Effective Database Transformation and Efficient Support Computation for Mining Sequential Patterns

Chung-Wen Cho[1], Yi-Hung Wu[1], and Arbee L.P. Chen[2]

[1] Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
[2] Department of Computer Science, National Chengchi University, Taipei, Taiwan
alpchen@cs.nccu.edu.tw

Abstract. In this paper, we introduce a novel algorithm for mining sequential patterns from transaction databases. Since the FP-tree based approach is efficient in mining frequent itemsets, we adapt it to find frequent 1-sequences. For efficient frequent k-sequence mining, every frequent 1-sequence is encoded as a unique symbol and the database is transformed into one in the symbolic form. We observe that it is unnecessary to encode all the frequent 1-seqences, and make full use of the discovered frequent 1-sequences to transform the database into one with a smallest size. To discover the frequent k-sequences, we design a tree structure to store the candidates. Each customer sequence is then scanned to decide whether the candidates are frequent k-sequences. We propose a technique to avoid redundantly enumerating the identical k-subsequences from a customer sequence to speed up the process. Moreover, the tree structure is designed in a way such that the supports of the candidates can be incremented for a customer sequence by a single sequential traversal of the tree. The experiment results show that our approach outperforms the previous works in various aspects including the scalability and the execution time.

Keywords: Data mining, Sequential patterns, Database transformation, Frequent k-sequences.

1 Introduction

Sequential pattern mining [2][3][4][5][6][8], which discovers interesting patterns from transaction databases, is an essential problem in the data mining field. This problem was first introduced in [2]. A transaction database has three fields, i.e. customer id, transaction-time, and the items purchased. An itemset is a non-empty set of items and a sequence is an ordered list of itemsets. In this way, each transaction corresponds to an itemset. Each customer with a unique customer id may have more than one transaction with different transaction-times. All the transactions from a customer are ordered by increasing transaction-times to form a sequence, called the *customer sequence*.

The *size* of an itemset is the number of items in it. A *k-itemset* is an itemset with size k. The *length* of a sequence is the number of itemsets in it. A *k-sequence* is a

sequence with length k. Moreover, the *size* of a sequence is also defined as the number of items in it. Given sequences $X=<X_1X_2...X_n>$ and $Y=<Y_1Y_2...Y_m>$ where all X_i's and Y_j's are itemsets and $n \le m$, X is a *subsequence* of Y, i.e., *contained* in Y, if there exist n integers $1 \le i_1 < i_2 < \ldots < i_n \le m$ such that $X_1 \subseteq Y_{i1}$, …, $X_n \subseteq Y_{in}$. The *support* of a sequence is the number of customer sequences containing it in the entire database. Given a minimum support threshold *minsup*, a sequence is *frequent* if its support is not lower than minsup. We call the database composed of customer sequences the *sequence database*. The problem of sequential pattern mining is to efficiently find all the frequent sequences from a sequence database.

In the remainder of this paper, for brevity, we will use frequent itemsets and frequent sequences to mean frequent 1-sequences and frequent k-sequences for k>1, respectively, unless explicitly specified otherwise. To our knowledge, AprioriAll is the first algorithm [2] that mines frequent sequences in two phases. That is, in the first phase, only frequent itemsets are found and then frequent sequences are mined in the second phase. In both phases, patterns are mined in the bottom-up fashion, i.e., frequent k-itemsets/sequences first, then frequent (k+1)-itemsets/sequences, and so on. The stage of mining frequent k-itemsets/sequences is called pass k. Between two consecutive passes, the anti-monotonic property, that all subsets/subsequences of a frequent itemset/sequence must be frequent, is utilized for *candidate* pruning, where a candidate is a sequence whose support has not been computed yet. AprioriAll applies the Apriori algorithm [1], which is proposed to mine association rules, to Phase 1 for mining frequent itemsets from the sequence database. At the beginning of Phase 2, AprioriAll maps every frequent itemset to a unique symbol and then transforms each transaction into a set composed of all the symbols whose patterns are contained in the transaction. For example, given the sequence database in Table 1 and the minsup 2, the frequent itemsets and the corresponding symbols are listed in Table 2. Note that in our notation an itemset is enclosed with parentheses while a sequence is enclosed with angle brackets. Based on Table 2, Table 1 is transformed into Table 3. In this way, all the frequent itemsets are retained and the subsequences of customer sequence containing a non-frequent itemset will not be considered as candidates for the frequent sequences.

Table 1. A sequence database

CID	Cus. Seq.
C1	<(abc)(bc)>
C2	<(b)(abc)(ad)>
C3	<(bd)(bc)>

Table 2. Mappings

F.I.	S.	F.I.	S.
a	A	ab	E
b	B	ac	F
c	C	bc	G
d	D	abc	H

Table 3. The transformed database

CID	Customer Sequence
C1	<(ABCEFGH)(BCG)>
C2	<(B)(ABCEFGH)(AD)>
C3	<(BD)(BCG)>

In Phase 2, we call a sequence *singular* if each set in it has only one symbol. Consider a set having two symbols X and Y, where x and y are the corresponding frequent itemsets, respectively. On one hand, if x is a subset of y, a sequence containing (XY) can be replaced with the corresponding one containing (Y). On the other hand, let z be the union of x and y. If z is frequent, again a sequence containing (XY) can be re-

placed with the corresponding one containing (Z), where Z is the symbol corresponding to z. Therefore, after the transformation, only the singular sequences should be considered as the candidates for the frequent sequences in Phase 2.

During the support computation, AprioriAll enumerates the subsequences of each customer sequence and accumulates the support of each candidate in a *candidate tree*, where each path from the root to a leaf in it corresponds to a candidate. Two or more subsequences enumerated from a customer sequence might be identical and therefore the corresponding path in the candidate tree will be traversed and counted more than once. However, by the problem definition, no matter how many times a subsequence appears in the customer sequence, its support can be increased at most by one. Therefore, it is unnecessary to repeatedly enumerate identical subsequences from a customer sequence. In the applications, this problem is usual, e.g., a customer often buys the same items more than once. Owing to the large number of enumerated subsequences, the counting on the candidate tree can be time-consuming.

In addition to AprioriAll, SPAM [4], and Pseudo-Projection [6] find frequent sequences based on the *lattice* concept as follows. Given a set of items, a lattice is a layered graph where each node stands for a distinct sequence and each link indicates the parent-child relationship between one sequence with size k and another with size k+1. To find all the frequent k-sequences, all the two approaches recursively select a set of nodes as candidates and scan the database to compute their supports. The number of database scans in each of them is directly proportional to the total number of frequent k-sequences.

To sum up, AprioriAll retains all the frequent itemsets in the transformed database in Phase 2 to avoid enumerating the subsequences containing non-frequent itemsets. Given n distinct frequent items, if the maximal length of a frequent sequence is k, there can be in the worse case $(2^n-1)+(2^n-1)^2+\ldots+(2^n-1)^k$ frequent sequences to be discovered. Since the 2^n-1 frequent itemsets are in the minority, mining frequent sequences is relatively important in sequential pattern mining. The two-phase architecture that finds two kinds of patterns separately should enable the mining tasks for different types of patterns improved. However, previous works [4][5][6][8] have noted that the bottom-up approach suffers from the huge amount of candidates generated at one pass. Recently, as the price-to-capacity ratio of main memory shrinks, the pain of too many candidates in AprioriAll has been alleviated. However, AprioriAll still suffers from the transformed database that can be much larger than the original one and the high execution time for support computation. Accordingly, in this paper, we develop a new two-phase approach for mining sequential patterns, equipped with the facilities for effective database transformation and efficient support computation.

Finding frequent itemsets in Phase 1 is relevant to the problem of association rule mining, which have been extensively discussed, such as [1][7]. Among them, the FP-tree based approach [7] takes advantage of the common items among itemsets to reduce the cost of subset enumeration. This inspires us to adapt the same idea to the sequence database. A straightforward way that adopts the FP-tree to find frequent itemsets from a sequence database does not work because the support of an itemset may be overestimated. In this paper, we propose the concept of *transaction intersection* to compute the correct support of each itemset.

In Phase 2, we aim at effective database transformation and efficient support computation. First, from the anti-monotonic property, it is not necessary to retain all the

frequent itemsets in the transformed database. Take Table 3 as an example. Since combining A with any other symbol cannot form a frequent sequence, it guarantees that E and F (two supersets of A) cannot form any frequent sequence, either. Our approach skips this kind of symbols during the database transformation to have two advantages: no need to examine the sequences containing a symbol that cannot form any frequent sequence and the transformed database with a smaller size.
To speed up the support computation, we propose a technique to avoid redundantly enumerating the identical subsequences from a customer sequence. Moreover, the nodes in the candidate tree are sorted such that the supports of the candidates can be incremented for a customer sequence by a single traversal of the tree. In the experiments, our approach outperforms the previous works in both the execution time and database scalability. Moreover, the results also show that the main advantages of our approach lie in minimizing the number of symbols encoded and computing the supports of candidates in Phase 2 efficiently.

The rest of this paper is organized as follows. In Section 2, we present the method for mining frequent itemsets from the sequence database. In Section 3, we describe the method for mining frequent sequences. The performance evaluation and experiment results are shown in Section 4. Finally, we conclude this paper in Section 5.

2 Frequent Itemset Discovery

In Phase 1, we adapt the FP-tree based approach to finding frequent itemsets from the sequence database. The FP-tree based approach first constructs a compact data structure called *FP-tree* to keep all the transactions in the transaction database and then performs a mining algorithm on it to find all the frequent itemsets. In this paper, we adopt the mining algorithm named TD-FP-Growth. For lack of space, readers please refer to [7] for the detail.

The FP-tree based approach cannot be directly applied to finding frequent itemsets in the sequence database because the transactions in a customer sequence may contain identical itemset. In that case, the support of the itemset will be overestimated. For example, given a customer sequence <(abc)(bcd)>, as the two transactions are inserted into the FP-tree, this customer sequence will contribute 2 to the support of (bc). An intuitive response to this problem is to combine (abc) and (bcd) into a single transaction (abcd), which correctly provides the support of (bc). However, in this way the support of (ad) will be miscounted. In our approach, we introduce the concept of *transaction intersection* that can correctly compute the support of each itemset as we insert all the transactions in each customer sequence into the FP-tree.

Consider an itemset I and a customer sequence $S=\langle T_1T_2\ldots T_n\rangle$, where each T_i is a transaction. Our goal is to insert all T_i's into the FP-tree such that the correct support of I from S, i.e., 0 or 1, can be obtained from the FP-tree. This process is similar to the enumeration of I from S. If the intersection O_{ij} between T_i and T_j is I, I will be enumerated twice and its support from the FP-tree is overestimated as 2. Therefore, for every two transactions T_i and T_j in S, we consider their intersection O_{ij} as an extra transaction, called a *negative transaction intersection*, which is also inserted into the FP-tree with a negative support -1. Moreover, if the intersection O_{ijk} between T_i, T_j

and T_k is I, I will be enumerated from S thrice but decreased by the three negative instance intersections O_{ij}, O_{ik}, and O_{jk}. In that case, the support of I from the FP-tree is underestimated as 0. Therefore, for every three transactions T_i, T_j and T_k in S, we further consider their intersection O_{ijk} as a *positive transaction intersection*, which is also inserted into the FP-tree to increase the support of I by one. In this way, the two kinds of transaction intersections (abbreviated as TI) are alternatively derived from the intersections among transactions in S. The transaction intersection that is the intersection between L transactions in S is called the *L-TI*. Without loss of generality, each transaction in S is treated as a positive 1-TI.

For example, given a customer sequence S = <(abc)(bcd)(abd)(ac)>, for every two transactions, we have the 2-TIs {(bc), (ab), (ac), (bd), (c), (a)}. Moreover, for every three transactions, we have the 3-TIs {(b), (c), (a), ∅}. Finally, the 4-TI is ∅. From the 4 transactions in S, i.e., 1-TIs, the support of (a) is overestimated as 3. Since (a) appears in three 2-TIs, its support will be decreased to 0. Finally, (a) appears in only one 3-TI and therefore its support will be correctly computed as 1.

Our approach employs all the transaction intersections generated from each customer sequence to construct the FP-tree, in which the negative transaction intersections can decrease the supports of the corresponding itemsets contained in them. After the FP-tree is constructed, the count of a node in the FP-tree can be positive, zero, or negative. Finally, the TD-FP-Growth algorithm is directly applied to this FP-tree to find all the frequent itemsets.

3 Frequent Sequence Discovery

All the notations used in this section and their definitions are shown in Table 4.

The two main components of Phase 2 in our approach are *database transformation* and *frequent sequence discovery*, respectively. In the database transformation, unlike AprioriAll that encodes all the frequent itemsets at once, our approach divides the frequent itemsets into groups based on their size and encodes them at different rounds in pass 2. The groups of frequent itemsets having smaller sizes are encoded earlier, i.e., frequent 1-itemsets, frequent 2-itemsets, and so on.

In our approach, frequent k-sequences are discovered at pass k, where k≥2. In pass 2, at the first round, all the frequent 1-itemsets are encoded as symbols and the frequent 2-sequences that contain at least one of them are discovered. These frequent 2-sequences are immediately used to check the frequent 2-itemsets to see whether

Table 4. Notations and their definitions

Notations	Definitions
FE_r	The set of the frequent r-itemsets encoded in round r
$FE_{1..r}$	The set of all the frequent itemsets encoded in rounds 1…r
FS_r	The set of the symbols corresponding to FE_r
$FS_{1..r}$	The set of all the symbols generated in rounds 1…r
TD_r	The database after the transformation in round r
F_2^r	The set of the frequent 2-sequences found in round r

they should be encoded in the next round. This process repeats until there is no frequent itemset to be encoded. In pass k (k≥3), candidate k-sequences are generated according to the frequent (k-1)-sequences found in pass k-1, and then the transformed database is scanned to compute their supports. Note that, although main memory space is getting larger nowadays, it is still possible to generate too many candidates in one pass. Therefore, in each pass, our approach monitors the number of candidate sequences currently generated. If this number reaches a predefined threshold, the supports of these candidate sequences are computed via one database scan. In this way, during the support computation, all the candidate sequences involved can fit in the main memory. In the following, our procedure of database transformation is introduced first. After that, the main stages, candidate generation and support computation, for frequent sequence discovery are presented, respectively.

3.1 Database Transformation

In the first round of pass 2, each frequent 1-itemset is put into FE_1 and encoded as a distinct symbol in FS_1. Moreover, each transaction in a customer sequence that contains the frequent 1-itemsets in FE_1 is replaced with a set of the corresponding symbols in FS_1 to produce TD_1. After that, all the frequent 2-sequences in TD_1, i.e., F_2^1, are discovered. Finally, we determine whether a frequent 2-itemset should be encoded for the second round according to the following property.

Property 1. Given a frequent (r+1)-itemset X, let the r-itemsets contained in X be denoted as X_1, X_2 … and X_{r+1}. For any frequent itemset Y, the following two rules always hold:

1. If <XY> is frequent in round r+1 $\Rightarrow$ $<X_iY> \in F_2^r$ $\forall 1 \le i \le r+1$.
2. If <YX> is frequent in round r+1 $\Rightarrow$ $<YX_i> \in F_2^r$ $\forall 1 \le i \le r+1$.

According to the anti-monotone property, the above property can be easily observed. As a result, only the frequent 2-itemsets should be encoded in round 2, if there exists a frequent itemset Y in $FE_{1..2}$ such that one of the rules in Property 1 holds. In round r (r≥2), the database TD_{r-1} is transformed into TD_r by adding the symbols in FS_r to the transactions that contain the corresponding frequent itemsets in FE_r. After that, to avoid the frequent 2-sequences that have been generated in previous rounds, only the 2-sequences that contain at least one symbol in FS_r are considered as candidates and enumerated from TD_r. Finally, the frequent (r+1)-itemsets satisfying Property 1 are encoded for round r+1. When no more frequent itemset is encoded, the database transformation is terminated in pass 2.

For example, consider the sequence database in Table 1. and the symbols in Table 2. In the first round, all the frequent 1-itemsets are encoded (A, B, and C) and TD_1 is shown in Table 5. The frequent 2-sequences composed of these symbols are <BB> and <BC>. In the second round, only (bc) has to be encoded because (ab) and (ac) do not satisfy the rules in Property 1. Symbol F is appended to each transaction containing both B and C, and the result is shown in Table 6. In this round, only <BF> is a frequent 2-sequence. Finally, pass 2 is terminated because no frequent 3-itemset should be encoded.

Table 5. TD_1

CID	Customer Sequence
1	<(ABC)(BC)>
2	<(B)(ABC)(A)>
3	<(B)(BC)>

Table 6. TD_2

CID	Customer Sequence
1	<(ABC)(BCF)>
2	<(B)(ABCF)(A)>
3	<(B)(BCF)>

Distributing the database transformation task into rounds has two advantages. First, the transformed database generated in each round is much smaller than the one generated by AprioriAll. Second, in AprioriAll, every frequent itemset is encoded at the beginning of Phase 2 and therefore the number of candidates can be huge. By contrast, in our approach, the sequences containing the frequent itemset that is not encoded are not generated as candidates. Moreover, our approach will not enumerate such kind of sequences from the customer sequences for support computation.

3.2 Candidate Generation

We describe the procedure of candidate generation in two cases, i.e., pass = 2 and pass = k for $k \geq 3$. In round r of pass 2, for each symbol α in FS_r, α is combined with each β in $FS_{1...r}$ to generate the candidate sequences $<\alpha\beta>$ and $<\beta\alpha>$. Since only the 2-sequences that contain at least one symbol encoded in round r are considered as candidates, our algorithm will not generate redundant candidates in different rounds.

We adopt the same method as described in [2] to generate candidates in pass k for $k \geq 3$. Each frequent (k-1)-sequence $<\alpha_1 \ldots \alpha_{k-2}\alpha_{k-1}>$ is combined with each frequent (k-1)-sequence $<\beta_1 \ldots \beta_{k-2}\beta_{k-1}>$, where $\alpha_1=\beta_1$, …, and $\alpha_{k-2}=\beta_{k-2}$, to generate the candidate sequence $<\alpha_1 \ldots \alpha_{k-1}\beta_{k-1}>$. Based on the anti-monotonic property, we further check whether any subsequence with length k-1 of $<\alpha_1 \ldots \alpha_{k-1}\beta_{k-1}>$ is non-frequent. If there exists such a subsequences, $<\alpha_1 \ldots \alpha_{k-1}\beta_{k-1}>$ cannot be frequent and therefore will be pruned.

In each pass, the generated candidates are stored into the candidate tree, where the support of each candidate is kept at the leaf node of the corresponding path. For each non-leaf node, all its children are stored in lexicographic order as an ordered list. This ordering of nodes in the candidate tree is used to reduce the cost on support computation, which will be detailed in the next section.

3.3 Support Computation

In pass k, for each customer sequence, all the subsequences with length k are enumerated by combining the symbols in different transactions. Given a subsequence $\alpha=<\alpha_1\alpha_2 \ldots \alpha_k>$ enumerated from a customer sequence, where α_i is a symbol, we say α *matches* a path $p=n_1n_2...n_k$ in the candidate tree if $n_i=\alpha_i$ for $1 \leq i \leq k$. For each subsequence enumerated, the candidate tree is traversed to find its match. As described in Section 1, repeated enumeration of identical subsequences from a customer sequence is unnecessary and time-consuming. Therefore, in this section, we introduce a novel method to avoid it such that only a single traversal of the candidate tree is required for counting all the subsequences enumerated from a customer sequence.

The rationale of our approach is as follows. First, since a subsequence may have multiple occurrences in a customer sequence, we need a particular representation of the customer sequence such that each distinct subsequence is enumerated exactly once from that representation. Second, the symbols in that representation should be ordered such that the subsequences enumerated from it are in the same order as the sequential traversal on the candidate tree. In the following, we first define the proposed representation and then describe the subsequence enumeration algorithm on it.

Consider a customer sequence $S=<T_1T_2...T_n>$, where T_i is the i^{th} transaction. If a symbol α appears in T_i, this occurrence of α is denoted as α^i. For example, <(BDE)(A)(B)(BC)> can be denoted as $<B^1D^1E^1A^2B^2B^3B^4C^4>$. Let *i-suffix*$_S$ denote the subsequence $<T_iT_{i+1}...T_n>$ of S. If α appears in $T_{j_1}, T_{j_2} ... T_{j_m}$, where $i \le j_1 < j_2 ... j_m \le n$, we call α^{j_k} the *k^{th} instance* of α in i-suffix$_S$.

Definition 1. Given k occurrences of symbols $\beta_1^{i_1}$, $\beta_2^{i_2}$ … $\beta_k^{i_k}$ in S, where $1 \le i_1 \le i_2 \ldots \le i_k \le n$, $<\beta_1^{i_1}\beta_2^{i_2}\ldots\beta_k^{i_k}>$ is called a *necessary subsequence* of S if $\beta_1^{i_1}$ is the 1st instance of β_1 in the 1-suffix$_S$ and for $2 \le j \le k$, $\beta_j^{i_j}$ is the 1st instance of β_j in the $(i_{j-1}+1)$-suffix$_S$.

For example, in $<B^1D^1E^1A^2B^2B^3B^4C^4>$, $<B^1A^2C^4>$ is a necessary subsequence because B^1 is the 1st instance of B in 1-suffix$_S$, A^2 is the 1st instance of A in 2-suffix$_S$, and C^4 is the 1st instance of C in 3-suffix$_S$. On the contrary, $<B^1A^2B^4>$ is not a necessary subsequence because B^4 is not the 1st instance of B in 3-suffix$_S$.Obviously, each subsequence enumerated from S must have exact one occurrence in S, which is a necessary subsequence. As a result, only the necessary subsequences instead of all the subsequences in a customer sequence should be enumerated.

Definition 2. For each i-suffix$_S$ S', if we remove each occurrence of a symbol that is not the 1st instance of that symbol in S', the resultant subsequence is called a *pivot* and denoted as *i-pivot*.

For example, the 4 pivots of $<B^1D^1E^1A^2B^2B^3B^4C^4>$ are 1-pivot: $< B^1D^1E^1A^2C^4>$, 2-pivot: $<A^2B^2C^4>$, 3-pivot: $<B^3C^4>$, and 4-pivot: $<B^4C^4>$, respectively. For ease of presentation, we denote the j^{th} element of the i-pivot as *i-pivot[j]* and the total number of elements in the i-pivot as |*i-pivot*|. The following lemma shows the relationships between the pivots and a necessary subsequence.

Lemma 1. If $<\beta_1^{i_1}\beta_2^{i_2}\ldots\beta_k^{i_k}>$ is a necessary subsequence of S, it must be in the form of <1-pivot[j_1], (i_1+1)-pivot[j_2], …(i_{k-1}+1)-pivot[j_k]>, where $j_1 \le$|1-pivot| and $j_h \le$|(i_{h-1})-pivot| for $2 \le h \le k$.

From Lemma 1, each necessary subsequence of a customer sequence S can be formed by the elements in the pivots of S. Therefore, our approach first derives all the pivots from S and then generates all the necessary subsequences from the pivots. Since the pivots derived from S can be much larger than S, we derive and keep the pivots of S only when S is scanned for support counting. For each customer sequence with size n, we *sort* its elements in the form of $<\beta_1^{i_1}\ \beta_2^{i_2}\ \ldots\beta_n^{i_n}>$ such that for $1 \le j < n$, $\beta_j < \beta_{j+1}$ or ($\beta_j = \beta_{j+1}$ and $i_j < i_{j+1}$). For instance, the sorted form of $<B^1D^1E^1A^2B^2B^3B^4C^4>$ is $<A^2B^1B^2B^3B^4C^4D^1E^1>$. Given the sorted form of a customer sequence $<\beta_1^{i_1}\ \beta_2^{i_2}\ \ldots\beta_n^{i_n}>$, our algorithm named *Pivot Derivation* with two nested for-loops is proposed as follows.

Initially, all the pivots are set empty. Each entry of an array Index[k] is used to keep the number of elements in the k-pivot. For each symbol β_j^{ij} (in the i_j^{th} transaction), we check whether β_j has been stored in i_j-pivot, (i_j-1)-pivot, ... or 1-pivot. If the k-pivot does not have β_j, β_j^{ij} is immediately stored into it. The idea of this algorithm comes from two observations. First, the input is the sorted form of S. If k-pivot[Index[k]] does not equal β_j, this implies that β_j^{ij} must be the 1st instance in k-$suffix_S$ and also an element in the k-pivot according to Definition 2. On the contrary, if the k-pivot has β_j, there must exist an occurrence of β_j that is the 1st instances of β_j in h-$suffix_S$ for h=1…k. In this case, β_j^{ij} cannot be an element in the h-pivot for h=1…k and therefore will be skipped.

For example, consider the sorted form of $<A^2B^1B^2B^3B^4C^4D^1E^1>$ as the input. First, after A^2 is processed, both the 2-pivot and 1-pivot are $<A^2>$. Second, after B^1, B^2, B^3, and B^4 are processed, we have 1-pivot=$<A^2B^1>$, 2-pivot=$<A^2B^2>$, 3-pivot=$<B^3>$, and 4-pivot=$<B^4>$, respectively. Third, after C^4 is processed, all the 4 pivots are appended with C^4. Finally, after D^1 and E^1 are processed, only the 1-pivot is changed to $<A^2B^1C^4D^1E^1>$.

In our approach, sorting each customer sequence is done during database transformation. During the support counting, the above algorithm is invoked to derive all the pivots from a customer sequence. In the following, we will present our algorithm that can enumerate all the necessary subsequences from the pivots. Given the pivots derived from the sorted form of a customer sequence $S=<T_1T_2...T_n>$, the candidate tree, and pass k, our algorithm named *Necessary Subsequence Enumeration* (abbreviated as NSE) is shown as below. Note that the matching and counting for the candidates on T are also included in it. Given a node *x* on the candidate tree, its j^{th} child and the total number of its children are denoted as *x.children[j]* and |*x.children*|, respectively. In addition, the serial number of the transaction having r-pivot[i] is denoted as *r-pivot[i].no*.

```
Algorithm NSE(pivots, x, L, r)

(1) If (|x.children| ≠ 0) do {
(2)   i = j = 1;
(3)   While (i≤|r-pivot|) and (j≤|x.children|) do {
(4)     If (r-pivot[i] == x.children[j]) {
(5)       If (L == k)
(6)         x.children[j].count++;
(7)       Else If (r-pivot[i].no<n) and (L<k)
(8)          NSE(pivots, x.children[j], L+1,
                 r-pivot[i].no+1);
(9)       i ++; j ++; }
(10)    Else If (r-pivot[i] < x.children[j]) {
(11)      i++; }
(12)    Else    j++; }
```

Initially, NSE(pivots, the root of the candidate tree, 1, 1) is invoked. The parameter L is used to keep the path length currently traversed, indicating the length of candidate sequences to be counted. The parameter r will identify the pivot currently used.

In the while-loop, each symbol α in the r-pivot is compared with the symbol β in every child y of the node x. If α matches β, three cases exist. First, if a candidate k-sequence is found, its count is increased by one. Second, if y is not a leaf node and α is not in the last transaction (i.e., T_n), a further traversal of the candidate tree is required. Notice that the parameter r passed in the recursive call is determined according to Lemma 1. If neither case holds, both the next element of r-pivot and the next child of x are considered in the next iteration. On the other hand, if α does not match β, either the next element of the r-pivot or the next child of x is considered.

4 Performance Evaluation

In this section, we compare our approach with AprioriAll, Pseudo-Projection, and SPAM under different parameter settings. We implement all the approaches and follow the standard procedure in [2] to generate the synthetic databases. The number of distinct items is fixed to 1000, and the remaining parameters unmentioned are set as the default values. In addition, we modified AprioriAll to divide the candidates generated in a pass into groups such that each group of candidates can be placed in the main memory. In the following, we show the results from three experiments. At first, we compare the four approaches on the various settings of minsup. A number of databases are tested and the results are consistent. For lack of space, we only report the results from the database D10kC10T10S10I10.

In Figure 1, each point of a curve stands for the execution time of one approach. Generally, Pseudo_Projection and SPAM perform worse than our approach when minsup gets small, because of the larger amount of frequent sequences which leads to more database scans for support computation. Moreover, SPAM reports the worst processing time due to the following reasons. The computer hardware architecture cannot fully support very large bit-maps. SPAM needs additional CPU time to do the logical operations on bit-maps. As the minsup gets small, a large number of candidates need to be processed. As a result, the load of doing bit operations becomes very heavy.

Our approach outperforms M-AprioriAll (the modified AprioriAll) lightly when minsup is high. This is because only a few frequent itemsets exist. M-AprioriAll encodes all the frequent itemsets at the beginning of Phase 2, while our approach encodes them in different rounds. As a result, our approach needs more database scans than M-AprioriAll for the encoding process. However, when considerable frequent itemsets exist, the transformed database will be very large produced in M-AprioriAll, and many non-frequent sequences are generated as candidates. Moreover, the problem of repeated enumeration of identical subsequences from a customer sequence may also become worse when more frequent itemsets exist. Therefore, M-AprioriAll is worse than Pseudo_Projection and our approach.

When the database size is getting larger, all the approaches spend more time on database scans. In the second experiment, we compare the four approaches on different numbers of customer sequences, and the results are shown in Figure 2, where minsup is fixed to 50. All the three approaches are worse than our approach because Pseudo-Projection and SPAM frequently scan the databases, and M-AprioriAll produces a large transformed database.

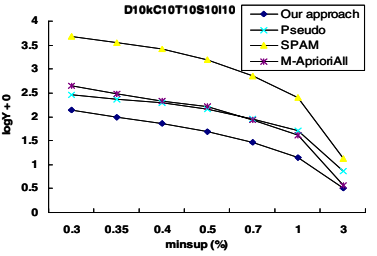

Fig. 1. Processing time for different minsup values

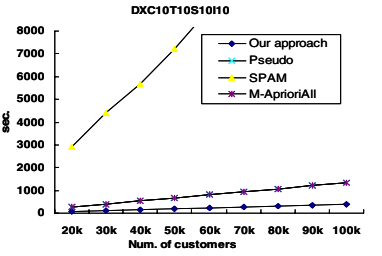

Fig. 2. Processing time on different database sizes

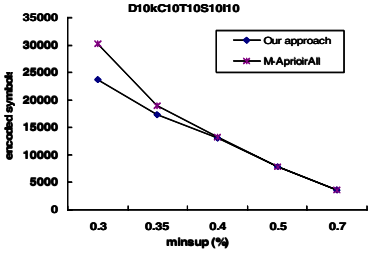

Fig. 3. The number of encoded symbols on different minsup values

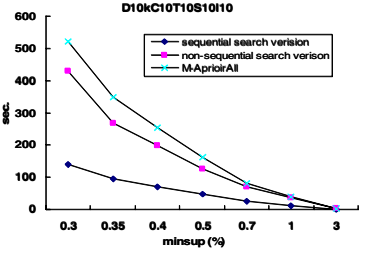

Fig. 4. Processing time of Phase 2

Finally, we compare Phase 2 in our approach with that of M-AprioriAll in two aspects, i.e., the number of symbols encoded and the process time on various values of minsup. For the number of symbols encoded, in Figure 3, as the minsup gets smaller, the number of encoded symbols reduced by our approach gets larger. Figure 4 shows the detailed comparisons between our approach and M-AprioriAll. We implemented two versions of our approach. One follows the idea proposed in Section 3.3, while the other does not provide the sequential search on the candidate tree. The former is called sequential version and the latter is called non-sequential version. As a result, the sequential version of our approach has the best performance than the others and the non-sequential version still outperforms than M-AprioriAll.

5 Conclusions

In this paper, we propose a novel approach to mining frequent sequences in large sequence databases. We adopt the two-phase architecture that generates frequent itemsets and frequent sequences separately. In Phase 1, we propose the concept of transaction intersection to successfully adapt the FP-tree approach for association rules mining to the sequence database. In Phase 2, we encode the frequent itemsets in different rounds such that the size of the transformed database, the number of candidates, and the number of enumerated subsequences are reduced. In addition, we also avoid enumerating the redundant subsequences from a customer sequence and

sequentially traverse the candidate tree for the subsequence enumeration of a customer sequence. We perform experiments based on various database sizes and minimum supports to compare our approach with M-AprioriAll, Pseudo-Projection, and SPAM. Moreover, we compare our approach with that of M-AprioriAll in the number of symbols encoded. We also implemented various versions of Phase 2 in our approach to compare with Phase 2 of M-AprioriAll. The results show that our approach is more efficient than the others in total processing time.

Acknowledgements

This work was partially supported by the NSC Program for Promoting Academic Excellence of Universities (Phase II) under the grant number 93-2752-E-007-004-PAE, and the NSC under the contract number 93-2213-E-004-013.

References

1. Agrawal R., Srikant R.: Fast Algorithm for Mining Association Rules. Proceedings of International Conference on Very Large Data Bases. (1994) 487-499.
2. Agrawal R., Srikant R.: Mining Sequential Patterns. Proceedings of International Conference on Data Engineering. (1995) 3-14.
3. Agrawal R., Srikant R.: Mining Sequential Patterns: Generalizations and Performance Improvements. Proceedings of the Fifth International Conference on Extending Database Technology. (1996) 3-17.
4. Ayres J., Gehrke J., Yiu T., Flannick J.: Sequential PAttern Mining using A Bitmap Representation. Proceedings of ACM SIGKDD Conference. (2002) 429-435, 2002.
5. Chiu D. Y., Wu Yi. H., Chen A. L. P.: An Efficient Algorithm for Mining Frequent Sequences by a New Strategy without Support Counting. Proceedings of International Conference on Data Engineering. (2004) 375-386.
6. Pei J., Han J., Mortazavi-Asl B., Pinto H., Chen Q., Dayal U., M. Hsu.: PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. Proceedings of International Conference on Data Engineering. (2001) 215-224.
7. Wang K., Tang L., Han J., Liu J.: Top Down FP-Growth for Association Rule Mining. Proceedings of Advances in Knowledge Discovery and Data Mining. (2002) 334-340.
8. Zaki M. J.: An efficient algorithm for mining frequent sequences. Machine Learning, Vol. 42(1/2). (2001) 31-60.