattern mining. (4) It can use memory reasonably. FSPM has also another very useful property: in addition to mining all patterns satisfying a support threshold, it can directly mine all the patterns related to an item specified by a user. Other algorithms can obtain no results until the pattern mining is finished. FSPM, on the other hand, can take an item as the target item, and go directly to finding the result. We have shown that it is the inherent potential embodied by the above properties that enables FSPM to perform better.

Extensive experiments have been conducted to compare FSPM with PrefixSpan. The results demonstrate that FSPM outperforms PrefixSpan in most cases. It was also found that FSPM requires more memory space than PrefixSpan. Therefore, for applications where memory is extremely limited, FSPM would not be a proper candidate. However, this is not a major disadvantage since the memory of modern computers has been very cheap and very large, and is becoming cheaper and larger.

In this paper we have considered sequence databases that consist only of items. Although many applications have these kind of databases, to be more widely used, FSPM must be extended so that it can deal with a sequence database in which a sequence consists of itemsets. This is for future work. Another interesting idea for future work is to enable FSPM to mine frequent closed subsequences that contain no subsequences with the same support.

Acknowledgments This work was supported in part by the Japanese Society for the

Promotion of Science (JSPS).

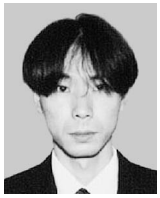
References

1) Agrawal, R. and Srikant, R.: Fast algorithms for mining association rules, *Proc. 20th International Conference on Very Large Data Bases*, Santiago, Chile, pp.487–499 (1994).
2) Agrawal, R. and Srikant, R.: Mining sequential patterns, *Proc. 11th International Conf. on Data Engineering*, Taipei, Taiwan, pp.3–14 (1995).
3) Ayres, J., Gehrke, J., Yiu, T. and Flannick, J.: Sequential Pattern Mining Using Bitmaps, *Proc. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.429–435, Edmonton, Alberta, Canada (2002).
4) Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J.: Models and Issues in Data Stream Systems, *Proc. 21th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Madison, Wisconsin, pp.1–16 (June 2002).
5) Chiu, D., Wu, Y. and Chen, A.L.P.: An Efficient Algorithm for mining Frequent Sequences by a New Strategy without Support Counting, *Proc. 20th International Conference on Data Engineering*, pp.375–386, Boston, MA, USA (2004).
6) Ezeife, C.I. and Lu, Yi: Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree, *International Journal of Data Mining and Knowledge Discovery* (*DMKD*), Vol.10, No.1, pp.5–38, Kluwer Academic Publishers (June 2005).
7) Garofalakis, M.N., Rastogi, R. and Shim, K.: Spirit: Sequential pattern mining with regular expression constraints, *Proc. 25th International Conference on Very Large Data Bases*, September 7-10, Edinburgh, Scotland, UK, pp.223–234 (1999).
8) Han, J., Pei, J. and Yin, Y.: Mining Frequent Patterns without Candidate Generation, *Proc. ACM SIGMOD Conf.*, pp.1–12 (2000).
9) Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U. and Hsu, M.-C.: FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining, *Proc. Int. Conf. on Knowledge Discovery and Data Mining* (*KDD'00*), Boston, MA, pp.355–359 (2000).
10) Li, Z., Chen, Z., Srinivasan, S. and Zhou, Y.: C-Miner: Mining Block Correlations in Storage Systems, *Proc. 3rd USENIX Conference on File and Storage Technology* (*FAST'04*), San Francisco, pp.173–186 (2004).
11) Masseglia, F., Cathala, F. and Poncelet, P.: PSP: Prefix Tree For Sequential Patterns, *Proc. 2nd European Symposium on Principles of Data Mining and Knowledge Discovery* (*PKDD'98*), Nantes, France, LNAI, Vol.1510, pp.176–184 (Sep. 1998).
12) Pei, J., Han, J., Mortazavi-Asl, B. and Zhu, H.: Mining Access Patterns Efficiently from Web Logs, *Proc. Pacific-Asia Conf. on Knowledge Discovery and Data Mining* (*PAKDD'00*), Kyoto, Japan (2000).
13) Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.-C.: Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth, *Proc. International Conf. on Data Engineering*, pp.215–224 (2001).
14) Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.-C.: Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach, *IEEE Transactions on Knowledge and Data Engineering*, Vol.16, No.11, pp.1424–1440 (2004).
15) Srikant, R. and Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements, *Proc. 5th Int. Conf. Extending Database Technology* (*EDBT'96*), Avignon, France, pp.3–17 (1996).
16) Wang, J. and Han, J.: BIDE: Efficient Mining of Frequent Closed Sequence, *Proc. 20th International Conference on Data Engineering* (*ICDE*), pp.79–90 (2004).
17) Wang, K., Xu, Y. and Yu, J.: Scalable Sequential Pattern Mining for Biological Sequences, *Proc.13th ACM international conference on Information and knowledge management* (*CIKM 2004*), pp.178–187, Washington, D.C., USA (2004).
18) Yan, X., Han, J. and Afshar, R.: CloSpan: Mining Closed Sequential Patterns in Large Datasets, *Proc. 3rd SIAM International Conference on Data Mining* (*SDM'03*), pp.166–177, San Francisco, USA (2003).
19) Yang, Z. and Kitsuregawa, M.: LAPIN-SPAM: An Improved Algorithm for Mining Sequential Pattern, *Proc. Int'l Special Workshop on Databases For Next Generation Researchers* (*SWOD'05*) *in conjunction with ICDE'05*, pp.8–11, Tokyo, Japan (2005).
20) Zaki, M.J.: SPADE: An efficient algorithm for mining frequent sequences, *Machine Learning*, Vol.42, No.1/2, pp.31–60 (2000).

(Received March 7, 2006)
(Accepted September 14, 2006)
(Online version of this article can be found in the IPSJ Digital Courier, Vol.2, pp.768–782.)

Jiahong Wang received his B.E. degree from Northeast University, China, and his M.E. degree from the Shenyang Institute of Computing Technology of the Chinese Academy of Sciences, both in computer science. He received the Dr. Eng. degree from University of Tsukuba, Japan. He is currently an associate professor in the Faculty of Software and Information Science of Iwate Prefectural University, Japan. His research interests include concurrency control, transaction processing, distributed/parallel processing, data mining, and modeling and performance evaluation. Dr. Wang is a member of IEEE and IEICE.

Yoshiaki Asanuma received his B.E. degree in computer science from Iwate Prefectural University, Japan, in 2004. Presently he is with the Product Development Division of Internet Initiative Japan Inc.



Eiichiro Kodama received his master's degree in mathematical science from the University of Tokyo in 1994. From 1994 to 1996, he was with the Hitachi Information Technology Co., Ltd. (Hitachi IT). From 1998 to 2003, he was a Research Associate at Iwate Prefectural University. In 2003, he received his Ph.D. in information science from Tohoku University. He is Currently an Assistant Professor at Iwate Prefectural University. His main research interests are semantic web, natural language processing and algebraic number theory in mathematics. He is a member of the Information Processing Society of Japan, the Institute of Electronics, Information and Communication Engineers of Japan, the Japanese Society for Artificial Intelligence and the Association for Natural Language Processing.

Toyoo Takata received his Ph.D. in information and computer sciences from Osaka University in 1989. From 1989 to 1993, he was a Research Associate at the Department of Information and Computer Sciences, Osaka University. From 1993 to 1998, he was an Associate Professor at the Graduate School of Information Science, Nara Institute of Science and Technology. Since 1998, he has been a Professor of Software Information Science at Iwate Prefectural University. Dr. Takata is a member of IEEE, ACM, IEICE, IPSJ and SITA.

Jie Li received his B.E. degree in computer science from Zhejiang University, China, in 1982, his M.E. degree in electronic engineering and communication systems from the China Academy of Posts and Telecommunications, in 1985, and his Dr. Eng. degree from the University of Electro-Communications, Japan, in 1993. From 1985 to 1989, he was a research engineer at the China Academy of Posts and Telecommunications. From 1993, he has been with the Department of Computer Science, Graduate School of Systems and Information Engineering of the University of Tsukuba, Japan, where he has been an Associate Professor since 1997. His research interests are in distributed multimedia computing and networking, OS, transaction processing, network security, reliability, modeling and performance evaluation of information systems. He received the best paper award from IEEE NAECON'97. He is a senior member of IEEE, and a member of ACM. He has served as a secretary for Study Group on System Evaluation of IPSJ and on the editorial boards for IPSJ Journal, IEEE Trans. on Vehicular Technology, and International Journal of High Performance Computing and Networking. He is also serving on steering committees of the SIG of System EVAluation (EVA) of IPSJ, the SIG of DataBase System (DBS) of IPSJ, and the SIG of MoBiLe computing and ubiquitous communications of IPSJ. He has also served on the program committees for several conferences such as IEEE INFOCOM, IEEE GLOBECOM, IEEE MASS, IEEE ICDCS.